

Crank Storyboard

Crank Storyboard

Table of Contents

1. What Is Storyboard	1
A New Way to Build Embedded User Interfaces	1
Storyboard Designer and Storyboard Engine	1
A Non-Compiled Solution	2
Storyboard Software Updates	2
Compatibility with Previous Versions	2
2. Storyboard Architecture	3
Application Model Hierarchy	3
Event Driven Interaction Model	5
Events to External Applications	6
Data Binding for Dynamic Behavior	7
Context and Variables	9
Storyboard Model Internal Variables	10
Animation Definitions	12
3. Typical Development Workflow	14
Start a New Project: From Photoshop or from Scratch	14
Create and Organize Screen Content	14
Bind Events to Invoke Actions	14
Simulate and Export Model for Engine	15
4. Storyboard Designer Environment	16
Storyboard Designer Workbench	16
Anatomy of a Storyboard Designer Project	17
Storyboard Designer Editor	19
Editing Content	19
Editor Toolbar	20
Direct Editing	20
Storyboard Resource Export Configuration Editor	21
Managing Configurations	21
Resource Tree	22
Resource Export Options	22
Application Footprint Preview	25
Storyboard Designer Views	25
Actions View	25
Application Model View	26
Animation Timeline View	27
Images View	31
Layers View	32
Metrics View	33
Navigator View	35
Outline View	36
Problems View	37
Properties View	38
Component View	39
Variables View	39
Notes View	42
Storyboard Designer Utilities	43
Design Notes	43
GoTo Dialog	44
Storyboard Search Dialog	45
Resize Storyboard Application	46
Resource Clean Up Wizard	50

Consolidate Images Wizard	51
Trim Images Wizard	52
Split Images Wizard	53
Merge Control Images	54
Storyboard Performance Log Viewer	54
Predefined Record Plots	55
Custom Record Plots	56
Animation Plots	57
Analyzing Applications Using Record Plots	58
Toggling Event Lines	59
Plot Selector	60
Chart UI Elements and Usage	61
5. Creating A Storyboard Project	65
New Empty Storyboard Project	65
New Project from a Photoshop PSD File	66
New Project from a Storyboard Sample	67
New Project from Storyboard Embedded Engine File	68
Existing Project Import	69
Working with Multiple Application Design Files	71
Controlling Feature Enablement on Projects	72
6. Adding Content to your Application	74
Object Naming and Reserved Namespaces	74
7. Working with Events	76
Event Editor	76
Opening the Event Editor	76
Adding an Event Definition	76
Data Element Types	79
Changing Element Array Length	79
Reordering Event Data Elements	80
Editing Min and Max Attributes	80
Using the Storyboard IO Connector	80
Live Mode	81
Saved Mode	83
Data Element Editors	84
Storyboard IO Event C/C++ Header Export	85
8. Connecting Events to Actions	86
Event and Action Matching Rules	88
9. Using Variables to Create a Dynamic UI	91
Table Variables	92
Triggering Events on Variable Changes	94
10. Creating and Applying Animations	95
Animation Action	95
Timer Keyframe Animations	96
Screen Transition Animations	96
11. Simulating your Application	98
12. Scripting with Lua	100
Lua Action Callback Function	100
Passing Extra Parameters to Functions	101
Lua Execution Environment	101
Asynchronous Lua Support	102
Lua Debugger	102
Lua Executables	105
13. Working with C Callbacks	107
What are C Callbacks?	107

C Callbacks on Windows	107
Example C Callback	107
C Callback Export Labels	108
Exporting C Callbacks	109
C Callback Export Wizard	109
C Callback Command Line Export	110
14. Working with Design States	112
What are Design States?	112
Creating a Design State	112
Editing a Design State	113
Master/State Context	113
Storyboard Editor	114
Properties View	114
Variables View	114
Manually Editing Design States	115
Removing Changes From Design States	115
Hiding/Showing Design States	116
All Design States	116
Individual States	116
Converting a Design State	117
Animation Preview with Design States	117
15. Working with Images	119
Image Rotation	119
Rotate At Center	119
Rotate At Custom Point	120
Alpha and Transparency in Images	121
Creating Scalable 9-Patch Bitmap Images	121
Multi-Frame Animated GIF Images	122
16. Working with Text	123
Rich Text Styling and Markup	123
Translation and Internationalization	126
Text Translation View	127
Translating a Storyboard Application	129
Creating and Editing Translation Content CSV Files	131
Script Specific Text Shaping and Layout	133
17. Working with Touch, Gestures and User Input	134
Configuring Touchscreen Input	134
Windowed Applications	134
Gesture Support	134
Multi-Touch Gestures	135
gre.mtmove	135
gre.mtpinch	136
gre.mtrotate	136
Enabling Gesture In Your Application	137
18. Creating Lists and Tables	140
19. Working with Scrolling Content	143
Scroll Synchronization	144
20. OpenGL and 3D Rendering	146
Storyboard 3D Rendering Model	146
3D Rendering Fundamentals	146
The Scene Graph and Transformations	146
Material Support	147
Animation and Variable Support	148
Mapping FBX Animation data into meaningful structures	148

Support for Animation Takes	149
Troubleshooting 3D Problems	149
Working with OpenGL Shaders, Transforms and Compressed Textures	150
3D Transforms and Custom Shaders	150
Custom Shader Support	153
Compressed Textures	154
21. Working with Audio and Video	155
Media Backend Services	156
GStreamer Application	156
GStreamer Pipeline	156
FFmpeg Plugin	156
22. Multi-File Application Development	158
Simulating and Exporting Multiple Model Files	159
Resolving Conflicts and Synchronizing Changes	160
23. Reusable Graphical Components	162
Creation Guidelines and Conventions	162
Editing Components and Propagating Changes	163
24. Collaboration and Team Development	164
Revision Control System Integration	164
Comparing and Merging Model Files	164
Comparing and Merging Projects	166
Exporting Storyboard Projects for Sharing	167
25. Exporting and Running on your Embedded Target	169
Export Workflow	169
Selecting Files For Export	171
Deployment Bundle Packages	171
Storyboard Embedded Engine (GAPP)	171
Storyboard Compressed Package (SBP)	171
Native Android Application (APK)	171
Native iOS Application	172
Windows Standalone Launcher (EXE)	172
Storyboard Embedded Resource Header (C/C++)	172
Export Transfer Methods	173
Filesystem	173
SCP Transfer	173
Exporting from the Command Line	174
Exporting from the Command Line using Export Configuration	176
Additional Options	176
Setting up Storyboard Engine	177
Font Environment Variable	177
QNX Screen Environment	177
Running Storyboard Engine	178
Target Specific Configurations	178
Linux x86, armle	178
Microsoft WinCE, Compact7 win32, armle	180
Yocto Jethro Linux kernel (3.14) OpenGL, FBDEV, armle	180
26. Working with Storyboard Lite Platforms	181
What is Storyboard Lite	181
Storyboard Lite Installers	181
Design Considerations	181
27. Working with Mobile Platforms: Android and iOS	182
Exporting to iOS Devices	182
Xcode	182
iOS Developer Account	182

Code Signing Certificate	182
Device IDs	182
Application IDs	183
Provisioning Profile	183
Adding Extra Libraries for iOS	183
Exporting to Android Devices	184
Adding Extra Libraries for Android	185
Storyboard Lua Android Integration	187
28. Sending and Receiving Data with Storyboard IO	193
Storyboard IO Transport protocols	193
Connecting to a Storyboard Application	194
Storyboard IO Over TCP	194
Sending Events to a Storyboard Application	195
Event Naming Conventions	196
Serialized Data and the Event Format String	196
Setting Application Data	199
Receiving Events from a Storyboard Application	200
Debugging Storyboard IO	201
Debugging egress (Storyboard Engine sending to backend)	201
Debugging ingress (Storyboard Engine receiving from backend)	202
Storyboard IO Utilities	203
iogen	203
iorcv	204
29. Optimizing Your Storyboard Application	205
Measuring Performance	205
Action Execution Performance Considerations	206
Choosing the Right Image Format(s) Bit Depth	206
Framerate (Frames Per Second)	206
Scaling Images	207
Reducing Output Verbosity	207
Adjusting Engine Rendering Options	207
Managing Resource Memory	207
OpenGL Scene Graph Optimization	208
30. Extending Storyboard Functionality	209
User Defined Action Templates	209
User Defined Render Extension Templates	210
31. Structuring Your Photoshop Import Content	212
PSD file Structure for Import into Storyboard	212
PSD File Requirements	212
Application Model Hierarchy	212
Artboards	214
Layer Effects and Blending Modes	215
Naming Requirements	215
Naming Conventions	215
name_layer	216
name_group	216
name_control	216
name_up and name_down	216
Common Photoshop Elements	216
Transparency	216
Naming Convention Deviation	217
Sub Group Folders	217
32. Re-Importing Photoshop and Updating Content	218
Re-Importing Photoshop Content	218

A. Storyboard Lua API	222
Storyboard Lua API	222
gre.APP_ROOT	222
gre.SCRIPT_ROOT	222
gre.PLUGIN_ROOT	223
gre.LOG_constants	223
gre.LEFT, gre.RIGHT, gre.CENTER, gre.TOP, gre.BOTTOM	223
gre.OPAQUE, gre.TRANSPARENT	224
gre.set_data	224
gre.get_data	224
gre.set_value	225
gre.get_value	226
gre.resolve_data_key	226
gre.get_control_attrs	227
gre.set_control_attrs	228
gre.get_table_attrs	229
gre.set_table_attrs	230
gre.get_table_cell_attrs	230
gre.get_group_attrs	231
gre.set_group_attrs	232
gre.get_layer_attrs	233
gre.set_layer_attrs	233
gre.set_layer_attrs_global	236
gre.screen_attach_layer	236
gre.resize_control	237
gre.move_control	238
gre.move_layer	238
gre.set_focus	239
gre.get_focus	240
gre.send_event	240
gre.send_event_target	241
gre.send_event_data	242
gre.greio_disconnect	243
gre.clone_object	244
gre.delete_object	245
gre.clone_control	245
gre.delete_control	245
gre.poly_string	246
gre.get_string_size	246
gre.load_resource	247
gre.dump_resource	248
gre.walk_pool	249
gre.load_image	249
gre.timer_set_timeout	250
gre.timer_set_interval	251
gre.timer_clear_timeout	251
gre.timer_clear_interval	252
gre.animation_create	253
gre.animation_add_step	254
gre.animation_destroy	254
gre.animation_trigger	255
gre.animation_stop	256
gre.animation_pause	257
gre.animation_resume	258

gre.animation_create_tween	259
gre.touch	260
gre.key_up	261
gre.key_down	261
gre.key_repeat	262
gre.redraw	263
gre.quit	263
gre.thread_create	264
gre.receive_event	265
gre.env	266
gre.log	267
gre.mstime	268
gre.rgb	268
gre.torgb, gre.to_rgb	269
gre.to_alignment	270
gre.to_alpha	270
gre.rtext_text_extent	270
gre.perf_trace_point	271
gre.perf_trace_duration	271
Storyboard Lua Canvas API	272
gre.get_canvas	272
CANVAS:get_dimensions	272
CANVAS:fill	273
CANVAS:fill_rect	273
CANVAS:fill_poly	274
CANVAS:stroke_line	274
CANVAS:stroke_rect	275
CANVAS:stroke_poly	275
CANVAS:clear_rect	276
CANVAS:set_pixel	276
CANVAS:set_alpha	277
CANVAS:set_line_width	277
CANVAS:draw_image	278
CANVAS:draw_text	278
Storyboard Lua DOM Module	279
gredom	279
DOMOBJECT	281
CONTROL	284
TABLE	290
GROUP	291
LAYERINSTANCE	293
Lua DOM Samples	297
B. Storyboard IO API	299
Storyboard IO API	299
gre_io_add_mdata	299
gre_io_close	299
gre_io_free_buffer	300
gre_io_grow_buffer	300
gre_io_open	300
gre_io_receive	301
gre_io_send	301
gre_io_send_mdata	302
gre_io_serialize	302
gre_io_size_buffer	303

gre_io_unserialize	303
gre_io_zero_buffer	304
gre_io_get_error_codes	304
gre_io_get_error_message	305
C. Storyboard Engine and Plugin Options	306
Storyboard Engine Plugin Options	306
D. Standard Event Definitions	320
Standard Event Definitions	320
System Events	320
Pointer Events	321
Keyboard Events	329
Screen Manager Events	330
Focus Events	330
Table Events	331
Table Scroll Events	332
Layer Scroll Events	333
Mobile Events (Android and iOS)	334
Android Events	335
Windows Embedded Compact 2013 (WEC2013) Events	335
Plugin Specific Event Definitions	336
Timer Events	336
Animation Events	336
Gesture Events	337
Screen Display Capture (ScreenDump) Events	337
Screen Event Capture/Playback Events	338
Media Events	338
Logger Events	340
E. Standard Action Definitions	341
Built-in Action Definitions	341
gra.screen	341
gra.screen.fade	341
gra.screen.hold	341
gra.screen.release	341
gra.sendevent	341
gra.datachange	342
gra.screen.focus.set	342
gra.screen.focus.next	342
gra.screen.focus.prev	342
gra.screen.focus.direction	343
gra.table.scroll	343
gra.table.resize	344
gra.table.navigate	344
gra.scroll.stop	345
gra.log	345
gra.resource.dump_def	345
gra.playback	345
gra.capture	345
Plugin Action Definitions	346
gra.lua	346
gra.ccallback	346
gra.animate	346
gra.animate.stop	347
gra.audio	347
gra.greio	347

gra.perf_state	348
gra.redirect	348
gra.screen.path	348
gra.screen.scale	349
gra.screen.glswitch	349
gra.screen.glrotate	350
gra.screen.glflip	350
gra.screen.gldoors	351
gra.screen.gltip	352
gra.screen.glcube	352
gra.screen.rotate	353
gra.screendump	353
gra.system	353
gra.timer	353
gra.timer.stop	354
Media Actions	354
F. Standard Render Extension Definitions	357
Common Render Extension Options	357
Canvas	358
Circle and Arc	359
External Buffer	359
Fill	359
Image	359
Polygon	360
Rectangle	361
Text	361
3D Model	362
G. Storyboard Engine Public API	364
gr_application_create_args	364
gr_application_create	365
gr_application_free	365
gr_application_run	366
gr_application_quit	366
gr_application_debug	367
gr_app_log	367
gr_application_set_data	368
gr_application_set_data_variable	368
gr_application_get_data	369
gr_application_get_data_variable	370
gr_application_add_event_listener	370
gr_application_rem_event_listener	371
gr_application_send_event	371
gr_context_get_application	372
gr_context_max_fqn	373
gr_context_get_row	373
gr_context_get_column	373
gr_context_get_control	374
gr_context_get_group	374
gr_context_get_layer	375
gr_context_get_screen	376
gr_context_get_fqn	376
gr_context_get_event_name	377
gr_context_get_event_data	377
I. Storyboard Tutorials	379

33. Importing Sample Projects from Crank's Public SVN	381
34. Working with Multiple Application Design Files	386
Creating a Project	386
Resolving Conflicts	388
35. Creating a 3D Model Application	389
New Project	389
3D Model Control	389
Resize Model	391
II. Storyboard Demo Images	392
36. NXP	395
i.MX 6QuadPlus	395
Demo Details	395
i.MX 6UltraLite	395
Demo Details	395
i.MX 6ULL	396
Demo Details	396
i.MX 6DualLite	396
Demo Details	396
i.MX 6SoloX	396
Demo Details	396
Toradex i.MX 7Dual SoM	397
Demo Details	397
Flashing the Image	397
U-Boot	398
Copying Image to an SD Card - Linux	398
Copying Image to an SD Card - Windows	398
Running the Image	398
Turn On or Off the Console	398
37. STMicroelectronics	400
STM32F429	400
Demo Details	400
STM32F439	400
Demo Details	400
STM32 F7	400
Demo Details	400
Required Hardware and Software	401
Package Contents	401
STM32 ST-LINK Utility	402
Running the Demo	404
Importing the Demo into Storyboard Designer	405
Creating Your Own Demo	405
38. Microchip	406
SAMA5D2	406
Demo Details	406
SAMA5D2 with 7 inch Display	406
Demo Details	406
Copying Image to an SD Card - Linux	407
Creating an Image - Windows	407
Running the Image	407
39. Renesas	408
RZ/A1	408
Demo Details	408
Flashing the Image	408
40. Linux	409

TI AM355 Starter Kit	409
Step 1: Importing A Storyboard Sample	409
Step 2: Exporting A Storyboard Application	411
Step 3: Selecting The Storyboard Embedded Engine	413
Step 4: Configuring The Target Platform	413
Step 5: Running The Storyboard Application	413
Raspberry Pi	414
III. Release Notes	415
41. Release Notes 6.2	417
New Features and Functionality	417
Designer	418
Engine	418
Lua Scripting	419
Behavioural Changes and Deprecations	419
Known Issues	419
42. Release Notes 6.1	421
New Features and Functionality	421
Storyboard Designer	421
Storyboard Engine	422
Lua Scripting	422
Behavioural Changes and Deprecations	422
43. Release Notes 6.0	423
New Features and Functionality	423
General	424
Storyboard Designer	424
Storyboard Engine	424
Behavioural Changes and Deprecations	425
IV. Licensing	427
44. Storyboard Licensing	429

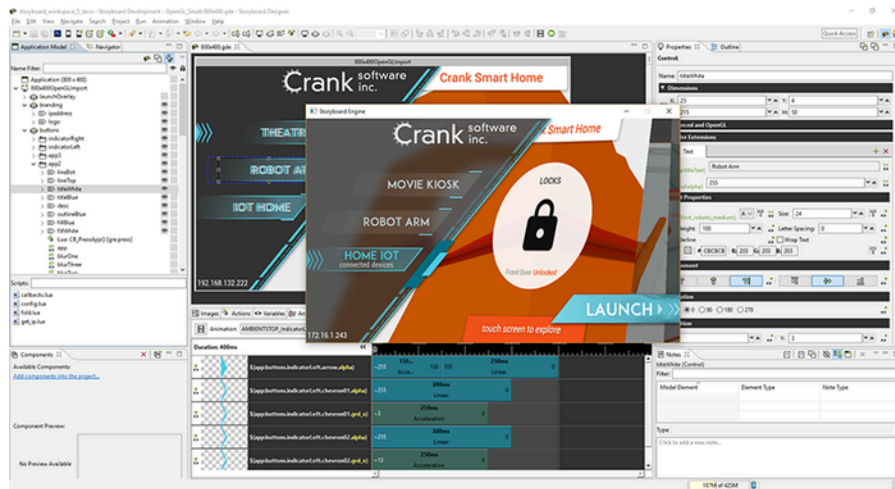
List of Tables

16.1. Supported Rich Text Tags	123
16.2. Supported Style Attributes	124
25.1. Options	178
28.1. No events are being received	201
28.2. A particular event is not being received	202
28.3. Event data not correct	202
28.4. The trouble shooting steps for egress Storyboard apply in this scenario, so see the above steps.	202
C.1. Plugin Options	306

Chapter 1. What Is Storyboard

A New Way to Build Embedded User Interfaces

Storyboard takes a design centric approach to creating embedded user interfaces. Users create interfaces in a desktop development environment by directly importing content from graphic design tools such as Photoshop and by dragging and dropping standard image assets into position to build up screens.



Storyboard isn't a traditional development environment. With traditional development methodologies software developers build user interfaces using functionality libraries and are unable to easily validate the look and feel of the interface while they are coding. User interfaces developed using Storyboard are immediately visible and teams can test and validate the interaction, graphics, and user experience during development via desktop simulation.

Unlike widget frameworks where you instantiate pre-configured elements and attempt to customize them, Storyboard Components offer a way of quickly creating reusable display elements that can be associated with logic that is independent of the presentation.

Storyboard Designer and Storyboard Engine

Storyboard is composed of two parts: Storyboard Designer and Storyboard Engine.

Storyboard Designer is the desktop development environment that designers and engineers use to assemble their user interface application. In this environment you can quickly import structured content from existing tools, such as Photoshop or 3ds Max, or as standard image and font files and assemble the visual structure of your application.

Once an application structure is assembled, Designer exports the application as a sophisticated state machine application model that is used as input to the Storyboard Engine. This application model is not generated source code that needs to be compiled (i.e. C/C++) and is completely independent of the target execution environment.

Storyboard Engine is the target-specific executable (or library in some scenarios) that loads the model generated by Designer and starts the execution of the state machine. Storyboard Engine is a highly optimized execution environment. Each engine is configured for a specific CPU architecture (arm, x86, mips...), operating system (Linux, QNX, Android, FreeRTOS...) and rendering technology (OpenGL, OpenVG,

software framebuffer, custom ...). This structure makes porting the Storyboard Engine to a new platform configuration relatively risk free and a deterministic process.

A Non-Compiled Solution

The application model that is generated as output from Designer does not require compilation and is target environment agnostic. This makes it straightforward to generate applications that execute in a variety of different environments, such as an application that can work both on an embedded device (for example a wall thermostat) and in a mobile context.

Another advantage of being hardware and operating system agnostic is that elements of the user interface can be used as part of a product's hardware evaluation process. It takes only minutes to create a rich animated application when working with Designer's Photoshop import functionality. This application can be immediately evaluated on any one of the over forty different platform configurations spanning more than a dozen operating systems and popular rendering technologies.

Storyboard Software Updates

Storyboard minor updates are provided through a software update installer within Storyboard Designer.

When Storyboard Designer starts, it automatically checks for updates and again every four hours while running. Users can also force a check for updates from within Storyboard Designer by selecting Help > Check for Updates from the main menu. If an update exists you will be notified to apply it and a wizard will guide you through the download and installation process.

Updates won't take effect until you restart Storyboard Designer.

Major updates are provided as a separate stand-alone installer. It is not recommended to install new versions of Storyboard on top of each other as there may be a number of file conflicts leading to instability. Install Storyboard versions to separate directories. You can install and run multiple versions of Storyboard on a system at the same time.

Compatibility with Previous Versions

Storyboard Designer 6.0 can import and work with projects from Storyboard Designer 4.0 and later (i.e. 4.0 - 4.2, 5.0 - 5.3). Projects created with versions of Storyboard Designer older than 4.0 must first be converted to an updated file format using a Storyboard Designer 4.x or Storyboard Designer 5.x release. Contact support@cranksoftware.com if you need assistance in performing this conversion.

When migrating to new versions of Storyboard save and archive a snapshot of the project if you plan to continue to work with it in an older Storyboard Designer environment.

To use a Storyboard project from an earlier release than it was created (e.g., using a version 6.0 project with a version 5.0 environment), the best approach is to 'back migrate' the project using a Storyboard Engine export and then a Storyboard Engine import. The export to a Storyboard Engine creates a runtime file (*.gapp) from the design file (*.gde). The Storyboard Engine file contains only the information required to run the application and does not contain all of the design material (such as notes and unused layers) so that content will be lost. Features that are not present or used by the older Storyboard version will be lost on import since it is not possible to use them with an older environment.

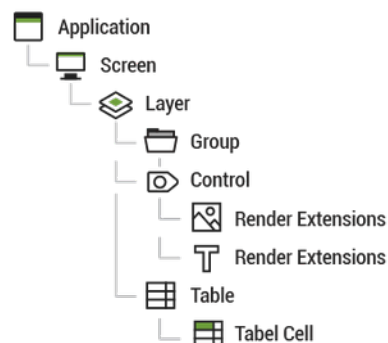
Chapter 2. Storyboard Architecture

There are three fundamental Storyboard concepts that you must understand when building user interface applications with Storyboard: The application model hierarchy, the event based interaction, and using variables for manipulating dynamic content. Having a solid grasp of these fundamentals will guide you from creative concept to implementation.

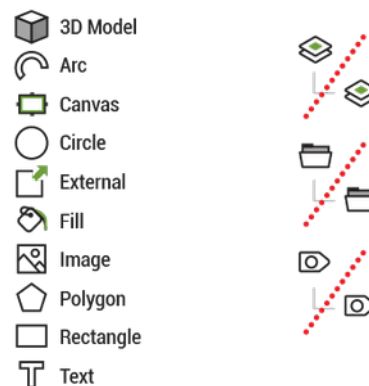
Application Model Hierarchy

An application is organized as a hierarchy of screens, layers, groups, and controls that are collectively called model elements.

Application Model

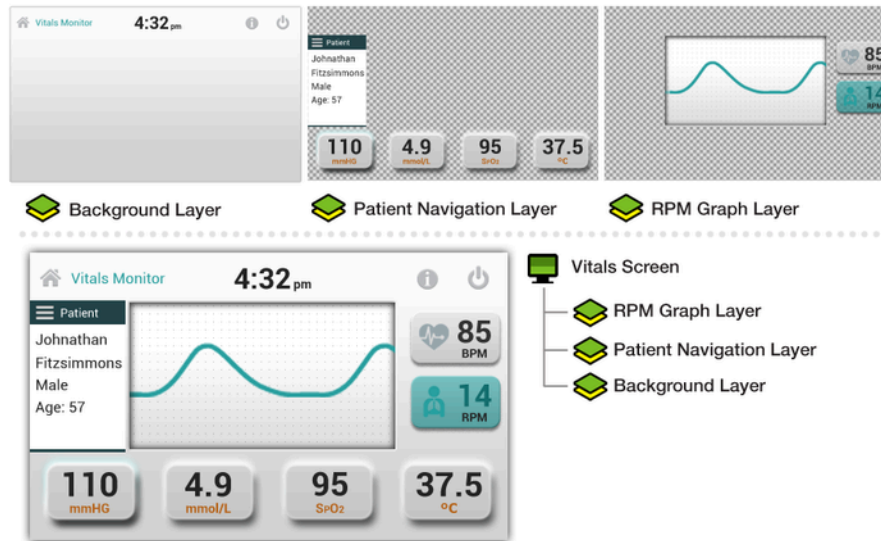


Render Extensions



Screens are used to represent contextual visual states of your application. When working within Designer your primary viewport into how the application is organized will be through the screens of the application that are defined. Structurally a screen is an ordered list of layers that are composited together to form the final screen display. In a Storyboard application only one screen can be active at a time.

Layers are used to organize related visual elements together in a reusable manner. In this case, reuse means reusable in different screen contexts so that the same layer can be associated with several different screens. When layers are shared among multiple screens, for example, as a common menu or background, then the content of the layer is exactly the same in those different screen contexts. If content changes on the layer in one screen context, then the content will change across all screens that use that layer. While the content for layers may be shared, there are several properties that are tied specifically to how the layer is used within a screen. These properties are called layer instance properties since they relate directly to a particular instance of a layer and include the x/y position of the layer, the visibility/hidden state, and the transparency or alpha blending value of the layer.



Since the layers have a configured size, content can be positioned within the layer and will clip to the bounds of the layer and not draw outside of it. By adjusting the position of a layer, you can move a significant amount of visual content with minimal effort. Layers however, are still containers for other structural objects, groups, and controls.

Controls are the primary user interaction points for most applications. Controls contain an ordered list of render extensions which are the base Storyboard rendering and drawing primitives. Example render extensions include image, text, fills, arcs, and polygons. A control has a size and all of the drawing performed by the render extension happens within the area of the control, which means a control is a clipping region. Similar to layers, controls also have properties such as position and visibility.

Controls are the basic building blocks for assembling a user interface. The same visual effect may be achieved by using a single control with multiple render extensions stacked on top of each other and positioned within the area of the control, or by using multiple controls that together make up the same visual presentation, or a hybrid combination of multiple controls and multiple render extensions. In cases where multiple controls are used to achieve a visual effect, we want to maintain cohesion. For this, it can be useful to put the controls into a group.

Groups are a collection of controls. Unlike a layer however, a group does not clip its content and the size of the group is determined by the bounds of the controls it contains. A group does have a position and all of the controls within the group will have a position that is relative to the group's position. In this way, groups are similar to layers in that changing the position of a group will translate all of the controls at once or changing the visible/hidden state will cause all of the controls contained to either be shown or hidden.

Groups and controls are both child objects of a layer, but groups can only contain controls. There is currently no grouping of groups, though that may be a future consideration.

Tables are a special form of templated control that is repeated over and over in a grid or list like pattern. Tables and their table cell control templates are treated the same as controls, but use a different representation for dynamic data. This topic is covered in more detail in the chapter Chapter 18, *Creating Lists and Tables*.

The application is the root of this hierarchy. The application maintains the list of all screens, layers, groups, and controls and determines at the start of execution which screen is the first displayed. The start screen setting is a property of the screen model object and can be set within Designer. Every application must have at least one screen and every screen must have at least one layer to be valid. Designer will generally raise an error on export if these conditions aren't met within the application design.

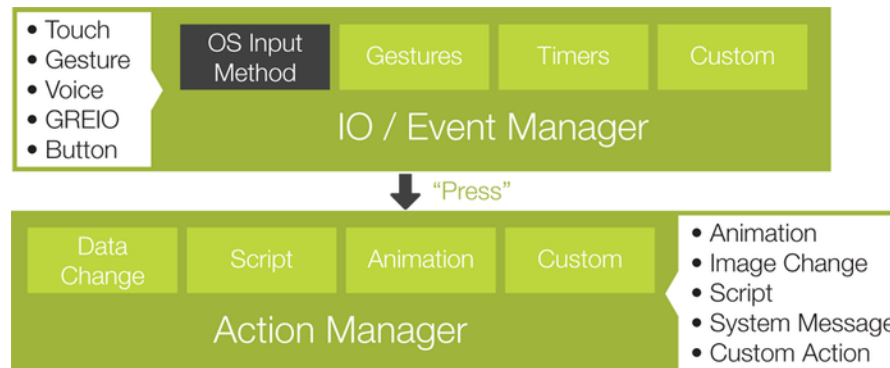
Event Driven Interaction Model

Storyboard is an entirely event-driven user interface state machine. Events are the basic communication mechanism for triggering activity and passing data in a Storyboard application.

Events are named entities and optionally carry an extra binary data payload. Events come in three main flavours:

- Traditional events such as press, release, motion, and keydown are generated by standard input sources. As these events are generally common, Storyboard provides a public definition for these events so that they can be externally generated in a consistent manner.
- Application events are events that are specifically related to the user interface and are designed to trigger activity from within the user interface itself. Events used this way form an internal API for the application and are generally private as they may be used solely for triggering a common UI experience such as "start_animation" or "goto_screen".
- External events are events that typically make up the data communication API with a backend service that contains the core data model of the product. These events frequently contain a domain-specific data payload related to the application. External events represent the API between front-end user interface and backend service and are specific to a product and should be well-defined to allow parallel development to occur among front and backend developers.

Events may be received from multiple different input sources, but are always processed serially by the Storyboard Engine.



Events trigger actions. Actions perform tasks; manipulate data, interact with the system, log messages, generate more events. There are many different actions that perform different things. Screen transition actions perform visual screen transitions. Data change and animation actions modify data within the UI that can cause a visual change. Timer actions are used to trigger new events in either a repetitive or one-shot manner, while Lua script actions invoke a callback function scripted in Lua (Lua is a script programming language used for glue logic in Storyboard). In addition to all of the fixed actions, the Storyboard SDK provides users with the ability to write and contribute their own actions to the engine.

The event/action association can be made on any of the model elements. For example, a control with an image render extension that is acting like a push button might have an event/action association that would match a press event and cause a data change action that hides or shows a control or perhaps changes the value of a variable.

Where the event/action association is made is important. Associations that are made on controls must be triggered by position-related events, such as press and release, otherwise the control must use focus settings to ensure that non-position based events (such as external events) are delivered to their context.

Associations that are made on layer or group model objects are more generic, but in order to be triggered the event must first be potentially triggered against a control in that context. Screen actions will always be matched as long as the screen is the currently visible screen. Application event/action bindings are the most general and will match in all contexts and are generally the most suitable when matching external events and processing model data.

The event-driven model that Storyboard uses for user interaction has significant benefits that may not be immediately apparent. The first is that the events form a discrete and defined interface point between the UI and the rest of the product. By isolating the external events:

- Graphic designers working on the user interface have a clear expectation regarding the data they have available to interpret.
- Test developers can exercise both the frontend and backend by generating simulated events with artificial data.
- Backend software developers have a clear direction on what data they are making available and do not have to overexpose or overdevelop their interfaces for functionality that will never be used.

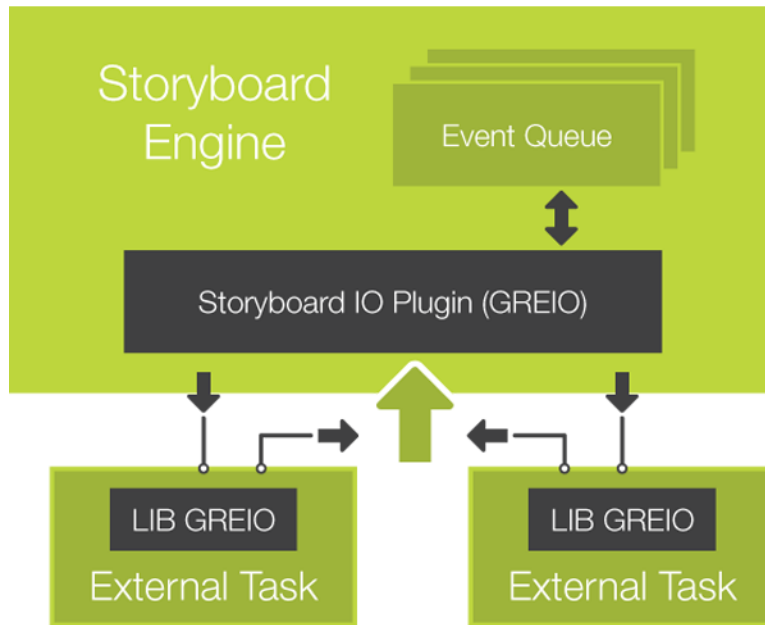
The event model also ensures a complete decoupling of the UI from the implementation and is what makes desktop simulation such a practical reality. Simulated data can be easily injected in environments where the real target hardware may not be ready, prohibitively expensive, or just plain awkward to work with. Building the UI in a comfortable environment and knowing that it will move readily to the embedded target at the time of deployment with small adjustments is an invaluable boost to overall productivity.

Events to External Applications

Communication with external processes in the embedded system can be accomplished in several ways. One approach that provides a strong API while maintaining a loose coupling for the implementation is to use Storyboard IO.

Storyboard IO is provided as a plugin for the Storyboard Engine and includes a C API and library for external applications to link against.

When the Storyboard IO plug-in is loaded a channel is created in order for processes to inject events into the system. A single event queue is used to serialize the events and therefore any events sent via Storyboard IO will be placed in the queue with standard Storyboard system events. If the external application wishes to receive events, it can create its own Storyboard IO channel which can have events sent through. Applications can have multiple receive channels and the Engine has a single input channel. The following diagram illustrates an application which can send events to the Engine and review events on a named channel.



The chapter Chapter 28, *Sending and Receiving Data with Storyboard IO* discusses how to work with Storyboard IO to send and received events while the C API is covered in detail in the appendix Appendix B, *Storyboard IO API*.

Data Binding for Dynamic Behavior

Render extensions associated with controls render the content, from text and images to arcs and polygons. Each render extension's behavior is controlled through a set of properties that the user can configure, such as fill color, text size and font, and image filename. By default, the settings for these properties are fixed at design time and can't be changed dynamically at runtime. However, these settings can be dynamically adjusted at runtime by using Storyboard variables.

Attributes, for both render extensions and actions, that may require dynamic adjustment at runtime can be associated to named Storyboard variables.

Similar to the event/action association, variables can be bound to any of the model elements in the application. This ownership dictates how that variable will be referenced when it is associated with a property in a render extension or action. Storyboard variables are managed as loosely typed key/value pairs in the Storyboard Engine's data manager component. The key is the fully qualified name of the variable as it is declared within the model and the value is the value of the variable set initially within Designer and changeable at runtime through various actions, animations, or using the Lua API.

The Storyboard model is hierarchical, so the construction of a fully qualified model path is a straightforward process of joining model element name segments with a . (dot) in between them. The following list demonstrates how the fully qualified model name is formed for a variable, `varname`, associated with different contexts in the model.

<code>varname</code>	This identifies a variable, <code>varname</code> , as an application level variable
<code>screen_name.varname</code>	This identifies a variable, <code>varname</code> , as associated with the screen <code>screen_name</code>

<code>layer_name.varname</code>	This identifies a variable, <code>varname</code> , as associated with the layer <code>layer_name</code>
<code>screen_name.layer_name.varname</code>	This identifies a variable, <code>varname</code> , as associated with the layer instance <code>layer_name</code> associated with the screen <code>screen_name</code> . Most variables are not defined as layer instance variables, but rather as layer variables.
<code>layer_name.control_name.varname</code>	This identifies a variable, <code>varname</code> , as associated with the control <code>control_name</code> that is located on the layer <code>layer_name</code> . Groups variables can also be addressed in this same manner as groups are also children of layers. The variable, <code>layer_name.group_name.varname</code> identifies a variable <code>varname</code> within the group <code>group_name</code> .
<code>layer_name.group_name.control_name.varname</code>	This identifies a variable, <code>varname</code> , as associated with the control <code>control_name</code> that is located in the group <code>group_name</code> on the layer <code>layer_name</code> .

Note

There is some overlap in the Storyboard namespace that could lead to ambiguous resolution. To maintain a clear name resolution, layers and screens may not have the same names. Within a container, such as a layer or a group, all of the model element names must be unique. Storyboard Designer enforces this restriction.

Which model element owns a variable is usually a good indication of its scope. Variables that are owned by control model elements will typically have an association with render extensions or actions local to that control. A variable with a broader ownership, such as at the screen or application level, will likely have several different associations within the model.

For example, to represent a button with a changing image, one might create a variable on a control that is used by an image render extension on that control. To easily configure the fonts and text sizes used within the application, you might create several variables at the application level and have all of text render extensions refer to those variables.

Variables represent data that is used within the user interface but are not tied to any single context. The same variable may be referenced by different render extension clients or even associated with action properties.

When you want to make a change to the visual presentation of the user interface, you don't perform drawing calls directly, instead you make changes to one of the variables that you have created. The Storyboard Engine monitors the values of all variables. As variable content changes, Storyboard determines if the variable is used by the current on-screen content. If there are changes, the Engine automatically schedules a refresh operation and groups the refresh operations together for efficiency.

To associate a property with a variable or to change the value of a variable, it's important to understand how a variable is addressed within the application. The addressing scheme generally follows the model hierarchy. When a variable is sourced from the application root using this full path, it is called a fully qualified name.

Note that variables associated with layers and variables associated with screens have the same addressing path. This is because a layer and it's content can exist on multiple screens, and removing the screen component allows a consistent address path to groups and controls on that layer.

Context and Variables

It can be cumbersome to always use a variable's fully qualified name. In some cases it is more convenient to address a variable by its context of use. For example, render extensions are always associated with a control and in many instances the properties that they associate to variables will also be local to that control. In this situation it would be more convenient to refer to a variable as being 'local to the control'. Storyboard supports this through the use of a data context shorthand that can replace a variable's fully qualified name.

The data context is the context of execution of a particular activity. For render extensions, this means the parent control of the render extension. For actions, this means the model element associated with the triggering event. The shorthand notation is a string formatted as `${@@CONTEXT:NAME}` where CONTEXT is going to be one of app, screen, group, layer, or control indicating the model element in context to resolve, while the NAME is the name of the variable being addressed.

To reduce complexity of using the fully qualified model paths and to minimize the maintenance effort as a project evolves or changes, Storyboard defines several variable shortcuts that will expand their value based on the current model context in which they are being resolved.

<code>\${app:varname}</code>	Refers to the application variable <code>varname</code> .
<code>\${screen:varname}</code>	Refers to the screen variable <code>varname</code> .
<code>\${layer:varname}</code>	Refers to the layer variable <code>varname</code> .
<code>\${group:varname}</code>	Refers to the group variable <code>varname</code> .
<code>\${control:varname}</code>	Refers to the control variable <code>varname</code> .

For example, in the following Storyboard model:

```
Application
+ MainScreen
  + ALayer
    + AGroup
      + AControl
```

where the current focus is associated with the control `AControl`, reference to a variable `varname` would resolve to a fully qualified path as follows:

<code>\${app:varname}</code>	<code>varname</code>
<code>\${screen:varname}</code>	<code>MainScreen.varname</code>
<code>\${layer:varname}</code>	<code>ALayer.varname</code>
<code>\${group:varname}</code>	<code>ALayer.AGroup.varname</code>
<code>\${control:varname}</code>	<code>ALayer.AGroup.AControl.varname</code>

For tables, where the context also includes the row and column index of the table cell being adjusted, a special `${cell:CONTEXT:NAME}` notation can be used that converts the variable shorthand into a variable that contains the row and column as a postfix to the standard variable name. For example, if the current focus or selection was directed a table control named `ATable` on the cell that was occupying row five and column one then the following would be variable expansions:

<code>\${cell:app:varname}</code>	<code>varname.5.1</code>
<code>\${cell:group:varname}</code>	<code>ALayer.AGroup.varname.5.1</code>

<code>\${cell:control:varname}</code>	<code>ALayer.AGroup.ATable.varname.5.1</code>
---------------------------------------	---

Storyboard Model Internal Variables

Not all variables have to be defined by the user and associated with render extension and action properties. Storyboard defines a set of variables for the model element properties such as position, size, and visibility. Each class of model elements contains a slightly different set of values, but all of the variables are prefixed with the `grd_` (Graphical Runtime Data) variable namespace to avoid any confusion with user created variables.

These variables are generally accessed using `${model_object:varname}`, for example `${control:grd_x}` indicates the x position of the current control

Layer and Layer Instance Variables

The following values can be queried and changed through normal data management channels. If these variables are being changed directly the fully qualified path, including the screen on which the layer resides (`<screen>.<layer>.<variable>`), should be used.

<code>grd_x</code>	<code>4s1</code>	The layer instance's x position relative to the screen
<code>grd_y</code>	<code>4s1</code>	The layer instance's y position relative to the screen
<code>grd_xoffset</code>	<code>4s1</code>	The x pixel offset that will be used to determine the origin of the layer instance
<code>grd_yoffset</code>	<code>4s1</code>	The y pixel offset that will be used to determine the origin of the layer instance
<code>grd_alpha</code>	<code>1u1</code>	The layer instance's transparency value. The values range from 255 (opaque) to 0 (transparent)
<code>grd_hidden</code>	<code>1u1</code>	The layer instance's visibility. A value of 0 indicates that the layer and all of its controls are visible and a value of 1 hides the layer and all of its controls
<code>grd_scroll_enabled</code>	<code>1u1</code>	The layer instance's content scrolling enablement. A value of 0 indicates that the layer can not be scrolled and a value of 1 indicates that the layer can be scrolled. This value can only be modified on layers that have had scrolling enabled in Storyboard Designer.

Note

Any change to the following values affects all layer instances.

<code>grd_width</code>	<code>4s1</code>	The layer's width
------------------------	------------------	-------------------

grd_height	4s1	The layer's height
------------	-----	--------------------

Group variables

The following values can be queried and changed through normal data management channels.

grd_x	4s1	The group's x position relative to its layer
grd_y	4s1	The group's y position relative to its layer
grd_zindex	4s1	The group's z-index position. This sets the stacking order of groups within its layer where 0 is at the back (furthest from the eye).
grd_hidden	1u1	The group's visibility. A value of 0 indicates that the control is visible and 1 that it is hidden

Control variables

The following values can be queried and changed through normal data management channels.

grd_x	4s1	The control's x position relative to its layer
grd_y	4s1	The control's y position relative to its layer
grd_width	4s1	The control's width
grd_height	4s1	The control's height
grd_zindex	4s1	The control's z-index position. This sets the stacking order of controls within its layer where 0 is at the back (furthest from the eye).
grd_hidden	1u1	The control's visibility. A value of 0 indicates that the control is visible and 1 that it is hidden
grd_active	1u1	A value of 1 states that the control is active (can receive and react to events) and 0 for an inactive control (cannot receive or react to events)
grd_opaque	1u1	Indicates if the control is opaque to events. If opaque (1), the control will block events from being handled by other controls. If the value is 0, the events flow through the control to ones behind it.
grd_findex	4s1	The control's focus index. This sets the focus on a control in a navigation sequence, where 1 sets the

		focus on the first control, 2 sets the second, etc. A value of 0 indicates that the control is not focusable. In order for a control's focus index to be changed dynamically at runtime, the focus value must be initially set to a non-zero value in Storyboard Designer.
--	--	--

Table variables

A table contains all of the control variables and also a set of table specific variables. These table specific variables can be queried but not dynamically changed. In order to change these values in a table, actions are provided: `gra.table.resize`, `gra.table.scroll`. The variables are as follows.

<code>grd_rows</code>	4s1	The number of rows in the table
<code>grd_cols</code>	4s1	The number of columns in the table
<code>grd_visible_rows</code>	4s1	The number of visible rows in the table
<code>grd_visible_cols</code>	4s1	The number of visible columns in the table
<code>grd_active_row</code>	4s1	The row index of the currently active cell
<code>grd_active_col</code>	4s1	The column index of the currently active cell
<code>grd_row</code>	4s1	The table's current top left row
<code>grd_col</code>	4s1	The table's current top left column
<code>grd_xoffset</code>	4s1	The x pixel offset that will be used to determine the origin of the 1,1 table cell
<code>grd_yoffset</code>	4s1	The y pixel offset that will be used to determine the origin of the 1,1 table cell
<code>grd_scroll_enabled</code>	1u1	The table's content scrolling enablement. A value of 0 indicates that the table can not be scrolled and a value of 1 indicates that the table can be scrolled. This value can only be modified on tables that have had scrolling enabled in Storyboard Designer.

Animation Definitions

Animations can have an enormous impact on the application's user experience and are an important part of any modern user interface. Used well, they can make a relatively dull user interface appear modern and intriguing. Used poorly, they can waste a user's time and ruin what might have been a highly useful application.

An animation in Storyboard manipulates variables exposed by the application model. These variables may be the user-defined variables or they may be the internal variables associated with model element properties. When the variables are numeric there are a variety of easing functions that can be applied to transition from one value to another. Many variable changes can be orchestrated to occur at points relative to one another and edited on a timeline within the design environment and ultimately saved into a named animation block that accompanies the exported model.

A specific animation can be created that relies on the fully qualified paths to specific variables or a generic animation can be created that relies on variables that use the context relative naming convention. In both situations, the definition of an animation is independent from its use.

In order to use or trigger an animation, the definition has to be referenced by an action, which in turn is associated with an event. The Animation Action is specifically intended for the purpose of launching user-defined animations designed in Storyboard Designer.

When the visual content of an application is dynamically created, it may not be practical to create animations within Designer in advance. In these situations it is possible to create rich animations programmatically using the Lua Script Action.

Chapter 3. Typical Development Workflow

Storyboard offers a variety of ways for designers to get started developing their user interface application. The following section walks through a common approach that is used when the design content is already available.

Start a New Project: From Photoshop or from Scratch

The first step is to create a Storyboard project within the Storyboard Designer desktop environment. When Photoshop design content is available, it can be directly imported and forms the basis of a new Storyboard application. In this scenario, the initial width and height of the screens are set to be the Photoshop canvas size. All of the layers within the Photoshop file are maintained as Storyboard layers and all of the rendered Photoshop content is imported as controls with image or text render extensions.

If Photoshop content is not available for import to bootstrap an application, a Storyboard project can be started from scratch. A new project wizard walks through the process of setting up the initial project parameters such as the target screen size. This creates a blank project with a single screen and a single layer.

Regardless of the starting point, using Photoshop import or starting with a blank application, a Storyboard project initially has a single model file named the same as the project and the file extension GDE. This GDE file contains the structure of the application, the event and action bindings, and the variable and animation definitions.

Create and Organize Screen Content

The application view within Designer shows an outline of the structure of your application. From this view you can add new screens, layers, and controls to your application and expand it to include the visual structure and organization appropriate for your product.

Storyboard uses industry standard formats for assets, such as images and fonts, as much as possible. When a project is created it is automatically populated with an images and fonts directory. To import new images or fonts to use in your project, you can simply copy files with supported formats (png, jpg, bmp, tga, ttf, otf) into these directories. The assets are available for use within your project to set as a property on a render extension or by dragging and dropping into a particular screen canvas directly.

Bind Events to Invoke Actions

As you introduce multiple screens into your application, you can switch between them using screen transition actions. There are a number of different screen transitions ranging from an immediate transition to elaborate animated screen transitions using 3D transforms to flip and shape the displays.

In addition to actions for screen transitions, there are actions for starting and controlling animations, adjusting the values of variables, sending events to other applications, and integrating with Lua to invoke scripted glue logic to further control the application. After the main structure of an application is created, you can add and customize actions to modify the user interface.

Simulate and Export Model for Engine

To enable immediate visual validation of the user interface, Storyboard Designer includes a built-in simulator. This simulator uses the same runtime engine that runs on your target but is configured to execute in the desktop environment. The simulator uses the same export process that you would use for an embedded environment but automatically incorporates a launch stage to make it fast to implement structure, bind actions, and check the behavior in a rapid development cycle.

Export to Android and iOS platforms using the an export wizard that creates a package that's ready to run on your mobile platforms. Exporting an application to a touch-enabled mobile device provides a simple, yet effective, method of experimenting with user groups on different design scenarios without having to deal with the instability (or incompleteness) of an early embedded target hardware configuration.

For deep embedded systems where a filesystem may not be available to host the assets from the application, the export process involves creating a C/C++ header file. This header file is included in your BSP or system image and compiled using the same toolchain you use to build all of the non-user interface elements of your product.

Chapter 4. Storyboard Designer Environment

Storyboard Designer is a design and development environment for creating full-screen, deployment-ready applications for embedded platforms using the Storyboard Embedded Engine.

Using Storyboard Designer designers incorporate graphic content and artwork directly into the application design process. The imported images (gif, jpeg, png, psd) are used as control surfaces that application developers can bind action behavior to, based on externally generated input events.

Graphic designers already have a set of design tools for creating and manipulating images. Storyboard Designer is not meant as a replacement for these tools, but is intended to provide a binding environment where static images can be animated into multi-screen applications by allowing graphics designers to easily import their work into an application design.

Storyboard Designer is intended for use by both graphic designers and embedded applications developers and is based on the extensible Eclipse framework (www.eclipse.org) [<http://www.eclipse.org/>].

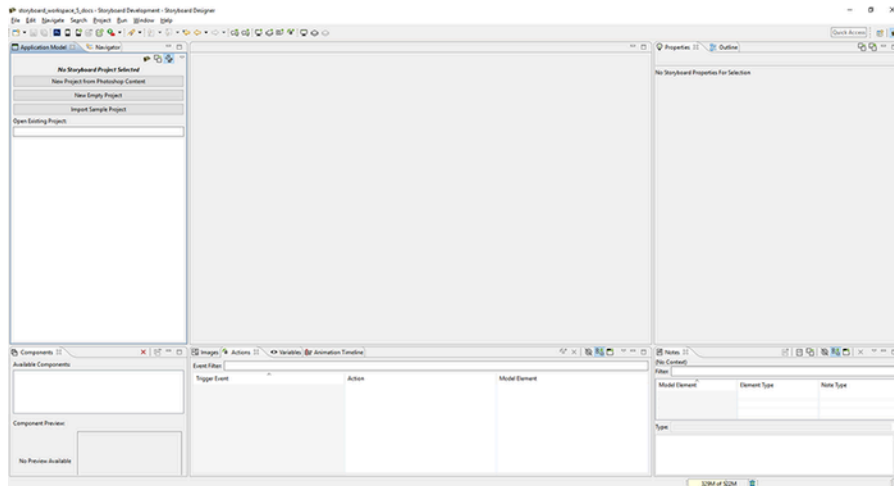
Embedded software developers typically work in C or C++ development environments. Storyboard Designer integrates into the CDT, the most common Eclipse based embedded development environment, so application user interface development can easily be done in parallel with other embedded software development.

Storyboard Designer Workbench

When Storyboard Designer starts, the user is presented with an initial empty working environment for application development as shown in the following image. Storyboard Designer presents the user with a main editing area that displays a visual, WYSIWYG, representation of the application screens as they are being developed. The editor is the primary interface for development and design of the application. The editor is opened, like any other standard editor in the Eclipse environment, by double-clicking on any Storyboard Designer file, or right-clicking and selecting File > Open.

The editor area is surrounded by dockable views that present editing information to the user as the application is being developed. Many of these views, such as the layer or application view, provide information relative to the selection in the current editor. Additional views can be added into the current display by selecting Window > Show View and then selecting the additional views.

The selection of views and their arrangement around the editor area is called a perspective. The default Storyboard Designer perspective layout can be customized by dragging, resizing, and re-docking views in an arrangement that is convenient to individual developer or designer workflows. It is always possible to reset the layout of the perspective to its default by selecting Window > Reset Perspective from the main menu.



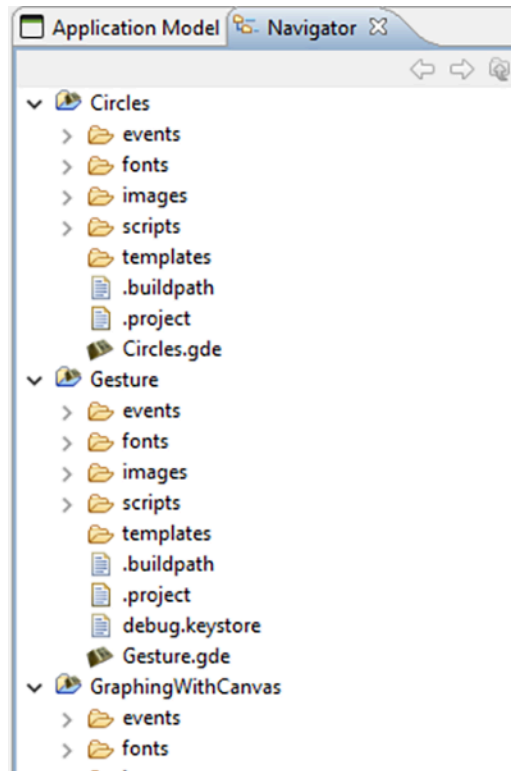
For more details on configuring the workbench refer to [Help > Help Contents > Workbench User's Guide](#)

Eclipse is an extensible framework with a rich set of plugins available from multiple software vendors. Among other integrations, team collaboration plugins for GIT, Mercurial, ClearCase, SVN and CVS are all readily available. The Eclipse marketplace (marketplace.eclipse.org) contains a comprehensive listing of available plugins and extensions.

Anatomy of a Storyboard Designer Project

Storyboard Designer manages its projects within a filesystem directory referred to as a workspace. The workspace is used to limit the scope of file resources to just those files in the host filesystem that are relevant for the application design. Storyboard Designer projects correspond to the root directories contained within the workspace directory.

When a new Storyboard project is created, using [File > New > Storyboard Application...](#), it creates an initial project structure in the workspace that contains several default directories in addition to the main Storyboard application design file.



The images, fonts, scripts, templates and events directories are automatically scanned for content and that content is integrated into the application designer tools. In order to import content from the filesystem into these directories, you can use the File > Import > General > File System option or the standard system copy and paste or drag and drop from other applications.

Each directory scans for a different type of content:

events	This directory and its subdirectories are scanned for event definition files, which are text files that have an extension of .evt. The events contained in the event definition files are automatically included in the action trigger event list. Event definitions are automatically managed by the Designer framework when new events are added or removed using the New Action Wizard.
fonts	This directory and its subdirectories are scanned for TrueType™ font files. In general, these font files have the extension of .ttf. The fonts discovered are automatically added to the list of available fonts in the font selection dialog. OpenType™ and TrueType container formats are not supported by Storyboard at this time.
images	This directory and its subdirectories are scanned for image file content. Supported image file formats include gif, jpeg, bmp and png files. Photoshop™ PSD files can be imported directly as an application or as the component images using the File > Import > Storyboard Development > Photoshop PSD File menu option.
scripts	This directory is scanned for Lua (www.lua.org) [http://www.lua.org/] scripts which have the extension of .lua. The functions that are found in these scripts are automatically added to the list of available functions presented in the Lua action argument configuration.
templates	This directory is scanned for Storyboard Designer template files. Valid templates are automatically added to the list of available templates or new actions. For more details on

creating and working with templates, refer to the document sections Working With Templates and User Defined Actions.

As changes are made in the filesystem, the workspace should refresh automatically and the changes are reflected in the Storyboard Designer user interface. An automatic refresh may be delayed due to system activity and can be forced at any time by selecting a project or file in the Navigator view and selecting Refresh from the right-click menu.

Storyboard Designer Editor

The Storyboard Designer editor is the central location for all design activities for your application. It provides a visual representation of all of the screens of the application and allows designers to edit the screen content and get immediate feedback about the look and feel of the application.

Editing Content

The default editing mode is to display and edit the entire application, showing all screens and their composited layers together.



If, instead of looking at all of the screens of an application together, you want to focus on editing and working with one particular screen, you can right-click in the editor and select Edit > Screen, which opens up a new editor window with that screen's content shown.

If you want to edit the layer contents only, independent of the screens to which they are bound, you can right-click in the editor and select Edit > Layers which opens up a new editor with all of the selected layers shown individually. If changes are made to a layer in this mode, the change will be reflected in all of the screens that reference the changed layer.

The right-hand side of the editor contains a fly-out palette toolbar that provides the basic visual design elements for the application; screens, layers, and controls. These can be selected and dropped onto the editor to start building up the application.

Additional editing functionality is available through the right-click menu while in the editor, as well as from the main menu. This is where you will find functions to manipulate control size, alignment, and z-order/front-back placement, as well as the creation of new controls, layers, and screens.

Content can be moved within its container by selecting one or more items and then using the arrow keys to move the item. By default, the movement is in 10 pixel increments when using the keyboard, but if the

SHIFT key is held down while using the arrows the content will move one pixel at a time in the desired direction.

There are a number of keyboard shortcuts for common operations these can be displayed on-screen by selecting Help > Key Assist... from the main menu.

Editor Toolbar



In addition to the editing options available right click menu, the toolbar provides functionality that is context sensitive to the editor being used. When a Designer file is being edited (and the editor area has focus) the toolbar provides short cuts to several common operations.

Storyboard Simulator	This will export the Designer file being edited to a Storyboard Engine file and simulates it using the host based Storyboard Engine configuration.
Zoom Display	<p>This controls the current zoom level of the display. There are two ways to zoom in an out with the scroll wheel.</p> <ol style="list-style-type: none">1. CTRL+MouseScroll (Windows/Linux) or COMMAND+MouseScroll (Mac) using a wheel mouse or a touchpad will zoom the editor in/out while keeping the selected model object centered on the screen.2. CTRL+SHIFT+MouseScroll (Windows/Linux) or COMMAND+SHIFT+MouseScroll (Mac) using a wheel mouse or touchpad will zoom the editor in/out on the area at the center of the editor.
Align and Resize	These toolbar actions provide a convenient alternative to manual alignment by aligning the selected controls automatically with one another. When a single control is selected the alignment is performed relative to the screen. When multiple controls are selected, the alignment is performed relative to each other.
New Model Elements	These toolbar actions provide an alternative to the palette for the quick construction of screens, layers and controls.
Control Outline	These toolbar actions control how the control content is displayed within the editing environment. By enabling the control outline, a border will be drawn around controls and layers. By enabling the wireframe mode, no control content will be drawn, but an outline of the controls is drawn. These modes can be used to optimize the application layout to avoid unnecessary damage and redraw operations.
Wireframe	<p>The wireframe mode turns off all of the render extension drawing within the controls and shows only an outline of the controls, layers, and screens similar to what is provided by the Control Outline functionality.</p> <p>This functionality is useful to minimize the amount of control overlap that occurs so as to prevent excessive redraw damage areas.</p>

Direct Editing

In addition to the standard model editing functionality, some render extensions have direct edit functionality. To enable the direct edit on a control with one of these render extensions, the user needs to do a slow double click on the control.

3D Model

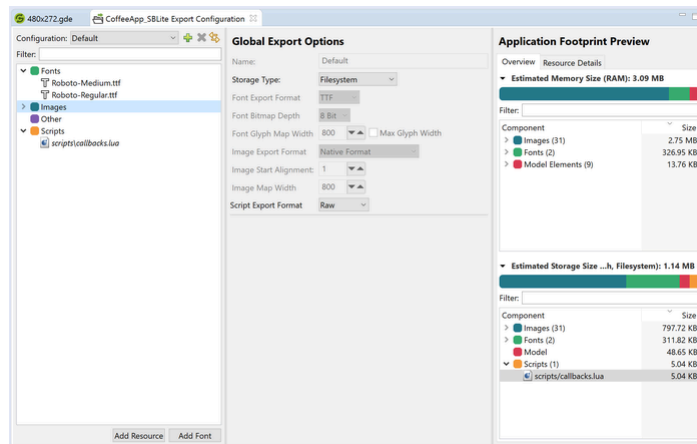
When the direct edit mode is activated, a set of xyz axes will appear in the bottom left corner of the control. At that point the user can use their mouse to move the model. Holding the shift key allows the user to rotate the model using their mouse. As well, holding either the x, y, or z key isolates that axis when either translating or rotating the model. When the user is finished editing the model, they just need to click anywhere outside of the control to end the direct edit.

2D Polygon Editing

When direct edit is activated, the user can create and edit a polygon on the screen using hotkey toggles. Hold 'shift' and click within the control bounds to create new vertices (ideally, 3 vertices are needed to have a sufficiently visible polygon). Hold 'control' and click on a particular vertex to delete it from the polygon. Hold 'alt + shift' and click near an edge to create a new vertex splitting that particular edge. Vertices can be moved by simple click and drag. Exit quick edit by moving the mouse out of the control bounds.

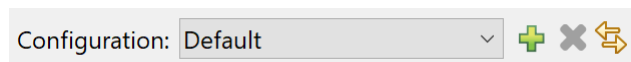
Storyboard Resource Export Configuration Editor

The Storyboard Resource Export Configuration editor provides the ability to create configurations for how resources should be exported from Designer. This allows for fine grained control over how and when resources should be exported from the design environment.



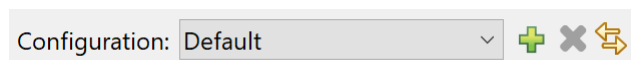
Managing Configurations

The Resource Export Configuration editor allows for the creation, deletion, and resetting of a configuration. The following toolbar controls these operations.



The combo box allows for the selection of a configuration.

The add button will invoke a dialog that allows for the creation of a new configuration. The following dialog is invoked



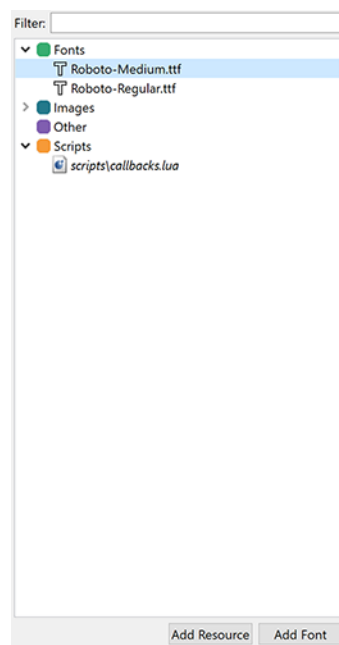
A name needs to be entered for the new configuration. The name must be unique. You can select a base configuration to base the new configuration off of. This will make a copy of the options that have been applied to the resources in the base configuration and apply them in the new configuration. When finished click on the "OK" button to create the new configuration or the "Cancel" button to abort the creation of the new configuration.

The delete button will remove the currently selected configuration. Note that the default configuration cannot be deleted.

The revert button will reset all of the options on the resources to their default values.

Resource Tree

The resource tree allows for the selection of individual resources so that options can be changed on individual resources.



The filter allows for the searching of resources based on name.

The tree is grouped by each of the different resource types, images, fonts, scripts, and other. There is a right-click menu that can be invoked that will allow for the creation, deletion, and reverting of the current configuration. It also allows for the generation of image maps if the image map option is selected. References to currently selected resource can also be searched for in the model.

The "Add Resource" button at the bottom of the tree allows for the addition of resources that are not in the images, fonts or scripts directories. The "Add Font" button allows for the addition of a font that is shipped by Designer. If the storage type for the configuration is "VFS" and the font export format is "Bitmap" the size can also be selected for the font.

Resource Export Options

Resource options are set at a global level and at an individual level. All resources will use the global setting for an option if the option is not changed at the individual level. This allows for a quick setting of options for the general case of how a resource should be exported but then also allows for more granular control over individual resources that need to be exported in a different way. For example, most images in an

application may have alpha. So in an VFS configuration, most images should probably be exported as the "DIRECT RGB8888" format. However for a big background image, that has no alpha, it may make sense to export that image as the "DIRECT RGB565" format as this will save on space.

The image shows a configuration window with two sections: "Global Export Options" and "Resource Export Options".

Global Export Options

- Name: Default
- Storage Type: Virtual Filesystem
- Font Export Format: TTF
- Font Bitmap Depth: 8 Bit
- Font Glyph Map Width: 480 (with a checkbox for Max Glyph Width)
- Image Export Format: Native Format
- Image Start Alignment: 1
- Image Map Width: 480
- Script Export Format: Raw

Resource Export Options

- ☒ Include in Export
- Script Export Format: Compiled

There are two options that are applied to the configuration. There is the name and the storage type. The name must be unique and the "Default" configuration cannot be renamed. The options for the storage type are: All of the global options are mirrors of the individual options that can be set on an individual resource. There are global options and individual resources for images, fonts and scripts.

Filesystem	This will export resources to a filesystem. This selection disables the need for a majority of the options on the resources as they will be loaded and interpreted from the filesystem based on the type of resource they are the exception to this rule is script files. They can be exported as compiled scripts to a filesystem.
Virtual Filesystem	This will export resources in a virtual filesystem format. This selection is only available for use with the C Header exporter. This enables a majority of the option as the C Header export is used with systems that have constrained resources.

When the VFS format is selected as the storage type, the "Font Export Format" option is enabled for the configuration. This is a global option as the mixing of bitmap and truetype font configurations is not currently supported. The choices for the "Font Export Format" are:

- | | |
|--------|---|
| TTF | This will export all fonts as a ttf file. This file will require the TrueType font manager to be used. Each ttf file that is needed will be loaded in to memory and the glyphs that are used will be stored in a cache in the heap. |
| Bitmap | This will export each individual glyph in a font file as an alpha map. The size of each individual pixel in the alpha map can be controlled through the "Font Bitmap Depth" option. |

When the "Font Export Format" is set to the "Bitmap" option then the "Font Bitmap Depth" option will be enabled. This option is something that can be controlled on each individual font size resource as larger sizes may take up more space. The choices for the "Font Bitmap Depth" option are:

- | | |
|--------|--|
| 1 Bit | This will export the bitmap font as an alphamap that has 1 bit per pixel in the alphamap. This bit determines if a pixel should be on or off. |
| 2 Bits | This will export the bitmap font as an alphamap that has 2 bits per pixel in the alphamap. These 2 bits represent the alpha value that should be used for that pixel |
| 4 Bits | This will export the bitmap font as an alphamap that has 4 bits per pixel in the alphamap. These 4 bits represent the alpha value that should be used for that pixel |
| 8 Bits | This will export the bitmap font as an alphamap that has 8 bits per pixel in the alphamap. These 8 bits represent the alpha value that should be used for that pixel |

If the "Font Export Format" is set to the "Bitmap" option then the "Font Glyph Map Width" option will be enabled. This option is an option that can be set on each individual font size. It controls how wide a glyph map width can be. It is useful if a system has alignment restrictions when rendering buffers. The "Max Glyph Width" checkbox will set the "Glyph Map Width" to the width of the largest glyph in the font. This creates a single strip of glyphs in the glyph map.

The "Image Export Format" option will become enabled when the "VFS" option is specified as the "Storage Type". The available image export formats are:

- | | |
|------------------------|--|
| Native Format | This will export the image in its native format, such as a PNG or JPEG. This will require the image to be decoded at runtime which will require heap. |
| Direct RGB8888 | This will export the image as an uncompressed image that has 32 bits per pixel. This image can then be drawn directly from storage without the need for heap. |
| Direct RLE RGB8888 | This will export the image as run length encoded image that has 32 bits per pixel. This image can then be drawn directly from storage without the need for heap, but will require the pixels to be expanded as it is being draw, which may incur a performance hit. It will in most cases take up less space. |
| Direct RGB565 | This will export the image as an uncompressed image that has 16 bits per pixels. This image can then be drawn directly from storage without the need for heap. The image will take up less space than the DIRECT RGB8888, but it will not support alpha. Also gradients may show banding due to a reduction in color fidelity. |
| Direct RGB565 Dithered | This will export the image as an uncompressed image that has 16 bits per pixels that has been dithered. This image can then be drawn di- |

rectly from storage without the need for heap. The image will take up less space than the DIRECT RGB8888, but it will not support alpha. Gradients should not have banding due to the dithering.

Direct Indexed

This will export the image as an uncompressed image that is palette based. This image can then be drawn directly from storage without the need for heap. The image will take up less space than the DIRECT RGB8888. Colors may look different because they will be matched to colors in the palette.

Direct RLE Indexed

This will export the image as a run length encoded image that is palette based. This image can then be drawn directly from storage without the need for heap, but will require the pixels to be expanded as it is being drawn, which may incur a performance hit. It will in most cases take up less space.

The "Image Start Alignment" option will become enabled when the "VFS" option is specified as the "Storage Type". The value that is specified for this option is the alignment that is used for the image data in any of the direct rendering image format options. The "Image Map Width" option will become enabled when the "VFS" option is specified as the "Storage Type". This is an option that can be set individually on image maps. This will set the width that the image map will use. The height of the image map will be set to whatever is required to fit all images in the image map.

The "Script Export Format" option controls how lua scripts are exported. The options are:

- | | |
|--------|--|
| Raw | This will export the script file as plain text. It will need to be compiled by the Lua engine when it is loaded. |
| Bitmap | This will precompile the Lua file when it is exported, so that the Lua engine does not need to do it at runtime. |

Additionally each resource has an "Include in Export" option. If this option is unchecked, the resource will be omitted from the export.

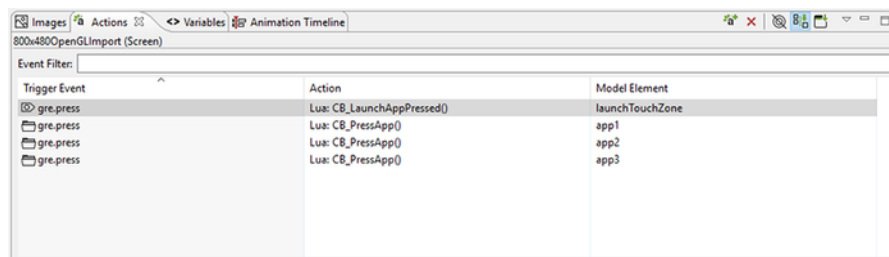
Application Footprint Preview

This preview provides feedback on how much memory or storage a resource export configuration will take up when it is used. For more information on this, see the Metrics view section in this document.

Storyboard Designer Views

Actions View

The Actions view provides a display of all of the available actions that are in context for the given selection in the editor.



Actions can be added through the right-click menu in the editor, **Add > Action...** or in the Actions view directly.

The content of the action list can be sorted by selecting the action table title. When actions are sorted by their triggering event, the order in which they appear will also correspond to the order in which they will be evaluated within the same context. If two actions are bound to a `gre.press` event on the same control, for example a Data Change and then a Screen Fade, then the first action in the list will be executed (Data Change) before the next action (Screen Fade). The order of these events can be adjusted by right-clicking the event and selecting Move Up or Move Down as required.

The content of the action list is automatically populated based on the Designer model object selected in either the editor or in the Application Model view. The content of the list can be populated in several ways:

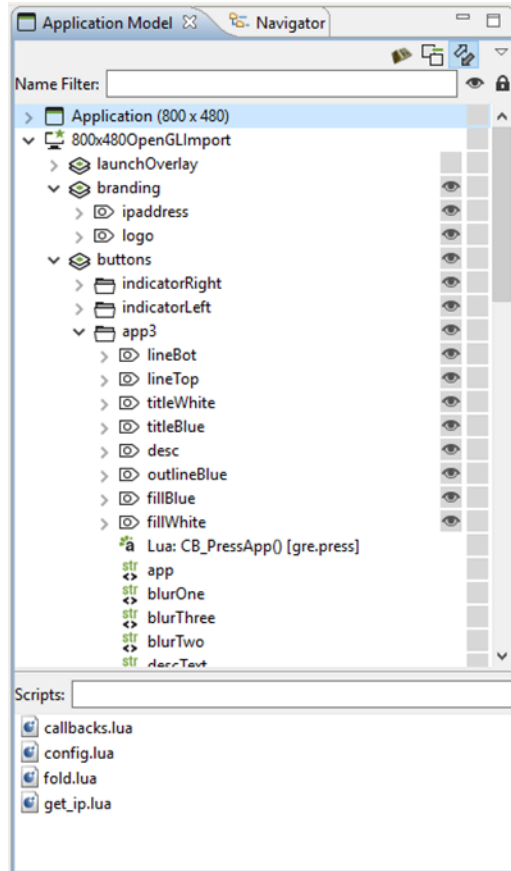
Selection Only	This shows only the actions associated directly with the selected model object.
Sub Hierarchy	This shows the actions of the selected model object and all of its child model objects.
Application Hierarchy	This shows all of the actions in the project.

In addition to controlling how the list is populated using the toolbar selections, it is also possible to use the name filter at the top of the list to match against specific event names. This is particularly useful when used in conjunction with the Application Hierarchy to search the entire project for custom events.

The triggering event, action type, and context can all be edited inline in the action table. Each action also has its own set of parameters or configuration options. These values can be changed in the lower display area of the Action view once an action selection is made. When the action types are changed, as many argument values from the original action will be migrated to the new action as long as the argument names and types match.

Application Model View

The Application Model view displays a tree representation of the model objects that make up the Storyboard application.



The tree representation aligns with the Storyboard model representation and allows editing operations to be performed on elements that may not be visible within the editor. For each of the application, screen, layer, and control objects the Application Model displays the Actions and Data Variables associated with that model object.

The visibility of the layer instances and controls can be quickly adjusted through the Application Model view by toggling the setting in the visibility column. The changes made here will be immediately reflected in the editor and will also be reflected in the Storyboard Engine runtime file as the initial setting for the layer instance or control.

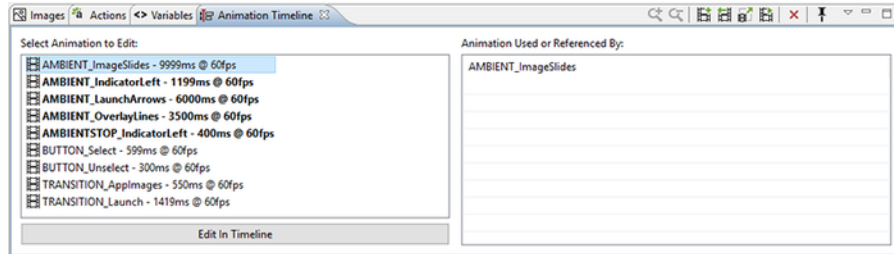
Since layers are displayed as layer instances, the tree will show layer and control content several times in the tree. If the Link with Editor toolbar option is enabled, when a model object is selected in the Application Model view the editor will automatically scroll to present the appropriate context of their selection. The same behavior can be achieved by double-clicking on a model object if the view's content is not synchronized with the editor.

The Application Model view tree also displays all of the scripts, animations, and unused layers that are currently a part of the application design.

You can copy and paste model elements, such as screens, layers, controls, and actions, from one application to another using this view.

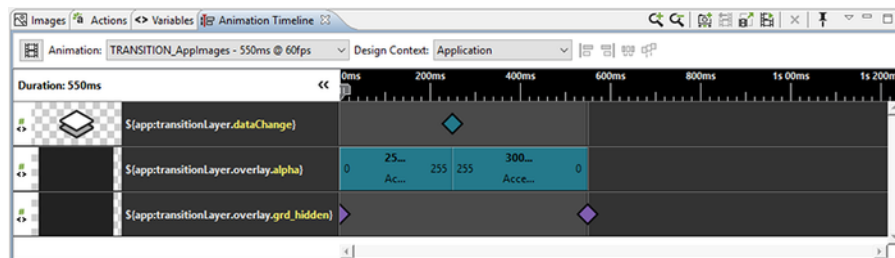
Animation Timeline View

The Animation Timeline view provides an editing environment for creating the animation blocks that are used by the animation action.



The initial Animation Timeline presentation displays a list of all of the animation blocks defined in the application. Those animations whose names are highlighted in bold represent animations that are currently referenced. The list on the right is updated, based on the selected animation, to show a list of all of the locations in the project where this animation is currently referenced.

Double-click on an animation or select the Edit in Timeline button to change the display mode of the Animation Timeline from a list of all animations to a timeline-based editor focused on the selected animation block. To create a new animation, right-click in the list and select Add Animation, or select Add Animation from the toolbar.



The timeline editing mode displays a list of the variables that are modified during the animation and a timeline view indicating the starting time offset and step duration for a particular animation step relative to all of the other animation steps running in this block.

To see the properties view to modify values, click on an animation step.

Property values can be edited / calculated inline.

To duplicate a step, double-click beside the animation step.

Clicking an animation step displays a pop up window that displays all of its information when the step is too small to do so.

To easily line up the animation steps in the timeline, select multiple animation steps and use the align left, align right, distribute, or pack tools.

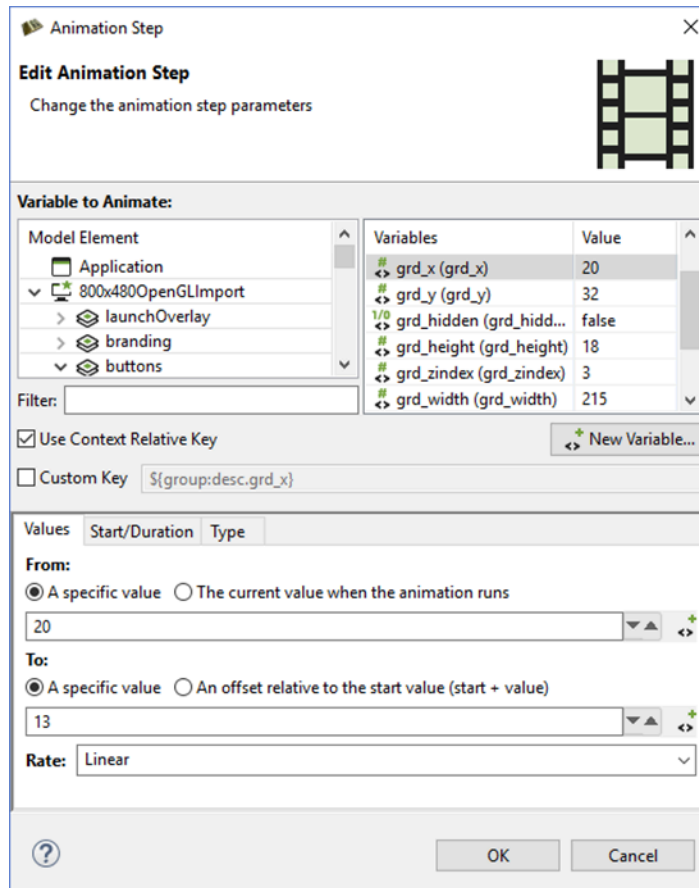
Using the 'Design Context' drop-down list, select a context that the animation will use to resolve variables. This context is also used to determine which screen is displayed during an animation preview (see below).

Right-click on an animation group to duplicate its functionality and assign it to another variable.

You can drag and drop the block within the display to adjust the starting offset or lengthen or shorten its duration by resizing the block. The start, end values, and the rate of change can be adjusted inline by double-clicking the values or by right-clicking the block. Animation steps can also be reordered by dragging and dropping them in the list. For animations with many steps, you may want to choose the compressed display option from the menu.

If you want to make a more significant change to the animation step, or to change several values all at once, select a step and click on the 'Change Variable...' button in the properties view and the Animation

Step Dialog will open. You will also get this dialog if you right-click and select Add Animation Step to add a new variable to animate in this block.



The Animation Step Dialog allows you to select the variable that you want to animate. You can fine tune the characteristics of the animation step including how you want the values to change and when.

Working with Animations

There are a number of ways to get started creating animations. You can manually create a new animation from the Animation Timeline View and add the variables to the animation one-by-one, however Storyboard provides a few shortcuts to make this process easier.

Record Animation



To create new animations, Storyboard provides a mechanism that allows you to record a series of changes to your application and have those changes automatically incorporated into a new animation. This capability, Animation Recording, can be accessed from the top menu bar.

The animation recorder tracks all changes that are made to model objects in the application. When the recording is finished it automatically creates new variables for those static values that have changed. It then gathers all of the changed variables and places them into a new animation ready to replay that visual change.

While recording, you have the ability to take snapshots. Snapshots cause subsequent steps created by changes during the recording to be offset by the duration of the previous step. This way, you can create

a sequence of consecutive animation steps rather than have them execute simultaneously. The snapshot action is located right next to the recording action, and will only be enabled if you are currently recording.

Note

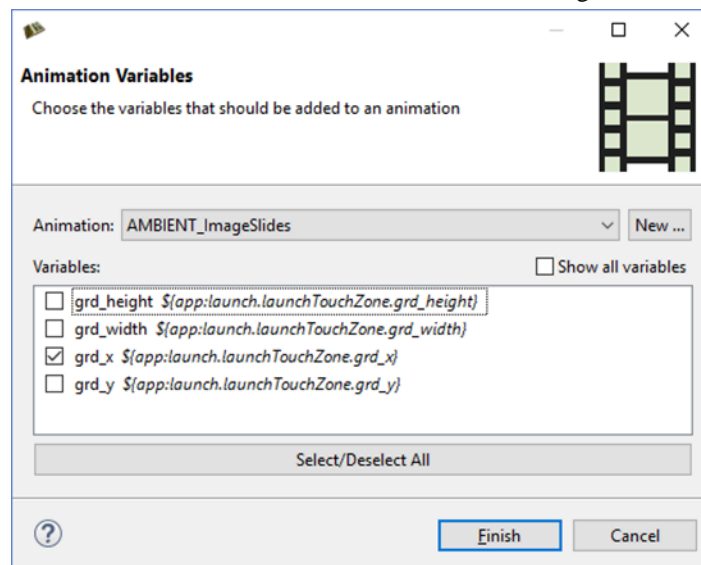
When using the animation recorder, any structural changes made to the model will be automatically reverted and lost at the end of the recording. The model recording only tracks changes to existing model properties and variables, creating variables automatically for any static elements that change.

This means that, if during the recording any controls, variables, actions, or any other model elements are created, they will not be captured in the recording and they will also not be present in the model when the recording is stopped.

Add Animation

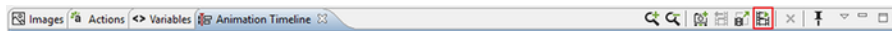


To quickly add new variables to either a new or existing animation, use the Animation Variable command that is available from the editor toolbar or from the right-click menu on many model objects.



A dialog displays all of the variables that are associated with the current selected object. You can quickly select the variables you are interested in, select the animation that you would like them to be applied to, and immediately start fine-tuning that animation.

Preview Animation



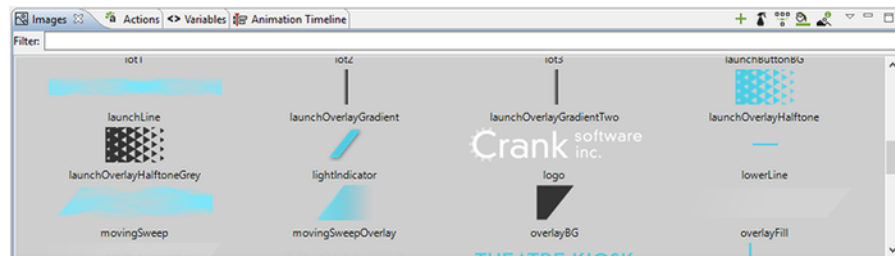
Preview the currently selected animation by clicking the animation preview action. An uneditable version of the application is displayed in a dialog and runs the animation. The bottom bar contains actions that control the preview. Play/Pause will stop and restart the animation from the current frame. While the preview is paused, use the Fast Forward/Rewind buttons to move ahead/back one frame at a time. Interact with the progress bar to jump to any frame and click on the Replay button to reset the animation to frame 0 in preparation for another run.



By default, the preview window chooses the screen to display by looking at the animated variables. However, if the user has selected a design context for the animation in the timeline, that context is used to determine the screen to display in the preview.

Images View

The Images view provides a thumbnail presentation of the images that are currently included in the application design.



The content for the Images view is automatically pulled from the image file content in the *images* directory of the Storyboard project. New content imported into this directory, upon a workspace refresh, is automatically shown in the Images view. Supported image file formats include PNG, JPG, GIF and BMP files. In order to import Photoshop PSD file content, the images must be imported using the File > Import > Photoshop PSD File wizard.

You can drag and drop images from the Images view directly into an application design. By default, a new control is created that matches the image's size and the image source is a static value pointing at the drag and dropped image.

Within the Images view it is also possible to quickly switch an image that is used in one context to another using the Swap Image With... right-click menu. For example, if a number of controls are using an image as a button background and a new image is available to update the look, you can import the new image, select the existing image, and choose Swap Image With... to quickly change all instances within the application to the new look.

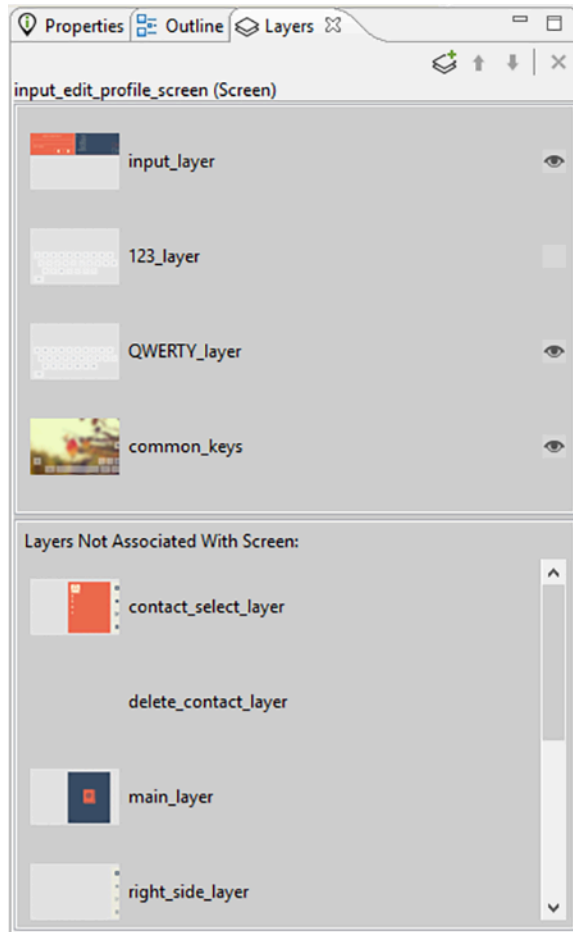
All files contained within the `images` folder are processed as images, regardless of their file extension. If an unrecognized image file is encountered, by default it will not be displayed. This behavior can be adjusted by deselecting `Show Only Images` in the view's drop-down menu. This menu also provides the ability to group images by directory which is a convenient way to classify images that are used in different parts of your user interface.

As an application evolves, it will often accumulate a number of unreferenced or duplicate resources. The Image view offers a few utilities to help manage these images.

- The Resource Clean Up toolbar action prunes and deletes unreferenced images from the application design.
- The Consolidate Images toolbar action identifies duplicate images, references all consolidated into a single resource, and removes duplicate images from the workspace.
- The Trim Images toolbar action works on selected images to remove all of the extraneous transparency that surrounds non-transparent content. This can significantly impact the rendering performance on systems without hardware graphic rendering capabilities.
- The Split Images toolbar action works similarly to the Trim Images option but transforms a single image with significant areas that are completely transparent into multiple images with that transparency trimmed away.

Layers View

The Layers view provides a thumbnail display of the layers used by the currently selected screen and also displays a thumbnail of all of the other layers that are available for use but are not bound to the selected screen.



Layers are categorized as either being included on the current screen or not. Layers can be added or removed from the current screen by dragging and dropping them either into or out of the current screen area, or by using the delete key.

Layers are listed according to their front (top) to back (bottom) order z-order presentation. Manipulate the order by dragging and dropping the layers within the view or using the toolbar buttons. In addition, the Layer view provides the ability to change the visibility within the application. If the visibility or z-order are changed, the change is immediately reflected in the editor and will also be reflected in the Storyboard Engine runtime deployment.

Using the toolbar controls in the Layers view it is possible to create new layers and also to open up the Layer editor mode to work with layers independent of the screens with which they are associated.

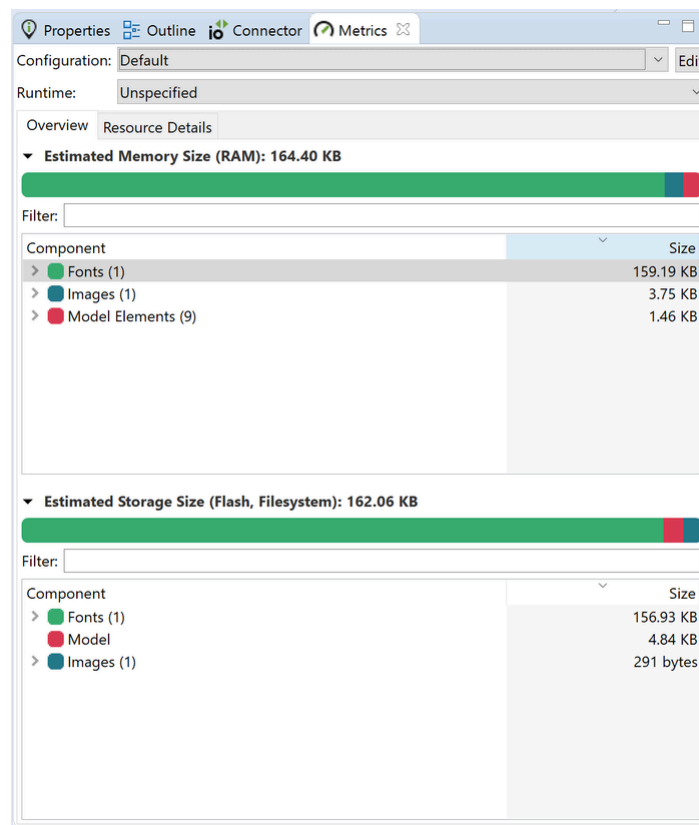
Metrics View

The Metrics view shows how much memory and storage is going to be used by the resources that are included in a Storyboard application. Resources account for a large majority of the memory and storage cost associated with a Storyboard application. If a Storyboard application needs to reduce its memory and storage costs, then its resource use is an area that should be investigated to see if there are sections of the application that can be redefined to reduce the use of resources and therefore save on the memory and storage costs.

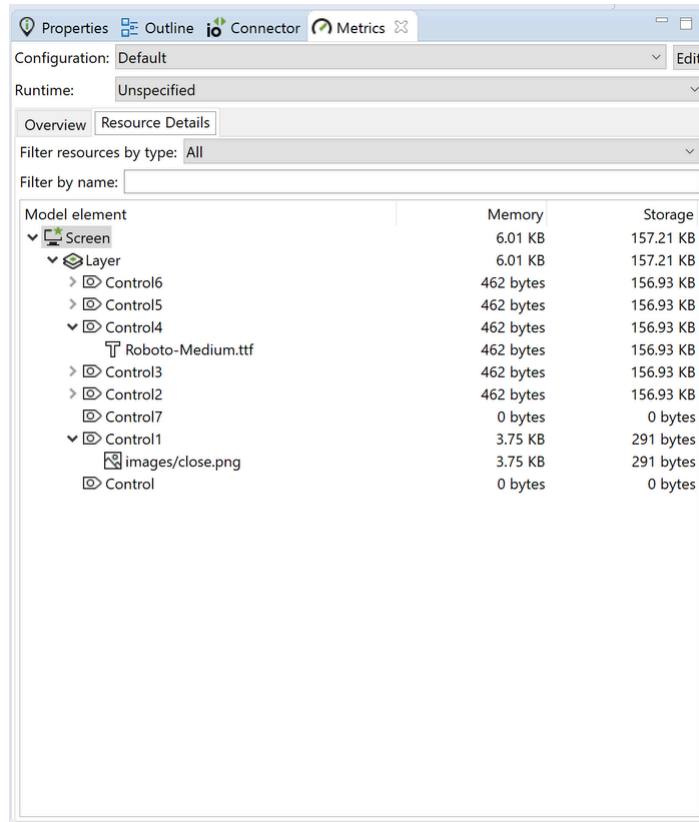
The memory and storage cost of a resource is affected by how it is stored. In the case of a storing resources on a conventional filesystem, images are decompressed into memory and font glyphs are managed by the

freetype font engine in a cache. In the case where a virtual filesystem is being used, there are options to control how images and fonts are stored and accessed and this will have an impact on how much memory and storage they will consume. The configuration combo box allows you to select the resource export configuration to show the memory and storage costs for. The edit button next to the configuration combo box will open the Resource Export Configuration editor, which allows for creating new configurations that can have different settings applied to individual resources. For more information on the Resource Export Configuration dialog, see the section in this document.

The model cost for a Storyboard application is tied to the runtime that is being used. Different runtimes will have different model costs due to the differences in internal storage types used in the runtimes. Selecting different runtimes will show how much the model will cost while the engine is running with the Storyboard application.



The outline section in the Metrics view shows the breakdown of the memory and the storage taken up by each type of resource, which are images, fonts, scripts, and other. These sections can be expanded to show each individual resource. A resource can be selected and a menu that is invoked by right-clicking on the resource will allow for searching of the model to show which model elements are referencing that resource. There is also the option to view what an image looks like to provide context, as well as the option to edit the resource in the currently selected resource export configuration. The filter text field can be used to search for a resource by name.

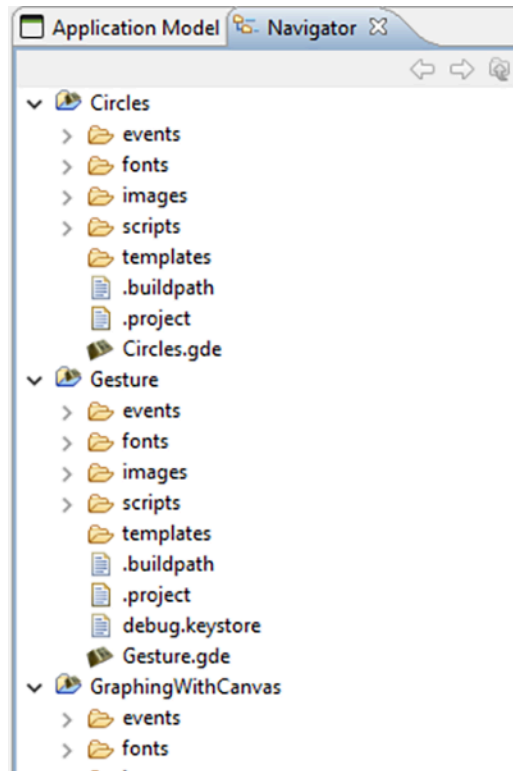


The resource details tab shows a break down of resources being used by each element in the Storyboard application. The element can be selected and show in the application view by selecting that option from the right click menu. This tab is useful for seeing which elements in the Storyboard application are taking up the most memory and storage. The filter by name text field can be used to search for an element by name.

It should be noted that this is just the memory cost for resources. Things like circles, polygons, complex screen transitions, and code that is written in Lua will have a memory cost associated with it. For example, complex screen transitions will need to composite the current screen and the next screen in to offscreen memory buffers. This would cost an addition width * height * bit depth * 2 in memory usage to perform that complex screen transition.

Navigator View

The Navigator view is a standard filesystem style explorer limited to showing only content available in the Storyboard workspace.

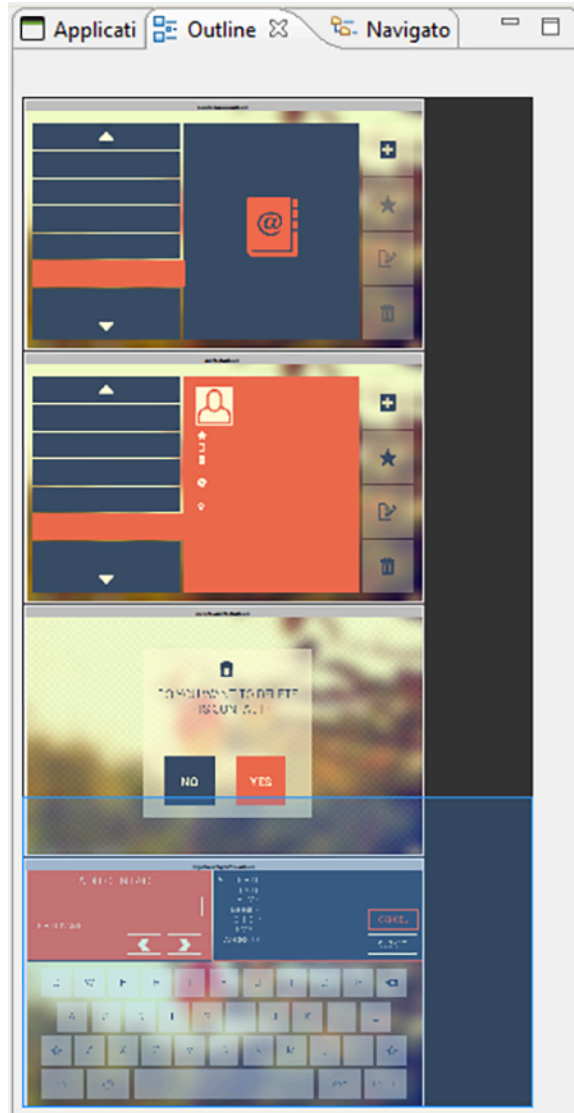


The Navigator view only displays content that has been imported into the workspace and starts with the top level Storyboard Project directories that have been created. Since the workspace allows multiple Storyboard projects to be shown, it is possible to work on multiple projects concurrently all within the same workspace with multiple editors open targeting different Storyboard projects. Content and resources can be copied and pasted among the different editors.

The Navigator view provides a variety of filters to hide/show different file types as well as the ability to group projects together as 'working sets' and then to only display the content from those working sets. For more details on configuring the Navigator and filtering workspace content refer to the Eclipse help [Help > Help Contents > Workbench User Guide](#).

Outline View

The Outline view provides an overview of the entire editor. The outline content changes to reflect an outline of the editor that currently has focus. Within the Designer environment there are two outlines of interest, the Storyboard editor outline and the Lua editor outline.



The Storyboard editor outline displays a scaled visual presentation of the entire contents of the editor. If the editor is in Application mode then all of the screens of the application will be shown. If it is in Layer mode then all of the layers will be shown, and similarly for the other editor modes. By moving the highlighted area within the Outline view it is possible to change the viewport of the current editor.

The Lua script editor outline displays a listing of all of the identified functions in the file. Double-clicking the function name provides quick navigation to those functions.

Problems View

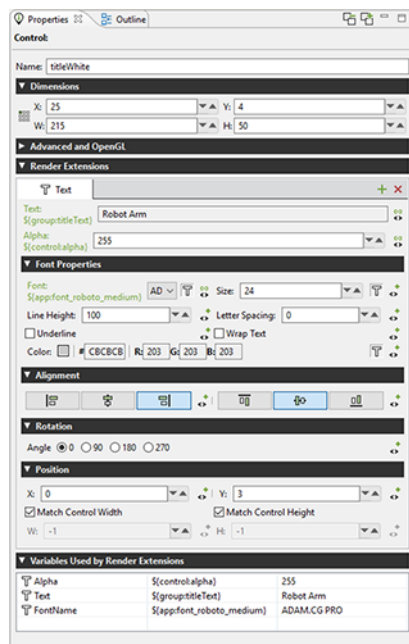
The Problems view shows a list of all problem markers that are created within the workspace. Storyboard provides a project analysis mode that runs in the background to examine Storyboard models and report on design concerns. To enable or disable this analysis mode you can change the Storyboard workspace preference setting in Windows > Preferences > Storyboard > Enable Background Storyboard Project Analyzers.

When the Storyboard Project Analyzers are enabled, the model will be scanned and analyzed and discovered issues will be reported into the Problems View as the workspace resources change. Issues that are scanned and reported include:

- Mismatched image color depths. If a 32bit color image is used with a 16bit color display there is a possibility the image will be distorted on the target.
- Fill hides content. If a fill masks on top of other content it can be an ineffective use of processing resources on target devices.
- Missing render extension content. For example, a control with an image render extension that is not defined or does not exist. This may cause the embedded target to perform additional computations.
- Scaled or rotated static image content. Since static images do not change at runtime, the work required to scale or rotate an image could be replaced by a fixed cost of rotating or scaling the image during the design. This will reduce the amount of processing required at runtime.

Properties View

The Properties view displays information about the current selection in the editor and also provides the ability to change and adjust the properties of that selection. The Properties view is the primary editing location for fine tuning the visual presentation and adjusting data bindings of the application.



For each model element, application, screen, layer, and control, there is a different property interface that provides access to those items that are most relevant to the selected context. The following is a list of some of the property pages.

Application	The application properties, active when the editor background is selected, displays the application name, size, and color properties.
Screen	The screen properties, active when a screen is selected, displays the screen name and indicates if the screen is the start screen for the application.
Layer	Layers and Layer Instances share a common set of property pages. A Layer Instance is simply a Layer that is associated with a particular screen. Changes made to a layer's size or its controls will propagate to all layer instances. Position, opacity, and 3D rotation properties are properties associated with a layer instance, not the layer itself.

Group	Group properties contain the name of the group as well as information about the group's origin. In this property panel you will also find the functionality for automatically reconfiguring a group's origin based on its control content. This is useful when you are taking a group and then converting it for use in a more generic Storyboard Designer template.
Control	Controls contain the most sophisticated property pages. In addition to the name, size, and position information, the property page also contains the configuration parameters for the render extensions associated with that control. The render extensions are listed in the Z order (front to back) that they will be rendered within the control. This can be adjusted by dragging and dropping the render extensions within the list entry.
Render Extensions	Render extension property pages show the argument details of the selected render extension. This is the same information as is shown within the Control's property panel, but without all of the additional details associated with the control.
Actions	The property pages for actions show the parameters that are available for editing and the presentation changes based on action type. The content that is shown here is the same as the information presented in the Action View. In the case where multiple actions are selected, the content can be changed across the entire selection.

When multiple elements are selected, the Properties View attempts to show the most suitable content possible. If all of the selected elements are the same, then the properties view will display the common properties and any changes that are made will apply across all of the selected elements.

In certain cases, such as when multiple controls with different render extensions are selected, it may not be possible to provide a completely synchronized display. In these cases the display will show a common set of properties and hide the properties that are not common among the selected elements.

Component View

The Component view provides users with a list of Storyboard Designer components that are available for use in this project. The list of available components is generated automatically from the contents of the project's `templates` directory.

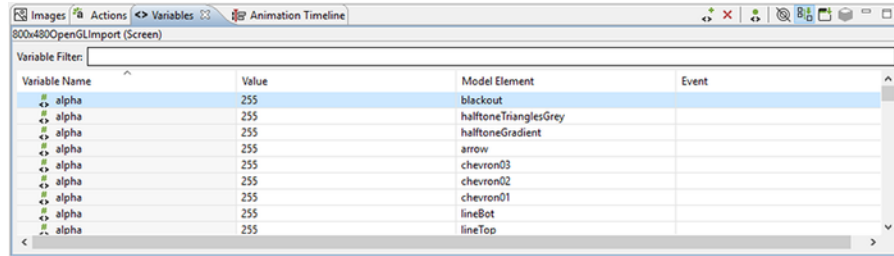
When a component is selected, its description, along with a graphical preview of the component, are shown if available. To create a new model object based on that component, select one of the components in the list and drag and drop it into your application.

Not all components create new Storyboard model objects. Some components simply enhance the functionality of an existing object. In this case, when a model object is selected you can right-click the object and select Component Apply to access a list of available components that can transform the selected object.

For more information about creating components, see the document chapter [Chapter 23, Reusable Graphical Components](#).

Variables View

The Variables view is similar to the Actions view in that it displays all of the data variables that are in the context of the current selection.



Once a variable is defined and associated with a particular model object context (application, screen, layer or control) then the variable can be referenced as a parameter for actions and render extensions.

There are two different types of data variables that can be defined. A normal variable contains a name, a type (i.e. number, string), and the value matching the type that should be used when the variable is referenced. To facilitate working with repetitive data within a table control, a special type of variable called a table cell variable can be created. This variable contains all of the same properties as a normal variable, but is extended to contain additional row and column information that can be used to specialize a particular value at a given table row and column.

The content of the variable list can be sorted by selecting the appropriate variable table title and the variable values can be edited inline in the variable list by double-clicking on the appropriate field that you wish to change.

Similar to the Actions view, the variables that are shown in the Variables view are automatically populated based on the Designer model object selected in either the editor or in the Application Model view. The content of the list can be populated in several ways:

Selection Only	This shows only the variables associated directly with the selected model object.
Sub Hierarchy	This shows the variables of the selected model object and all of its child model objects.
Application Hierarchy	This shows all of the variables in the application, regardless of what the current selection may be.

In addition to controlling how the list is populated using the toolbar selections, it is also possible to use the name filter at the top of the list to match against specific variable names. This is particularly useful when used in conjunction with the Application Hierarchy to search the entire project for a variable.

Variable Creation

New Variable

Variable Creation
Select a variable name and the type of variable to be defined.

Model Object: 800x480OpenGLImport

Name: txt

Type: Text String

Value: Hello World!

☐ Create a table cell variable

Cell row,column entries:
Specify the table cells to which variables will be associated.
Indicate a range of cells using '-' and separate entries using ','

Table Cells: 1,1

Examples:
2,3 Selects a single cell at row 2, column 3
2,3; 3,3 Selects two cells: 2,3 3,3
1,1 - 2,2 Selects four cells: 1,1 1,2 2,1 2,2

NOTE ON DEFAULT VALUES:
All unassigned table cell variables in Designer will use a default value for display purposes.
This behaviour differs from the Storyboard Engine and Simulator where a variable is required to exist in order to be rendered.

New variables are frequently created at the point where they are required, for example, within the property display for a render extension or the properties for an action argument. When variables are created in this context, then their types will automatically be determined from the context of use. However, variables can also be created directly from within the Variables view, in which case the user can select the type of the variable. It's important to match the type of the variable to its intended use. For example, text variables can't be used as adjustments for numeric values and vice versa.

The variable creation opens the New Variable wizard. From within this dialog you can select the name of the variable, its data type, and the value to associate with the variable. From this dialog you can also create table cell variables that span a particular row/column range

By default, the variable will be created and associated with the current application, screen, layer, or control that was selected when the New Variable wizard is launched. However this association can be changed on the second (optional) page of the New Variable wizard where the variable can be explicitly assigned to a different model object.

Note

The type of a variable is important for the Actions or Render Extensions that may use them. If a variable is mistyped, such as a string variable that is created but referenced in a location expecting an RGB color value, then the results are undefined. In general, you should create variables from the Actions or Render Extensions that will be using them to ensure the proper typing occurs.

Generating Events on Variable Change

It is possible to associate a user-defined event to be generated when a variable's value is changed. These events are designed to facilitate the synchronization of user interface elements that may not be directly associated with the variables whose data is changing. A typical scenario would be to monitor the position

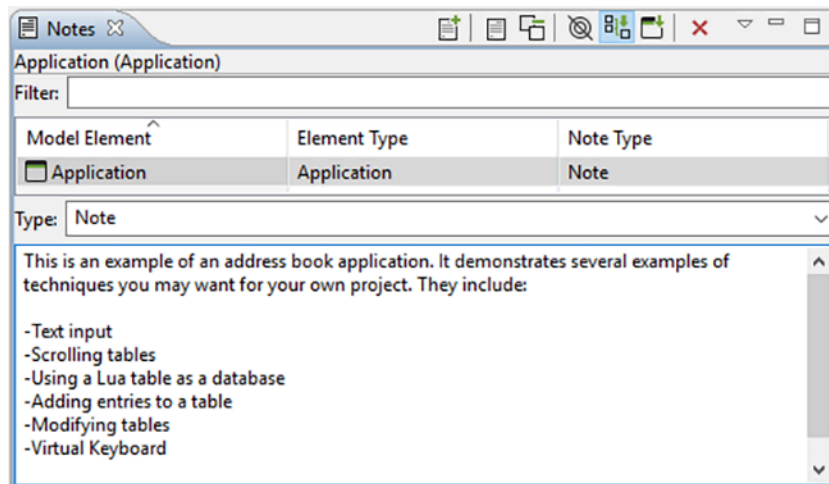
or location of a control and send a notification when it changes in order to maintain a corresponding relationship in another control.

In order to specify the event to be generated, simply enter the event name into the Event column of the desired variable or select the variable from the list and right-click and select Bind Event which will open the event definition list allowing you to pick from existing events or create a new one.

The variable change events are designed to be used to synchronize the user interface display with an updated variable value and are not meant to be used as counters for each changed value of a variable. For each variable change an event is added to the event queue only if there is not already an event with the same name in the queue waiting to be processed. Until that event is serviced, no additional events will be queued for that variable, or any variable generating the same event name.

Notes View

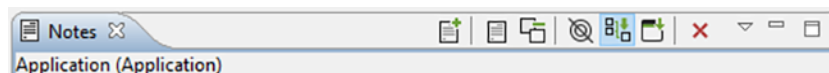
The Notes view displays notes attached to model objects within the current project.



Notes contain text to help organize a project or keep track of useful information specific to a particular model object. Notes also have a type associated with them which can be selected using the combo box in the editing area. To create a custom type, simply type in the combo box instead of using the drop down. When a note with a custom type is added, that type will be added to the combo box so that it can be quickly selected from the drop down when making other notes.

Notes attached to the current project are displayed in the table. It can display in three modes:

1. In context - only displays the note associated with the select model object.
2. Sub hierarchy - displays the note associated with the selected model object and any notes associated with model objects with are children on the selected object.
3. Full application - displays all the notes in the application.

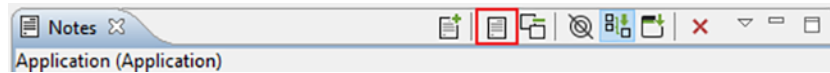


You can search for notes by typing in the filter text box. Sort the contents of the table by clicking on the header of one of the columns (e.g., clicking Model Element will sort the notes alphabetically by the name of the model object the note is attached to).

Add notes to any model object through the right-click menu by selecting New > Note, or by selecting the model object and clicking either the New... icon in the Notes view or by clicking in the text box where it indicates Click to add a new note.... To delete a note, select it in the table and then click the red X, or right-click it in the table and select delete.

To display the model object that a note is associated with, right-click a note in the table and select Go To > Screen.

All the notes in the current project can be quickly summarized in a list by clicking Toggle Full View (the yellow page icon).

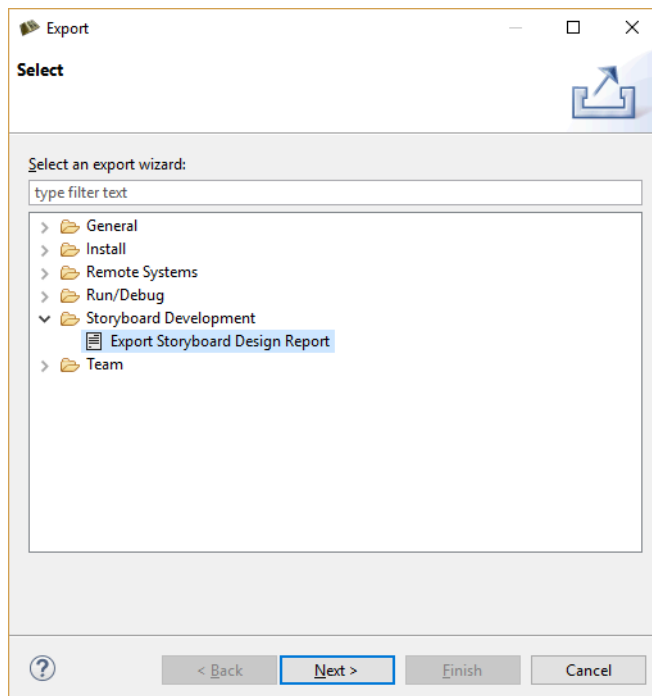


Storyboard Designer Utilities

Storyboard Designer offers many features to help you efficiently develop your embedded user interface. Some of these features improve performance by reducing potential runtime inefficiencies, while other features speed the development of the application by providing greater insight into an existing design or allowing multiple designers to work in parallel on the application's user interface.

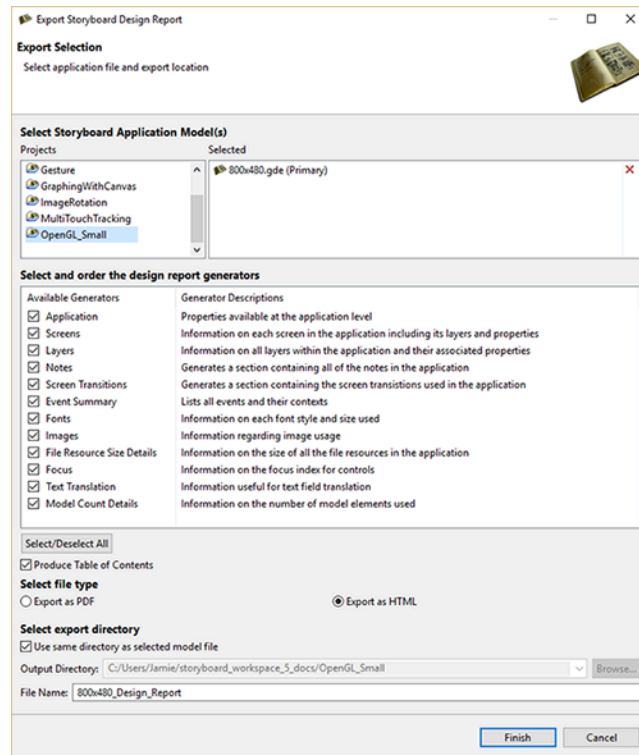
Design Notes

The Design Notes export wizard is used to generate an HTML or PDF report of the Storyboard project.



Access the Design Note export from File > Export > Storyboard Design Report or right-click a Storyboard GDE file and select Storyboard Export > Export Storyboard Design Report. This opens a dialog for you to select the file system location for the design report and the format, HTML or PDF, that the design report should be exported to.

The Design Notes report can be customized to contain a variety of information about the design model. Some of the available content options include screen transition information, content thumbnails, resource usage, and event and variable bindings.



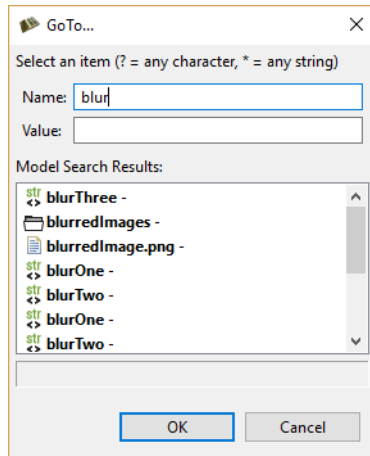
It is also possible to generate headless Design Note exports from the Storyboard Design files that can be used from a command line or scripting environment.

```
PATH_TO_INSTALL/Storyboard_Designer/storyboard/Storyboard -application
com.crank.gdt.designreport.core.designreportcore model=PATH_TO_GDE_MODEL format=[pdf|html]
output=PATH_TO_OUTPUT_FILE
```

The `model` is the full path to the Storyboard Designer model file and `output` parameter specifies the file system path where the report and associated resources will be created. The `format` parameter can be set to either `html` or `pdf` to indicate HTML or PDF outputs respectively.

GoTo Dialog

The GoTo dialog provides a quick way to locate and navigate to items in the application design. The GoTo dialog is activated when the CTRL+1 (Windows/Linux) or COMMAND+1 (Mac) keys are pressed while working in the main editor.



The Name and Value entries allow you to narrow the search criteria for the object you wish to find. The text entered here filters the results in the list to only display search results matching what you have typed:

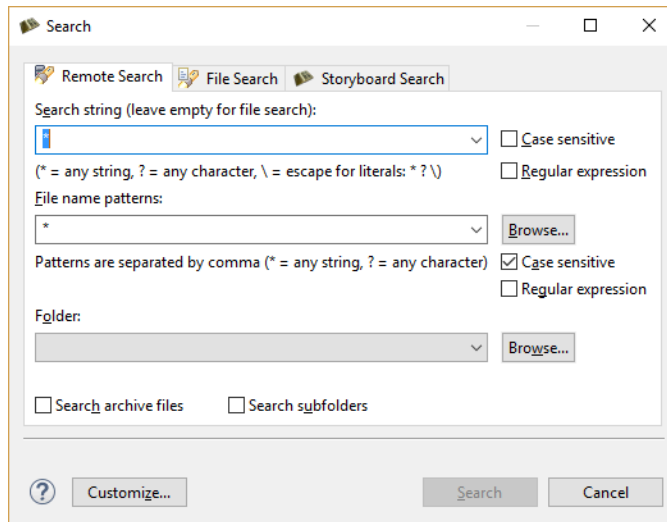
Name This selection limits the search to the primary name of the objects being searched. The name field also searches the name properties of render extension and action properties.

Value This selection limits the search to the value field of actions or render extensions being searched.

Storyboard Search Dialog

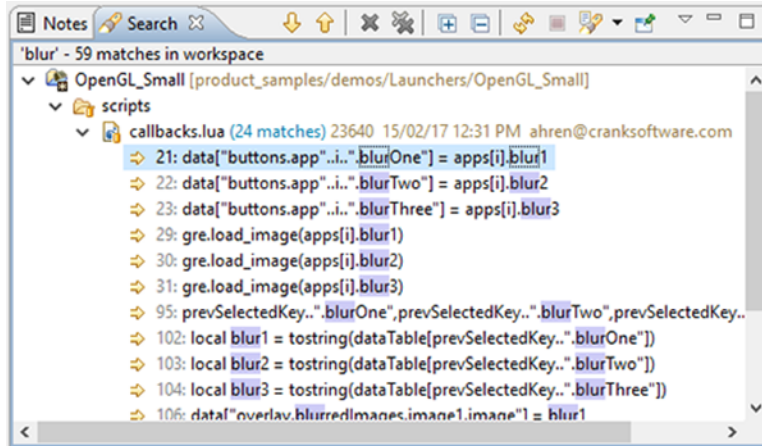
Storyboard Search provides an extension of the Go To functionality. Rather than providing the ability to see and navigate to a single selection, the results of a search are displayed in a tree format.

To search, invoke the Search dialog via the Search menu item or the CTRL+H key command.



Similar to the GoTo dialog, the Name and Value search criteria allow model object names to be searched or in the case of actions and render extension arguments and variables also their values.

The search results are displayed hierarchically in the Search view:



Where applicable, double-click the results to select and bring forward the appropriate model element in the main Storyboard editor.

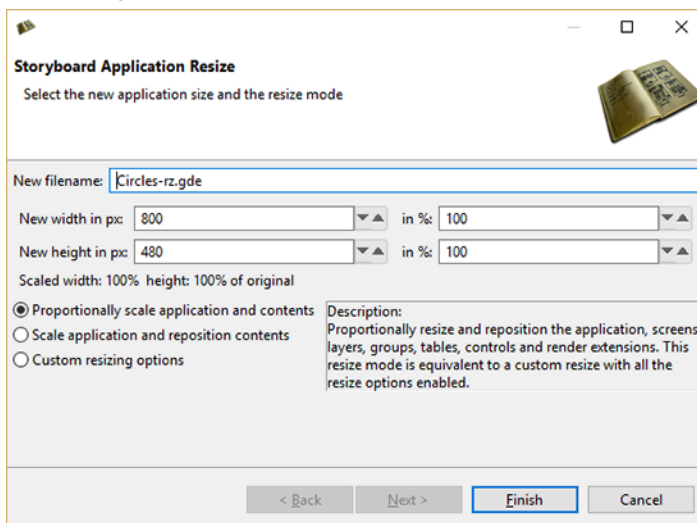
Resize Storyboard Application

Use the Resize Storyboard Application dialog to create a new Storyboard Designer model file, with different screen width and height settings, based on an existing model file.

The application resize action dialog allows a Storyboard Designer model file to resize its screen dimensions. Since all of the layers, controls, and render extensions in a model are placed at specific locations within the model, the resize operation provides several parameters to allow the layers, controls, and render extensions to be either re-positioned or re-scaled as appropriate.

The output of the resize action is a new model file located in the same file system location as the source model file. Creating this new scaled model file next to the original model file allows the designer to validate and inspect the scaled result before deciding to replace the original file.

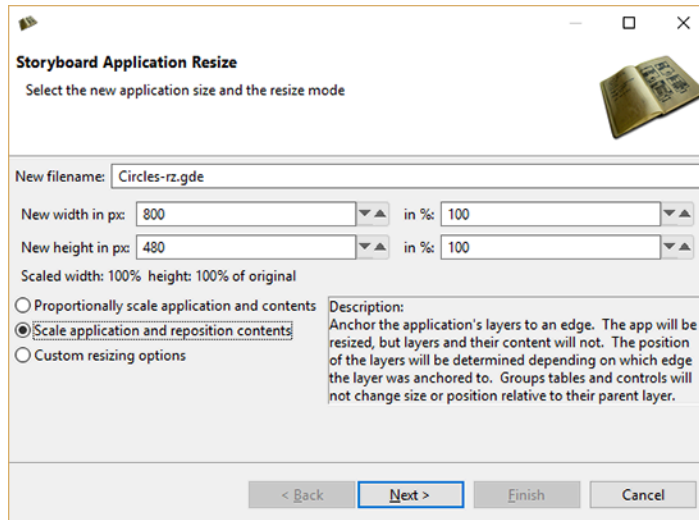
Proportionally Scale Application and Contents



This is the 'one size fits all' resize operation which will be suitable for most resize operations.

Proportionally resize and reposition the application, screens, layers, groups, tables, controls and render extensions. This resize mode is equivalent to a custom resize with the resize options enabled (Minus the Use Layer Content Position Anchors)

Scale Application and Reposition Contents



Storyboard Application Resize
Select the new application size and the resize mode

New filename:

New width in px: in %:

New height in px: in %:

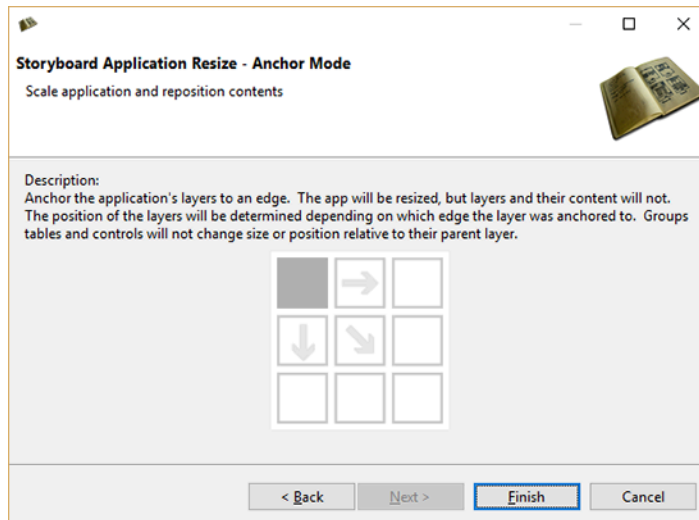
Scaled width: 100% height: 100% of original

☐ Proportionally scale application and contents
☒ **Scale application and reposition contents**
☐ Custom resizing options

Description:
Anchor the application's layers to an edge. The app will be resized, but layers and their content will not. The position of the layers will be determined depending on which edge the layer was anchored to. Groups tables and controls will not change size or position relative to their parent layer.

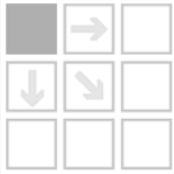
< Back **Next >** Finish Cancel

This resize operation is useful for making your application larger without changing the size of the content. This can be useful when extra space is required for adding additional buttons for testing purposes.



Storyboard Application Resize - Anchor Mode
Scale application and reposition contents

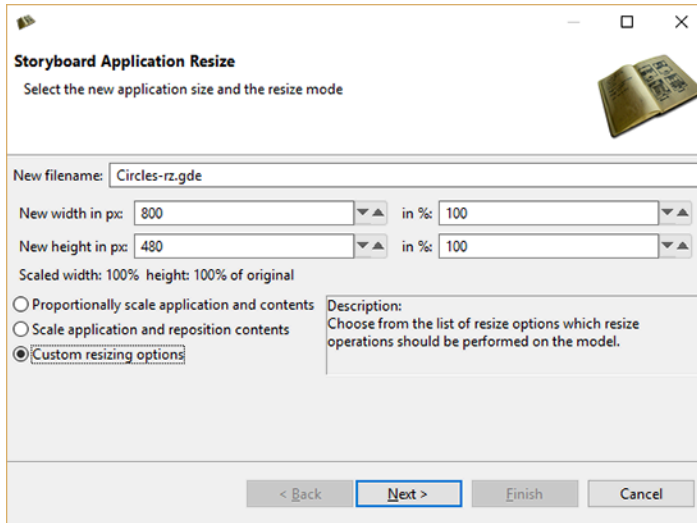
Description:
Anchor the application's layers to an edge. The app will be resized, but layers and their content will not. The position of the layers will be determined depending on which edge the layer was anchored to. Groups tables and controls will not change size or position relative to their parent layer.



< Back Next > **Finish** Cancel

Anchor the application's layers to an edge. The app will be resized but layers and their content will not. The position of the layers will be determined depending on which edge the layer was anchored to. Groups, tables and controls will not change size or position relative to their parent layer.

Custom resizing options



Storyboard Application Resize
Select the new application size and the resize mode

New filename:

New width in px: in %:

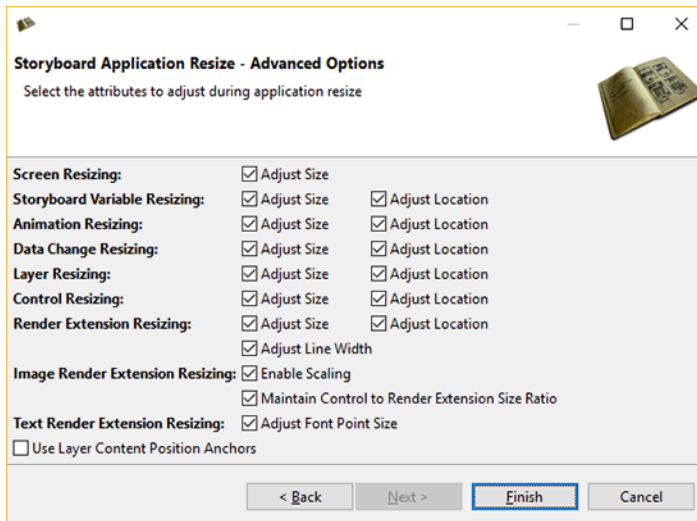
New height in px: in %:

Scaled width: 100% height: 100% of original

☐ Proportionally scale application and contents
☐ Scale application and reposition contents
☒ Custom resizing options

Description:
Choose from the list of resize options which resize operations should be performed on the model.

< Back **Next >** Finish Cancel



Storyboard Application Resize - Advanced Options
Select the attributes to adjust during application resize

Screen Resizing: ☒ Adjust Size

Storyboard Variable Resizing: ☒ Adjust Size ☒ Adjust Location

Animation Resizing: ☒ Adjust Size ☒ Adjust Location

Data Change Resizing: ☒ Adjust Size ☒ Adjust Location

Layer Resizing: ☒ Adjust Size ☒ Adjust Location

Control Resizing: ☒ Adjust Size ☒ Adjust Location

Render Extension Resizing: ☒ Adjust Size ☒ Adjust Location

☒ Adjust Line Width

Image Render Extension Resizing: ☒ Enable Scaling

☒ Maintain Control to Render Extension Size Ratio

Text Render Extension Resizing: ☒ Adjust Font Point Size

☐ Use Layer Content Position Anchors

< Back **Next >** **Finish** Cancel

Screen Resizing

Adjust Size: Scales the size of all the screens in the application.

Storyboard Variable Resizing

Adjust Location: Scales the location of all the user defined variables in the application that have been bound to a render extension location or center of rotation points.

Adjust Size: Scales the size of all the user defined variables in the application that have been bound to a render extension size.

Animation Resizing

Adjust Location: Scales the start and end values for `grd_x`, `grd_y`.

Adjust Size: Scales the start and end values for `grd_width` and `grd_height`.

Data Change Resizing

Adjust Location: Scales values for `grd_x`, `grd_y`.

Adjust Size: Scales values for `grd_width` and `grd_height`.

Layer Resizing

Adjust Location: Scales the location for all of the layer instances in the application.

Adjust Size: Scales the size of all the layers in the application.

Control Resizing

Adjust Location: Scales the location for all of the controls in the application.

Adjust Size: Scales the size of all the controls in the application.

Render Extension Resizing

Adjust Location: Scales the location for all of the render extensions in the application.

Adjust Size: Scales the size of all the render extension in the application.

Adjust Line Width: Changes the width of lines that appear in any render extension with the style set to "Line". The line width will be scaled by the value of the smallest scale factor, be it that of height or that of width.

Image Render Extension Resizing

Enable Scaling: Enables the scale flag on all image render extensions.

Maintain Control to Render Extension Size Ratio: "Enable Scaling" by itself stretches images to fit the entire control and may not have the desired effect. This option preserves the ratio from Control to Render Extension so the images will not look disproportionate or out of place within the control.

Text Render Extension Resizing

Adjust Font Point Size: Scales the text font size by the value of the smallest scale factor, be it that of height or that of width.

Others

Use Layer Content Position Anchors: Resizes the layers, but not the controls, groups, and render extensions. This option overrides the options to resize those elements. The 'next' button of the Resize Wizard will be enabled which transitions to anchor page, a selection for the anchor point for the controls/groups within their respective layers.

Resizing GDE from the Command Line

It also possible to perform a headless resize of a Storyboard Application (*.gde) file from a command line or scripting environment.

All of the Storyboard executables will be located in the `PATH_TO_INSTALL/Storyboard_Designer/storyboard` directories, though on each desktop platform they are named slightly differently

Windows:

```
"....\Crank Storyboard.exe" -application com.crank.gdt.ui.gderesize
    -output=<PathToResizedGDEFile>
    -w=800
    -h=480
    <PathToGDEFile>
```

Mac:

```
....../Storyboard.app/Contents/MacOS/Storyboard -application  
com.crank.gdt.ui.gderesize  
-output=<PathToResizedGDEFFile>  
-w=800  
-h=480  
<PathToGDEFFile>
```

Linux:

```
....../Storyboard -application com.crank.gdt.ui.gderesize  
-output=<PathToResizedGDEFFile>  
-w=800  
-h=480  
<PathToGDEFFile>
```

Note: Storyboard requires a display, so to run in a true headless environment a Virtual frame buffer needs to be setup.

```
Xvfb :1 -ac -screen 0 1024x768x8 export DISPLAY=:1  
....../Storyboard -application com.crank.gdt.ui.gderesize  
-output=<PathToResizedGDEFFile>  
-w=800  
-h=480  
<PathToGDEFFile>
```

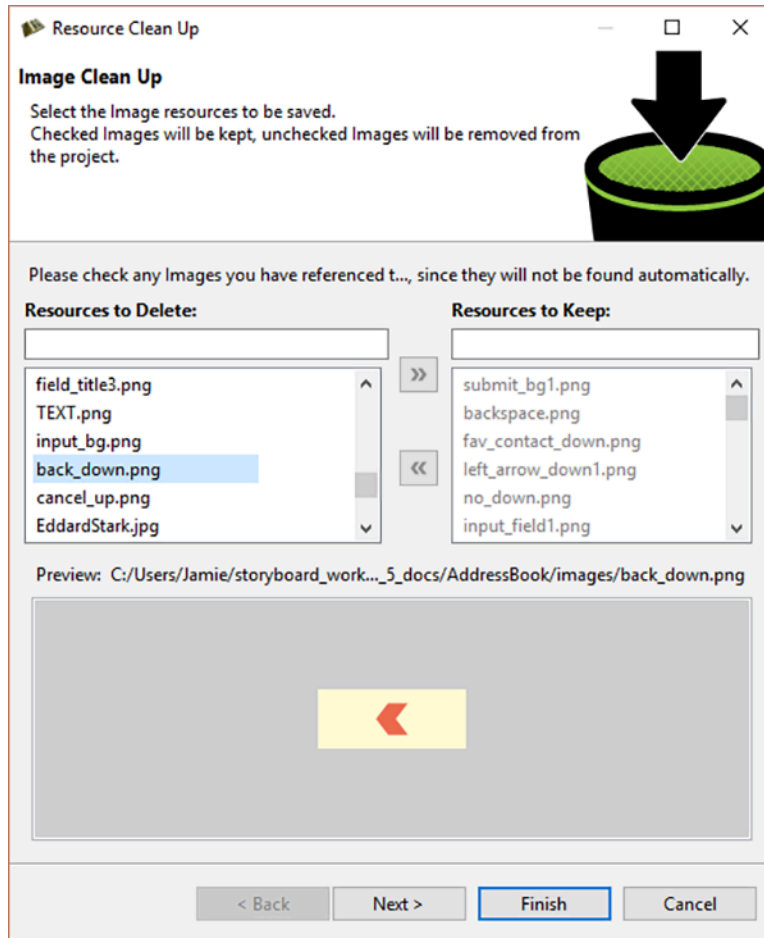
Where output is the full path to the output Storyboard Designer model file. Where w and h are the new width and height for the resized application.

By default the Storyboard Application will be resized using the "Proportionally Scale Application and Contents" resize mode. In order to perform a "Proportionally Scale Application and Contents" resize or "Custom resizing options" resize, The wizard option must be used.

```
....../Storyboard -application com.crank.gdt.ui.gderesize  
-wizard  
-output=<PathToResizedGDEFFile>  
<PathToGDEFFile>
```

Resource Clean Up Wizard

Run the Resource Clean Up wizard from within the Images view or by selecting the model file in the Navigator view and selecting **Resource Clean Up...** from the menu.



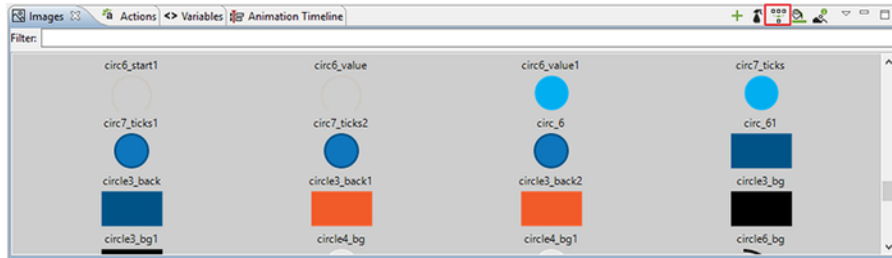
The Resource Clean Up wizard is used to remove resources that are present in the workspace but are no longer referenced by the project. In the wizard, the list on the left side contains the resources (fonts, images) and is used to select the resources that should be maintained/kept in the project. All resources that are not selected are permanently removed from the project and file system. The preview on the right side contains a preview of the selected resource and the file system location of that resource.

This tool can only detect those resources that are directly referenced by the project. It is important that resources that are referenced from external programs or scripts be manually checked so as to prevent their removal from the project.

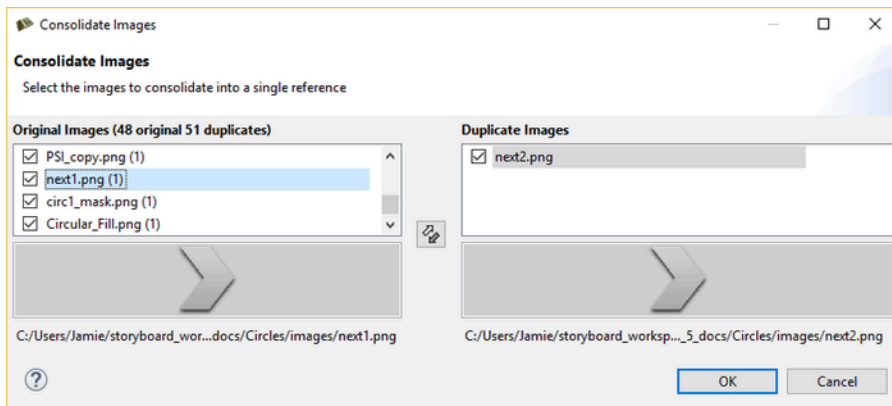
Once all of the resources to remove have been identified, selecting OK permanently removes those files from the file system.

Consolidate Images Wizard

Launch the Consolidate Images Wizard from the toolbar in the Images view.



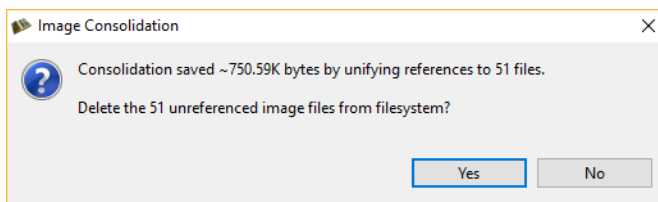
Launching the Consolidate Images wizard analyzes all of the images in the workspace. These images are compared byte by byte to determine if their content is identical and can safely be consolidated together into a single reference.



Once the analysis is complete, a dialog presents the results showing the duplicate images that have been detected and provides a visual comparison of the source and reference images to ensure that they are different.

By default, all duplicates are consolidated into a single reference. To remove a reference from being consolidated, deselect the item.

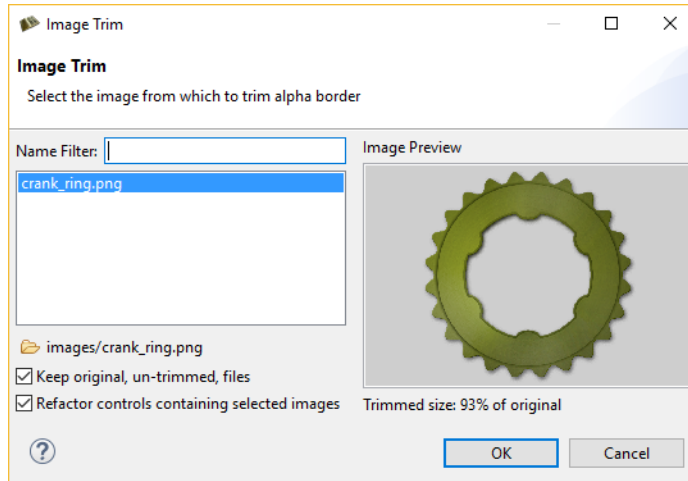
Selecting OK searches the model and consolidates variable and argument references to unify their references.



After all of the references have been combined there are likely to be a number of images that are no longer used. These can be immediately deleted from the workspace at this point.

Trim Images Wizard

The Trim Images Wizard can be launched from the toolbar in the Images view.



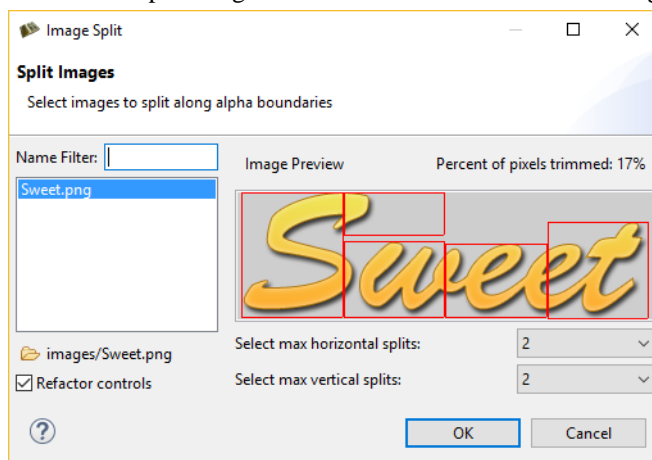
The Image Trim wizard analyzes all (no selection made) or just the selected images from the Image view. It scans the image looking for transparent pixels on the borders of the image, in order to shrink the image to it's smallest possible size.

The Wizard displays the selected images (or all images, if no selection) that have any pixels to be trimmed. The image preview shows the candidate image, and provides information on how much of the image will get trimmed. By default, the wizard makes copies of the original, untrimmed files and keeps them in the file system. If you would like to delete the original untrimmed files, simply deselect the appropriate checkbox.

The wizard also refactors all controls containing this image, while maintaining the proper control size and the position of the image within each control.

Split Images Wizard

Launch the Split Images Wizard from the toolbar in the Images view.



The Image Split wizard analyzes all (no selection made) or just the selected images from the Image view. It scans the image once, looking for transparent pixels and calculates how the image should be split to get rid of the maximum amount of transparent pixels.

The preview displays a red outline of how the image will be split, and will update according to how many horizontal and vertical splits you specify. The percent of pixels eliminated is displayed in the top right corner.

When you select OK, the image split operation proceeds on those images you have selected. After splitting the images, all references in your app to the previous image are refactored such that the controls containing those images are replaced with the new split images positioned appropriately.

To leave the controls as they are and not perform the refactoring, simply deselect the Refactor controls check box at the bottom of the wizard.

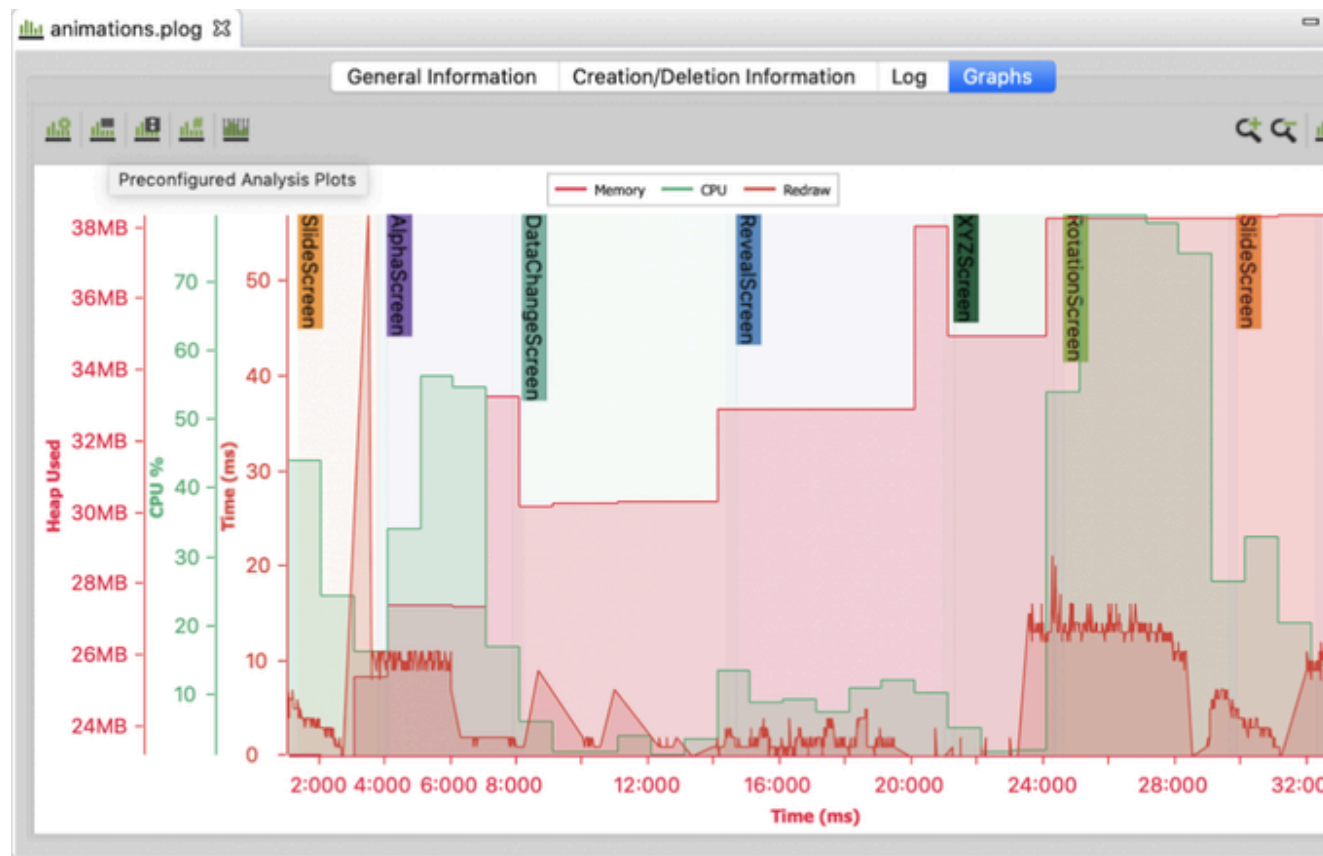
Merge Control Images

Launch Merge Control Images from the Application Model view or by selecting Manage > Merge Control Images from the right-click menu on the desired control.

This can be used when you have one or more controls with static image content that will not change during runtime. It can be more efficient to flatten those render extensions into a single static image. The controls must be in the same layer for the wizard to work correctly.

Storyboard Performance Log Viewer

Performance logging is a general record tracing and logging mechanism that is built into the Storyboard engine. This functionality allows the capture of fine grained execution information about operations being performed within the engine. This can be enabled by enabling performance logging in the Storyboard Simulator Configurations under logger (.plog file).



Each performance record is made up of three pieces of identifying information along with the timestamp of when the record was generated and in certain cases the duration of the operation represented by the

record. The three pieces of identifying information represent the payload of the performance records and provide context to the overall record stream:

- A *class* identifies the classification of the record. An example of a class might be an event, an action or a control.
- An *operation* identifies the record within the class in terms of context. For example events can be queued or they can be dispatched, both are examples of operations within the event class. Similarly the control class identifies the allocation, destruction and rendering operations that take place with controls.
- The *value* or *name* of the record is the final piece of data. This field has context based on the class and operation of the record. The field may include information such as the name of the event in the case of an event dispatch or queue operation or the name of the resource being loaded for a resource load record.

The class, operation and value fields can be represented hierarchically as not all operations apply to all classes.

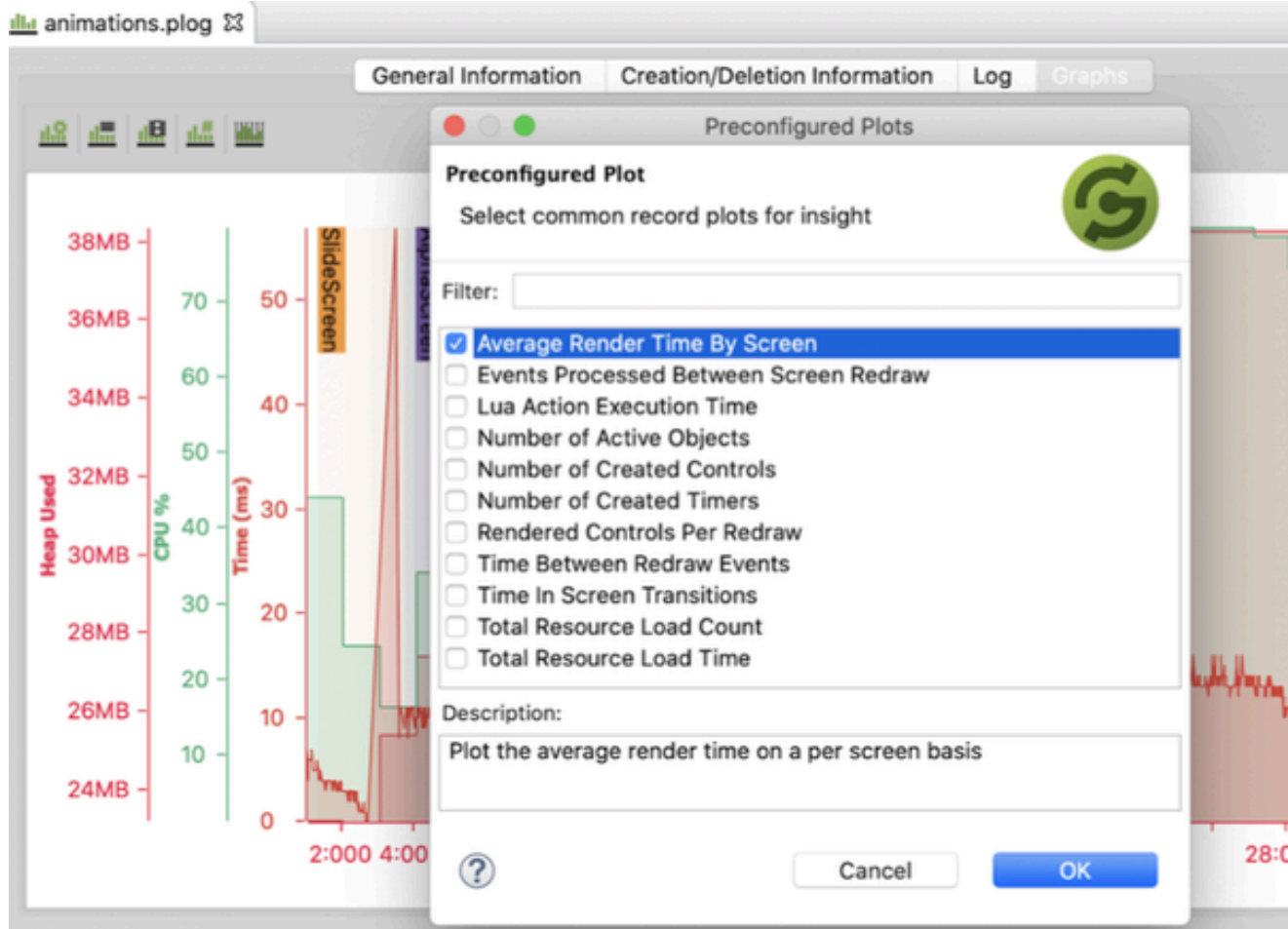
The new plotting functionality allows these records to be filtered, analyzed and visualized within the context of a Storyboard application execution timeline.

A number of predefined record plots are included that represent common analysis scenarios and a custom plotting mechanism gives you the ability to perform record filtering that is contextual to the problem you may be trying to investigate with your own application.

asdasd

Predefined Record Plots

A number of predefined plots have already been defined that align with common filtering and analysis that are both common and useful. For example displaying the screen rendering time, resource load times or CPU usage information.

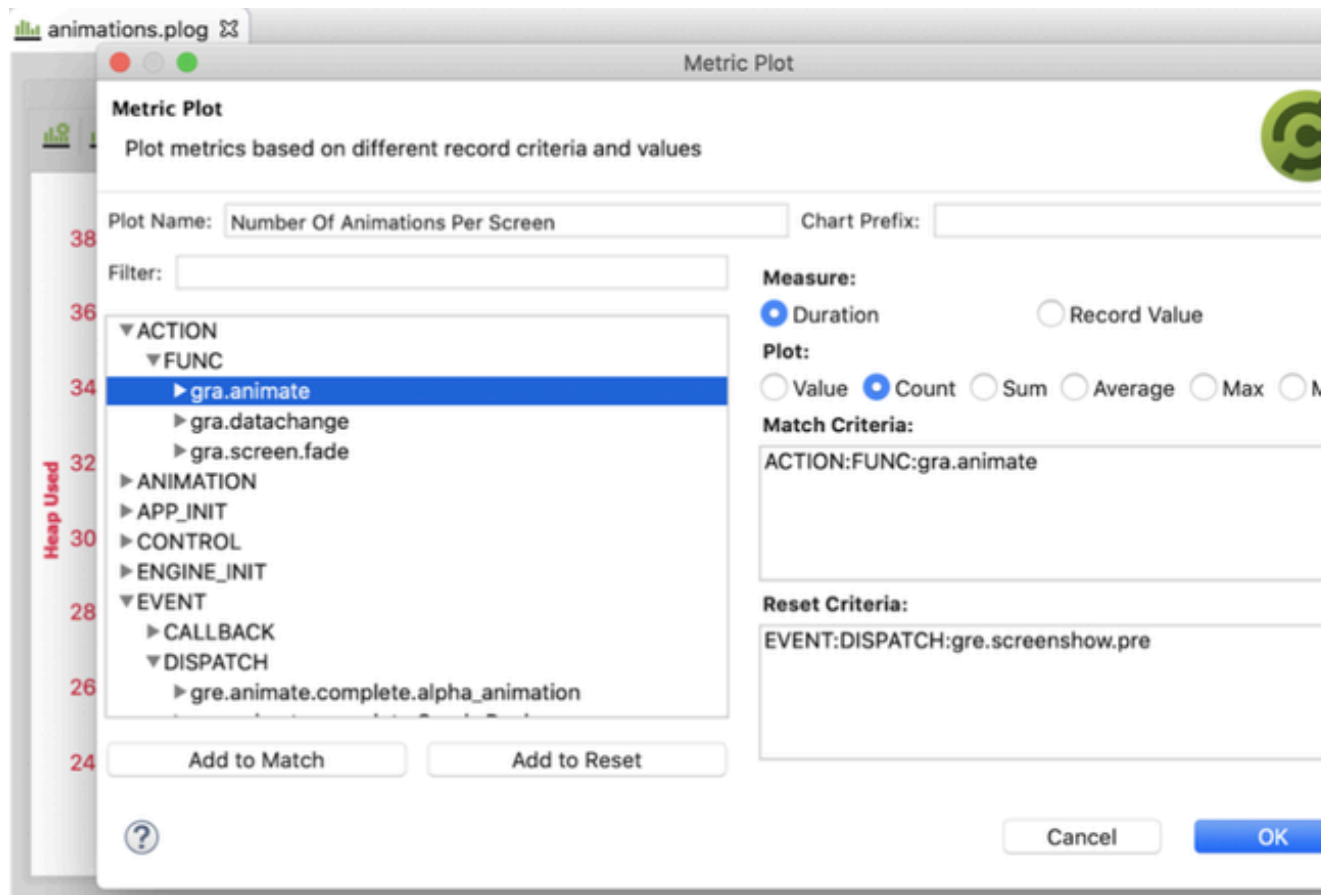


Not all of these plots may be applicable for the performance log that you are analyzing, for example CPU and memory metrics are only available if the trace information contains those records. On some engine platforms all metric information is not always available.

Each of the pre-configured plots contains a description of the metric that is being generated and plotted and further description of how you might use these plots is described below.

Custom Record Plots

The custom metric plot dialog provides an advanced functionality to create custom filters and match criteria for various records in the event log. A number of statistical operations are available to be applied to the filtered records in order to generate plottable data.

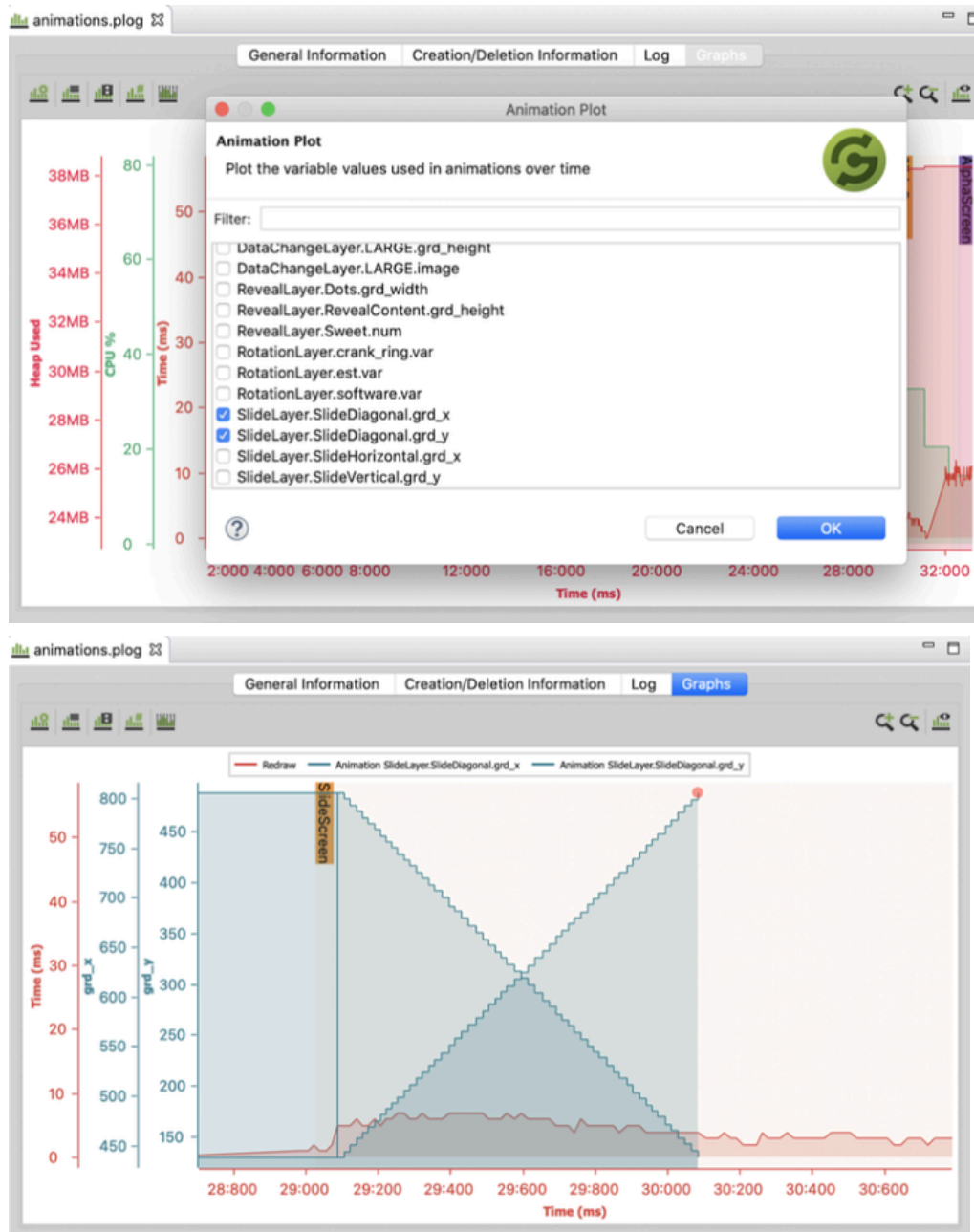


A basic operation can involve counting the number of instances of a record, for example running animations, or determining the current, total or average time for a particular action such as Lua.

You can select a record match criteria that will cause records to be included in the statistical operation set as well as a reset criteria that will cause the statistical operation to reset and start over. A common reset operation would be to reset at every screen transition to gain insight on per-screen costs. Altogether this provides a very powerful mechanism that you can create visual plots of data for different contexts.

Animation Plots

The performance log also contains records that describe the state of animations that are executed during the course of the log capture. More specifically the variables used in those animations and the values that are being changed over the course of the animations execution. A separate pre-configured animation plot dialog shows these variables and allows their values to be plotted on the application timeline against other plots.



Analyzing Applications Using Record Plots

Here are some quick analysis scenarios that you can take leverage with your performance logging plots.

Visual Accountability

How an application renders can be greatly influenced by a large number of external factors. On all systems it is possible for the rendering time to be measured and plotted. This is included as the *Redraw* pre-configured plot and is one of the default record plots that will be shown.

In general redraw time will correlate to the complexity and composition of the visual scene and the goal is for it to remain below certain thresholds such as 60fps (~16ms/redraw), 30fps (~33ms/redraw) or 15fps (~66ms/redraw).

If you observe spikes in the rendering time, it may be due to an external influence interrupting the operation of the rendering engine or it may be due to an internal application factor such as a new resource being loaded on demand or additional complexity of a scene by way of more content being rendered. Overlaying on top of the redraw plot you can use the pre-defined plot for resource load count to identify when resources are being loaded and if they correlate to a rendering delay or use the rendered control count pre-defined plot to determine if the number of rendered elements on the screen have increased. With this information you can take corrective action in your application such as pre-loading resources or merging control content together at design time rather than engine render time.

CPU Accountability

Another first thing you might want to do is to take a look at the CPU profile of your application where it is available. For this you need to ensure that you are logging with metrics enabled. The pre-defined CPU usage plot will indicate what your average CPU usage over the application and how that usage correlates to various screens of operation.

Similar to the redraw time, loading of resources can cause a spike in CPU usage, but so can the processing of different kinds of events and as such the action processing time for those events may be something you want to correlate. For this the pre-defined action duration plot will be helpful to allow you to understand what actions or what stream of events are correlated to your CPU bursts and take corrective action if required.

Memory Accountability

Similarly to CPU accountability, the memory usage information is captured when metrics are enabled (on some platforms). This memory profile offers insight to dynamic behaviour of an application and the goal with this plot is for an initial growth to be observed and then a relatively flat plot for repetitive parts of the user interface as the application enters a steady state.

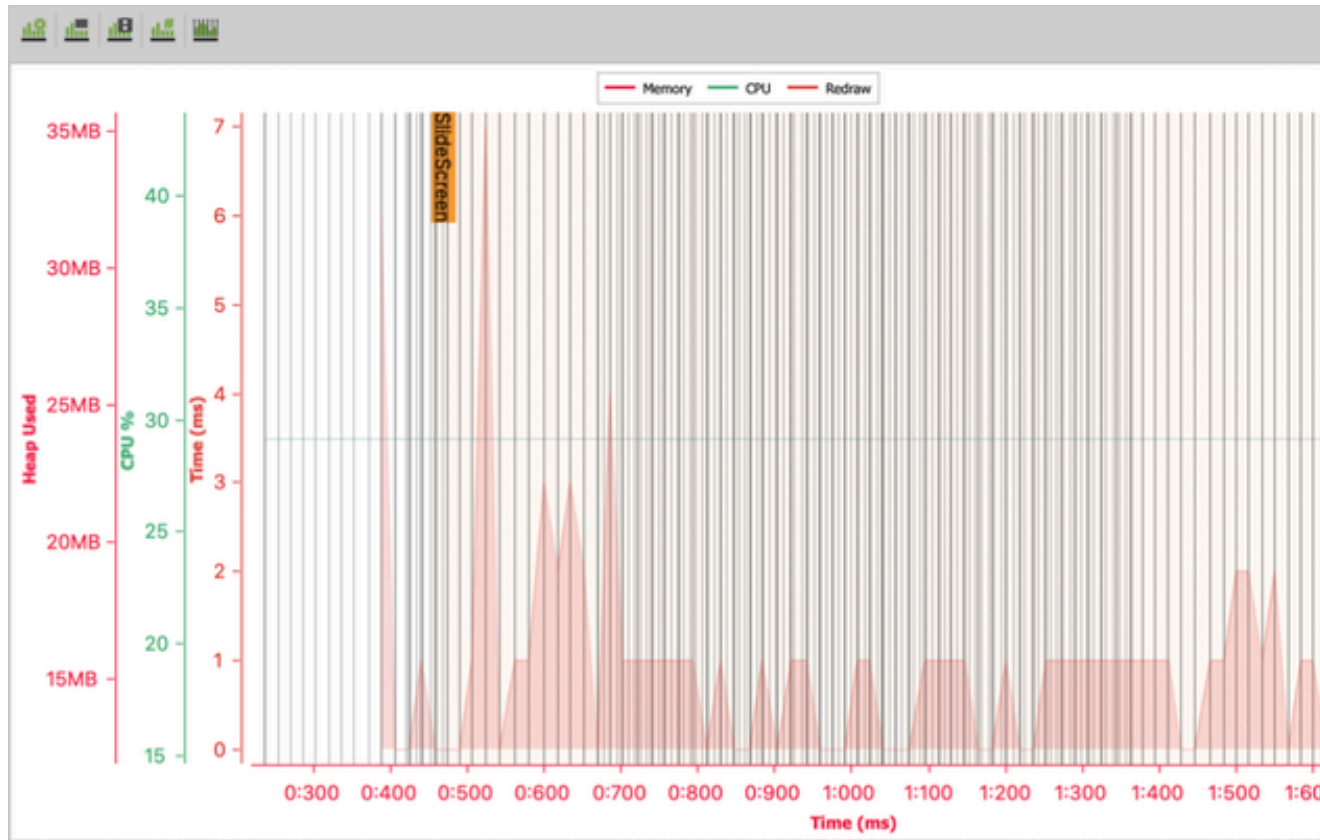
Memory is typically used by resources such as fonts and images and these are loaded on-demand. If you see that your memory usage is continuously climbing then you might want to correlate that with some of the rendering activity. It could be that you are experiencing a dynamic resource leak. Dynamic resources include controls and groups that are cloned, tables that are resized and not virtualized as well as timers and animations that are created from Lua. You can use the pre-configured resource count plot to see whether or not these resources collectively are growing. If they are then further investigation may lead you to look at what kind of resources are being created and to ensure that they are being properly cleaned up within the application.

External Data Source Accountability

Storyboard is an event-driven system and as such interfaces with external applications using events. It may be that the frequency of events arriving from outside sources may be affecting the operation of the UI. This might manifest itself as longer render times as they're observed in the render time pre-configured plot. In order to visualize the number of events that are being generated of any particular type you can use the custom metric plot and specify the events that you are interested in and plot either count or duration as a cumulative value on a per screen or application basis.

Toggling Event Lines

The Chart also has the ability to enable or disable vertical line markers that allows users to visualize the event on each screen. As Storyboard is entirely event driven, it provides the essential ability for a user to be able to determine which screen has a cluster of event activity and how it relates to some of the other metrics discussed in the sections above.



Plot Selector

The plot selector tool allows a user to select which plots are to be shown or hidden in the plot area. Adding plots from the animation, custom record, and predefined record plots will appear in this view. The plots that are currently in that view can also be deleted.

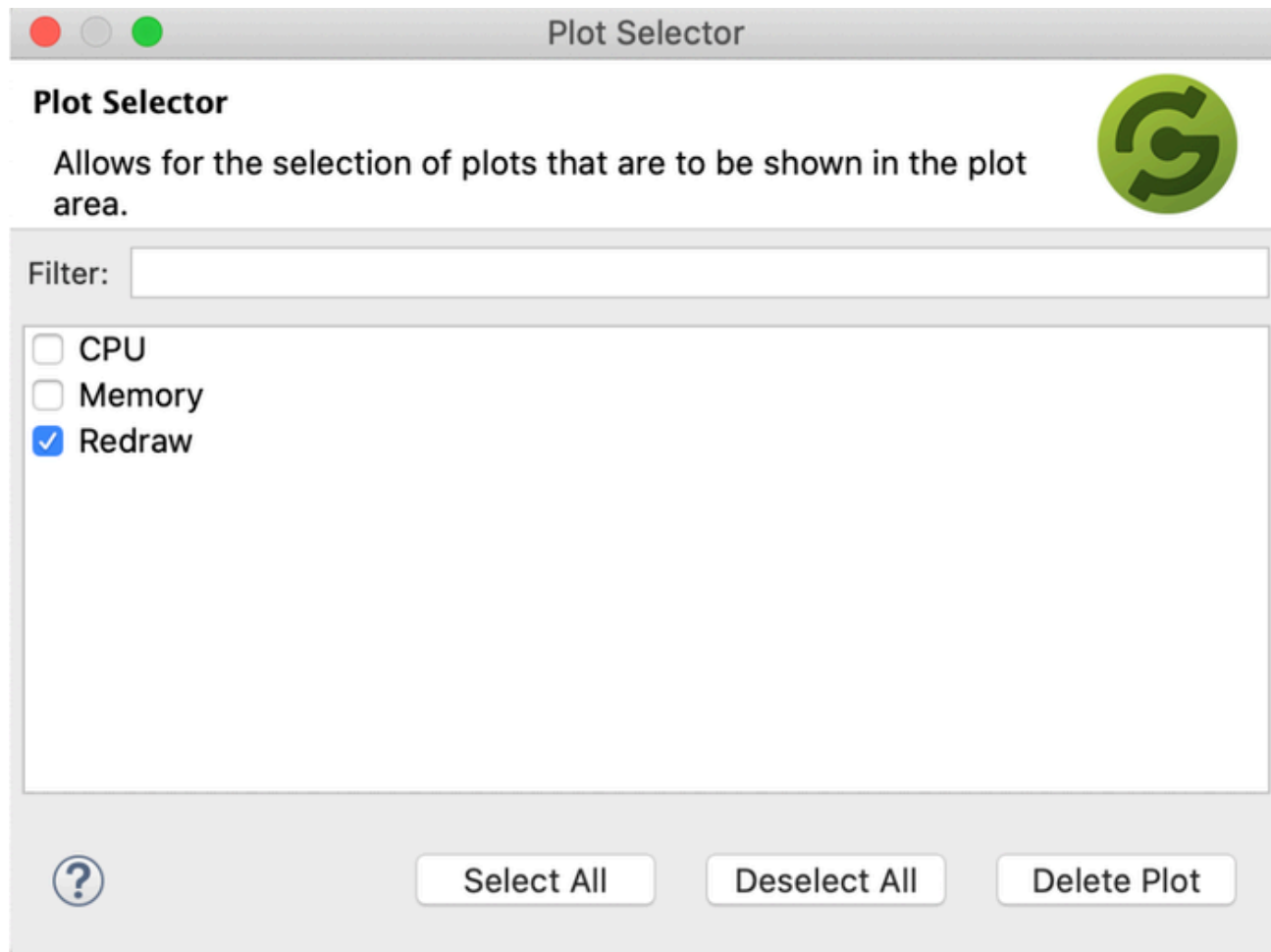





Chart UI Elements and Usage

The Tool Bar



The left tool-bar consists of the tools discussed in the sections above (from left to right):

-  The *Plot Selector* Tool
-  The *Preconfigured Analysis Plots* Tool
-  The *New Animation Plot* Tool

-  The *New Metric Plot* Tool

-  The *Event Lines Toggle* Tool

The right tool-bar consists of the tools that take actions on the appearance of the charts (from left to right):

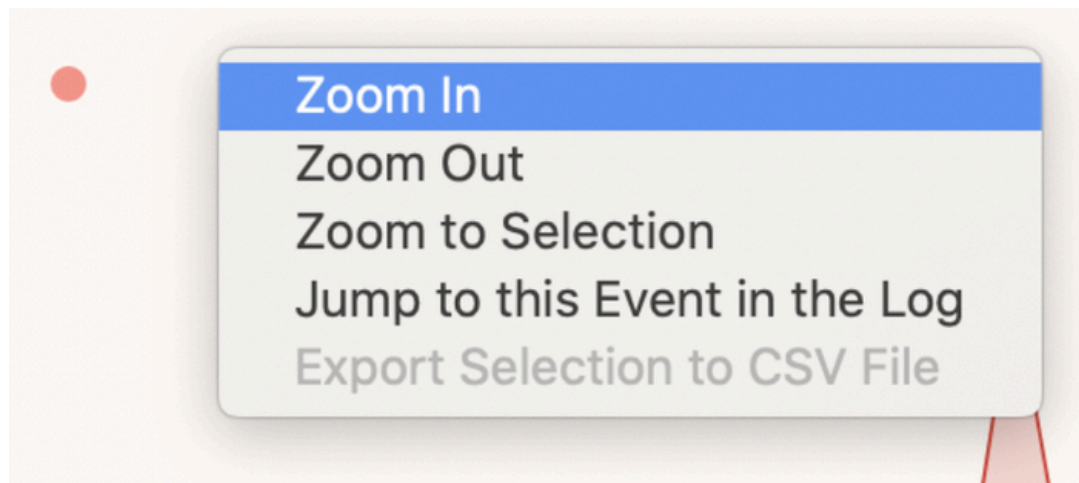
-  The *Zoom-In* Tool

-  The *Zoom-Out* Tool

-  The *Reset-View* Tool

The Right Click Menu Items

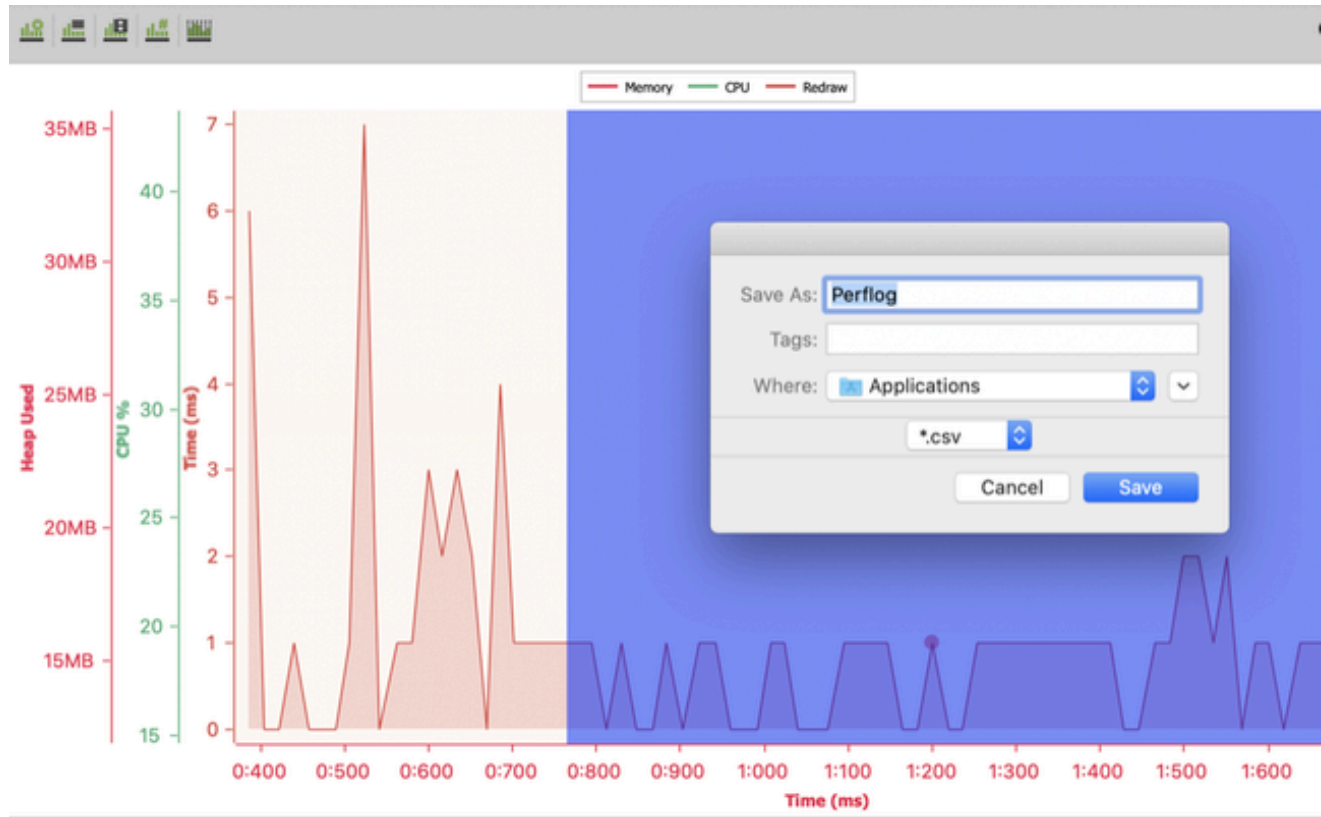
The user also has the ability to use some of the functions to “*Zoom In*”, “*Zoom Out*” and “*Zoom to Selection*” abilities from the menu items below.



“*Jump to this Event in the Log*” provides the user the ability to jump to the specified event in the Log tab. From here, a user has the ability to look at the grouped performance record involving that single event or group of events in the case that there were multiple events for that particular time instance.

General Information Creation/Deletion Information Log Graphs				
Description	Type	Operation	Duration	Start Time
image	sb_RenderExtension	render	0	970
images/slide_bg.png	sb_Image	render	0	970
SlideHorizontal	sb_Control	render	0	970
image	sb_RenderExtension	render	0	970
images/slide_horz....	sb_Image	render	0	970
SlideVertical	sb_Control	render	0	970
image	sb_RenderExtension	render	0	970
images/slide_vertic...	sb_Image	render	0	970
SlideDiagonal	sb_Control	render	0	970
image	sb_RenderExtension	render	0	970
images/slide_diag....	sb_Image	render	0	970
ArrowSign	sb_Control	render	0	970
image	sb_RenderExtension	render	0	970
images/arrow_sign....	sb_Image	render	0	970
SlideXY	sb_Control	render	0	970
image	sb_RenderExtension	render	0	970
images/slide_x_y.png	sb_Image	render	0	970
display_update	sb_RenderManager	render	1	970
display_push_front	sb_RenderAPI	render	1	970
gre.motion	sb_EventQueueByExter...	queue	0	971
gre.motion	sb_Event	dispatch	0	971
gre.motion	sb_EventQueueByExter...	queue	0	979
gre.motion	sb_Event	dispatch	0	979
Internal Callback	sb_EventQueueByExter...	queue	0	986
gre.redraw	sb_EventQueueByExter...	queue	0	986

“Export Selection to CSV File” provides an export functionality to extract the records between a selection range of the gathered trace. The user can select the selection range that they wish to investigate by clicking and dragging horizontally across the plot area to start the selection. This is shown in the image below. They can then right click and export the selection to their desired location.

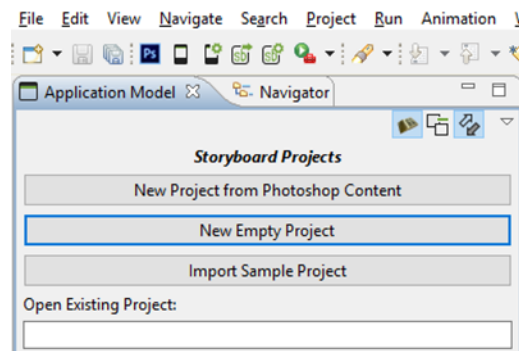


Chapter 5. Creating A Storyboard Project

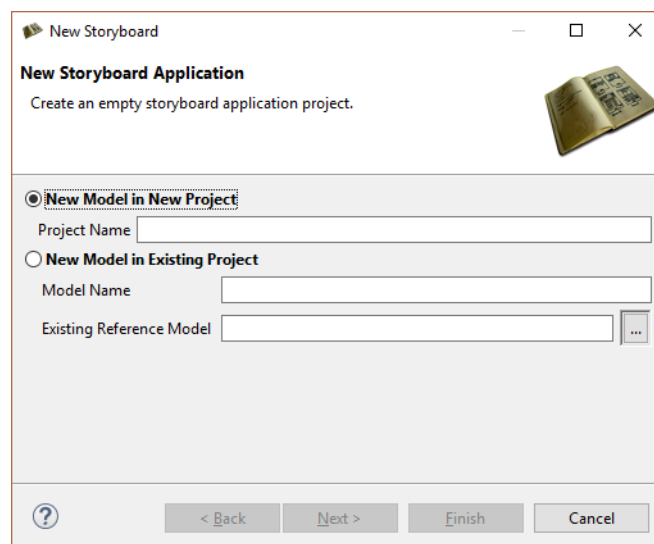
You can create a new Storyboard project in various ways: as a new empty application, as a Photoshop™ import, as an import of a sample, or from an existing Storyboard Designer project or Storyboard Engine *.gapp file.

New Empty Storyboard Project

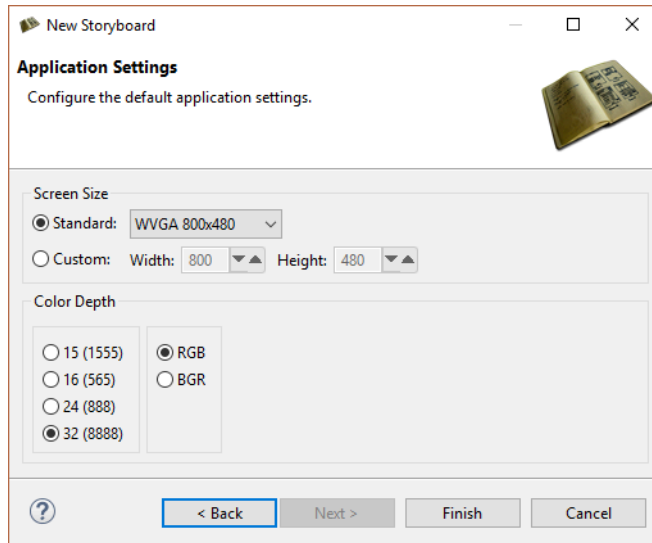
In order to create an empty Storyboard project that contains only minimal application structure, select File > New > Storyboard Application.



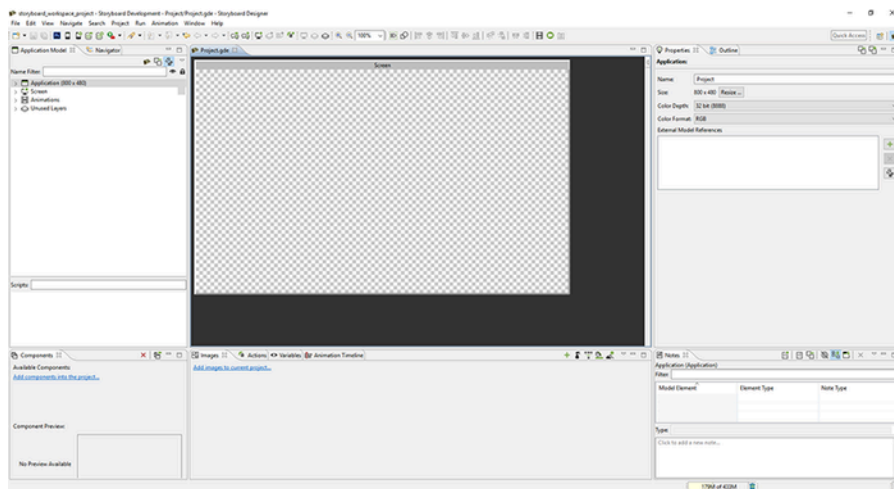
This opens the New Storyboard Application wizard and prompts for a name to use for the Storyboard project. This name will also be used for the initial Storyboard application file.



Next, set the size of the screen display and its color depth characteristics. Once configured, these settings remain configured for the life of the application.



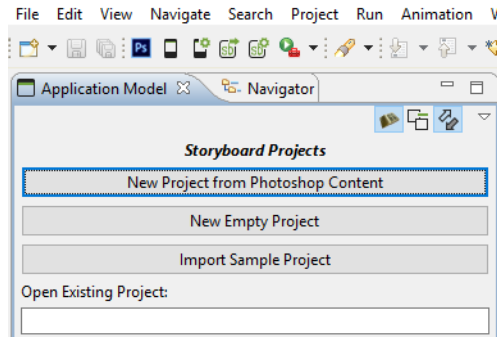
Select Finish to close the wizard, create the new project, and automatically open the Storyboard editor on the project:



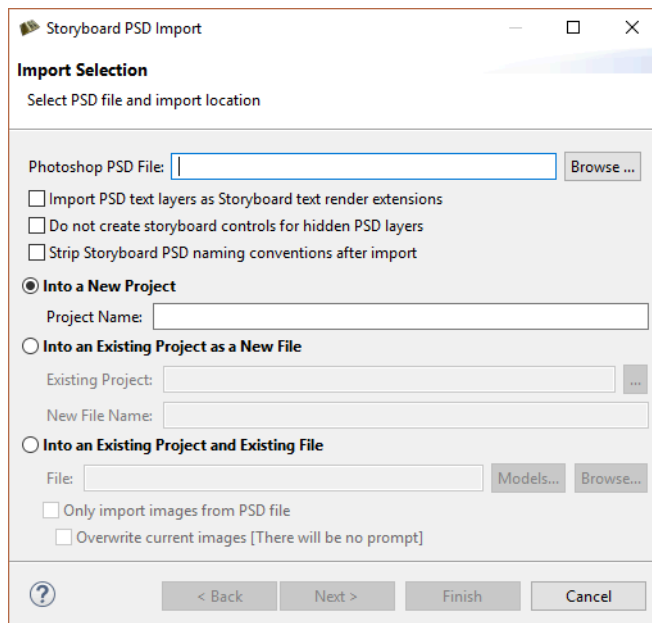
New Project from a Photoshop PSD File

A common place to start with application development is the artwork and graphics, or the visual content. Visual content for a Storyboard application is typically prepared in Photoshop™. The Photoshop™ import feature provides the ability to use these design files to jumpstart application development.

To use the import feature, select **File > Import**.



Select Photoshop PSD File and click Next, then select the PSD file that you want to import from the local filesystem.



Select whether to import in to an existing project or in to a new project. You can control how the PSD import is executed by changing the PSD import options.

Once the Photoshop™ file is imported the application opens in the Storyboard editor. The layer information from the Photoshop™ file is maintained, and the layers created as part of the model are automatically included in the Application View as children of the screen(s) that they were associated with or as part of the *Unused Layers* category for layers that were hidden during the initial import.

The screen with the name of the PSD file that is now a part of the application reflects the last visible state of the file when it was saved and contains those layers that were visible during that editing session.

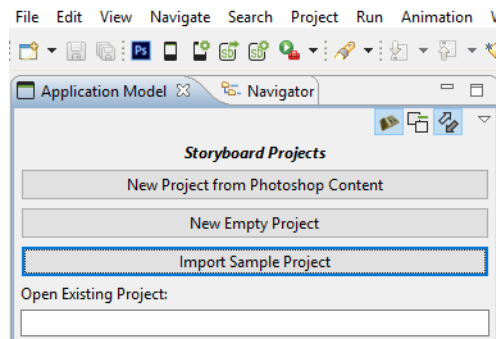
New Project from a Storyboard Sample

The Storyboard installation contains a number of samples that you can use as reference for different UI development scenarios. These sample projects are generally not full-featured programs but are snippets designed to demonstrate how to use features of Storyboard including render extensions, actions, animations, or other visual effects.

Use the following options to launch the Import Storyboard Sample dialog:

- Select Import Sample Project from the Application Model view
- Select File>New>Storyboard Sample
- Select File>Import and then Storyboard Development>Storyboard Sample

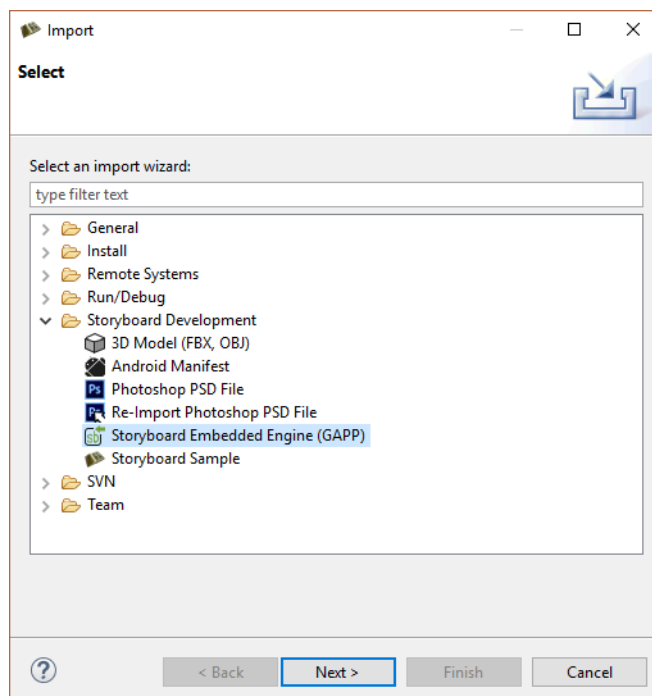
These options bring up a dialog allowing you to select one or more projects to import. The keyword filter allows you to search for projects that contain functionality that you may be interested in by searching both the project name as well as the project descriptions.



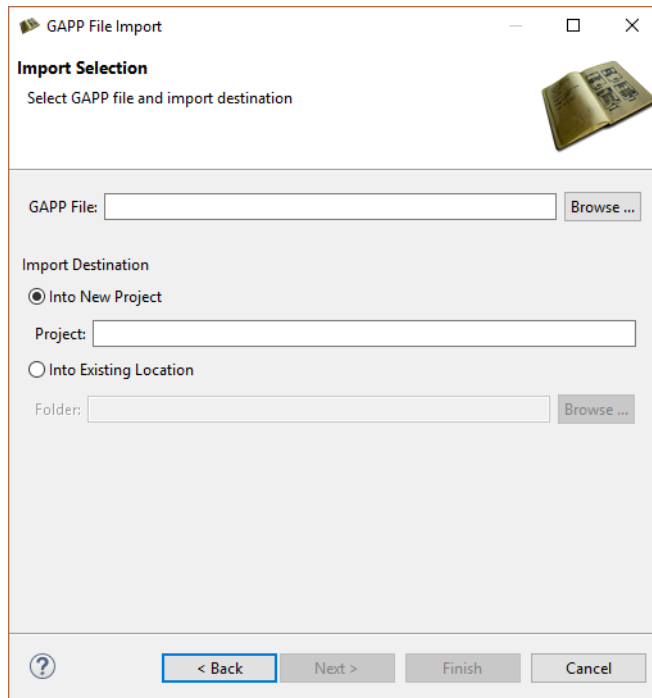
Once selected, a project will be created and the sample will automatically be opened into the Storyboard Designer editor.

New Project from Storyboard Embedded Engine File

Storyboard Designer can also round-trip export/import the files created as part of the Storyboard Embedded Engine, also known as *gapp* files since they typically have a .gapp file extension. These files are imported in the same way as a Photoshop™ file, using the import wizard from File > Import > Storyboard Development > Storyboard Embedded Engine (GAPP).



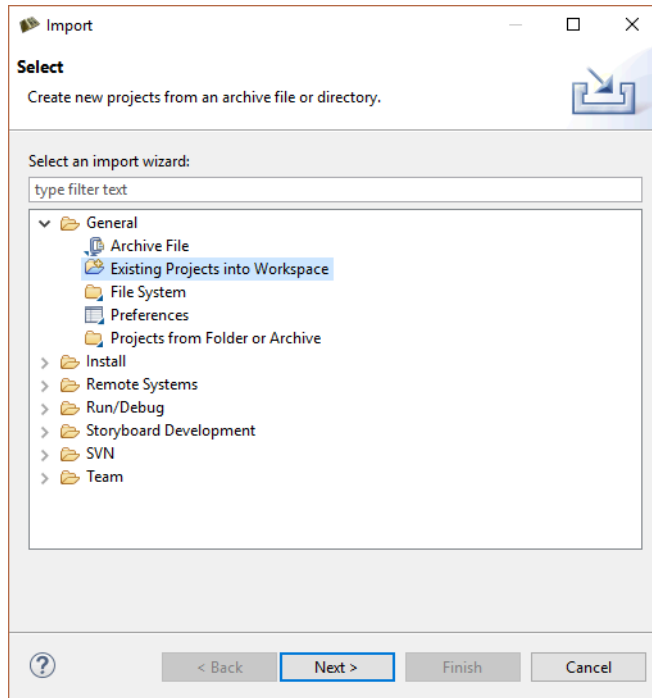
Select Next, and then select the embedded engine file to be used for import.



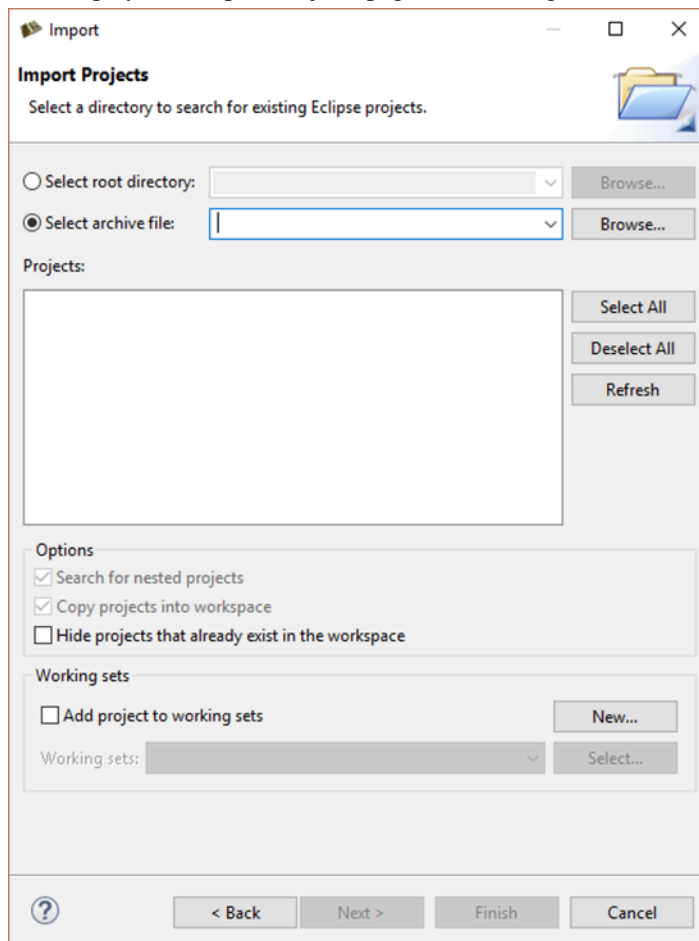
Existing Project Import

Occasionally it may be convenient to share an existing project from one workspace to another. If the project is archived or its directory structure completely copied to a new location then it is possible to import the Storyboard Designer project as an existing project.

Select File > Import to launch the wizard and select Existing Projects into Workspace.



This displays the Import Projects page of the dialog.



Search for existing projects contained in the filesystem by selecting the Select root directory option or projects that are contained within an archive file by selecting the Select archive file option.

In either situation, the Projects panel will populate with the content that can be extracted from the location selected. If projects exist in the workspace with the same name, then those projects will be shown but disabled in the list. Select the projects you want to import to your workspace and press Finish to exit the dialog and return to the workspace. Note that projects that are imported will not automatically be opened in the Storyboard editor.

For projects that are being imported from the filesystem you also have the option to Copy projects into workspace which, as the name indicates, will copy the project content into the workspace. Without making this copy selection a new project will be created that links to or references the existing location of the project in the filesystem.

Working with Multiple Application Design Files

In large projects it might be desirable to split the application design work, not only among multiple application designers, but also among multiple design files in order to minimize the amount of conflict that needs to be managed by a revision control system such as Subversion, Git, or Mercurial.

In order to facilitate working with multiple files, Storyboard maintains each design as a stand-alone free runnable application, even when it may be later incorporated into a larger unified application. This separation allows a more rapid development cycle as application developers are simulating and tuning content in a more focused environment rather than having to consider all other system functionality.

To create a unified application, the Storyboard simulators and exporters have been modified to allow them to accept multiple application files which they will merge together to produce a single unified output for the Storyboard Engine.

Starting a multi-file application development is the same as starting a regular project since before you can have two projects, you need to have one project. Typically the initial project created will be the master application and will host the majority of the shared resource content. This is only a convention and not a technical requirement.

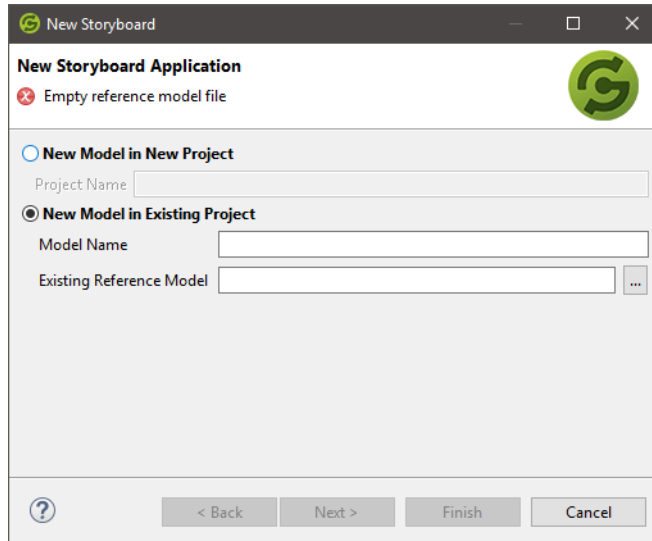
Once another application is required, for example to represent a distinctly themed area of the unified application, then you will want to create another Designer application model file within the base project.

Note

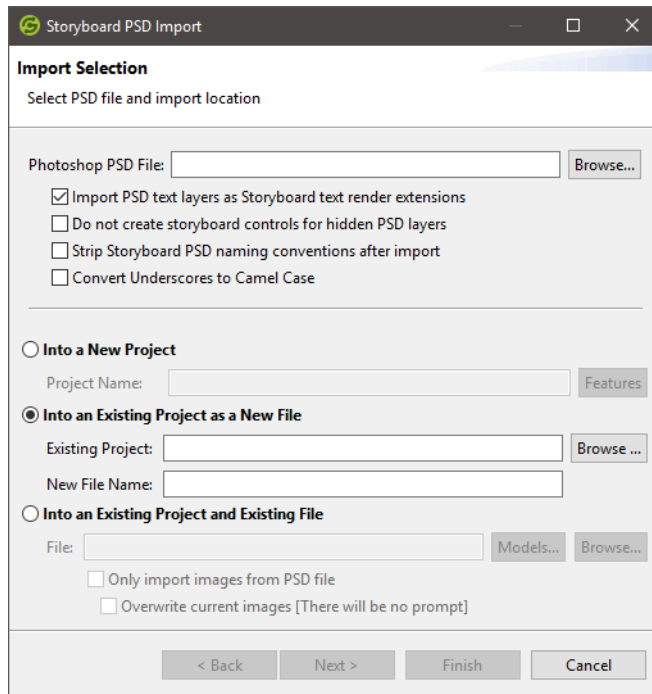
Currently, multi-file applications are all hosted in the same project so that they can ensure proper sharing of image, font, and script resources.

You can create another application file in several ways:

Create a blank application using: File > New > Storyboard Application ... and select New Model in Existing Project.



Import PSD content into a application: File > Import > PSD ... and select Into an Existing Project as a new File.



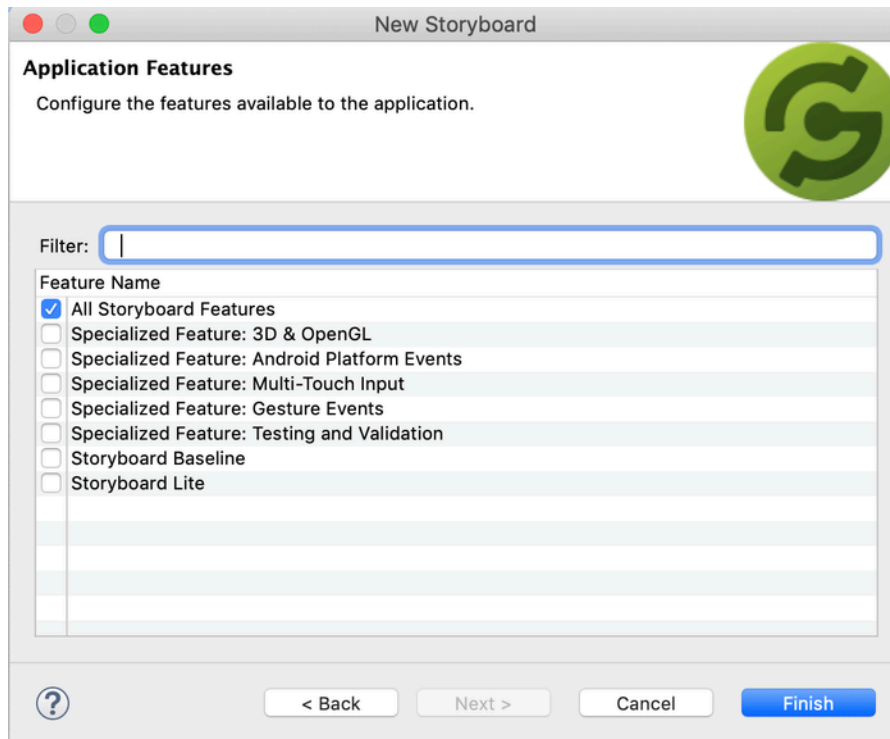
You can also copy and paste an existing application and rename the *.gde file to bootstrap a new design.

Doing this will result in multiple application model (*.gde) files that are all contained within the same project. They all reference the same image and script resources. Working with multiple project files offers a great deal of flexibility but requires special development consideration. These issues are discussed in more detail in Chapter 22, *Multi-File Application Development*.

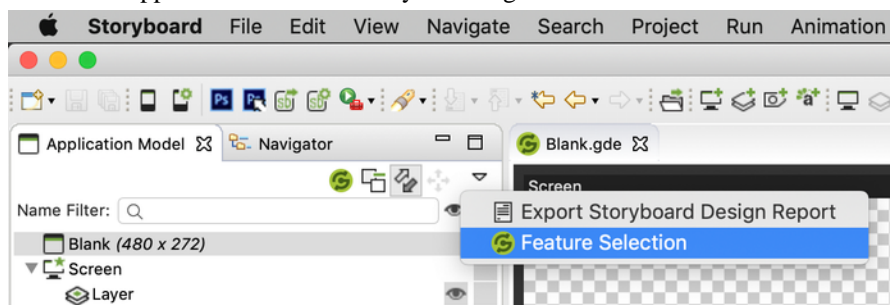
Controlling Feature Enablement on Projects

Storyboard Designer provides a very rich set of functionality for application developers to use as they are building their applications. However, it may be the case that the embedded target system constrains which

functionality should be used. In these situations, and in order to provide an appropriate level of guidance to application developers about what they can and can't use, Storyboard Designer provides the ability to limit what features are available. This filtering happens only in the Designer user interface to filter the events, actions and render extensions that are appropriate for the configuration selected.



The feature selection display shown can be accessed in the new project wizards when a project is created from a PSD file or when an empty project is created. The feature selection can also be changed after the fact in the Application Model view by selecting the view menu:



Features are grouped into two main categories, specialized features and platform configurations. The platform configurations *Storyboard Baseline*, *Storyboard Lite* identify types of target platform configurations as either fully featured or as resource constrained. The specialized features are generally selected in addition to one of the platform configurations to add in specialized functionality. The default feature configuration for is to select *All Storyboard Features*.

Chapter 6. Adding Content to your Application

Once your application project has been created the next step is to start adding new model elements, screens, layers, and controls that reflect the display and organization of the user interface you want to build. You can add much of this new model content through the Storyboard Designer editor or via the Application Model View using the Add ... menu item. This will generally add the selected item as a child of whatever the current selection is.

Storyboard provides a flexible model for you to organize your application, however there are some general guidelines that you should follow. Applications should have screens that represent the different modalities of the application. For example, a thermostat application may have a main screen, a separate screen for managing settings, and another for a manual override behavior. Having all of the screens of the application line up with the different operating modes that a user might encounter with the product means that a new developer can immediately engage with a new project. When compared to a traditional code-based approach this visual roadmap allows for faster iteration and development.

Similar to using multiple screens to provide immediate visual guidance on how an application operates, layers can be used to consolidate common visual experiences. For example, an application may have a common navigation or feedback interface that should be represented in all modes of operation. This common visual presentation can be placed into one or more layers and those layers used on multiple screens of the application. When a layer is shared in Storyboard, it is truly shared. The control content on the layer is not replicated or copied, but is a single reference. For this reason it can be an efficient design approach to segment content to multiple different layers that can be reused across multiple different screen contexts.

Using multiple screens and layers in a Storyboard application does not impose a significant memory or CPU overhead cost. Similarly, due to resource references, all image and font resources in Storyboard are pooled and centrally managed. Once an image (or font) is referenced you can use it in a different visual context without incurring any additional memory overhead. This can be an important consideration for deeply embedded applications that may want to offer a compelling user interface but have a limited amount of storage Flash or dynamic RAM. With a judicious selection of common images, or by using techniques such as nine-patch images, it is possible to have a rich and engaging experience with minimal resources.

Object Naming and Reserved Namespaces

As you create your application you will encounter a number of situations where you will need to determine the name of an object. This might be the name of a screen, layer or control if you aren't importing from Photoshop or this might be name of an event or event a Lua file or function. Being consistent with your names will help your application stay organized and avoid conflict.

Storyboard establishes and reserves several namespaces to ensure that system events and actions do not conflict with user defined functionality.

- `grd_` This prefix is reserved for all of the Storyboard internal variables. These are the variables that are associated with specific properties of model objects and are generally manipulated from Lua scripts.
- `gre.` This prefix is reserved for all of the Storyboard standard events. You will see this prefix when you are picking events to trigger your actions. When you create your own events, you should avoid the `gre.` prefix.

- gra. This prefix is reserved for all of the Storyboard defined actions. This action is the runtime identifier and is something that you would encounter if you were creating your own actions using the Storyboard SDK product.

In addition to the reserved namespaces, Storyboard Designer enforces a number of naming rules:

- Starts with a character a-z/A-Z not a digit or special character. Only _ (underscore) is supported as a special character and there can be no spaces in names.
- Screens and layers must be uniquely named. Controls and groups can use the same names as long as they have different parent containers. Two controls within the same layer can't have the same name.
- Variables can not have the same name as model objects in their parent container. For example you can't declare a layer variable `button` and have a control on the same layer named `button`.

Chapter 7. Working with Events

Event Editor

The Event Editor is used to add and edit event definitions.

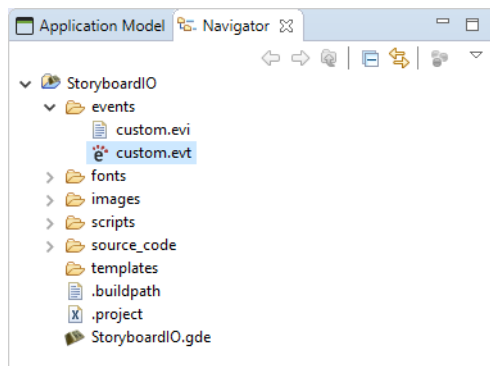
Opening the Event Editor

The different ways to open the Event Editor are:

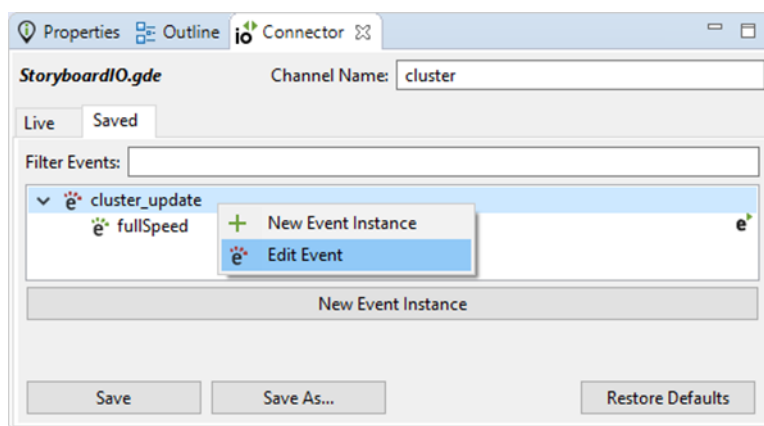
- From the main toolbar, with a Storyboard Editor open, click the Open Event Editor button.



- In the Navigator View, double click a .evt file.



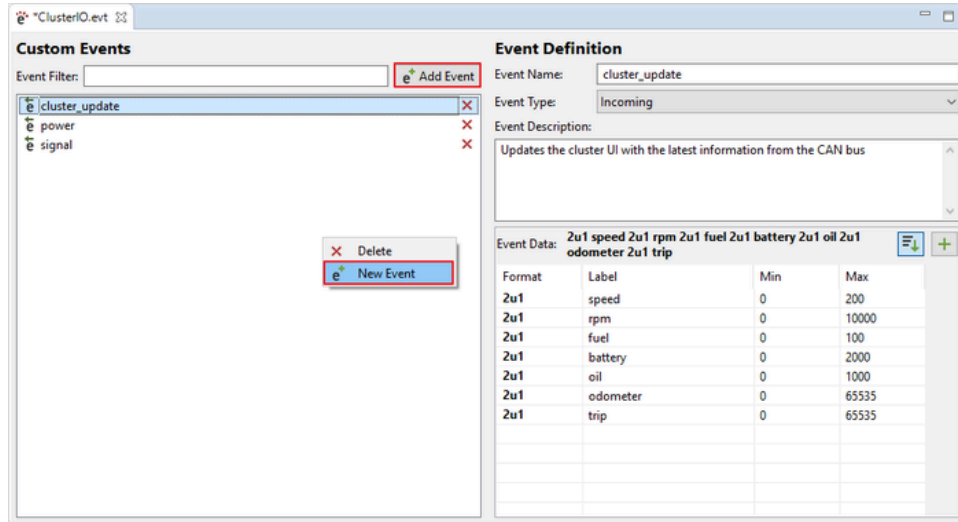
- In the Storyboard IO Connector View, in the Saved mode, right click an event > Edit Event.



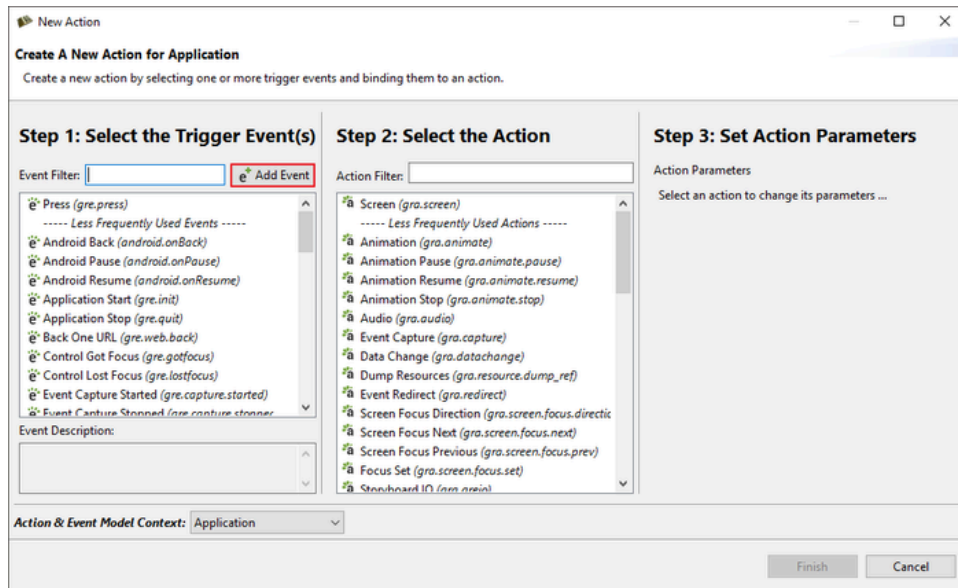
Adding an Event Definition

The different ways to add a new event are:

- From the Event Editor, select the Add Event button or right-click the event list, select the New Event menu item



- Right-click a model object and select Add > Action menu item to open the New Action dialog, then select the Add Event button.



An event definition is made up of four parts, name, type, description and format.

New Event

Event definition
Create a new event.

Event Name:

Event Type:

Event Description:

Event Data: **2u1 speed 2u1 rpm 2u1 fuel 2u1 battery 2u1 oil 2u1 odometer 2u1 trip** +

Format	Label	Min	Max
2u1	speed	0	200
2u1	rpm	0	10000
2u1	fuel	0	100
2u1	battery	0	2000
2u1	oil	0	1000
2u1	odometer	0	65535
2u1	trip	0	65535

Finish Cancel

- **Name** - Must not be blank and must only contain alphanumeric characters, periods and underscores.
- **Type**
 - **Incoming** - Events coming into the user interface from external applications.
 - **Outgoing** - Events from the user interface into external applications.
 - **Self** - Events generated from the user interface to itself and are generally used to trigger additional behavior in the UI.
- **Description** - This field is used to store information about the event definition. Different users and teams will have their own style of event descriptions. A place to store information such as; conditions for events firing, expected behavior or any other descriptive information.
- **Format** - For more information about event format strings, read the section on Serialized Data and the Event Format String. To add elements to the format string, press the Add Event Data Element + button and open the New Storyboard IO Data Element Wizard.

New Storyboard IO Data Element

SBIO Data Element

Select the type of the new SBIO data element

Name: speed

Format:

- (4f1) Float 32-bit
- (8u1) Unsigned Integer 64-bit
- (4u1) Unsigned Integer 32-bit
- (2u1) Unsigned Integer 16-bit**
- (1u1) Unsigned Integer 8-bit
- (8s1) Signed Integer 64-bit
- (4s1) Signed Integer 32-bit
- (2s1) Signed Integer 16-bit
- (1s1) Signed Integer 8-bit
- (1s0) String

Array Size: 1

? Finish Cancel

- **Name** - Must not be blank, must only contain alphanumeric characters or underscores and cannot begin with a number.
- **Format** - The size(8,4,2,1 bytes) and type(signed, unsigned, float) of data.
- **Array Size** - If this data element is an array, the Array Size will be greater than 1.

Data Element Types

These are the different data element types and their default Min/Max attributes.

Event Data: 8u1 u8 8s1 s8 4f1 f4 4u1 u4 4s1 s4 2u1 u2 2s1 s2 1u1 u1 1s1 s1 1s0 string +			
Format	Label	Min	Max
8u1	u8	0	18446744073709551615
8s1	s8	-9223372036854775808	9223372036854775807
4f1	f4	-3.4028234663852886E38	3.4028234663852886E38
4u1	u4	0	4294967295
4s1	s4	-2147483648	2147483647
2u1	u2	0	65535
2s1	s2	-32768	32767
1u1	u1	0	255
1s1	s1	-128	127
1s0	string		

Changing Element Array Length

It is possible to change the size of an element array during the creation of the data element or after the event has been created by right-clicking on an event data element in the Event Editor > Change Array Length.

Event Definition

Event Name:

Event Type:

Event Description:
Updates the cluster UI with the latest information from the CAN bus

Event Data: **2u1 speed 2u1 rpm 2u1 fuel 2u1 battery 2u1 oil 2u1 odometer 2u1 trip**

Format	Label	Min	Max
2u1	speed	0	200
2u1	rpm	0	10000
2u1	fuel	0	100
2u1	battery	0	1000
2u1	oil	0	1000
2u1	odometer	0	100000
2u1	trip	0	100000

Context menu for 'battery':

- + Add New Element
- ✗ Delete Element
- n Change Array Length**


The Change Array Size Dialog will be opened. This is a simple dialog that accepts a number which will be used as the array length for the selected data element. The array length must be greater than zero.

Change Array Size

Set the number of data elements for this array

Array Size:

Reordering Event Data Elements

The event format can be changed by reordering the data elements. This is done using drag and drop operations. There are rules that enforce the order of data elements from the largest data types down to the smallest data types. Data elements that are of common size can be reordered amongst themselves, regardless of their type (i.e. 4u1, 4s1 and 4f1). To enable/disable the enforcement of sorting rules, toggle the  button.

Editing Min and Max Attributes

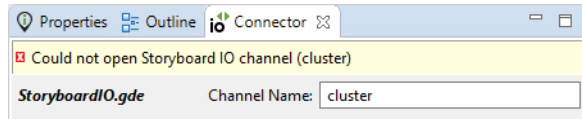
Each event data element has min and max attributes associated with it. It is a good approach to define these values when defining the event. These attributes are not used by Storyboard Engine, they are only used by Storyboard Designer Storyboard IO Connector. To edit these values, double-click in the cell to activate the editor.

Using the Storyboard IO Connector

The Connector View will show **(No Model Selected)** if a Storyboard Application hasn't been opened. In order for the Connector View to populate, a Storyboard application must first be opened. Once an application has been opened, it will show the name of the current Storyboard Application in the top left corner of the view.

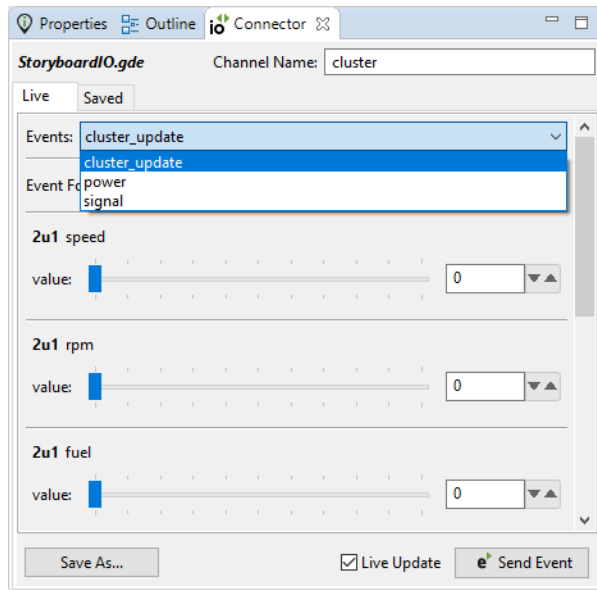
The Storyboard IO channel name is autopopulated with the name of the selected Storyboard Application. It can be overridden in the case where your application has specified a different SBIO channel name.

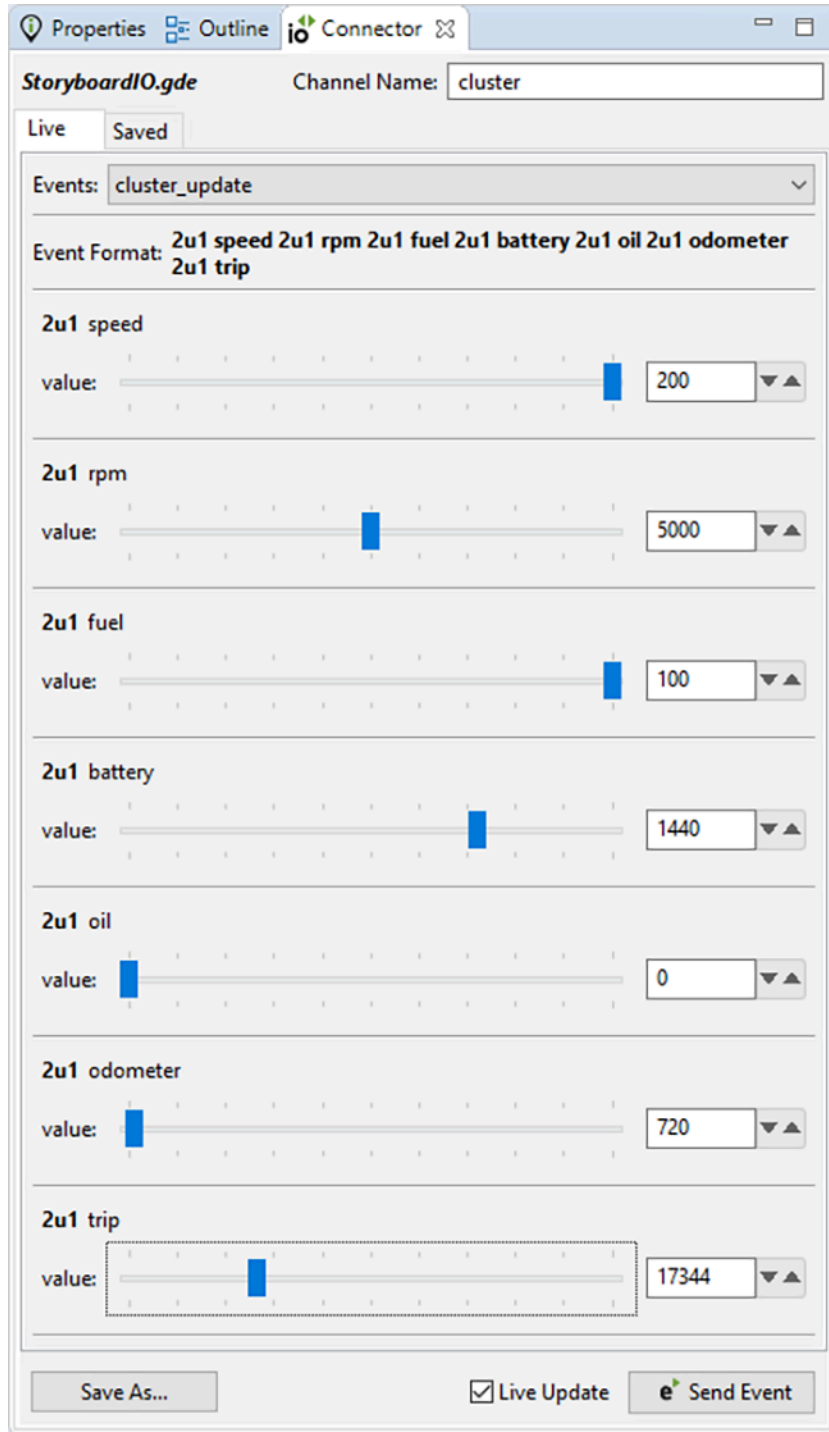
Your Storyboard application must be launched before sending an event from the Connector View or else a warning will appear.

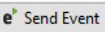


Live Mode

The Live Mode tab in the Connector View is used to send events on the specified SBIO channel. All of the user events defined for the current application are listed in the dropdown Events menu. Selecting an event from the list will populate the Event Format field and element editors.





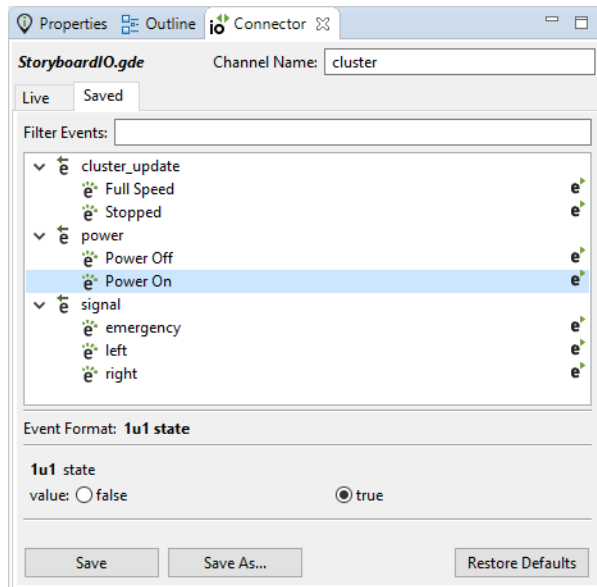
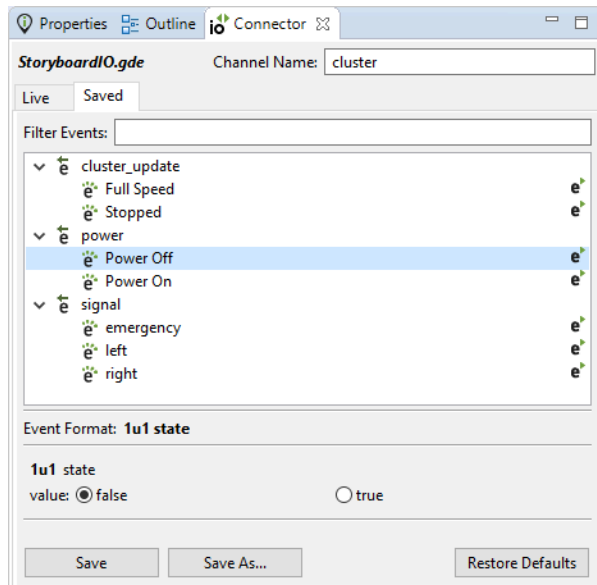
The Live Update button is enabled by default, it determines if modifying any event data element values should send a new event on the specified channel, this means dragging a slider will send a series of events to your application. If the Live Update button is disabled, one can change values as much as necessary and then the event can be fired by pressing the  Send Event button.


The Save As... button is used to save the current event configuration for reuse in Saved Mode, this is known as an **Event Instance**, these are the primary focus of the Saved Mode tab.

Saved Mode

The Saved Mode tab in the Connector View is used to send preset events, known as event instances. An event instance is an event that has been pre-configured for reuse.

Here we have the event 'power' and two event instances 'Power On' and 'Power Off'. The Connector View will default to using a boolean editor when an event element is defined with a min of 0 and a max of 1.



There are two ways to send an event from Saved Mode. The first method is to double click on an event instance in the events tree, the second is to press the  button that appears beside event instances in the event tree.

The Save button is used to save the changes made to the currently selected event instance.

The Save As... button is used to save the current event configuration as a new event instance. Provide a name for the new event instance in the Save As... dialog.

The Restore Defaults button is used to return the currently selected event instance back to its saved state.

The Filter Events field is used to search for events and event instances by name.

Data Element Editors

There are different editors available for working with event data elements in the Connector View.

- Number Slider - Contains a slider and a number spinner. Not available on 8u/8s/4u element types.

- Number Spinner - A number spinner, can perform mathematical calculations on, increment and decrement the current value. Not available on 8u/8s/4u element types.

- Number Field - A text field for inputting numbers. Only available on 8u/8s/4u element types.

- Color Picker - Open Color Dialog or input RGB color values. Only available on 4u type elements.

- Boolean Radio - True and False buttons.

- Boolean Toggle - A button with an up state (False) and a down state (True).

- Text Box - A text field for input of UTF-8 text. Only available on 1s type elements. 1s0 elements are null-terminated while 1sN elements are limited to N bytes. This is the only editor available for 1s0 element types.

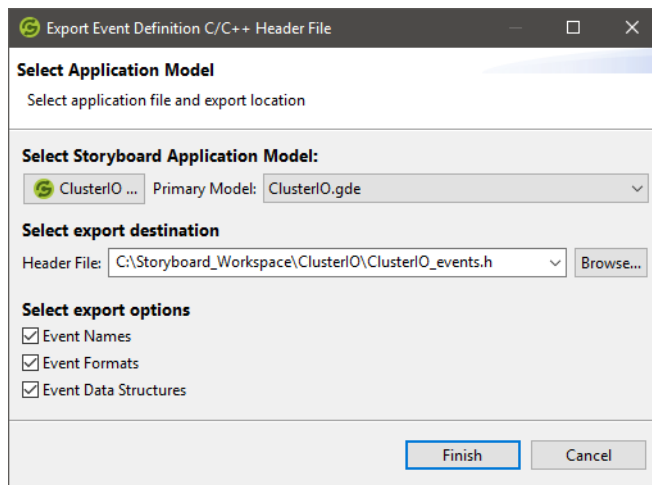
Storyboard IO Event C/C++ Header Export

When defining events, they are often meant to be injected from an external application. To facilitate this, Storyboard has the ability to generate a C/C++ Header file that contains event names, format strings and data structures for all events of type **Incoming** and **Outgoing**.

There are two ways to run an event header export.

- From the Navigator View, right-click on a GDE file > Storyboard Export > Storyboard IO Event Header (C/C++)
- File > Export... > Storyboard Development > Storyboard IO Event Header (C/C++)

The following dialog is used to configure the parameters of the export with options to export names, formats and data structures.



This chapter further discusses working with Storyboard IO

Chapter 8. Connecting Events to Actions

Events trigger actions. Actions perform the work of the application by making adjustments to the user interface or interacting with outside applications.

Every event has a name which is used as the matching trigger for actions. Events may also contain an optional data payload and a symbolic description of that payload in what is referred to as a format string. Events can be directed at a particular model element, contain position information to target a particular screen region (such as a press event) or be completely independent of the display state of the user interface.

Events fall into three broad categories:

- Standard Input Events are generic input definitions and are pre-defined within Designer and as such are not user defined events. These events represent expected user interactions such mouse or touchscreen press, release and motion as well as keyboard input. There are other standard input events defined for managing UI state changes such as focus management, screen change notification and application state management.

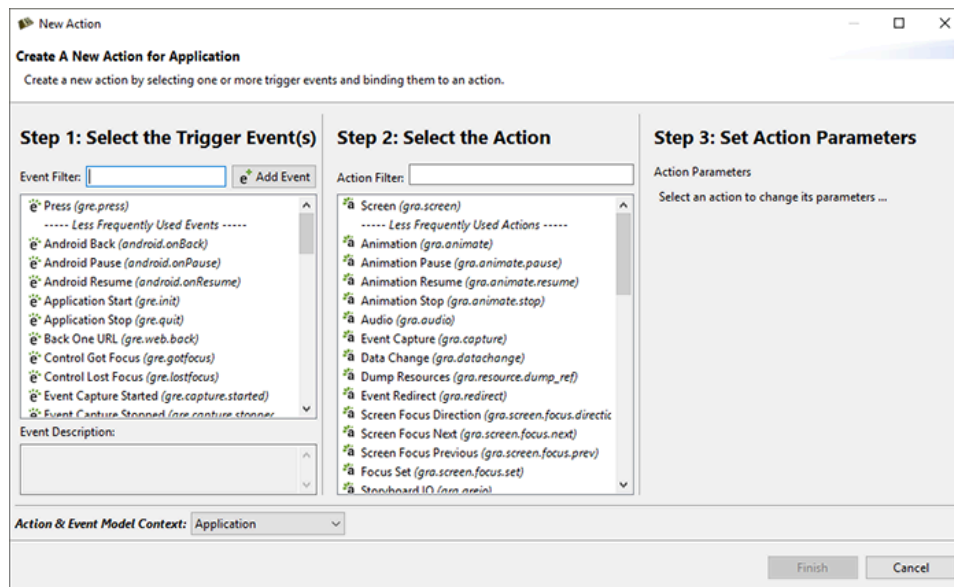
These event definitions and their data payloads are defined in the header file `idefs.h` and are also listed in the appendix Appendix D, *Standard Event Definitions*.

- UI Events are events that are defined to be used just within the context of the user interface. UI events are considered user events as they are custom to your application and you define them for your specific application needs. In general these events are generated from within the application in order to trigger some additional follow up behavior within the Storyboard application. Examples of this include the events generated in response to timers or animations. User defined events that are sent within the application are also considered as UI events, for example an application might define its own `PowerOn` or `PowerOff` events that it sends when the user initiates a touchscreen interaction. In the Event Editor, these types of events are labeled **Self Events**.
- External Events are also user events. These are events that you defined for the purpose of communicating information into and out of the user interface. These events will frequently represent the application programming interface between the user interface and the rest of the system. Due to the shared nature of these events, between the UI and the system, these events deserve a special level of diligence in their design as it requires a greater effort to change their meaning or data payloads once they are put to use. In the Event Editor, these types of events are either labeled **Incoming** or **Outgoing Events**.

The delivery of an event in a Storyboard application is what will trigger the execution of actions which, as we have mentioned, causes visual and behavioral changes in the application. Actions receive an association with an event when they are bound to a particular model element in the application hierarchy. For example the way in which a `gre.press` event triggers a `Data Change` action might look something like the following:



To add actions to your application model, right-click the application and select Add > Action menu to open the New Action dialog.



This dialog presents a three panel dialog containing the three pieces of information that you require to bind an action.

Trigger Event

This panel allows you select one or more events that you want to trigger actions. By default the list is populated with the standard events, but you can add your own user defined events as well at this point. Select the Add Event button to open the New Event dialog. In this dialog you can enter in the name of your new event and provide some optional descriptive information about the source and payload of this event. Once a new event is created it will be added to the list of available events shown the New Action dialog. All user defined events are stored in the project relative file `events/custom_evt`. This text file uses a simple formatting and can be edited directly if many user events need to be added all at one time.

Note

The name of an event is what is used to trigger actions, so event names should be carefully considered. All of the standard Storyboard event definitions are placed within a `gre.` prefixed namespace. As you develop your application and introduce your own custom events, you should take care to organize your events in a manner that allows them to be clearly identified and associated with specific functional domains within your overall product.

Action	The second panel in this dialog allows you to select one or more actions that you want to invoke when the triggering event is received. There are many different types of actions that you can draw from.
Action Parameters	This third panel's content will change based on the action selection and presents a user interface for adjusting the parameters of the selected action. If multiple actions are selected, then this panel will only show the parameters for the first action selection made, but all of the selected actions will be bound. The action arguments are presented for convenience in this dialog and can always be changed or reconfigured using the Properties View.

In order to complete the dialog at least one event and one action must be selected. The resulting actions are going to be bound to the selected model object, or if multiple objects were selected then bindings will be created for every one of the selections. You can verify that the action binding is being performed on the intended object by looking at the Action & Event Model Context information at the bottom of the New Action dialog.

After an action is bound to a model object, that action will show up in the Application View underneath the model object it has been bound to and will also be visible in the Action View when the model object is selected. The action's properties can be changed and re-configured by selecting the action in either of these contexts and then using the Properties View. For information about specific action parameters and properties, please refer to the Storyboard Action Definitions Appendix.

If you have mistakenly bound an action to the wrong model object or have it triggering on the wrong event, it is also possible to change this binding information from the Action View by double clicking on the field you want to change and making the required adjustment. Similarly you can also change the action that is being invoked when an event is matched by double clicking and selecting a new action from the drop down list. When the action is changed as many of the action parameters as possible will be transferred to the new action from the old action, but you should take care to double check your parameter settings when making this type of adjustment.

Event and Action Matching Rules

In general event processing proceeds starting with control actions and working upwards through the model until the event is finally matched by application actions. Which control(s) start this process depends on the type of event that is being processed.

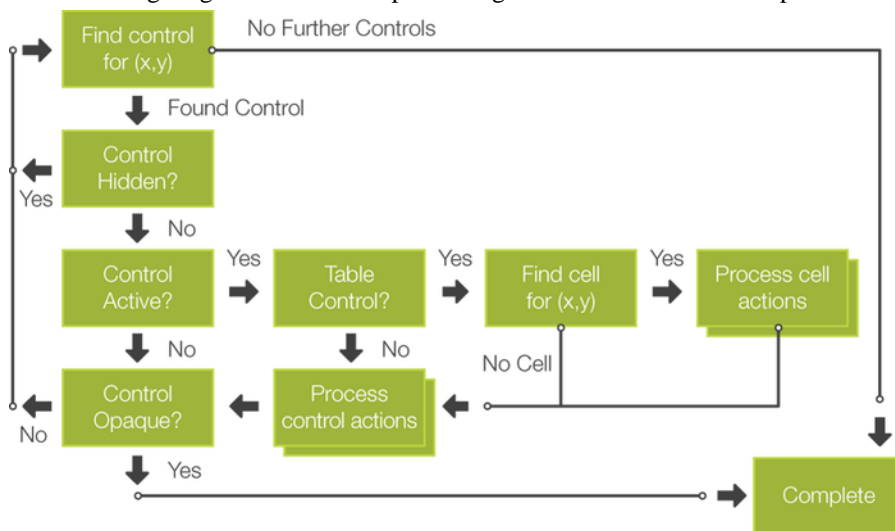
- Pointer type events such as press, release and motion are directed at a specific location within the display. For these events the controls that will be selected as the starting point for processing will be based on an intersection of the event position and the control location. Multiple controls may intersect with the event and as such the control list for processing will be ordered in the same visual front (first) to back (last) order. Events that are not visible or not active are not included in this processing.

The controls that are matched by pointer events also have the potential to become the newly focused controls if they have enabled a focus index property (off by default). Focus controls are used for non-targeted events that do not have a specific display location.

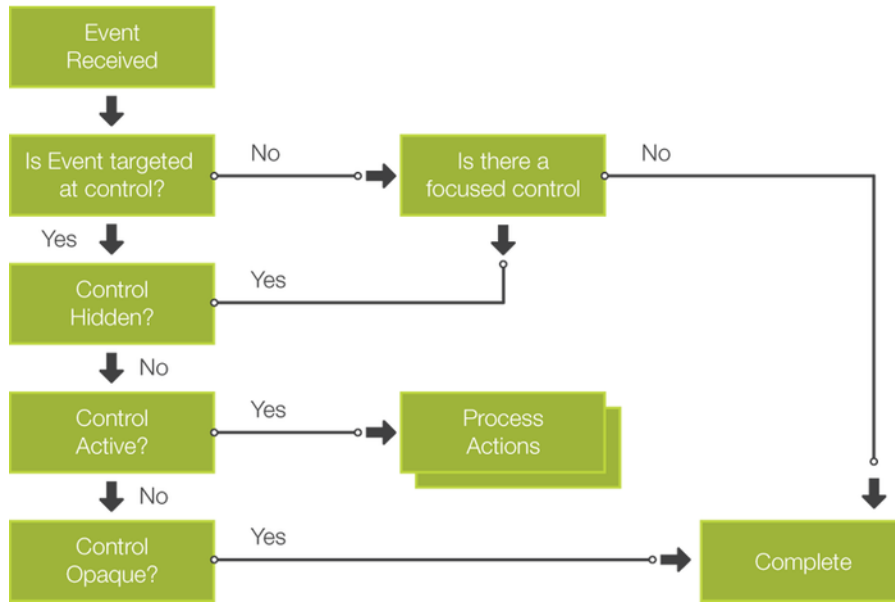
- Targeted events, which is what most user defined events will be, do not have specific screen display information and do not use an intersection rule to pick the control for processing. Instead these events will start matching at the currently focused control on the current screen if one exists. If no such control exists, then there is no control starting point and the event processing will start at the current screen.
- Targeted events. These are events that have a specific model target provided within the event payload. In these situations, the control list will be created based on what the target indicates. If the target is a control then it will be used to start processing.

If there is a list of controls then the event processing will proceed through the list with the events being processed sequentially: control, control's group (if applicable), control's layer. If there is an event to action match on a control (or group, or layer) and the properties on that model object indicate that the processing should stop, then event processing immediately passes up to the screen or application level with no further controls (or groups or layers) being processed. If through this control selection process, there is no applicable list of controls, then the processing of the event will start at the currently displayed screen and then move to the application.

The following diagram outlines the processing of an event that contains position information:

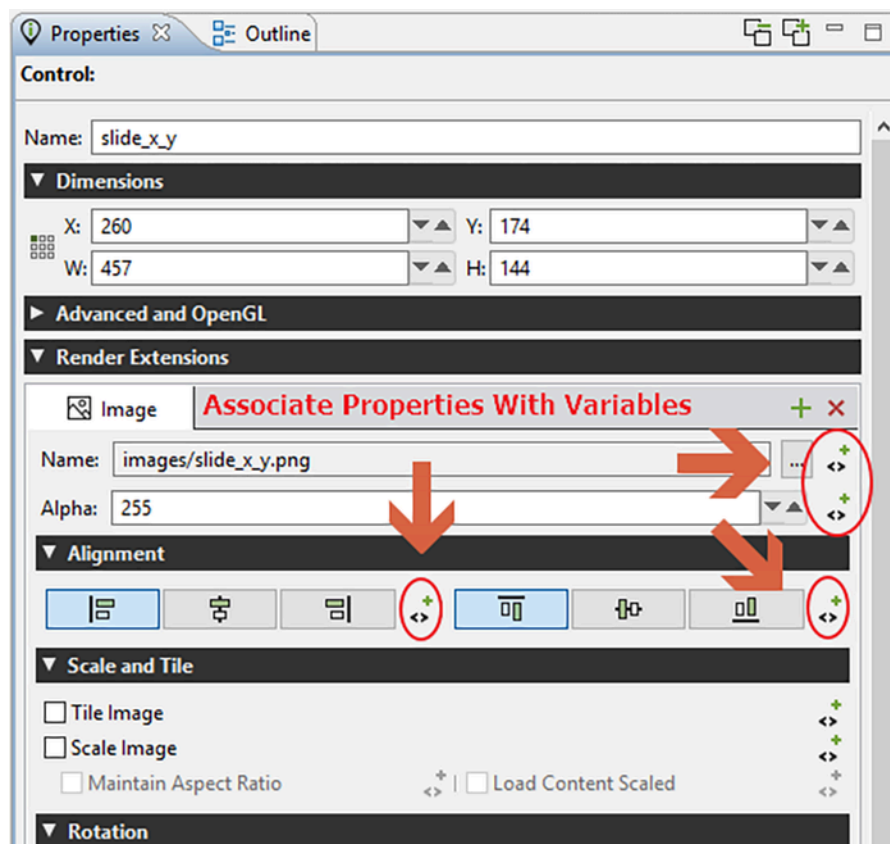


The following diagram illustrates the flow for an event which does not have positional information:



Chapter 9. Using Variables to Create a Dynamic UI

As outlined in the Storyboard Architecture chapter, Storyboard variables play an important role in making the presentation of a UI dynamic. Render extensions and actions have sets of properties, such as color, size, alignment, or name, that define their behavior or operation. By default, these properties contain fixed values that do not change from the values they are given at design time. This would nominally make for a static and fixed user interface and probably an unsatisfying user experiences. To give rendered content and executed actions dynamic behavior that can be modified when the application runs, these properties can be bound to variables.



Variables, like actions, are associated with a particular model element. The model element that they are associated with typically is an indication of the intended scope of use for that variable. For example if a variable is associated with a control, then it is likely only render extensions and actions local to that control that are going to be using that variable. If a variable is associated with the application or with a screen, then that implies a broader context of use for the variable. This is not a hard rule however since Storyboard allows references to variables using a fully qualified name as discussed in the Storyboard Architecture chapter.

The preferred way of creating a variable is to create it at the point where the variable is going to be used, for example associated with particular render extension's property. Within the Design environment, you will know if a property can be made dynamic and bound to a variable by the presence of a variable binding $\langle + \rangle$ indicator next to the property name in the Properties View. Pressing on this icon will automatically open up the Variable Selection dialog prompting you to select an existing variable from the application model. If the variable you want to associated is not listed and you want to create a new variable, then this

is easily accomplished from within this dialog by pressing on the New... button. This will then open the New Variable dialog allowing you the ability to create a new variable directly in place. The variable will be created and associated with the model element that you were looking at within the Variable Selection dialog and automatically selected for you.

The alternative way to create variables is to create them directly on the model objects that they will be associated with through the Variables View or the Application Model View using the Add > Variable functionality which will open up the New Variable dialog. This method is more direct, but more prone to error because you will be asked to provide a type association for the variable. When variables are selected, and created, through a particular property the type information is passed along and used as a filter so that the subsequent dialogs will only allow you to select or create variables that match the type of the property you want to bind to.

You can identify that a variable has been bound to a render extension or action property by the change in presentation of that property in the Properties View. Properties that have not been bounds will simply show the name of the property and it's current value. Properties that have been bound to a variable will have an additional annotation indicating the variable that has been associated in the form of `${...:variable_name}`. This annotation will show in green if the variable has been resolved in the Design environment and will show in red if the variable can not be resolved.

To change the variable binding to a new or different variable, then just repeat the process of clicking on the variable binding $\langle \rangle$ indicator to re-open the Variable Selection dialog. To remove a variable binding on a property you can right click on the variable binding $\langle \rangle$ indicator and then select Unbind. This will remove the binding but will not remove the variable definition. To remove the variable definition, delete the variable from the Application View or from the Variables View

Once a variable is created and associated with a property, any change in that variables value will cause the Storyboard Engine to automatically schedule a redraw operation of any elements that might be affected by the change. Variables can be bound to multiple properties at the same time. In fact this is a common Storyboard design pattern to create styles within the application for shared behavior that you want to treat in a uniform manner. An example of such a use would be the definition of application variables for font names and font sizes or a set of translation text values that should be used commonly throughout the application. During design, the value of the variable can be changed either directly when adjusting a property that has been bound to a variable by changing the value in the Properties View or by making a change directly in the Variables View. When the value of a variable changes in the Design environment the change will propagate through to all clients who are referencing that variable in a similar manner to what happens when the value of a variable is changed at runtime.

Variables are generally manipulated directly in one of two ways; via a DataChange Action or as part of a Lua callback function in a Lua Action. The DataChange Action's is a simple action that allows one or more variables to have their value set to a new value. The new value can be fixed or it can also be the value of another variable extracted at runtime. It is ideal for scenarios where there is no additional computation or logic required, simply a new value has to be assigned to a variable. If more sophistication is required, then the the section called "Storyboard Lua API" provides a comprehensive API for reading and writing variables defined as part of the Storyboard model and offers a rich set of conditional logic and IO facilities that can be used to guide the setting of variable values. The Animation Action provides a common third way of manipulating variables and that mechanism is covered in detail in the Chapter 10, *Creating and Applying Animations* chapter of this guide.

Table Variables

Tables variables are the same as any other Storyboard variable, but due to the potential number of such variables, they receive special treatment to facilitate their use. A table variable is different from a normal

variable only by the fact that they follow a special naming convention of name .<row>.<col> so that by using a single base name, many different rows and columns can be addressed.

New Variable

Variable Creation

Select a variable name and the type of variable to be defined.

Model Object: CRANK

Name: txt

Type: Text String

Value: Hello World!

☒ Create a table cell variable

Cell row,column entries:
Specify the table cells to which variables will be associated.

Indicate a range of cells using '-' and separate entries using ';'.

Table Cells: 1,1 - 5,1

Examples:
2,3 Selects a single cell at row 2, column 3
2,3; 3,3 Selects two cells: 2,3 3,3
1,1 - 2,2 Selects four cells: 1,1 1,2 2,1 2,2

NOTE ON DEFAULT VALUES:

All unassigned table cell variables in Designer will use a default value for display purposes.

This behaviour differs from the Storyboard Engine and Simulator where a variable is required to exist in order to be rendered.

Finish Cancel

Table variables are created using the same process that normal variables are created. Within the New Variable dialog there is a special check box Create a table cell variable. When this box is selected you will be able to provide information regarding the number of variables that you want to create and which row and columns they will be created for. Once these variables are created, they will be displayed in the Actions View as a tree item with the base name of the variable shown but then expandable to show the individual row and column variable entries with their name .<row>.<col> naming convention.

Table variables show up in the Variable Selection dialog the same way as normal variables do, but only as the basename and not as the individual row/column entries. This is done intentionally to reduce the clutter of the variable selection dialog. When a table variable is bound to a property and it is bounds in the context where the table cell information is available, then the label on the property it is associated should show an annotation that starts with `{cell: . . .}`. If this is not showing this special cell annotation but is showing the normal variable, either the variable was not created as a cell variable and must be re-created or the context of the selection operation did not include a table cell.

Once created, the values of the individual variables can be changed to reflect different representation for different cells. Similar to standard variables, these changes can be made at Design time from either the Properties View as changes to the value of a property or within the Variables View. If using the Properties View to make a change, the variable that will be changed will be the variable whose row and column is in the context of the selection.

Tables are generally filled with content dynamically at runtime, and resized accordingly. To this end the role of the table, and table variables, is to provide enough of a visual representation of the design to ensure that the representation is correct. For example if a table will only be displaying three visual rows of information, but may be dynamically filled to contain 100 rows, then there is no requirement to create a table variable with 100 entries. Create only as many table variables as you need to show the design and leave the remaining variables to be created dynamically at runtime. Doing this will result in a smaller and less cluttered design.

Triggering Events on Variable Changes

As discussed in the section called “Event Driven Interaction Model”, events can be generated from many different sources. The Storyboard Engine supports the creation of user events (see Chapter 8, *Connecting Events to Actions*) and these events can also be generated by the engine when the value of a variable changes. A typical example of where this type of correlated behavior might be applied is to synchronize a scrollbar style presentation with a section of a scrolling table or layer.

To bind an event to fire when a variable changes you add the event to the variable in the Variables View by selecting the variable you want to bind, right clicking and selecting Bind Event. This will open the standard event selection dialog and allow you to pick (or create) an event to fire when the variable changes.

Similar to screen redraw operations, a new event will not fire every time that a variable's value changes. Instead the first change will cause the event to be queued and until that event is serviced, no subsequent events will be generated. This behavior reflects the intended use of this event generation to allow an efficient coupling of display synchronization to data changes. This functionality should be used with care as overuse of this binding can result in a flood of events and excessive redraw operations.

When the event bound to a variable fires, it will be targeted at its parent model object. This means, for example, if you have a variable associated with a control that the event will be delivered first to that control and then fall down until it hits the active screen and application. This means that actions on objects that are not visible and part of the active screen may fire.

Chapter 10. Creating and Applying Animations

Animation Action

The Engine supports user defined animations using the animation action, `gra.animate`. This action starts executing an animation immediately, monitors the animation, and applies the specified changes as they have been defined by the user in Designer.

An animation is a named block of operations that will perform changes on Storyboard data values at a pre-determined frame rate. The individual data changes that occur within an animation are referred to as animation steps.

An animation step contains the following information:

key	The key is a reference to the data object that is going to be changed over the course of the animation. In general, keys are numeric items such as x or y position, width, height or transparency (alpha) values. However, it is possible with 0 duration animation steps to apply a change to any variable at a point in the animation. This includes text or images.
offset	This is the time in milliseconds from the time that the animation was started that this particular change will start to occur.
duration	This is the time in milliseconds over which the change will occur. This value may be 0 for changes that are not numeric (i.e. text or image values) or if the animation step is defined to occur at the start or end of the animation block.
rate	For non-zero duration animation steps, this is the change curve that will be applied to the numeric value from its start value to the end value. Example rates include linear, ease in (easein), ease out (easeout), ease in out (easeinout) or bounce.
starting value	This represents the starting value of the animation. The starting value can be either a specific value or variable reference, or it can be specified as the current value of the animation key at the time that the animation starts. Using the current value is good for animations that need to work generically to achieve some end value.
end value	This represents the end value of the animation. The end value can be either a specific value or variable reference or it can be specified as an offset from the starting value rather than as an absolute value. Using an offset (or delta) in an animation makes it easy to perform incremental animations on objects.

Animation steps are all synchronized within an animation block so that their data changes will occur in a synchronized manner. While it is possible to specify arbitrary time offsets and durations, these values will be mapped onto the nearest synchronized frame slot. The frame slots are dictated by the frame rate of the animation block.

Animation instances can be labelled with a string `id` which is an identifier used to provide exclusive execution. It is possible for many animations to run concurrently, however if two animations have the same `id` value, then only the last one invoked will actually run. For example, if you have an animation to shrink and grow a control, then you only want one of either the shrink or grow operations to occur at one time. This can be achieved by having the shrink and grow animation actions share the same identifier.

Animations may be stopped at any time by using the animation stop action, `gra.animate.stop`.

Timer Keyframe Animations

Animations can also be created in the more traditional method of setting a timer and operating on data on every timer firing. By manipulating data in the timer callback, clients can cause any number of custom behaviors to occur, as the data changes on variables will automatically be reflected through to the user interface as would be done at any other time.

The timer callback allows non-traditional data change rates to be applied, as well as flip-book style animations where a sequence of images is pre-defined and changed on each timer iteration. This could be achieved through a series of 0 duration animations as of Storyboard 3.0.

Screen Transition Animations

Animations are frequently used during screen transitions. A screen transition is a way to move from the visible screen to a new screen which may or may not have common layers. By default, screen transitions can be invoked by using one of the following actions:

<code>gra.screen</code>	Transition to a new screen immediately
<code>gra.screen.fade</code>	Fade the new screen into the current screen over time
<code>gra.screen.path</code>	Slide the new screen in and the old screen out over time, from one of the following directions <ul style="list-style-type: none">• Left• Right• Top• Bottom
<code>gra.screen.scale</code>	Grow the new screen over the current screen

All transition actions, which are time-based, take similar arguments that control the duration of the transition, the rate at which the transitions will occur, the orientation of the transition, and the number of frames that should be used. Using these arguments, the designer can control the user experience (e.g., duration and effects) as well as the overhead incurred on the system (frequency of frame updates).

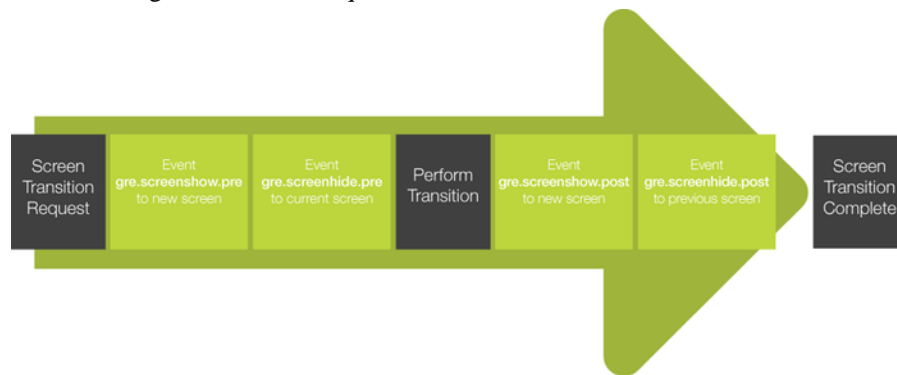
During a screen transition, four events will be generated to notify the system of the current state. These events are:

<code>gre.screenshow.pre</code>	This event is generated for the new screen being shown. The event will be generated before the transition starts. This event gives the user a chance to change data via the <code>gra.datachange</code> action or Lua before their transition content is updated.
<code>gre.screenhide.pre</code>	This event is generated for the previous screen being hidden. The event will be generated before the transition starts.
<code>gre.screenshow.post</code>	This event is generated for the new screen being shown. The event will be generated after the transition has completed.

`gre.screenhide.post`

This event is generated for the previous screen being hidden. The event will be generated after the transition has completed.

The following illustrates the sequence of events:



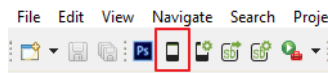
The transitions are written such that if graphics hardware layer support is available, then these layers, assuming they are available for use, will be leveraged to lower the processing overhead for the system during the transition period. Experience has demonstrated that it is possible to achieve smooth transitions at almost no CPU cost when the hardware capabilities can be properly leveraged.

Chapter 11. Simulating your Application

After an application has been created, it is a good idea to run it through the simulator to validate the runtime behavior before exporting the application to a Storyboard Embedded Engine deployment file. The simulator is a host-based instance of the Storyboard Embedded Engine and should exhibit the same operational behavior as the target, though there may be differences in the level of performance obtained because of the different CPU and graphic characteristics.

The simulation of a Storyboard application can be invoked from a number of different locations within the Designer environment.

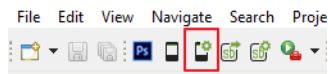
- From the main menu Run > Simulate Storyboard Project
- From the main toolbar using the Simulate Storyboard Project

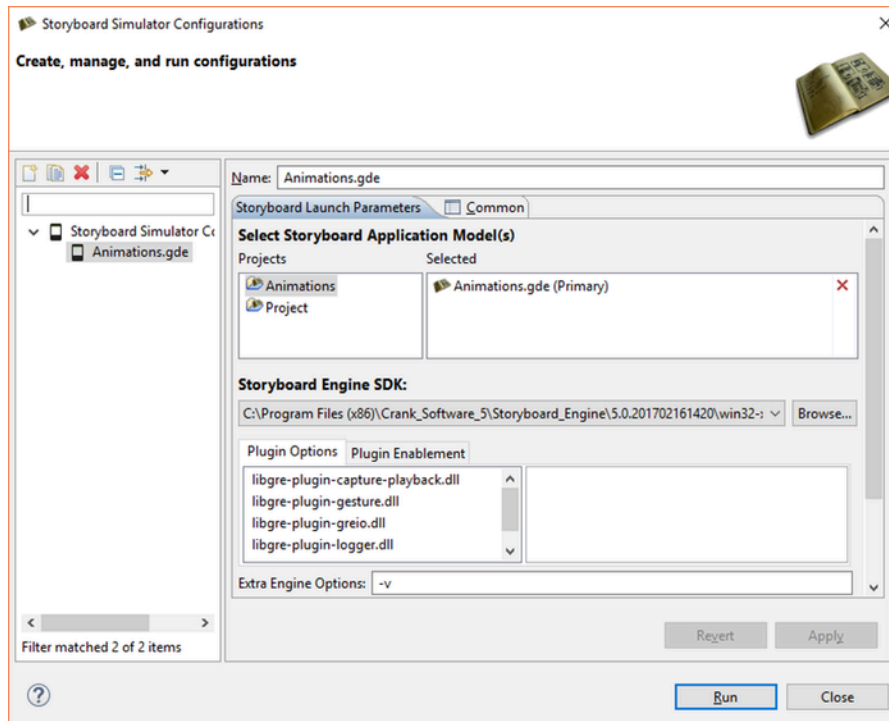


- By right clicking on an application model file (gde) and selecting Storyboard Simulator
- By right clicking within the Storyboard editor and selecting Simulate...
- By using the key command ALT/CMD+L within the Storyboard editor

If no simulator launch configuration exists, then this will create a default configuration for you and launch the simulator with your application.

If you need to change the simulator launch properties, or you would like to create a new configuration then you will want to use the Storyboard Simulator Configurations option instead.





Within the Storyboard Simulator Configurations dialog you have the ability to create new simulation launch configurations or change the properties and settings of an existing simulation launch configuration.

- Depending on your host platform, you may be able to select a different version of the Storyboard Engine to execute by selecting from the Storyboard Engine SDK drop down. For example you may want to select an OpenGL hardware accelerated renderer or a software renderer depending on which simulator better matches your target environment.
- Common plugin options are exposed within this dialog for easy access. These options translate directly to command line options described in Appendix C, *Storyboard Engine and Plugin Options*.
- The Extra Engine Options text entry allows you to pass options directly to the sbengine simulator. In addition to the `verbosity (-vvv)` option you can pass any of the command line options described in Chapter 25, *Exporting and Running on your Embedded Target* and Appendix C, *Storyboard Engine and Plugin Options*.

The Storyboard Engine Command Line area shows the final assembled command line that will be passed to the sbengine that has been selected as the simulator.

It is possible to have multiple different simulator launch configurations for a single project that perform different types of execution. For example you may choose to create a second launch configuration that enables the Lua debugger option or perhaps you have different launch configurations that allow you to perform and event capture and then another for event playback. When you launch the Storyboard Simulator, if there are multiple configurations available a dialog will prompt you to select the configuration that you want to run, allowing quick selection between different pre-configured options

In the event of multiple application project files within a project the Select Storyboard Application Model(s) presentation will change allowing you to select one or more application files to launch. If you are working with multiple Storyboard application models, then you should continue reading Chapter 22, *Multi-File Application Development* for more details around multi-file simulations and exports.

Chapter 12. Scripting with Lua

The Storyboard Lua API (Lua API) gives developers access to the Engine through a Lua scripting interface. This API is a library of functions which allow interaction with the Engine by manipulating data and working with events and user interface components. Through the Storyboard Lua API developers can:

- Get and set data values from the model
- Inject application events
- Manipulate model objects such as controls/layers

The Storyboard Lua plugin is built on top of the standard 5.1 release of Lua available from www.lua.org [<http://www.lua.org/>]. While the core Lua interpreter is unchanged from the standard release, two additional modules have been incorporated to facilitate development with Storyboard:

The bitwise manipulation module (bit32) from Lua 5.2 has been built-in to this Lua plugin. This module provides a native implementation of several standard bit operations, including those required for text conversion to/from UTF-8. The documentation for the bitwise functions available from this module can be found in the Lua 5.2 Reference Manual [<http://www.lua.org/manual/5.2/manual.html#6.7>]

The Storyboard module (gre) is included that provides function extensions to manipulate and work with the currently active Storyboard model. This module also incorporates the Storyboard IO communication API that can be used to send events to external programs.

Lua Action Callback Function

When a Lua callback function is invoked by the Lua action it will be invoked with a single parameter as in the following prototype:

```
script_function_name(table mapargs)
```

The argument *mapargs*, is a Lua table whose keys provide the context in which the action is being invoked along with any action specific argument and parameters. This context includes the application, screen and control the action was associated with, the currently focused control, any arguments provided to the action as well as all of the event data that cause the action to fire.

The following keys are always available inside the context's table:

context_app	The application context of the current action
context_screen	The screen context of the current action (the current screen)
context_layer	The layer context of the current action (the current layer)
context_group	The group context of the current action (the current group)
context_control	The control context of the current action (the current control)
context_row	If the context_control is a table then this is the row index of the current cell
context_col	If the context_control is a table then this is the column index of the current cell

active_context	The fully qualified name of the model object that invoked the action. If that object is the application, this will be the empty string.
context_event	The name of the event the triggered the action
context_event_data	A pointer to a Lua table containing any event data. The event data is different for each event and is defined in the event definition.

A Lua type called 'context' has been defined inside Storyboard's custom Lua module 'gre' to represent the mapargs object for the purpose of auto completion inside the editor. However since Lua is untyped by default, a Doxygen comment describing the mapargs parameter must be added to a function in order to get auto completion on it, as follows:

```
--- @param gre#context mapargs
function CBMyFunc( mapargs )
    -- I now have auto completion on mapargs
end
```

Functions created by Storyboard will automatically be prepended with this comment.

Example of using context data:

```
--- @param gre#context mapargs
function CBGetContext( mapargs )
    print("Triggered by event : " .. mapargs.context_event)
    print("Event was targeting : " .. mapargs.active_context)
end
```

Passing Extra Parameters to Functions

Lua actions are identified using an action type of Lua and setting the specific Lua function and extra parameters (if required) in the action arguments. Any extra parameters will be transferred directly to the Lua function through first argument (a Lua table) and the data can be accessed by using the parameter name as the table index.

```
--- @param gre#context mapargs
function CBUserParameter( mapargs )
    local p = mapargs.paramter1
    print("my_lua_func was passed : ".. tostring(p))
end
```

Lua Execution Environment

The Storyboard Engine Lua plugin provides a slightly different execution environment when compared to normal Lua script execution.

Normally a single Lua script serves as the starting point of script execution and all other scripts would be included using the Lua `require()` declaration. The Storyboard Lua plugin provides a slightly different

loading behavior in that it will pre-load all of the Lua scripts contained in the **scripts** directory at engine initialization time. The load ordering can be controlled by using the `require` statement to explicitly order dependencies. Since the `require` mechanism is used to perform the loading, any project files that use the same names as built-in Lua modules (i.e. `table.lua`, `string.lua` or `io.lua`) will generate a load time warning indicating the potential load time resource collision.

A side effect of this early module loading and execution is that any Lua script that is located outside of function blocks will have the opportunity to run before the application is fully initialized. This can be used to seed early execution environments or load preferences before the UI is in place and ready to render. Alternatively, this early initialization is possible by binding a callback to the `gre.init` event.

In addition to loading all of the script files in the **scripts** directory, the Lua plugin modifies the **package.path** variable and `;;` default search path to automatically search the scripts directory.

A convenience variable, **gre.SCRIPT_ROOT** is pushed into the execution environment that contains the path from the current working directory to the scripts directory. This variable can be used to locate additional resource files or to include extra script directories in a manner that is relative to the overall deployment bundle. Each separate path is delineated by a single `'`.

```
print("Script base directory: " .. tostring(gre.SCRIPT_ROOT))
-- Look for additional module files in the scripts/modules directory.
-- This will search the added directory first
package.path = gre.SCRIPT_ROOT .. "/modules/?.lua;" .. package.path
```

Asynchronous Lua Support

The asynchronous Lua support is provided in two fashions:

Lua Action: To create an independent Lua thread in response to an action, the user can add an 'async' parameter to any existing Lua script and it will automatically create and run that action outside of the main UI thread.

Lua Script There is a new Lua API call `gre.thread_create()` that takes a single parameter which is a function to execute. This function will be executed and scheduled to run in an independent thread of execution.

Threads are created using the system's underlying native operating system thread support. Operations are synchronized explicitly through locks in the Lua VM, however there is no explicit support for data synchronization (i.e. mutexes, condition variables)

The suggested communication pattern for inter-thread communication is to use Storyboard IO to inject event data into the system. This is similar to the idea behind `LuaLanes` or `WebWorkers` where inter-task communication is based on message passing.

Threads will be hard-terminated at exit. Clients should establish their own protocol where a soft shutdown is required to trigger any asynchronous threads to terminate.

Lua Debugger

The Storyboard Lua Debugger enables the developer to monitor the flow of execution of the Lua scripts used by the Storyboard application. Using the debugger it is possible to step line by line through a Lua script while examining the variable values that are being used by the Lua functions.

Note

The Lua debugger is configured such that it can only be used with the simulator runtimes on the host platforms that support Storyboard Designer. For assistance in configuring the debugger for embedded targets, contact Crank Software support (support@cranksoftware.com).

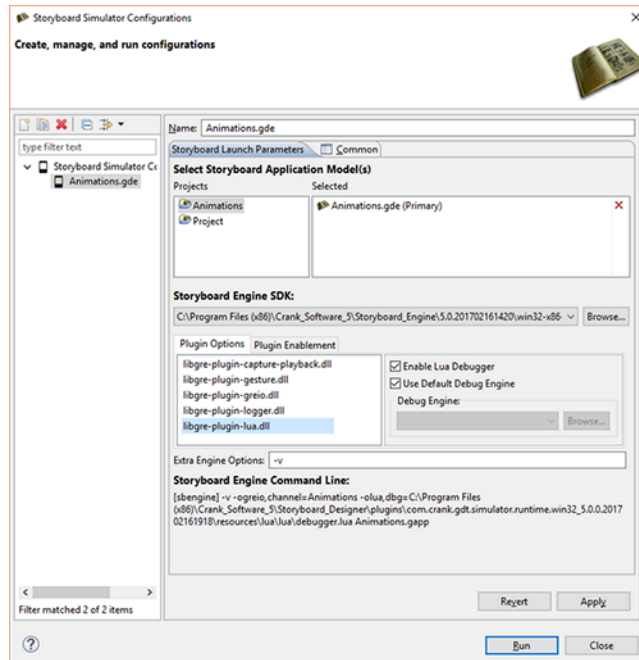
The Designer debugging environment communicates with the application's Lua script plugin using network sockets in a client/server model. The Storyboard application acts as the client and is controlled by the Designer debug environment which acts as the server.

Creating and launching the debug server is an automated process. To configure Lua debugging it suffices to take the following steps:

1. Create a Storyboard application launch configuration. This topic is discussed in more detail in the chapter Chapter 11, *Simulating your Application*.
2. Enable debugging in the Storyboard Lua plugin
3. Launch the Storyboard application

The first two steps are part of a one-time configuration required to set-up the simulator launch configuration. After the initial set-up, only the last step needs to be performed in order to launch an application with the Lua debugger running.

When enabling the Lua debugger you may find it convenient to create a second launch configuration for the same project with debug enabled. This will allow you to quickly switch from a development/debug simulation to a standard simulation. To enable the Lua debugger simply check the Enable Lua Debugger option associated with the Lua plugin options.

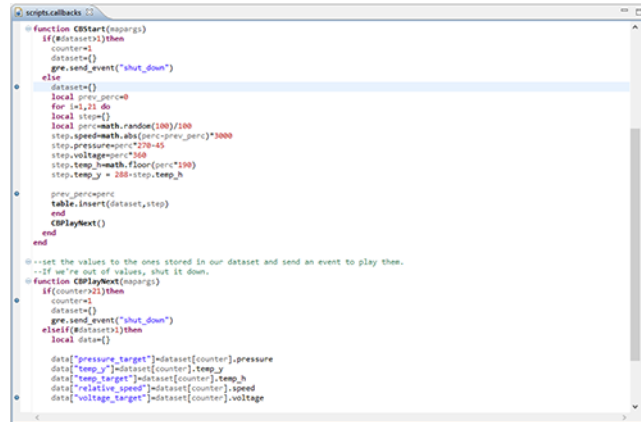


Once the Lua debugger is enabled, run the Storyboard application the same as you would a normal simulation.

This will launch the Storyboard application and at the same time initialize the debug client within the Storyboard application and the server running within the Designer environment. The two will connect automatically over a network socket and your application should immediately begin running. To confirm

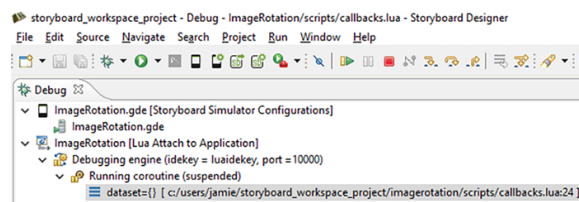
executed. Declaration breakpoints may not resolve to an execution stop point in the script.

Breakpoints can also be enabled and disabled and removed by selecting them from the listing in the Breakpoints view and performing the appropriate operation. It is also possible to navigate to directly to the script source file from within the Breakpoints view by right clicking and selecting Go To File.



Stepping, Continuing and Terminating

The Debug view provides a list of active debug sessions and the execution stack trace when a session is at a breakpoint. Once a breakpoint is hit, then it is possible to single step to the next line of code, to continue the execution until the next breakpoint is encountered or to terminate the application using the view's toolbar commands.



Lua Executables

Storyboard includes a few stand-alone Lua executables packaged for the convenience of customers who are working with Storyboard in environments where it is possible to use shell programs to interact with the environment, most notably Windows, Linux, Mac and QNX environments.

- sblua This is the standard Lua command line interpreter `lua` that has been linked against the Storyboard Lua shared library `libsblua.so`. All enhanced Storyboard functionality that is not associated with an Storyboard application can be accessed using this interpreter. This can be specifically useful for creating test and simulation scripts that generate Storyboard IO calls using the `gre.send_event` API.
- sbluac This is the standard Lua command line compiler `luac` that has been linked against the shared library `libsblua.so`. This utility can be used to pre-compile Lua script source files into platform independent byte code for faster load times. This utility is invoked automatically during a Storyboard Export if the Generate precompiled Lua selection is made in the export configuration.

The bytecode is platform independent and can be used the in the same way as non-bytecode Lua files by the Storyboard Engine at runtime. Note that currently, our Storyboard engine only looks for .lua extensions, so if you would like to use bytecode files, make sure to give them a .lua extension. Please be careful not to overwrite your Lua script files, because they cannot be retrieved from the bytecode files.

To see the usage parameters just type 'sbluac' on the command line with no other arguments.

```
usage: sbluac [options] [filenames].
Available options are:
  -          process stdin
  -l         list
  -o name    output to file 'name' (default is "luac.out")
  -p         parse only
  -s         strip debug information
  -v         show version information
  --        stop handling options
```

A basic compilation to produce a bytecode file with the name luac.out, containing debugging information, would look like this:

```
sbluac input.lua
```

To name the output bytecode file something other than luac.out, use this:

```
sbluac -o output.luac input.lua
```

To strip out debugging information, use this:

```
sbluac -s -o output.luac input.lua
```

Chapter 13. Working with C Callbacks

What are C Callbacks?

On resource constrained systems, you can reduce your memory footprint by using C instead of Lua. Typically, the systems that require this type of configuration are running with a virtual filesystem (VFS) and are required to use the Storyboard Embedded Resource Header (C/C++) functionality of Storyboard. A `gra.ccallback` action is used in place of `gra.lua` action. In order to support faster development cycles, it is possible to use the C callback plugin in dynamic environments as well. The C callback plugin needs to be pointed to the (dll/so) with the option `-ocallback,path=[path to lib]`. In the non VFS case, if the C callback action cannot find the requested function, it will fallback to Lua, to try and find a function with a matching name, this is to allow developers to use Lua to implement functionality quickly, then migrate the functionality into C once the desired behavior has been achieved.

C Callbacks on Windows

On Windows the C callback dll must be compiled with the `/MD` option to ensure that memory can be safely passed from Storyboard Engine to your dll and back. Use `__declspec(dllexport)` to ensure that `sb_ccallbacks` can be accessed from the dll. Usually a macro is used for this in order to support building for other targets, as seen in the example below.

Example C Callback

Here's an example of how we could take the `ClusterIO` sample application and replace the Lua functionality with C. The action that is triggered by the `cluster_update` event should be changed from a `gra.lua` to a `gra.ccallback` action, it should call the same function as before, `CBUpdateEvent`. This library has a dependency on `libgre`.

ClusterIO_ccallbacks.h

```
#include <stddef.h>
#include <gre/gre.h>

void cbClusterUpdate(gr_action_context_t *action_context);

DLLEXP const sb_ccallback_t sb_ccallbacks[] = {
    { "CBUpdateEvent", &cbClusterUpdate },
    { NULL, NULL },
};
```

ClusterIO_ccallbacks.c

```
#include <stdio.h>
#include <string.h>
#include "ClusterIO_ccallbacks.h"

#include <ClusterIO_events.h>

#if defined(GRE_FEATURE_VFS_RESOURCES)
```

```
#include <gre/sdk/sbresource_vfs.h>
#endif

void
cbClusterUpdate(gr_action_context_t *action_context) {
    gr_application_t *app;
    cluster_update_event_t *cluster_data;
    gr_data_union_t data;
    void *event_data;
    int nbytes;

    app = gr_context_get_application(action_context);
    if(app == NULL) {
        return;
    }

    event_data = gr_context_get_event_data(action_context, &nbytes);
    if (nbytes != sizeof(cluster_update_event_t)) {
        return;
    }

    cluster_data = (cluster_update_event_t *)event_data;

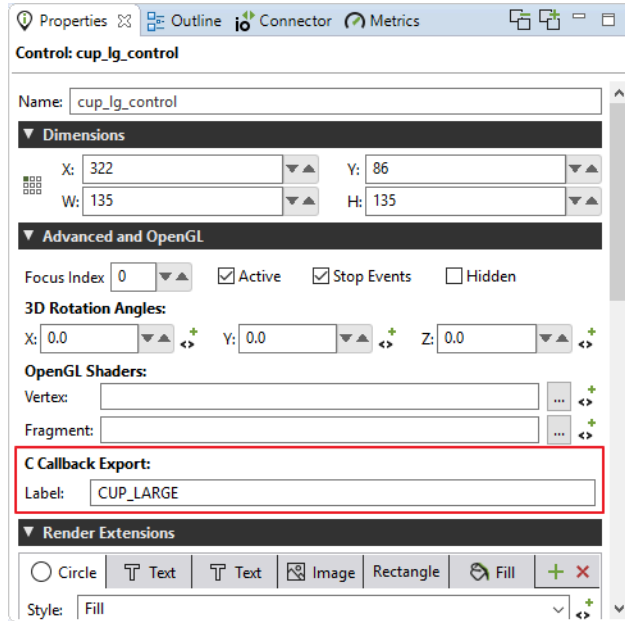
    memset(&data, 0, sizeof(data));
    data.i16 = cluster_data->speed;
    gr_application_set_data(
        app,
        "speedometer_content.speed.text",
        GR_DATA_FORMAT_2s1,
        &data
    );

    data.f32 = ((float)cluster_data->speed * (214.0f / 200.0f)) - 107;
    gr_application_set_data(
        app,
        "speedometer.pointer_speedometer.rot",
        GR_DATA_FORMAT_4f1,
        &data);

    data.f32 = ((float)cluster_data->rpm / 10000) * 49;
    gr_application_set_data(
        app,
        "tach_exterior.pointer_tach_exterior.rot",
        GR_DATA_FORMAT_4f1,
        &data);
}
```

C Callback Export Labels

You can specify labels to be exported to your C callback header file as defines. In the properties view, under Advanced, you will find the section called C Callback Export, where the label can be specified. This field exists for Screens, Layer Instances, Groups, Tables and Controls.



In the above example, the following would be exported into the .h file.

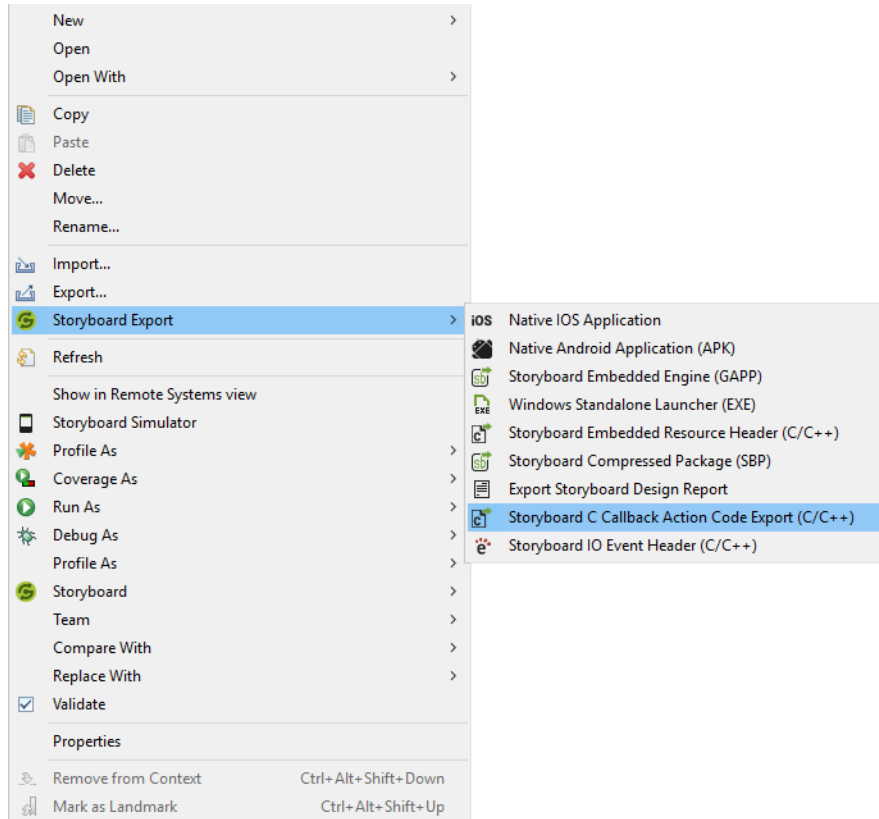
```
#define CUP_LARGE "size_layer.cup_lg_control"
```

Exporting C Callbacks

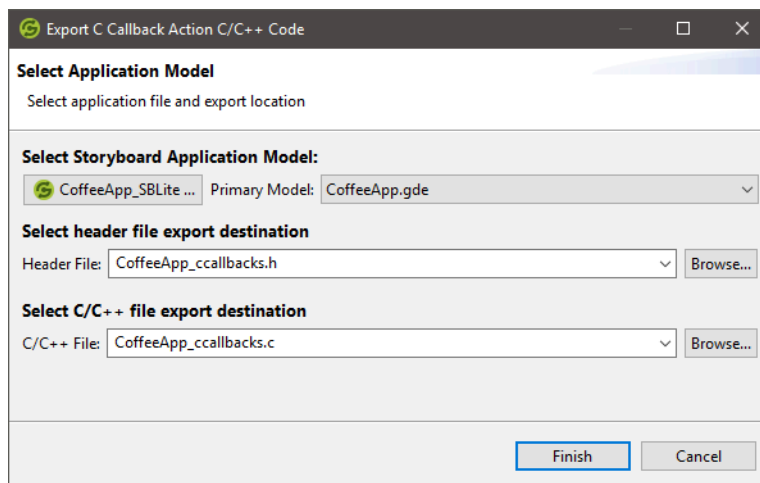
Storyboard will export stubs for C Callbacks in the form of a source file(.c) and header file(.h). There are a couple different ways to generate these files. Through the C Callback Export Wizard, through the Model Resource Header Export and from a command line application.

C Callback Export Wizard

The C Callback Export Wizard can be opened by right-clicking a .gde file in the Navigator View, Storyboard Export > Storyboard C Callback Action Code Export (C/C++).



The C Callback export wizard allows you to select the .gde file, and choose the output path for the c/c++ and header files.



The exported .h file will contain all the C callback export labels given to model objects in your project as defines. The exported .c file will contain function stubs for all the C callback functions used in your application. Keep in mind that this export will overwrite anything you may already have in your file, so you should not point the exporter to your working copy.

C Callback Command Line Export

All of the Storyboard executables will be located in the `PATH_TO_INSTALL/Storyboard_Designer/storyboard` directories, though on each desktop platform they are named slightly differently

Windows:

```
....\Storyboard.exe -application com.crank.gdt.ui.ccallback.application.ccallbackexport  
model=<PathToGDEFfile> c=<PathToCFile> h=<PathToHFile>
```

Mac:

```
....\Storyboard.app/Contents/MacOS/Storyboard -application  
com.crank.gdt.ui.ccallback.application.ccallbackexport  
model=<PathToGDEFfile> c=<PathToCFile> h=<PathToHFile>
```

Linux:

```
....\Storyboard -application com.crank.gdt.ui.ccallback.application.ccallbackexport  
model=<PathToGDEFfile> c=<PathToCFile> h=<PathToHFile>
```

Note: Storyboard requires a display, so to run in a true headless environment a Virtual frame buffer needs to be setup.

```
Xvfb :1 -ac -screen 0 1024x768x8 export DISPLAY=:1  
....\storyboard -application com.crank.gdt.ui.ccallback.application.ccallbackexport  
model=<PathToGDEFfile> c=<PathToCFile> h=<PathToHFile>
```

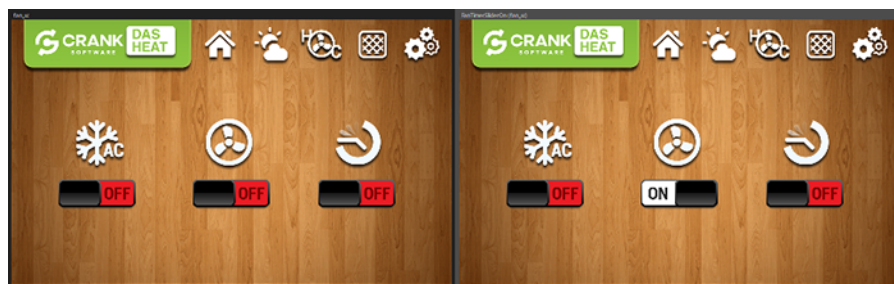
Where the `model` is the path to the Storyboard Designer model file. This model file will be used to search for a export configuration. The `configuration` parameter specifies the configuration name. If this is provided then only the configuration with that name will be used in the export.

Chapter 14. Working with Design States

What are Design States?

Design States are a visual representation of screen which is in a different state, because data has changed. A Design State may represent a screen after running an animation or a set of data changes. Design States allow users to communicate how a screen might change during the normal execution of your application.

In the screenshot below, we have the default Screen on the left and the Design State on the right. This Design State represents what the screen will look like if the user selects the fan toggle button.



Design States enable the user to work with content that may not be visible by default. In the example below, the Design State *5DayState* has a single argument which is the *show_5_day* animation, this is a representation of how the screen will look when a user presses the "5 DAY FORECAST" button.

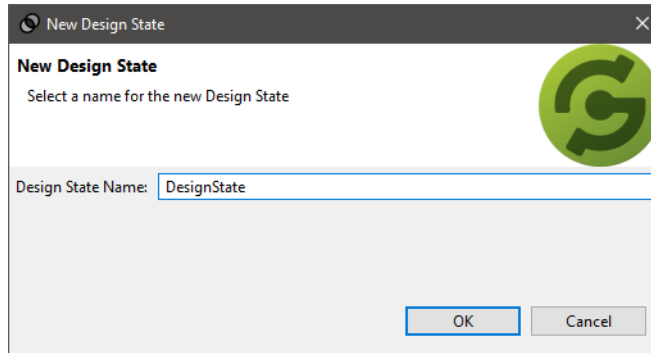


Model elements cannot be added/removed from Design States, these operations are always global. Removing a control, will remove the control from the application. Design States are limited to making changes that can also be made at runtime, this means setting internal model object attributes (x, y, width, height, hidden) and changing Storyboard variables.

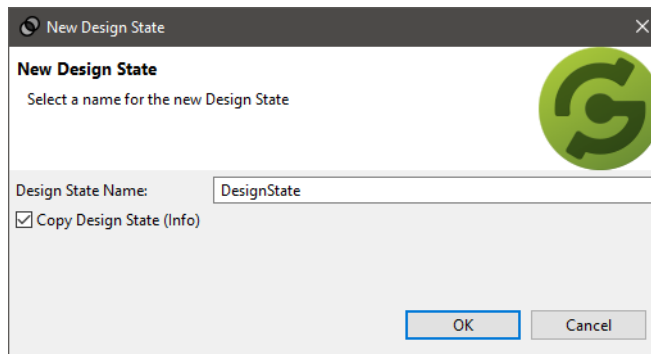
Creating a Design State

There are several ways to create new Design States.

- Add > New > Design State



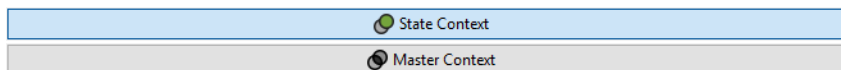
- Keyboard shortcut CTRL+N+D (Windows/Linux), COMMAND+N+D (Mac)
- When we right-click on an existing Design State and choose Add > New > Design State, you will be given the option to create a Design State from the current Design State, this is effectively going to copy the current Design State.



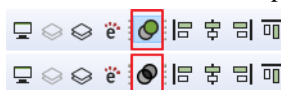
Editing a Design State

Master/State Context

Once a Design State is added to the application, a new button will appear at the bottom of the Storyboard Editor. This is the Master/State context selector. This button is responsible for determining what information is shown in the Properties View and Variables View, as well as how changes will be applied to the model.



This button also shows up in the Storyboard workbench toolbar.



Master Context and **State Context** can be thought of as Global and Local contexts.

- When in **Master Context**, changes made in the Storyboard Editor, Properties View or Variables View will affect the global state, the regular Screen and all Design States that do not specifically override a value will inherit the change.
- When in **State Context**, changes made in the Storyboard Editor, Properties View or Variables View will affect only the currently selected Design State. The exception to this is changes made to elements

on the regular Screen will still be propagated to all Design States that do not specifically override the value, since changes to the regular Screen will always be global.

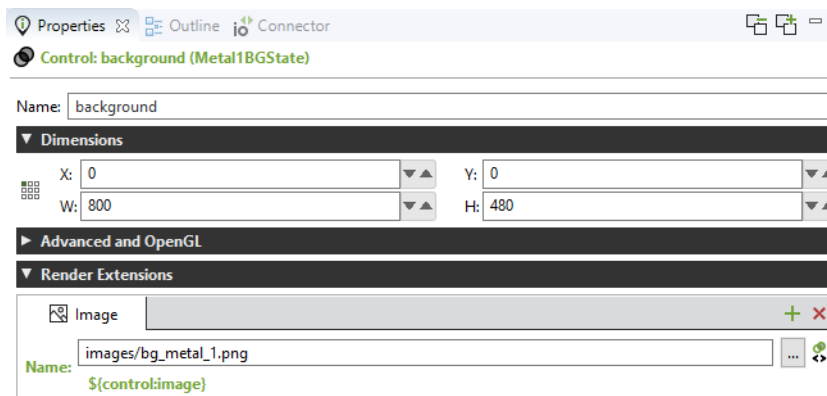
Storyboard Editor

Changes made to model objects in a Design State will automatically add the changes to the Design State. Changes to internal model object attributes such as x, y, width, height and hidden can be made directly through drag and drop operations as is typical when working with the Storyboard editor. The primary difference with the introduction of Design States is depending on your current **Context Mode** the changes will either be applied globally or locally.

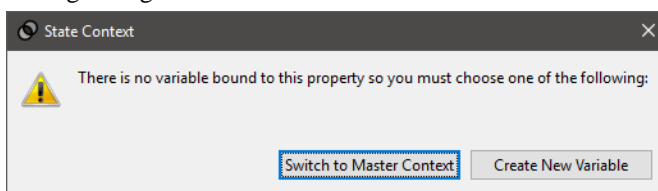
Keep in mind that additions and deletions are always global operations.

Properties View

When in **State Context** the title at the top of the Properties View will be green and marked with a Design State icon. All properties that are changed in a Design State will be highlighted green and styled in bold text.



When in **State Context**, changing a property that does not have a variable bound to it will open the following dialog:

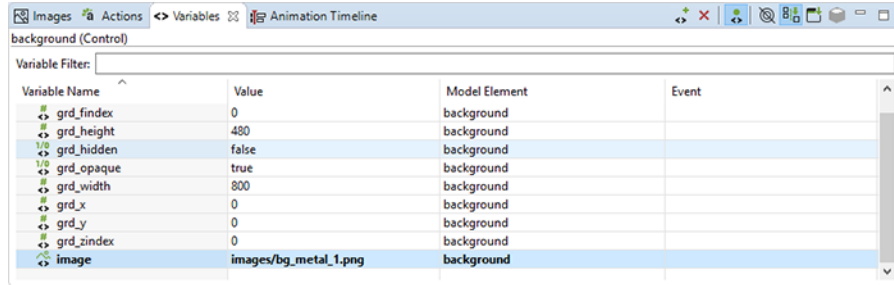


The purpose for this dialog is that Design States can only change properties that can be changed at runtime, in the case of render extensions, only properties with variables bound may be changed at runtime.

- If the change is intended to be a global change, then the user will select Switch to Master Context.
- If the user actually wants to remain in **State Context** and change the specific Design State, then creating a new variable will allow changes to be made specifically to the selected model object in the context of the current Design State.

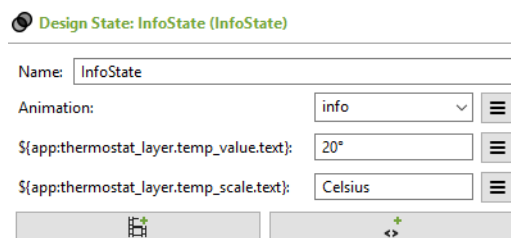
Variables View

When in **State Context**, variables that are changed in a Design State will be styled in bold in the Variables View.




Manually Editing Design States

When we select a Design State from the Application View or from the Storyboard editor, the Properties View will be populated with the Design State and its arguments.



Animation Arguments



A Design State can be made up of several arguments, any number of which may be animation arguments. Just like animations at runtime, Design State animation arguments can specify a context in which to run. To specify a context, choose the  button of an animation argument and select Choose Animation Context, animation keys which are context specific (i.e. \${control:...} or \${layer:...}) will be resolved with the specified context.

Variable Change Argument

Variable changes are simple key-value assignments. The most appropriate editor for the specified variable will be shown in the properties view such as an RGB editor for color variables or an image selector for image variables. At this time, context based keys are not supported so all keys must be fully qualified paths to variables.


Removing Changes From Design States

There are several ways to remove a change from a Design State.

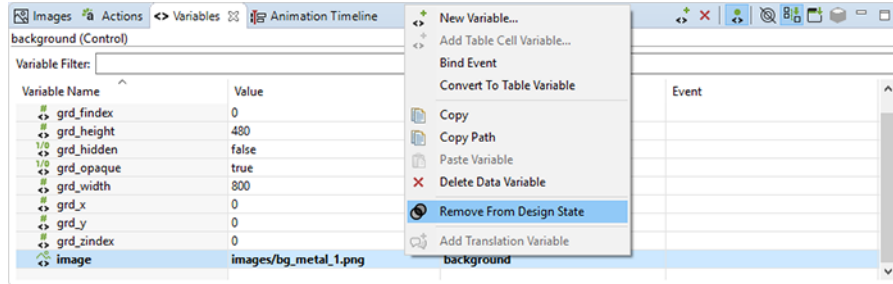
- With a Design State selected, in the Properties View, click the menu button  beside one of the Design State arguments and select  Remove Animation or Remove Variable Change.
- From the Properties View in **State Context**, right-click on a Design State icon on a Design State property (bold label, highlighted in green) and select Remove From Design State.
- For internal variables:



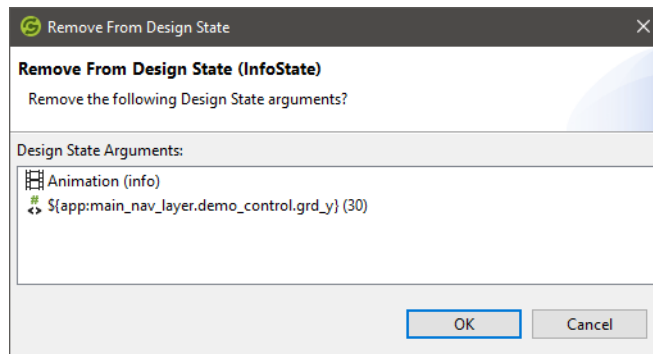
- For user variables:

Name: 
 \${control:image}

- From the Variables View in **State Context**, right-click on a Design State variable (bold label) and select **Remove From Design State**.



When removing a variable from a Design State, the following dialog will appear, listing all the arguments that will be removed from the Design State.



Hiding/Showing Design States

All Design States

When not actively working with Design States, they can all be hidden instantly. Toggling Hide All Design States will override the visibility of Design States so that none are visible. Toggling Show All Design States will restore them to their previous visibility, those Design States that were individually hidden will remain hidden.

- CTRL+D (Windows/Linux), COMMAND+D (Mac)
- View > Hide/Show All Design States
- In the Storyboard editor, right-click and select View > Hide/Show All Design States

Individual States

Individual Design States can be hidden/shown in the Application Model View. You may want to hide all the Design States that are not related to the current task you are working on. Visible Design States will have an effect on the performance of the GDE Editor. By hiding Design States, the Storyboard editor has less drawing to do, so it will be inherently quicker.

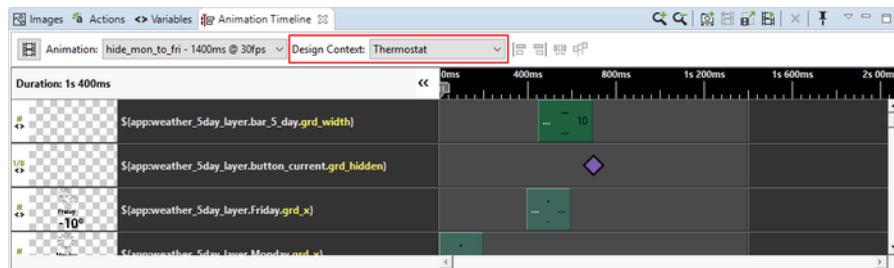
Converting a Design State

Design States can be converted into either an animation or into a named data change. To convert a Design State, right click on a Design State in the Application Model View, under Convert To... there are three options:

1. Animation an animation will be created with all steps having a duration of 500 ms for numeric data or 0ms in the case of strings or booleans.
2. Data Change a named data change will be created. Animations are 'flattened' such that only the final values will be included in the data change.
3. Design State when converting a Design State into a Design State, all animations arguments will be 'flattened' and included in the new Design State as a series of variable change arguments.

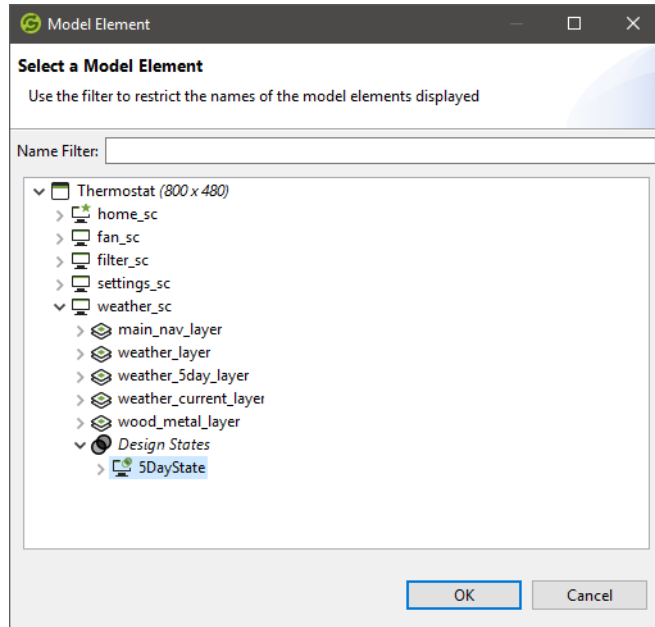
Animation Preview with Design States

One useful feature of Design States is the ability to create a custom starting point for the Animation Preview tool. Some animations have a forward and a reverse animation. With Design States, the Animation Preview tool can be instructed to begin the preview from a particular Design State as the starting point. In order to specify which context to use for the Animation Preview, there is a Design Context field, shown below.



Opening the Design Context dropdown and selecting More Model Elements will open the following dialog, which is used to select the specific context in which to run the animation. The selected model object may be in a Design State, or may be the Design State itself.

The current animation is *hide_mon_to_fri* which is the reversed animation of *show_5_day*. In this sample, we have a Design State that represents the *show_5_day* animation, *5DayState*, by selecting it as the starting point for the Animation Preview, we can preview *hide_mon_to_fri* starting from the end of the *show_5_day* animation.



Chapter 15. Working with Images

Custom imagery enhances the presentation of a rich user interface. A significant portion of a Storyboard application's user interface objects are made up of custom visual elements. The Designer environment works hard to ensure that the management and use of those images is as flexible as possible.

Images are applied to a UI design using the image render extension. The image render extension uses its name property to identify an image file (png, jpg, bmp among others) that is to be rendered when the control is redrawn. The name of this image file is generally relative to the project root and by convention is located in the `images` directory of the project. Like all properties, the name field can be bound to a variable in order to make it dynamic and changeable at runtime. The Storyboard Samples > Animation > ImageRotation and Thermostat (among many others) demonstrate how images and image properties can be manipulated to provide different visual effects.

Any image content that is directly in the `images` directory of a project will be loaded into image selection dialogs and made readily available to the user. Since the `images` directory is part of the filesystem, it is relatively straightforward to add content into it, however Storyboard also provides some import facilities directly within the Designer environment:

- Directly placing images into your project's `images` directory. This can be done using copy and paste, drag and drop directly to the underlying filesystem folder or into the folder within the Designer environment.
- An image may be dragged and dropped from your filesystem directly into the application editor. When this happens the image will automatically be copied into the `images` directory and then a new control and image render extension will be created where you dropped the image.
- The Images View has functionality to import images using a toolbar icon. Multiple images may be imported at a time.
- The Image Selection Dialog. This is the dialog associated with the Name property of the image render extension. By default this dialog will show you the currently available images, but if you need to bring in a new image then there is an import button within the dialog.

The Image Selection dialog is presented when creating a new image control or when we are selecting to change the Name property of an image render extension. The primary difference between creating a new image control and changing the name property on an existing image render extension is that only newly created image controls will have the option to Resize the control to the image dimensions. This option allows you to drag out an area for a control but then properly fit the control to match the size of an image.

Image Rotation

Rotate At Center

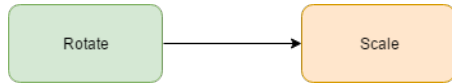
Storyboard image render extensions have an option to facilitate the rotation of an image at the center of the image. For images that are not scaled, this option behaves exactly as one might expect it would and as the rotation property is changed the image rotates accordingly.

When an image is scaled and there is need to rotate it the rotate at center provides an automatic scaling facility that may not be what you are expecting. In these situations the selection of rotating at center or selecting a custom rotation co-ordinate and making it the center of the image will provide different results.

In all situations, the option to rotate at center works best with the image render extension alignment configured center-center as such:

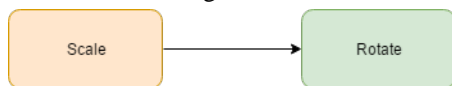


The rotate at center option uses a specific transformation path that will scale the image to fit the original size of the render extension after the rotation has taken place, this means that as it is rotated, it will be scaled to fit into its original bounds.



Rotate At Custom Point

The option to use custom rotation points works well with scaled images. The image will be scaled to fit its render extension bounds before being rotated and the image rotation will be performed without modifying the size of the image on screen.



In order to see the visual representation of the custom rotation point in Storyboard Designer, a render extension must have 'Rotate around center of control' disabled and must either have a non-zero rotation angle or must have a variable bound to the rotation property of the render extension. Hovering the mouse cursor over the crosshair will display the name of the render extension and the position of the rotation center point. Changes made to the rotation center x and y in the Properties View will be reflected by the crosshair in the editor.



Clicking on a control with a visible center rotation point crosshair will enter the control into direct edit mode. Upon entering this mode, the center rotation point crosshair will change to indicate that it is in direct edit mode and the mouse cursor will change to indicate that the point can be dragged. Direct edit mode allows you to drag the center rotation point crosshair within the bounds of the control, the center

rotation x and y properties will be updated to reflect changes to the crosshair position. When multiple image render extensions which are rotated at custom points exist on the same control, only the topmost render extension's crosshair will become active in direct edit mode.



When images are rotated dynamically at runtime, there is rarely a one-to-one correspondence between the original source pixels and the destination pixels so some level of interpolation, or mapping, is going to occur. There are many different algorithms for this interpolation each of which trades visual image quality for the cost of the computation to perform the interpolation. This trade off can be adjusted globally by configuring the `quality` render manager option for `sbengine`, with Storyboard defaulting to a higher visual quality mid-CPU cost option. On OpenGL based rendering platforms, another option that can have an equally significant impact on the visual quality of rotated images is the `multisample` render manager option which controls how the edges of image textures are blended with one another. Higher multisampling means smoother edges and lower multisampling results in more jagged edges.

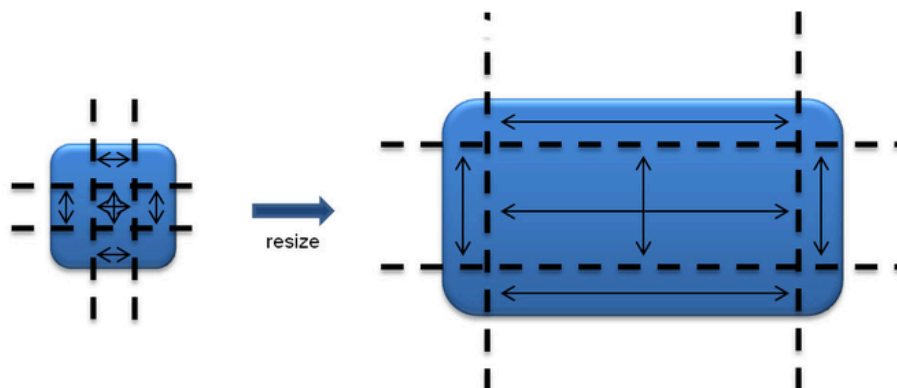
Alpha and Transparency in Images

Since much of a Storyboard UI is composed of images, the format and organization of those images can have a dramatic impact on performance. One of the key performance issues related to image rendering and composition is the use of full or partial transparency. Large areas of full transparency can result in many CPU or GPU cycles consumed needlessly traversing pixel areas. Similarly images that are fully opaque but in a format that indicates they may have some level of per pixel level transparency can be optimized at Design into a different format. Image format optimization and consolidation is covered in more detail in the optimization chapter of this manual.

Creating Scalable 9-Patch Bitmap Images

9-Patch is technique used to scale an image in such a way that the 4 corners remain unscaled. The four edges are scaled in one axis and the middle is scaled in both axis. 9-Patch support has been added to

Storyboard Designer to make scaling images on embedded applications easier. Instead of having multiple button images of various sizes, customers can now have one image that scales and maintains image quality.



9-Patch images can be designed and edited directly in the Storyboard Design environment. You can quickly analyze and convert existing large or scaled image content to 9-Patch format to achieve immediate memory and runtime performance improvements. In order to be converted to 9-Patch, the source image must have full 32 bit color depth.

Multi-Frame Animated GIF Images

While Storyboard animations [#ww_animations] provide a very power mechanism to create custom interactions, simple single image animations can be created using using the industry standard animated GIF format. Simply reference the animated GIF file in the standard image render extension and when the image is rendered for the first time it will start running the frames of the image. In the current implementation, animations specified as an animated GIF will loop forever. To stop the animation, change the image reference to a non-multi-frame image or mark the control as hidden.

Chapter 16. Working with Text

Storyboard gives you the ability to create and control text content with the text render extension. This render extension offers a single styled text display with options to control the text alignment, text overflow behaviour along with line and inter-character spacing. The text will be rendered using the font and point size selected in the property panel.

Storyboard uses standard TrueType (*.ttf) and OpenType (*.otf) font files and includes a number of commercially redistributable fonts. As these distributed fonts are selected and used the font will be copied to the `fonts` directory of the project. To use a custom TrueType or OpenType font in your application you just have to copy the font file into the `fonts` directory of the project. Once the font is parsed and analyzed it will show up for selection and use in the Font property selection.

By using the font files directly Storyboard offers developers total control over what resources are used on the embedded system. For example a system that uses a specific font for only numeric elements can be edited to remove all other glyphs. Similarly several different fonts can be merged together to provide broader language support for a particular product. The editing of these font files can be done with standard commercial editing tools such as `FontForge` or `TypeTool` among others.

To support internationalization and broad character support, Storyboard uses UTF-8 encoding universally for all text rendering. As long as the selected font contains support for the Unicode code points described by the encoding, Storyboard will be able to display the content. Support for advanced bi-directional text layout is provided by third party Crank Software partners and is discussed in more detail in [Text Shaping and Layout](#).

There are times when you will want to make formatting choices on your text based on the content you are displaying. Situations like this usually come up when dealing with internationalization and handling languages that have more characters than can be displayed or require a simple layout re-ordering to be displayed right to left. In these cases Storyboard provides you with ways to gather metrics about the text being displayed through the built in [Appendix A, Storyboard Lua API](#) the section called “`gre.get_string_size`” With the functionality provided by the Lua API you can choose to do things like reduce the font size of your text if the language you are displaying is more verbose. The [Scrolling Text](#) sample provides an example of how to use this Lua text API to size and measure text and adapt it to a particular display environment.

Rich Text Styling and Markup

Along with standard text support, Storyboard also supports the use of rich text. The rich text render extension allows the user to use a subset of HTML/XML tags to define the text formatting.

Table 16.1. Supported Rich Text Tags

NAME	DESCRIPTION	TAG	OPTIONS
Paragraph	The <code><p></code> tag specifies a paragraph.	<code><p> </p></code>	<code>style</code> - Set of attributes that define how to style the text.
Span	The <code></code> tag is used for styling text.	<code> </code>	<code>style</code> - Set of attributes that define how to style the text.
Bold	The <code></code> tag specifies bold text.	<code> </code>	None

NAME	DESCRIPTION	TAG	OPTIONS
Italic	The <code><i></code> tag specifies italic text.	<code><i> </i></code>	None
Underline	The <code><u></code> tag specifies underlined text.	<code><u> </u></code>	None
Break	The <code>
</code> tag specifies a line break.	<code>
 or </br></code>	None
Non-Breaking Text	The <code><nobr></code> tag specifies text that can't break.	<code><nobr> </nobr></code>	None
Font File	This is used to specify a font face for a font family (local file only, true-type fonts only).	<code><style> @font-face { font-family: roboto-bold; src: url('file:fonts/roboto_bold.ttf') } </style></code>	<p>1. font-style - Defines how the font should be styled. The values "oblique" and "italic" are treated the same, this specifies the font as italic. Default is "normal".</p> <p>2. font-weight - Defines the boldness of the font. Only support value is "bold", otherwise it's treated as "normal"</p>
Ordered Lists	The <code></code> tag is used to define an ordered list.	<code> </code>	style - Set of attributes that define how to style the list.
Unordered Lists	The <code></code> tag is used to define an unordered list.	<code> </code>	style - Set of attributes that define how to style the list.
List Items	The <code></code> tag is used to specify a list item.	<code> </code>	None

Table 16.2. Supported Style Attributes

NAME	DESCRIPTION	ATTRIBUTE NAME	PARAMETERS
Text Color	The color attribute specifies the color of the text.	<code><p style="color:red;">This is a paragraph.</p> or <p style="color:#FF0000;">This is a paragraph.</p></code>	Color can be a color name (ie. "red", "blue") or a hex value (ie. "#FF0000")
Background Color	The background-color attribute specifies the color of the background behind the text	<code><p style="background-color:red;">This is a paragraph.</p> or <p style="background-color:#FF0000;">This is a paragraph.</p></code>	Color can be a color name (ie. "red", "blue") or a hex value (ie. "#FF0000")

NAME	DESCRIPTION	ATTRIBUTE NAME	PARAMETERS
		F0000;">This is a paragraph.</p>	
Text Alignment	The text-align attribute specifies the alignment of the text block.	<p style="text-align:left">Left Aligned</p>	<ol style="list-style-type: none"> 1. left - Left justifies text. 2. center - Center justifies text. 3. right - Right justifies text.
Vertical Alignment	The vertical-align attribute specifies the vertical alignment of the text with it's line.	<p style="vertical-align:top;">Top Aligned</p>	<ol style="list-style-type: none"> 1. top - Aligns text at the top of the line. 2. baseline - Aligns text at the baseline. 3. bottom - Aligns text at the bottom of the line.
Font Family	The font-family attribute specifies the font family to use.	<p style="font-family:roboto-bold;">This is a paragraph.</p>	Name of the font family to use.
Font Size	The font-size attribute specifies the size of the font.	<p style="font-size:24px;">This is a paragraph.</p>	Text font size, point size syntax only.
Left Padding	The left-padding attribute specifies the left padding of the list.	<ol style="left-padding:0">	Number of tabs to use for list padding.

Example Rich Text

```

<style>
  @font-face {
    font-family: roboto-bold;
    src: url('file:fonts/Roboto-Bold.ttf')
  }
  @font-face {
    font-family: light;
    src: url('file:fonts/Roboto-Light.ttf')
  }
</style>
<p style="text-align:left">Left Aligned </p>
<p style="text-align:right"> Right Aligned </p>
<p style="text-align:center"> Aligned Center </p>
<p> <b> <u> I am Bold </u> </b> <br>I should be on my own line <br>
<i> I am italic </i> This long text should not be broken up. This
      long text should not be broken up.This long text should not be
      broken up.</p>

<p>
<ol>
<li> item 1 </li>

```

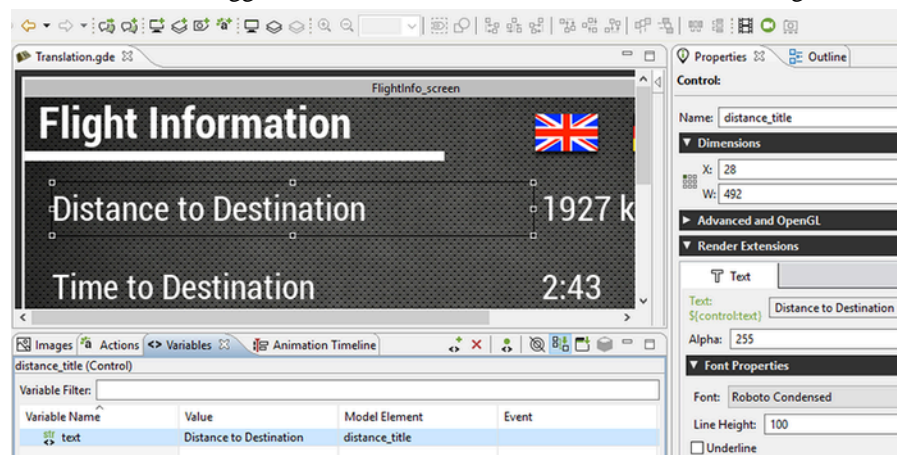
```

<li> item 2 </li>
<li> item 3 </li>
</ol>
<ul>
<li> item A </li>
<li> item B </li>
<li> item C </li>
</ul>
</p>
<p style="font-family:roboto-bold"> Roboto Bold </p>
  <p style="font-family:light"> Roboto Light </p>
  <p style="font-size: 50px"> 50px </p>
  <p style="text-align:right">right aligned <span style="color:blue">
    combined with blue</span></p>
  <p style="text-align:center">mixed styles: <span style="font-family
    :roboto-bold;color:blue">blue and bold</span> or <span style="color
    :green"><i>Green italic</i></span></p>

```

Translation and Internationalization

Storyboard makes it simple to translate and internationalize the text content of your application. Dynamic text content is treated the same as any other dynamic content that is rendered to the display. Within the text render extension, the translatable content should be associated with a variable. Any changes that occur to that variable, will trigger a screen re-draw to occur if that variable is being used in the current display.



Changes in string content of variables is automatically reflected in the user interface to those locations. This makes the translation activity significantly less labour intensive, as it is only required to identify text strings that are statically declared (ie not bound to a variable) and convert them to be associated with a variable. This variable to text string association forms the base of Storyboard's internationalization support. In order to change the UI to reflect a new language one has only to update the text variables with the appropriate UTF-8 encoded text strings.

There are many different ways to organize the association of language to text strings and then the association of those text strings to user interface variables. Storyboard does not impose any particular method, but generally speaking there are two main techniques that can be considered to accomplish these associations.

One method is to simply keep track of all of the text variables being referenced and for each variable have the translated string stored. This means that if I had a control, for example named mybutton on a layer

named `mylayer`, with a text render extension whose text property was associated with a variable (ie `myvariable`) that our language database would store the the fully qualified name, `mylayer.mybutton.myvariable`, along with the value of the string, perhaps "Hello". This technique is simple and works well for small applications where there is not a lot of duplication of text content. The sample *Translation* demonstrates this technique being used in conjunction with multiple independent comma separated value (CSV) files containing the language database information. Early versions of Storyboard provided a minimal level of automation to help manage these mappings.

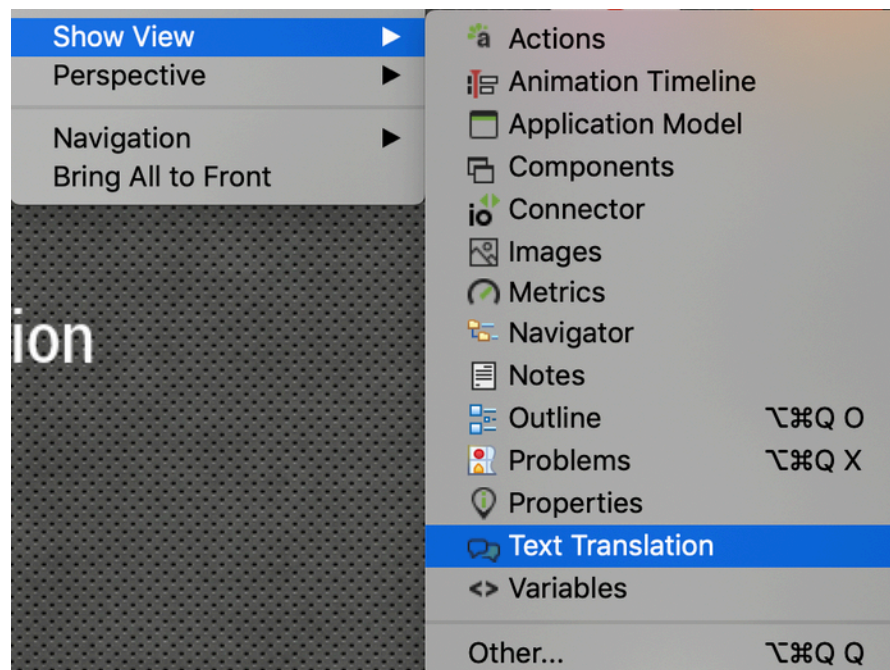
A second technique provides a more scalable approach to translated applications and is recommended for sophisticated translation and internationalization applications. Instead of having each UI instance of text declare a local variable all translated text refers to a single translation namespace with unique text strings identified by unique ids. This centralized approach allows unique text strings to be declared once and more easily allows for contextualized translations to be differentiated from one another. The sample *TranslationUsingId* demonstrates this technique and is supported within Storyboard Designer with automation tools that more easily integrate into workflows supported by text translation professionals.

The *TranslationUsingId* and *Translation* samples can be imported into your Storyboard Designer workspace via **File > Import > Storyboard Sample**.

In some circumstances, most notably with non-latin character sets, it may also be a requirement to dynamically change the fonts being used to map to an alternative font that provides the appropriate glyph support for the characters being rendered. Additionally it may be that a change in translated text requires additional properties to be adjusted, such as font point size or control dimensions, to accommodate the new translation. These can be adjusted as a straightforward change in value of a Storyboard variable. This is exactly the same technique that is used for translated text, the difference of course being that rather than changing text string values, the changes are to the font name or point size or numeric property of a control.

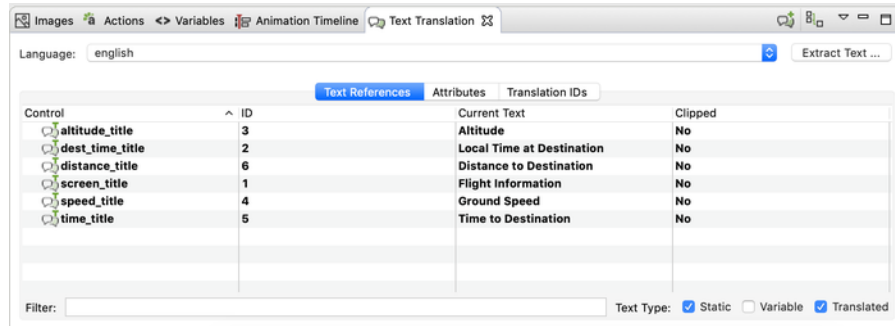
Text Translation View

Storyboard Designer contains a view specifically designed to assist with the process of converting and managing translated text using the common translation namespace technique mentioned above. To open the Text Translation view select **Window>Show View> Text Translation**.



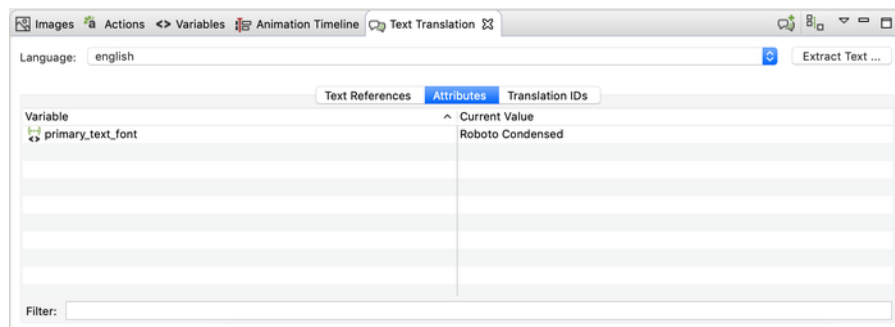
The Text Translation view shows the text and translation content for projects that have identified themselves as having a translation database. For information on getting started establishing a translation database, refer to the section Translating a Storyboard Application [#howto_translate].

For example, looking at the Text Translation view when then *TranslationUsingId* project is selected will show three areas of content.



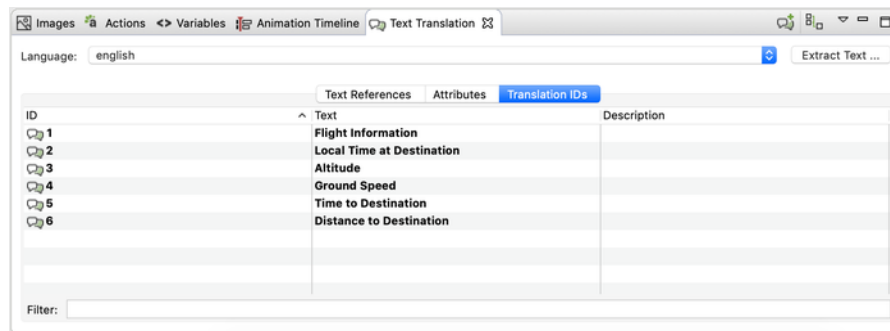
The Text References tab of the Text Translation view shows all of the text references in text render extensions that have been detected in the project. This includes text references that are static (not bound to variables), already associated with variables or associated with variables in the translation namespace. The filters at the bottom of the tab can help reduce the content displayed. This tab is the primary area where you would manage the associations of text in the application with translated values. Right clicking on any of the text entry fields will show a menu of choices:

Translate...	This will perform a lookup of the text string in the translation database. If a value is found then it will be used, otherwise a prompt to create a new text translation entry will occur. This option is only available for non-translated content.
Synchronize Selection	This is used to synchronize the value of the current selection with other language values using the same key. This can only be performed on translated content.
Switch Translation ID	This is used to switch an existing translation to another translation and is used when there are multiple translations with different contexts of use, or to change the translation id to something different entirely. This can only be performed on translated content.
Remove Translation	This will remove a translation id mapping and revert the value to a static string or local variable. This can only be performed on translated content.



The Attributes tab of the Text Translation view shows all of the non-text variables that are being tracked and adjusted in association with language changes. Unlike text references, these variables must be manually added but can include both user defined variables, such as font names or point sizes, and model internal variables, such as a control's position or size. This tab is the primary area where you would manage the associations of variables, but it is possible to add variables to the translation view from the Variables view or Application Model by right clicking on an existing variable and selecting Add Translation Variable. Within the Attributes tab, right clicking on any of the text entry fields will show a menu of choices:

Add Variable To Translation	This will open a variable selection dialog allowing you to manually select variables that have already been defined within the model to be associated with a translation language change.
Update Translated Variable	This is used to synchronize the value maintained in the attribute database for the current language with the current value used in the UI. This is a handy option to use when you want to interactively work through a set of language translations and make UI adjustments interactively and save the current value as shown in Designer.
Synchronize Selection	Similar to the the functionality in the Text References tab, this allows the synchronization of values for a field across multiple languages.
Remove Translation	This will remove the association of a variable as being changed when a language changes. This change is propagated to the translation attribute database.

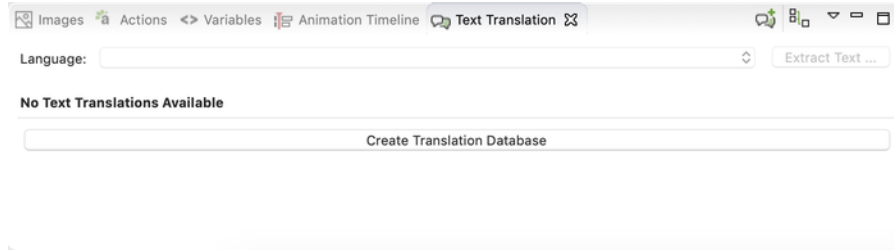


The Translation IDs tab of the Text Translation view shows all of the translation identifiers (ids) for the current translation set and the value of those ids for the currently selected language. Translation ids that are currently referenced and in use are shown in bold. At this time Storyboard does not offer support for the editing of these database values from within Designer but suggests that the values be edited in their source database, for example the comma separated value (CSV) file in the translations directory.

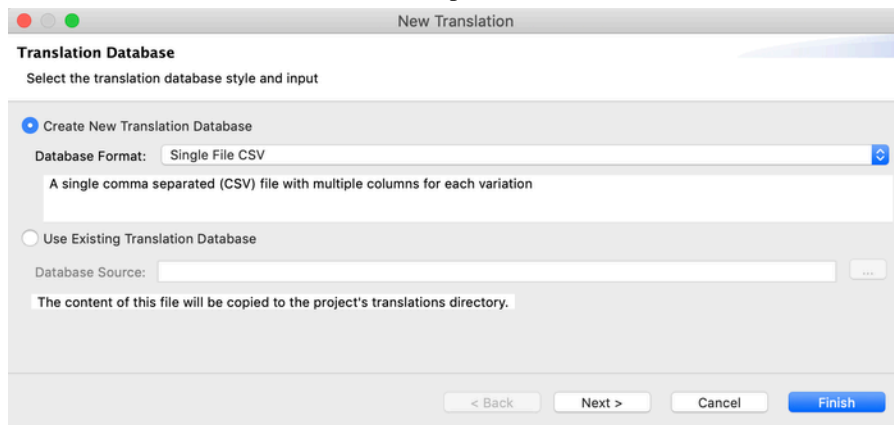
Translating a Storyboard Application

Many user interfaces do not start out with translation as a primary concern. Translation and internationalization of a UI is frequently an activity that occurs after the main user experience issues have been addressed and a workflow established in a primary language.

When the Text Translation view is opened and no translation database has been associated with the project, then what you will see will be an empty view prompting for a translation database to be created:



To get started translating your application, create a binding to a translation database where the text strings for the application in the default language can be stored along with any additional attributes that need to be considered as part of the internationalization effort for this application. Create this binding by selecting Create Translation Database which will open a New Translationwizard:



Currently Storyboard Designer only has the ability to interface with comma separated value (CSV) multi-column language files. In these files, commonly used by translators in programs such as Excel, a single column contains a unique identifier, then subsequent columns contain the different language values for that identifier. For example:

```
ID,English,French,Spanish
1,Hello,Bonjour,Hola
```

If a translation database such as this is already available, then you can select it in this dialog, otherwise a new one will be created for you. These files contain the text string information used for different text values in different languages. It is frequently necessary to also change non-text attributes that are Storyboard specific so these attributes are maintained in a separate database file using the same organization but instead of unique identifiers the fully qualified path to the Storyboard variable that should be changed is used instead:

```
ID,English,French,Spanish
Layer.Control.grd_width,192,310,192
```

These files are stored in the translations directory of the project with the name(s) that you have provided.

Once a translation database is established, then the first thing to do is to run an analysis of the current user interface and to establish translation mappings for the text content that is present in the default language of the application. This can be done on a field by field basis using the entries in the Text References tab or a full analysis can be performed using the Extra Text option. The Extract Text button in the Text Translation

view scans the all text field references and presents a dialog that allows the user to select and approve the automatically identified translations and to create new text translations for fields that do not already have a translation entry. This tool significantly helps bootstrap the translation process.

After this bootstrapping, the translation database file (default `translations/translations.csv`) can be provided to text translators or linguists to populate with new language entries. For more details on the best practices for editing these files to create UTF-8 encoded text strings, refer to the next section [Creating and Editing Translation Content CSV Files](#) [#ww_utf8csv]. If the translation database file is replaced in the project and the project re-opened, then the new language values will automatically become available for use.

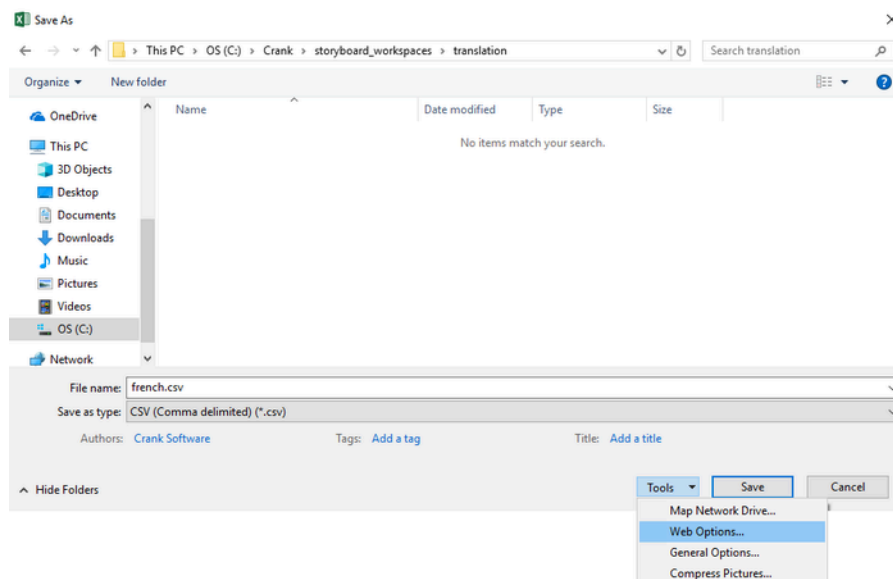
Creating and Editing Translation Content CSV Files

We have explored two tools for editing and creating translation content, Microsoft Office and Open Office Calc. Both tools are able to save UTF-8 encoded (csv) files.

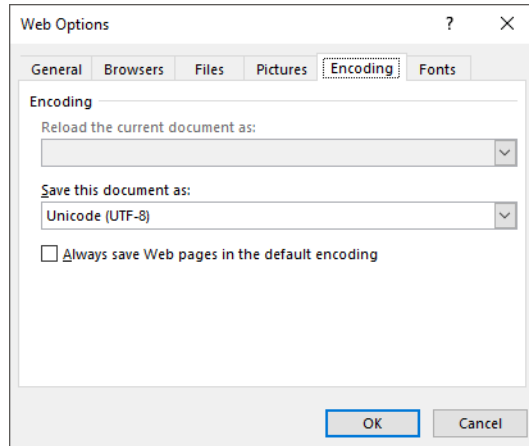
Microsoft Excel (Office 365 Version)

Microsoft Excel is a popular spreadsheet software solution that is widely adopted in many industries.

1. Using Excel, open a spreadsheet file via `File > Open...` (note: this can be a spreadsheet created using Excel or Calc)
2. Save the file as a (csv) file via `File > Save As > Browse`
3. Choose 'CSV (Comma delimited) (*.csv)' in 'Save as type:'
4. At the bottom of the dialog, select `tools > Web Options....`



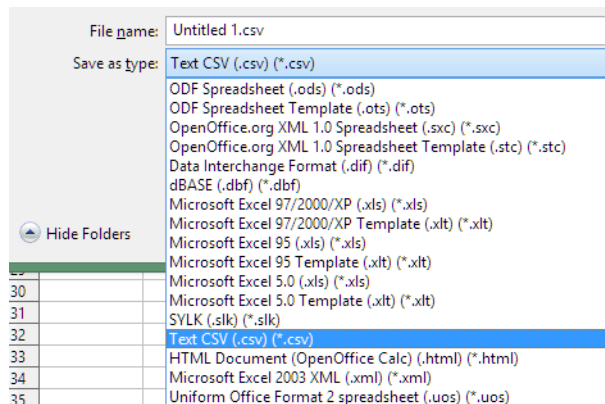
5. Select the 'Encoding' tab and choose to save this document as: Unicode (UTF-8).



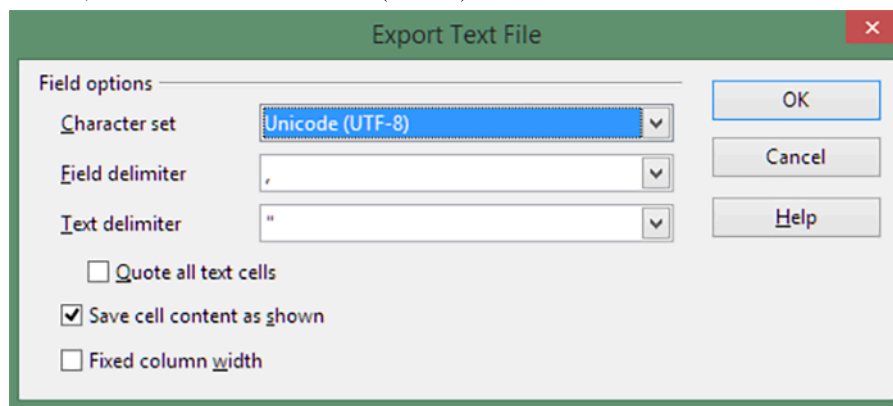
Open Office Calc

It is free and can be downloaded from www.openoffice.org [http://www.perforce.com/product/components/eclipse_plugin].

1. Using Calc, open a spreadsheet file via File > Open... (note: this can be a spreadsheet created using Calc or Excel).
2. Save the file as a (csv) file via File > Save as... > Text CSV(.csv)



3. When saving a (csv) file, Open Office Calc will ask which character encoding you wish to use for the file, be sure to choose Unicode(UTF-8).



Script Specific Text Shaping and Layout

Storyboard includes unidirectional text layout as part of the standard product offering. Support for bi-directional (mixed mode left-to-right and right-to-left) text is provided through commercial third party Crank Software partners.

Advanced text shaping and glyph positioning functionality for certain language scripts is provided in Storyboard by the `libgre-plugins-harfbuzz` plugin based on the functionality of the Harfbuzz text shaping library (harfbuzz.github.io) [<https://harfbuzz.github.io/>]. This text shaping technology is dependent on the FreeType font manager and advanced C++ toolchain support and is available for QNX, MacOS and most Linux platforms.

Advanced text shaping can have a significant impact on performance so it is disabled by default. In order to enable text shaping globally within an application for all text rendering set the text shaping script and language values into the application level `grd_text_shaper_attrs` variable. The value for this variable should be a string that identifies the desired text script and the language processing by their respective ISO codes and formatted as `script=<script_code>; language=<language_code>`

Script tags: https://en.wikipedia.org/wiki/ISO_15924 [https://en.wikipedia.org/wiki/ISO_15924]

Language tags: https://en.wikipedia.org/wiki/ISO_639 [https://en.wikipedia.org/wiki/ISO_639]

For example, a sample Lua script to set the text shaping for Thai script may look like:

```
gre.set_value("grd_text_shaper_attrs", "script=Thai;lang=th")
```

To disable the text shaping set the value to an empty string.

```
gre.set_value("grd_text_shaper_attrs", "")
```

Most latin languages do not require text shaping support and can receive an increase in performance by omitting the text shaping processing.

Chapter 17. Working with Touch, Gestures and User Input

Configuring Touchscreen Input

Touchscreen input is a very platform specific consideration. Storyboard works with a number of standard input devices and abstracts the implementation specific behaviours for `press`, `release`, `motion` and `multi-touch` system events into standard Storyboard events. These input events are described in the Storyboard standard event definitions section of this document.

The configuration details for setting up and troubleshooting two popular Linux input systems are described in this document. For other system or touchscreen specific configurations consult the operating system or touchscreen vendors documentation.

- Linux Target Specific Configuration: tslib
- Linux Target Specific Configuration: mtdev

Windowed Applications

On some systems, Storyboard Engine runs in a window. When running in a window there is specific behaviour for mouse or touch input leaving the screen. When leaving while pressed, Storyboard Engine will generate a `gre.release` event. When entering the window while pressed, Storyboard Engine will generate a `gre.press` event. Entering/leaving a window will not generate events if there are no mouse buttons pressed down.

Gesture Support

Storyboard gesture support is provided by the `libgre-plugin-gesture` plugin. The gesture plugin options are described in detail in the Storyboard plugin option appendix of this document.

The gesture plugin interprets the inbounds `press`, `release` and `motion` events and based on those observed events will generate custom gesture events. Gestures are only emitted once a `release` occurs and a pattern has been matched.

Gestures are made up of a series of numbers. The numbers represent the direction that the cursor was traveling as a grid arranged from one (1) to eight (8) ordered clockwise:

1. Up
2. Up and to the right
3. Right
4. Down and to the right
5. Down
6. Down and to the left
7. Left

8. Up and to the left

By default the gesture plugin registers some default gestures

<code>gre.gesture.up</code>	1
<code>gre.gesture.down</code>	5
<code>gre.gesture.right</code>	3
<code>gre.gesture.left</code>	7

Other gestures can be created by registering them in a custom gesture definition file that is loaded by the gesture plugin.

The gesture definition file is a comma separated value text file that contains a field for the name of the event followed by the numeric gesture sequence string that needs to be matched to generate the event. For example, to define a Z gesture, you could put the following in the a `gesture-definition.txt` file:

```
gre.gesture.zee,363
```

This definition indicates when the gesture plugin detects a right, down and left, left motion sequence that it should generate a `gre.gesture.zee` event.

You can point the gesture plugin at the custom gesture definition file by running Storyboard Engine with the option `-ogesture,file=filename`, where `filename` is the name of the project relative file, for example `gesture-definition.txt`.

Gesture sequences are currently limited to 30 movements after which a warning will be generated and the gesture entry will be ignored.

Multi-Touch Gestures

Unlike the single touch gestures, which state which gesture you have just entered, the multi-touch gestures are events that fire whenever you have more than one finger on the touchscreen. The plugin tracks up to five contact points, if 6 or more are present they will simply be ignored by the plugin. The events the plugin listens to are `gre.press`, `gre.release`, and `gre.motion` to track the touchscreen info while only one finger is present and `gre.mtpress`, `gre.mtrelease`, and `gre.mtmotion`, to track the touchscreen info while multiple touches are present. Note when using a multi-touch enabled device single the press, release and motion events will be sent only while there is only one touch point present. As soon as there are multiple touch points present, all events will be mt events.

After listening to the events, if more than one touch point is present and one or more touch points move, the plugin will do an update where it compares the old touch locations to the updated touch locations and generates the related multi-touch gesture events. To determine how many fingers are currently being used to generate these events, there is an `npoints` field in the event data.

`gre.mtmove`

This event uses `x_move` and `y_move` to communicate the difference in x and y of the midpoint of all present touch touches between the current and last event sent from the touchscreen.

Data

```
uint32_t    button
```

uint32_t	timestamp
int16_t	subtype
int16_t	x
int16_t	y
int16_t	z
int16_t	id
int16_t	spare
float	value
float	x_move
float	y_move
int16_t	npoints

Where:

x_move	The x difference between this event and last event
y_move	The x difference between this event and last event
npoints	The number of touch points used to generate this event

gre.mtpinch

This event uses the value data field, which will be the scale factor of the average spacing from all current touch points compared to the spacing of all the old touch points. The scale factor is calculated by $\text{newspacing}/\text{oldspacing}$, so a value of 1.1 indicates a growth of 10% and a value of 0.9 indicates a shrink of 10%

Data

uint32_t	button
uint32_t	timestamp
int16_t	subtype
int16_t	x
int16_t	y
int16_t	z
int16_t	id
int16_t	spare
float	value
float	x_move
float	y_move
int16_t	npoints

Where:

value	The scale factor event
npoints	The number of touch points used to generate this event

gre.mtrotate

This event uses the value data field, which will be the difference in rotation between the average angle of all current touch points compared to the average angle of all the previous touch points. The value will be in degrees.

Data

uint32_t	button
uint32_t	timestamp
int16_t	subtype
int16_t	x
int16_t	y
int16_t	z
int16_t	id
int16_t	spare
float	value
float	x_move
float	y_move
int16_t	npoints

Where:

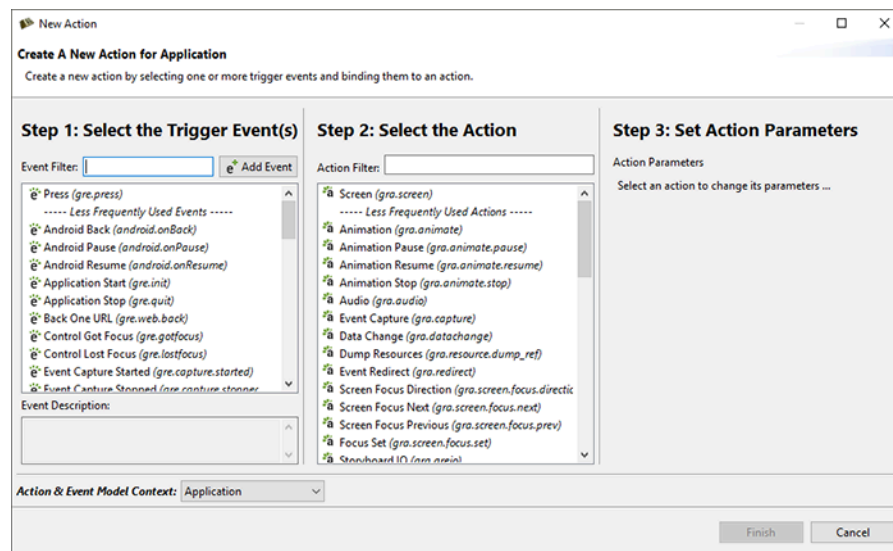
value The rotation difference event

npoints The number of touch points used to generate this event

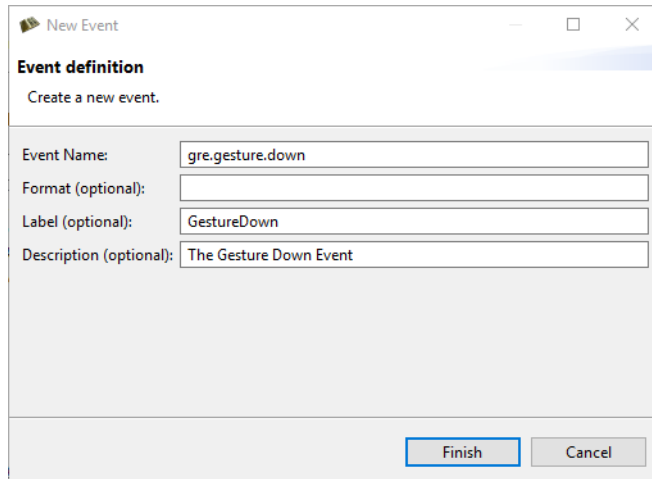
Enabling Gesture In Your Application

By default the gesture events are treated as custom events and are not included in the available events list for actions to bind with. You will need to add them manually the same way that custom user events are added as described in the chapter Chapter 8, *Connecting Events to Actions*.

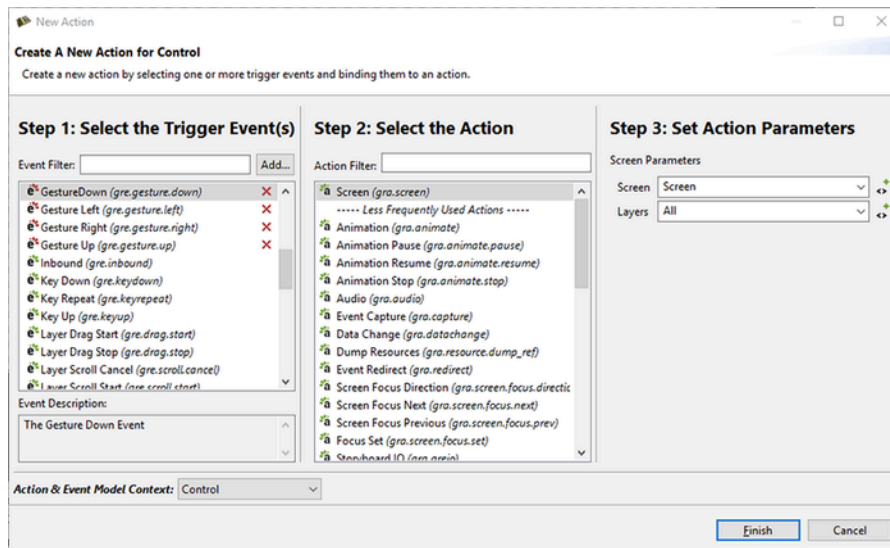
Right-click the control you want to add the action to, and select Add>Action. Then click the Add button to the right of the Event Filter text box.



You will see the New Event dialog box. This is where you add the gesture events to the list. You will need to do this for all the gesture events you want to use in your application, including `gre.gesture.up`, `gre.gesture.down`, `gre.gesture.left`, and `gre.gesture.right`.



Now that you have added the gesture events to the application you will be able to select them in the events list of the Add New Action dialog to trigger an action.



By default, sbengine will search your application for use of any gesture events and gestures will be enabled as required.

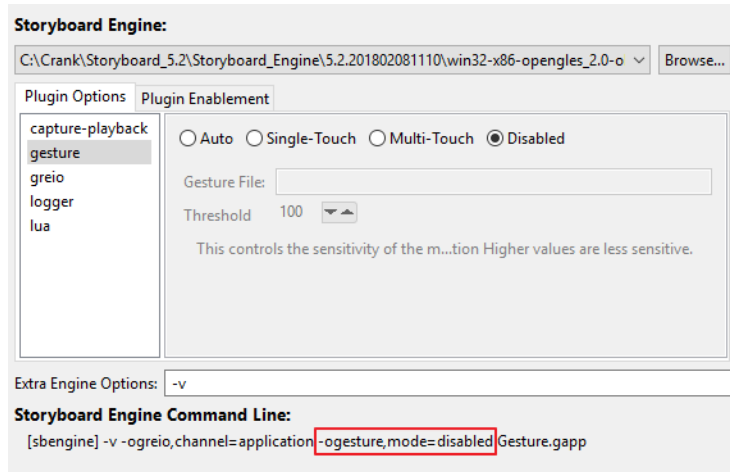
The gesture plugin can be disabled or forced to single or multi-touch mode by passing one of the following options to Storyboard Engine. See the section on Plugin Options for details about command line arguments.

When running from Storyboard Designer, the gesture plugin can be configured from the Simulation Configuration Dialog.

- auto This is the default setting, Storyboard Engine will decide disabled, single or multi.
- single Only single-touch gestures will be generated.
- multi Single and Multi-touch gestures will be generated.
- disabled No gesture events will be generated.

These gesture plugin options are discussed in more detail in the gesture plugin options section of this document.

Once you have configured the gesture plugin options in the Simulation Configuration Dialog, click Apply and Run to see your changes applied to the Storyboard Engine commandline arguments.



Chapter 18. Creating Lists and Tables

When you have a list of repetitive visual information to display then Storyboard's table's are a convenient tool to use. A table is equivalent to a control repeatedly applied, almost as a stamp or a template, to a grid of row and columns. A table with a single column or row can very effectively be used to represent list items. The controls that are created based on the number of rows and columns are referred to collectively as table cells. A table cell is a control at a particular one (1) based row and column offset. Tables offer automatic scrolling_layers capabilities, both interactively through touch events or programmatically using the Table Navigate and Table Scroll actions.

To create a scrolling table that is dynamically populated with data, you will generally follow these steps:

- Create a table
- Bind variables for dynamic table cell content
- Define the list data
- Resize the table to accommodate the list data
- Assign cell variables with the list data

The data that is being used for the table can come from a variety of sources, but is generally processed into the table cell variables using Lua. For example, your data from a database, a file on the filesystem or dynamically injected into the application as an event via Storyboard IO. Once you have your data you will be able to process it and assign the appropriate fields into cell variables that the control templates reference. The following is an example of Lua data to be used in a table that was taken from the AddressBook > Sample application shipped with Storyboard:

```
local contacts = {  
    {image="images/john_smith.jpg", first_name="John",  
     last_name="Smith", number="453-555-1685"},  
    {image="images/jane_smith.jpg", first_name="Jane",  
     last_name="Smith", number="466-555-1686"}  
}
```

With an idea of what the table data looks like, you can build a control template to be used in the table that will display this content. Tables are created the same way as standard controls, by right clicking in the editor or Application View and selecting Add > Table or by dragging a Table Control from the editor palette. The New Table dialog offers a few additional configuration options to allow you to specify an initial number of rows, columns as well as defining the render extensions for the table and the first table cell control template.

In the case of the data we have above, we might consider creating a table with a single column and a table cell control template that contains an image render extension (for the image) and a two text render extensions (for the name and number. The image name, and text content properties would each be bound to a table cell variable to allow each table cell to contain a unique value.

In general the number of rows and columns that you create on a table that will contain dynamic content is only going to be as large as you need it to be to show the design intent. With external data populating the table, you will size the table appropriately using the Lua API `gre.set_table_attrs()`. In our sample we have our data in a Lua table so we can easily extract determine the number or rows for the table to be, your application data may use a different protocol.

```
function CBInitTable()  
    -- Assume that tableLayer.table refers to our table
```

```
gre.set_table_attrs("tableLayer.table", {rows = #contacts})
end
```

This will configure the internal size of the table to the size of our `contacts` list. Once the table is dynamically sized, we can start filling content by assigning values to our table cell variables. This represents the transfer of model data to visual domain data as the Storyboard variables represent content that will be directly reflected in the UI. Table cell variables are accessed using their basename and a `.row.col` index:

```
function CBInitTable()
    -- Assume that tableLayer.table refers to our table
    gre.set_table_attrs("tableLayer.table", {rows = #contacts})

    -- Assume we've created cell variables named image, name and number
    local data = {}
    for (row=1,#contacts) do
        data[string.format("tableLayer.table.image.%d.1", row)]
            = contacts[row].image
        data[string.format("layer.table.name.%d.1", row)]
            = string.format("%s %s", contacts[row].first_name, contacts[row]
                .last_name)
        data[string.format("layer.table.number.%d.1", row)]
            = contacts[row].number
    end
    gre.set_data(data)
end
```

With this functionality in place the table is now being sized appropriately and its display content is being set.

If the content is too large to fit within the bounds of the table, then we will want to consider some sort of scrolling strategy. Storyboard offers two different scrolling strategies:

Cell based scrolling using Actions and cell re-positioning

Pixel based scrolling using table `xoffset` and `yoffset`

Both internally modify the x and y offsets, so it is safe to mix and match between the two techniques.

The cell based scrolling is performed by changing the cell that is located at the top left most corner of the visual table. This is performed using the Table Navigate or Table Scroll actions. These actions provide flexibility with regards to the roll-over of content as well since they are simply manipulating which cells are being displayed.

The pixel based scrolling is generally what users are expecting within a modern interactive UI. This scrolling is enabled by selecting the Enable Scrolling property in the table's Table section of the Properties View. Enabling this option will unlock various properties that regulate the rate at which the table scrolls through its decay and virtual friction. These properties, and the ability to synchronize scrolling content with other controls is discussed in Chapter 19, *Working with Scrolling Content*.

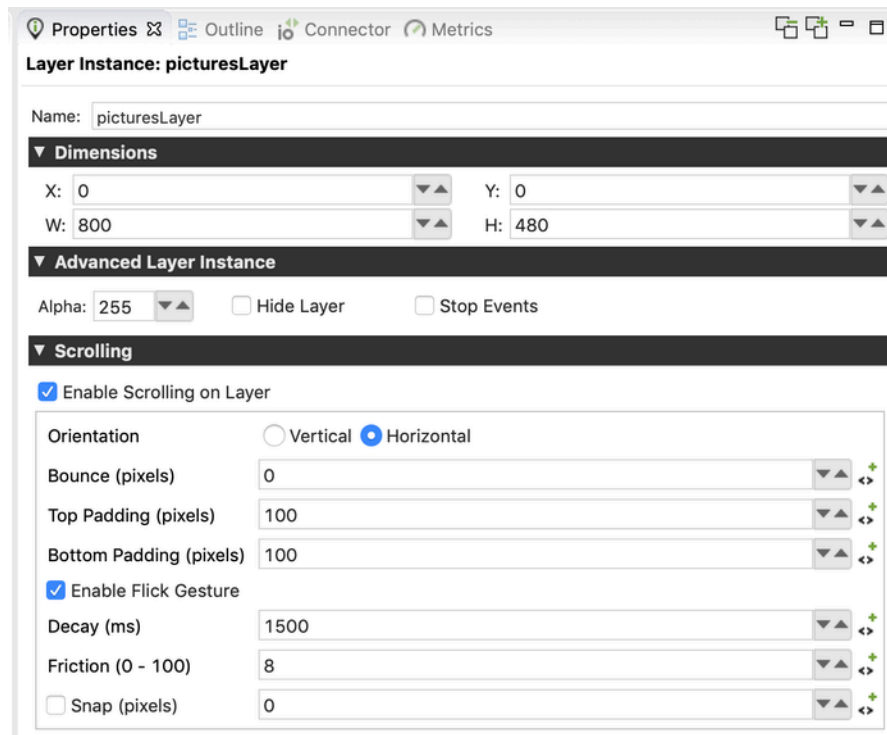
Most list based tables will only have a single column or single row and each cell will be rendered the same way but with different content (mapped via table cell variables). From time to time it is desirable to have different rendering capabilities for different cells and this can be done using different table cell templates. Storyboard tables only support different customized cell templates on a per column basis. To use a different table cell template, select the Cell Templates tab in the table properties section. Here you can add new table cell templates and assign those new templates to one or more table columns. It is recommended to name the table templates so that they can be more easily distinguished from one another. Once a table

template is created, then selecting the table cell in the corresponding table column will edit and configure that particular template rather than the default one.

For a more flexible and custom approach developers looking for per cell customization may want to explore using cloned controls and a custom layout on a scrolling layer rather than using tables.

Chapter 19. Working with Scrolling Content

Pixel based scrolling tables and scrolling layers provide the ability to scroll content in a smooth and uniform fashion. Configuring table's for scrolling is covered in the chapter Chapter 18, *Creating Lists and Tables*. Scrolling layers are based entirely on the bounds of their child group and control content and require a simple property enablement Enable layer scrolling behaviour within the Properties View.



When scrolling is enabled the object that is being scrolled, be it a layer or a table, is being moved as if it were positioned virtually and the object bounds provide the viewport into that virtual position. To track and measure this virtual positioning there are two internal Storyboard variables `grd_xoffset` and `grd_yoffset` that can be used to determine the virtual object position relative to the it's actual position. Considering the y offset value, when it is at 0 the content is aligned with the top of the control. When the y offset value is negative the content appears to be moving down.

When scrolling is enabled in the properties for an object the Storyboard Engine will start automatically tracking any press, release and motion events targeted at the object. While a press is active, the object will track exactly to the user motion. Once the press turns to a release, then the object will scroll automatically based on the scrolling properties configured.

Orientation	Vertical or Horizontal scrolling
Bounce	Number of pixels for the scrolling object to bounce when reaching the edge of the contents within the object
Top/Bottom Padding	Additional pixels at the top (left) or bottom (right) to allow scrolling past the edge of the contents within the object

Flick Gesture	If enabled, the object will continue to scroll after a drag and release based on the momentum of the gesture
Decay	Maximum time in milliseconds the scrolling object will scroll after a flick
Friction	A number from 0 to 100 to determine the friction level of the scrolling object. With higher friction it becomes more resistant to moving when swiped and more inclined to slowing down when released
Snap	If enabled, a scrolling table will automatically have its cell boundaries snap to the specified pixel offset after a flick gesture is triggered. A scrolling layer will have its offset snap to the nearest multiple of the specified offset

Scroll Synchronization

When we have scrolling content, we frequently want some sort of visual indication of where we are within the context of that content, like a scrollbar. Storyboard doesn't dictate what type of visual presentation is used for this tracking, but we'll describe here how you might make a scrollbar using a simple fill render extension. This technique can be readily applied to any other visual representation and is demonstrated in the Scrollbar > Storyboard Sample as well as the Address Book Sample

The first thing that we want to consider is what information we want to convey to the user. For a simple scrollbar we will use a proportional measurement that represents the percentage that our y offset $\text{positionPercent} = (-1 * \text{objectYOffset}) / (\text{contentTotalHeight} - \text{objectHeight})$. Here, we'll assume a vertically scrolling list, but the same principle applies for horizontally scrolling. With this `positionPercent` we have a representation of where our viewport is with respect to the total content available.

In order to apply this formula to synchronize a scrollbar representation with the content as it scrolls, we need to have a notification of the scroll change. This can be accomplished by making an event association with a variable as described in the section called "Triggering Events on Variable Changes". The variable that we want to bind an event to will be the internal variable `grd_yoffset` on the model object that we are scrolling.

With an event bound to an action, we might structure a synchronization function that might look something like the following:

```
-- Assume we are synchronizing a table named Layer.MyTable
function CBSyncScrollIndicator()
    -- These values could be cached, only yoffset would be changing normally
    local tableInfo = gre.get_table_attrs
    ("Layer.MyTable", "height", "yoffset", "rows")
    local cellInfo = gre.get_table_cell_attrs("Layer.MyTable", 1, 1, "height")
    local totalHeight = tableInfo.rows * cellInfo.height

    local positionPercent = (-1 * tableInfo.yoffset) /
    (totalHeight - tableInfo.height)

    -- Now apply the position_percent to your scrollbar object
```

end

Chapter 20. OpenGL and 3D Rendering

Storyboard 3D Rendering Model

3D Rendering Fundamentals

At the most basic level, rendering of 3D content is accomplished by using matrix and vector mathematics to transform points and directions between various coordinate spaces.

Understanding a few of the underlying concepts will help a designer make informed decisions when configuring 3D Model render extensions in Storyboard Designer. Below we will explain the coordinate spaces that are applicable to 3D rendering in Storyboard and explain how they relate to the properties of the 3D Model render extension.

World space is a three dimensional space that serves as the basis for defining all the other coordinate spaces. The locations of the camera and model in the 3D Model render extension properties are coordinates in world space. It is important to note that each 3D Model render extension instance references its own model data and is effectively a 2D portal into a distinct three dimensional world.

In Storyboard, we define the default position and orientation of the camera to be at the origin of World space and looking down the World space negative z-axis. There are 2 primary camera modes which determine the effect of the Camera parameters on defining View space (also called Camera space).

In “Orbit” mode, the Azimuth and Elevation parameters first rotate View space around the World space y-axis and x-axis (respectively). The camera X, Y, Z position then position the camera in this rotated space. By defining View space using transformations in this order, we can achieve a neat effect. If we set only the Z position of the camera, Azimuth and Elevation now spin the camera around the World space origin, with the camera always looking toward the origin.

In “Fly” mode, the camera X, Y, Z position define the position of the origin (0,0,0) of View space within world space. Azimuth and Elevation now rotate the View space around the y-axis and x-axis (respectively) of View space. This allows a camera that can freely look “away” from the World space origin in any direction.

You may notice in the above description that the above descriptions of Azimuth and Elevation are in terms of y-axis and x-axis, and not the z-axis. In order to simplify rotations, Storyboard does not allow the camera to be “rolled” along the View z-axis.

The 3D Model render extension takes as a parameter a single model file per instance. Storyboard supports .obj and .fbx files as 3D model input. Since FBX file support is provided by a closed-source library maintained by Autodesk. This library has support for limited number of platforms and architectures. To help mitigate this limitation, and provide an opportunity for offline optimization of model data, FBX files are converted on import to SSG (Storyboard Scene Graph) files.

The Scene Graph and Transformations

We support a hierarchical scene graph for defining a 3D scene. We define the Node to be the basic building block. Currently a node may be a:

- Group
- Mesh

- Light

All nodes inherit the transform (coordinate space) of their parent.

Groups define a set of children nodes, and a coordinate space which all children nodes inhabit.

The complete order of transformations within a Group node is the following:

- Inherited transform from parent
- Local (bind) transform from scene graph
- Deformation transforms
 - Translation
 - Rotation (around X-axis, followed by Y-axis and finally Z-axis)
 - Scaling

Meshes and Lights are leaf nodes.

Meshes define:

- Geometry
- Material information related to portions of the geometry.

Lights may be one of 2 types:

- Directional, best used for modelling distant constant light sources, such as the sun
- Point (or omni-directional) lights, best used for lights that emanate from a position, such as a lamp, etc.

Material Support

We support the following attributes for a material applied to a section of geometry:

- Ambient color
- Diffuse color
- Specular color (and a specular exponent)
- Emissive color
- Alpha (transparency, 0.0 completely transparent, 1.0 completely opaque)

We also support a diffuse texture map, which is currently used as a texture source for both diffuse and ambient color.

We store the following additional information, but do not have any support for rendering at this time:

- Reflectivity coefficient
- Separate ambient map
- Specular map

- Emissive map
- Bump map
- Normal map
- Reflection map (expected to take the form of plane, cube or spherical mapping of reflection information)

Animation and Variable Support

Information on what is possible with the FBX file format is included below, but the bottom line is that almost all 3D Modeling DCC tools dispense with almost all of this structure and bake the movements down into a single take/layer, so in Storyboard, for simplicity, we define a 3D scene animation to have:

- n Animation Channels, containing:
 - n Animation Curves

Channels are defined as a node/transform pair, such as "FrontDriversSideDoor"/RX (x rotation). These map to rows in the animation timeline in Designer.

Curves are defined by key frames, and include a key frame time and value for the transform. These will map to the endpoints of Animation Steps in Designer.

The Storyboard variables that are automatically associated with nodes in the 3D model are generalized as the following variable:

Rotation	RX, RY, RZ
Scale	SX, SY, SZ
Translation	TX, TY, TZ
Hidden State	hidden

Mapping FBX Animation data into meaningful structures

Animation data specified in an FBX file for a scene takes the following structure:

- n Animation Takes, containing:
 - n Animation Layers, containing:
 - n Animation Channels, containing:
 - n Animation Curves

Animation Takes (also called Stacks internally by FBX, but nowhere else it seems) define discrete animations that you might want to play. These quite easily map to our concept of animation clips in Storyboard Designer. Unfortunately, support for defining Animation Takes in many DCC tools is somewhat limited, see the note below. You can think of a Take in the film sense, "Action! ... do stuff, do stuff, do stuff... Cut!".

Animation Layers define a set of curves that you may want to play in parallel with another layer, allowing you to essentially modulate the defined motion of another layer. An example would be a sphere moving along a path (layer 1), while bouncing up and down (layer 2). These don't really map to anything in Storyboard, we would likely just import multiple layers of animation motion into a single clip.

Even though Animation Layers have little meaning to us, they are important because they are the container for a set of channels/curves.

Animation channels define what precisely we are deforming. These map to rows in our animation timeline. An example here would be "FrontDriversSideDoor, X rotation".

Each channel as mentioned above has a set of Curves, which basically map to the ends of Animation Steps in Storyboard. The curves are defined using key frames, with a time and a value.

In reality, most DCC tools (except MotionBuilder), will require any use of layers to be baked down into a single layer, and as mentioned above (and expanded on below), multiple takes are not natively supported either.

Support for Animation Takes

While FBX files can have multiple animation takes embedded in a single file, 2 of the most popular DCC tools, Maya and 3DS Max do not ship with the functionality to export the Animation Take data. These tools have a single animation timeline, and export the animation data a single take.

Artists desiring to specify multiple animations relating to a single model or scene have a few options, but all of them essentially defer defining this data to further down the asset pipeline.

The typical pipeline workflows are:

1. Export each separate animation into a separate FBX file. There are a whole bunch of problems with this idea.
2. Export modelling data to Autodesk MotionBuilder (previously called FilmBox, the origin of the FBX format) or another equivalent tool and use these to define the desired takes. These will import cleanly into separate takes.
3. Max and Maya have a paid plugin (fairly inexpensive - \$9 USD on TurboSquid as of the time of writing) allowing the artist to define multiple takes from the Maya and Max animation timeline. These are fairly simple tools, just defining a portion of the timeline to be each take, but are sufficient for most purposes.
4. Define all "takes" on a single timeline (with spacers between the desired takes) and export it as is. Use tools from the target middleware (Storyboard Designer, in our case), if they exist, to "slice" the animation into separate animations.

In order to support workflow 4, we would have to support the concept of slicing/splitting the incoming animations. As of Storyboard 4.2, this functionality is not supported.

Troubleshooting 3D Problems

When working with external 3D models, there are a number of issues that may occur when a model is imported and used within the Storyboard environment. If the model does not look correct while imported there are things that you can do both within Storyboard and within your 3D Design tools to troubleshoot:

- Zero Out/Freeze/Reset transforms. On 3ds Max this is done using Utilities > Reset XForm and in Maya this is in the right click menu Zero/Freeze Transforms.
- Set up a point for scene root. Attach geometry in proper hierarchy to that scene root, this allows a point to be referenced to move the whole model.

- If your textures are not showing up then make sure that you set up a project folder for your 3D model file and when exporting the FBX make sure that you select the media encoded selection and finally ensure that the textures that you are using are in the same directory as the imported model within Storyboard Designer.
- Different use cases call for different model co-ordinate systems. Use world coordinate system when using animations that have been pulled in from the FBX file. Alternatively use local coordinate system when placing the object dynamically through Lua or Storyboard variable manipulation.

In addition to the suggestions in the Storyboard optimization chapter, there are some specific 3D content related optimizations that can have both a visual and performance impacts.

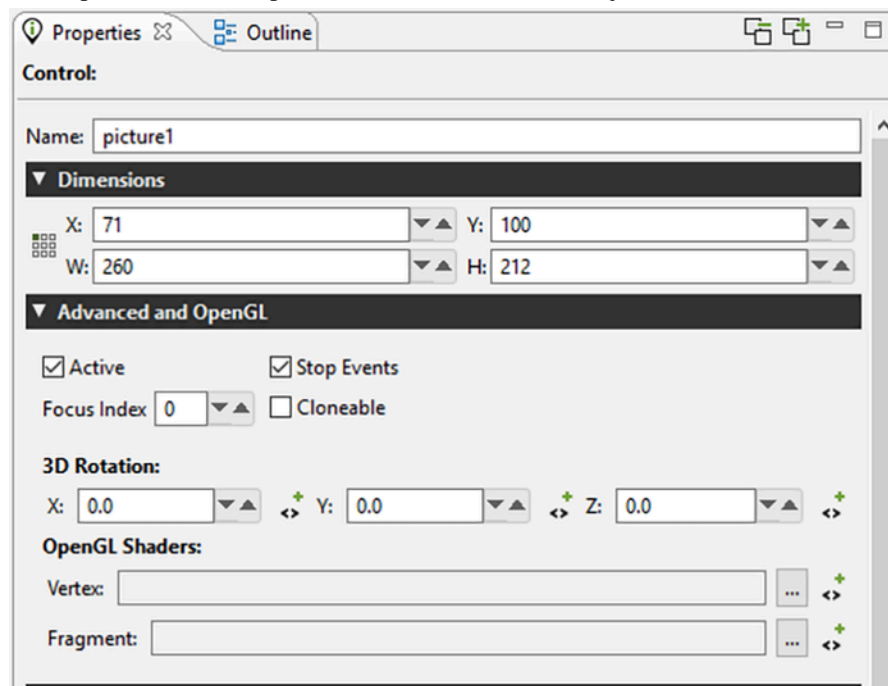
- Use an unlit shading model. Prebake lighting/ambient occlusion first to still have a nice lighting on the model without the runtime cost
- Turn culling on. This will have the impact of drawing less content and will speed up overall rendering operations.
- Use smaller texture maps and consolidate texture maps where possible. This will allow the rendering engine to use fewer resources and save on internal state transitions.

Working with OpenGL Shaders, Transforms and Compressed Textures

Storyboard provides the ability to leverage 3D capable OpenGL hardware to transform model elements, create custom GLSL shader effects and to display 3D model objects.

3D Transforms and Custom Shaders

In the Properties View for control and layer model elements there is an Advanced and OpenGL section that exposes transform options for the selected model object.



The X/Y/Z 3D Rotation transform properties allow the layer or control to be rotated around an imaginary axis that runs through the center of the object. This can be used to achieve simple perspective changes on an object, for example a cover flow style roll-over effect. These 3D rotation properties can be bound to variables and used any place a normal variable would be including animations and scripted data changes. The OpenGL3DModel Storyboard Sample rotates it's text field layer using a 3D rotation.

Also located within this Advanced and OpenGL property panel are properties for OpenGL Vertex and Fragment Shaders. These GLSL shaders can be provided as files and when provided will be used to process the control or layer after its base content has been rendered. Similar to the 3D Rotation properties, these files can be associated with variables and changed dynamically at runtime.

A complete description of OpenGL ES 2.0 shaders is beyond the scope of this document and the GLSL shader language itself is better treated in detail in the OpenGL ES Shader Language [http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf] specification. This chapter will focus on the shader programming aspects that are relevant to Storyboard developers.

Note

At this time, shader effects are not visible within the tool and are only visible when using an OpenGL based Storyboard Engine.

The shaders that are specified in the properties will be applied to the resulting texture that is the outcome of having all of the current render extensions applied to it. That is to say that the control is effectively rendered as it would be in the normal sense, but the final result is provided as a texture to the shaders to manipulate before it is finally rendered to the display.

Shader programs have three types of variables: attributes, varying, and uniforms. Attributes are passed into the shader from the render manager and contain data such as vertex locations, and texture coordinates. Varying variables are calculated in the vertex shader and passed into the fragment shader after being interpolated based upon the location of the fragment. Uniforms are also passed in from the render manager, but are typically used for purposes other than storing the geometry being rendered, for instance, containing a global alpha value which can be used to blend an entire model.

When writing a shader program, it is important to follow the conventions for attributes and uniforms established by the render manager. Otherwise, it will not be able to pass in geometric data and nothing will be rendered. Below is a minimal vertex shader which matches the functionality of the built in vertex shader for images.

```
attribute vec4 myVertex;
attribute vec4 myUV;

varying vec2 vtex;

uniform mat4 projMatrix;
uniform mat4 mvMatrix;

void main(void)
{
    gl_Position = projMatrix * mvMatrix * myVertex;
    vtex = myUV.st;
}
```

The myVertex and myUV attributes contain vertex and texture coordinates respectively. The projMatrix and mvMatrix contain the projection and modelview matrices, which are used to transform the input vertex

position, which is then assigned to the `gl_Position` for the vertex. The varying `vtex` is used to hold the interpolated texture coordinate which is then passed to the fragment shader. The render manager looks up `myVertex`, `myUV`, `projMatrix` and `mvMatrix` by name when it loads the shader, so these names must be used in any custom vertex shader. The varying name must match between the vertex shader and the fragment shader.

Below is a fragment shader which matches the functionality of the built in fragment shader for images, and shows the minimal code required to work with the Storyboard render manager.

```
#ifdef GL_ES
precision mediump float;
#endif
uniform sampler2D sampler2d;
varying vec2 vtex;

void main (void)
{
    gl_FragColor = texture2D(sampler2d, vtex);
}
```

The initial precision declaration is required by OpenGL ES, but not supported by OpenGL, and is set with a preprocessor conditional. The `sampler2D` uniform controls which texture unit is used when sampling a texture. The render manager only supports a single texture. The varying variable is interpolated based upon the vertex values in the vertex shader, and is passed into the sampler to look up the color at the fragment location. This is assigned to `gl_FragColor` and becomes the fragment color.

It is also possible to pass data from Storyboard variables to shader uniform variables, subject to two constraints: the model element for the variable must be the control for which the custom shaders are being used and the variable type must be float. When the custom shader is loaded, a list of all of the uniforms present is created. The name of each uniform is then compared to the list of variables attached to the control, and if a matching name of the appropriate type is found, it is used to set the value of the uniform when the control is rendered.

As an example, consider animating a custom shader to do a simple fade-in based upon the value of a timer. First, create a variable of type float called "current_time" for the control with the custom shader. Then create an animation using the animation timeline which changes the value of the variable from 0.0 to 1.0 over a few seconds, and create an appropriate trigger for the animation, for instance a mouse press event. Then, edit your fragment shader as follows:

```
#ifdef GL_ES
precision mediump float;
#endif

uniform float current_time;
uniform sampler2D sampler2d;
varying vec2 vtex;

void main (void)
{
    gl_FragColor = texture2D(sampler, vtex) * current_time;
}
```

When the control is rendered, the value of the uniform `current_time` will be set from the value of the control variable `current_time`, which will cause the color read from the texture to be scaled from 0.0 to 1.0 over the duration of the animation, causing a fade in effect.

Custom Shader Support

Storyboard supports custom OpenGL ES shaders written in GLSL. Shader programs can be attached to controls by creating a vertex and fragment shader program. These programs are then compiled at runtime and used by the Storyboard Engine. When creating a shader the uniforms can be manipulated in Storyboard Designer through variables. The naming of the shader uniform determines how it's variable is resolved. All shader variables must be float type variables. The uniform naming can be prefixed in order to tell Storyboard which context to resolve the variable:

<code>grd_a</code>	This variable is resolved at the Application level
<code>grd_l</code>	This variable is resolved at the Layer level (layer where the control is)
<code>grd_g</code>	This variable is resolved at the Group level (group where the control is)
<code>grd_c</code>	This variable is resolved at the control level (control where the shader is connected). This is the default if no prefix is used.

Fragment shader Example:

All variables can be created at the application level. The variables would be:

<code>r</code>	<code>float</code>
<code>g</code>	<code>float</code>
<code>b</code>	<code>float</code>
<code>a</code>	<code>float</code>

The program would be:

```
uniform float grd_a_r;
uniform float grd_a_g;
uniform float grd_a_b;
uniform float grd_a_a;

void main (void)
{
    gl_FragColor = vec4(grd_a_r, grd_a_g, grd_a_b, grd_a_a);
}
```

Vertex shader Example:

```
attribute vec4 myVertex;
attribute vec4 myUV;

varying vec2 myTexCoord;
```

```
uniform mat4 projMatrix;  
uniform mat4 mvMatrix;  
  
void main(void)  
{  
    gl_Position = projMatrix * mvMatrix * myVertex;  
    myTexCoord = myUV.st;  
}
```

Compressed Textures

The OpenGL ES 2.0 render manager now supports compressed textures on supported hardware. The formats supported are PVRTC1, both 4BPP and 2BPP. This compression format is supported on most PowerVR graphics chipsets.

To determine if the chipset supports it, running Storyboard with a verbosity level of 6 (-vvvvvv_ will print out, on startup, the GLES extensions supported by the chip. If PVRTC is supported, you will see GL_IMG_texture_compression_pvrtc in the extension string list.

Storyboard will manually decode these images if the runtime being used does not support them. Should a project that was running on a PowerVR chip and using compressed images be run on a SW runtime that does not support them, the images would still decode and render correctly, just without HW acceleration.

PVRTC provides a 8x improvement in memory size (A 1024x1024x4 BMP would take 4MB of memory, whereas a PVRTC image would take 512K)

Compression tools can be found at:

- PVRTexTool [<http://community.imgtec.com/developers/powervr/tools/pvrtextool/>]
- Using texturetool to Compress Textures [https://developer.apple.com/library/ios/documentation/3D-Drawing/Conceptual/OpenGL_ES_ProgrammingGuide/TextureTool/TextureTool.html]

Chapter 21. Working with Audio and Video

Storyboard includes a media plugin for Storyboard Engine that supports the playback of several different types of audio and video formats. The media plugin provides a common action interface for creating and controlling media playback using a variety of different backend media player services.

It is the backend services that perform the decoding of content. The display of that content may be to a Storyboard External Buffer render extension which will be composited into the Storyboard application, or the service may have the ability to render to discrete hardware layers available on the platform and composited to the Storyboard application directly in hardware.

External media backend services are provided for particular operating system and hardware based on their underlying media support. Currently Storyboard provides two external media backends, which are described in the section called “Media Backend Services”.

Note

Audio and Video support is not universally available on all embedded target platforms. It is however available on MacOS, Linux and Windows desktop simulation platforms.

Video and Audio playback and control are accomplished by using a set of defined actions. When creating a media application you must include these media actions in your Storyboard project. The media actions must be installed on a project by project basis and are defined as Action Templates. The media action template is located in your Storyboard installation root under the directory `Samples/ActionDefinitions/media.sbat`. Copy this file to the `templates` directory of your project in order to be able to use media actions in your project. Once you have copied the file over, you will have to close and reopen your application to see the media actions.

In order to create a Storyboard application that will play video, you must first add a control to your project containing an External Render Extension. This render extension will be the display target where the video content will be rendered, so size and position it appropriately in your application. In the properties of the external render extension you must set the Buffer Name and Object Path values. These values will be required to play the video properly and will be used to point to where the video content should be displayed.

To initiate video playback, you will use one of the media actions. You will need to add the predefined media action `gra.media.new.video` to your application, which will tell the media plugin to play your video. As discussed in Chapter 8, *Connecting Events to Actions* this action can be configured to be a response to any event.

In the action parameters, you will specify the `Channel_name` parameter, which will be used as the name of the Storyboard IO channel that will be created for communication with the media plugin. The channel name should not conflict with the application name and should be relevant to its use, a good suggestion might be `media`. Specify the project relative name of the video file to play in the parameter `Media_name`, for example `video/myvideo.webm`. Additionally fill in the parameters that will link the media command to the external buffer where the playback is to occur. For this you configure the `External_buffer_name` with the same name as the Buffer Name used in the external render extension and the `Object_name` with the same name as the Object Path. The `Output_width` and `Output_height` should also match the dimensions of the external buffer. Finally the parameter `Output_depth` should be set to 4.

Other media actions allow you to control the video playback operation once it has been started, including providing pause, resume, stop and seek action commands. Storyboard Designer includes a media appli-

cation in its samples named Media. For more information on loading samples, refer to the section called “New Project from a Storyboard Sample”.

Media Backend Services

The media backend service does the work of decoding, playing and controlling the media based on requests from the Storyboard application over a Storyboard IO channel. The default Storyboard IO channel name is `com.crank.media_backend`. This value can be overwritten by setting the `SBMEDIA_CHANNEL_NAME` environment variable to a new value. The FFmpeg plugin is loaded automatically when it is present in the Storyboard plugins directory. If the gstreamer-backend is going to be used to play media, the FFmpeg plugin needs to be removed from the Storyboard plugins directory or else both backends will compete to service media requests.

GStreamer Application

This media backend uses the gstreamer framework to play and control audio and video files. In order to use this backend the platform must have gstreamer and the required plugins installed. It is a good idea to try and play content with the “gst-launch” application to ensure a proper installation before running gstreamer-backend. This backend also uses Storyboard IO for communication with the Storyboard application so please ensure Storyboard IO is functional and the application has the “greio” plugin loaded.

Options:

`-e` : Render the video content with an external buffer
`-p pipeline`: Use the defined gstreamer pipeline to play the media
`-v`: increase verbosity, debug output

Action Data The “new.audio” and “new.video” actions take an `extra_data` argument. This argument is a string which can contain the following options which must be separated by a “;”.

GStreamer Pipeline

You can specify the gstreamer pipeline used to play the particular media by either passing it on the command line to gstreamer-backend with the `-p` option or by passing it to the actions. The pipeline can be passed in as:

“pipeline:[your pipeline]”

This pipeline can be similar to the one used with the `gst-launch` application with a few minor modifications. In order to allow the changing of the media file the first part of the pipeline must contain a named `filesrc` element as follows:

“pipeline:filesrc location=video.mov name=media-src”

Doing this will allow the code to find the named element and replace the location with a new video file.

FFmpeg Plugin

This is a plugin to Storyboard which uses the FFmpeg libraries to play and control audio and video files. In order to use this backend, the plugin must be included with in the runtime engine. You can play a video

from the Storyboard Designer Simulator, as well as using the Storyboard Engine on supported platforms. Note that at this time, FFmpeg ships with only WebM video format support and Ogg audio format support.

Action Data The “new.audio” and “new.video” actions take an extra_data argument. When using FFmpeg, no extra data is required and this argument can be empty.

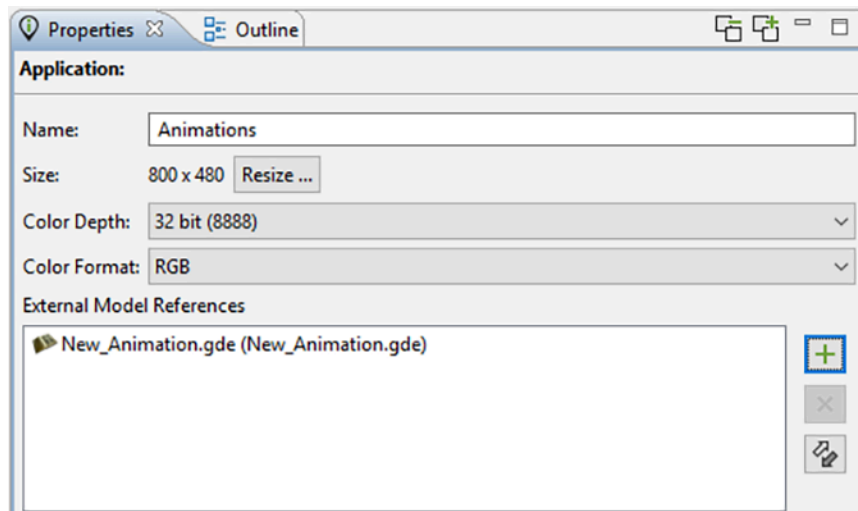
Chapter 22. Multi-File Application Development

Storyboard projects generally keep all of the model structure within a single application model file. For large products that have very discrete and well separated areas of development, then Storyboard offers the possibility of segmenting this functionality into multiple application development model files. How to create multiple model files is created is introduced in the section called “Working with Multiple Application Design Files”. Once multiple model files are created, then development can proceed in the same fashion as described in the chapters on general application development.

When it comes time to relate the disparate model files to one another and simulate and export the result, there are some differences from the standard Storyboard development process that are worth considering.

Screen transitions provide the means to tie together multiple applications to form a single, larger, unified application.

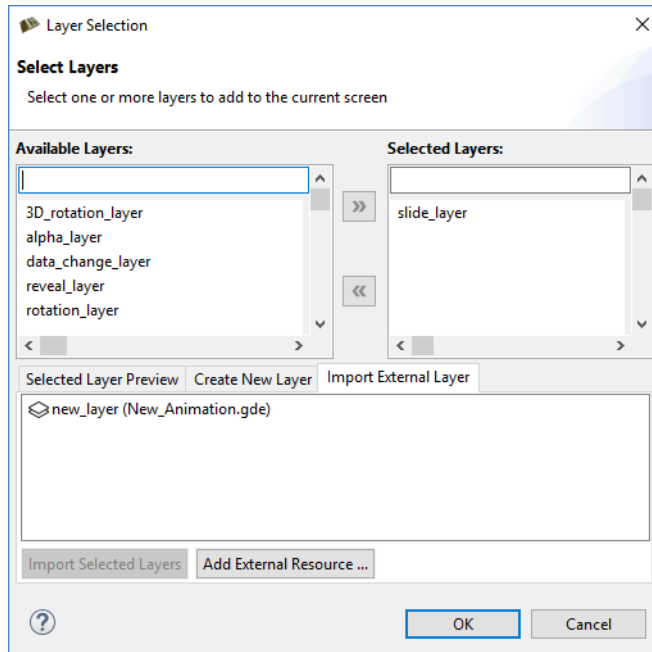
In order to indicate that one application (the source application) will be making a reference to screens in another application (external reference), the source application should list the external application in its Properties. Go to Application > Properties > External Model References



Once an application is added as an external reference, the screens from that application will show up in any of the Screen transition selection lists as well as any animation definitions..

Layers from any listed external applications can be added to an application by explicitly importing them using:

Add > Existing Layer > Import External Layer

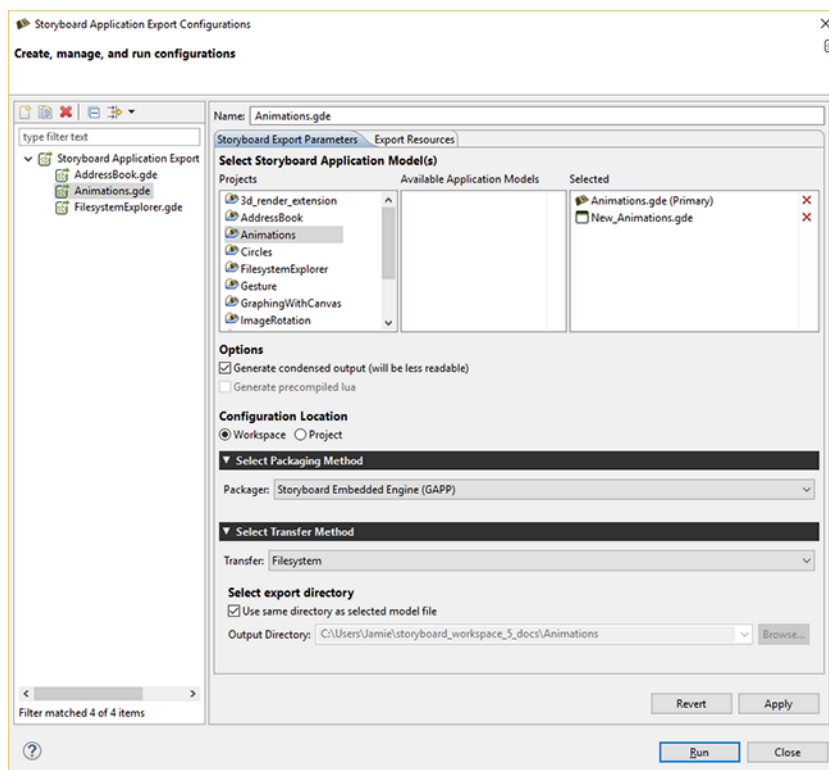


Once a layer is added as an external layer, then a copy of the original layer from the external application is snapshotted and incorporated directly into the source application. At this point, the external layer can be used with any source application screens just like any other layers.

Simulating and Exporting Multiple Model Files

Once applications have been developed, shared content leveraged, and transitions between application screens established the next step is to simulate or export the unified application.

The Storyboard simulation configuration dialog, accessed via Run > Storyboard Simulator Configurations allows multiple model files to be specified. All models in the 'Selected Models' block will be included in the application when launched.



The first source application in the list (tagged as Primary) will be used to determine the unified application's launch screen.

Similarly exporting a runtime application to be used with Storyboard Engine, Android, or iOS targets also allows multiple Storyboard application files to be selected. In these export scenarios, you will be prompted to select the source application that should be used for the unified application's launch screen.

If there are no conflicts among the selected applications they will be merged together and converted into a single unified application and used for the user-selected operation of simulation or export.

If there are conflicts among the resources then the differences will need to be resolved before continuing with an application merge.

Resolving Conflicts and Synchronizing Changes

When multiple applications are merged together to form a unified application, the following occurs:

- All layers from all applications are assembled together into a unified list of available layers. If two or more layer names are the same then those layers have their content compared. If the content is identical then the merge continues. If the layer content differs, then an error is flagged and the user will be prompted to resolve the differences and the application merge stops.
- All application/global level variables from all applications are assembled together into a unified list of global variables. If two or more variable names are the same then those variables have their values compared. If the values are identical then the merge continues. If the variable values differ, then an error is flagged and the user will be prompted to resolve the differences and the application merge stops.
- All animation definitions from all applications are assembled together into a unified list of available animations. If two or more animation names are the same then those animations have their definitions

compared. If the animation definitions are identical, the merge continues. If the animation definitions differ, then an error is flagged and the user will be prompted to resolve the differences and the application merge stops.

- All screens from all applications are assembled together into a unified list of available screens. If two or more screens have the same name, then an error is immediately flagged and the user is prompted to resolve the differences.

The external referencing of model elements relies on names remaining consistent during the application development. In some instances, if names change it may be possible for content to become unsynchronized and it may need to be resynchronized on an application by application basis.

The application properties page provides a synchronization action that scans the project for externally referenced content and then compares that content to the source reference. If there is a difference, then the difference is flagged as a conflict for resolution and the user is prompted for different ways to solve the conflict based on the nature of the issue

Chapter 23. Reusable Graphical Components

Storyboard Designer allows developers to create re-usable design components that can be shared among multiple projects. Components combine model elements, such as groups and controls, together with graphical assets such as fonts and images plus interaction rules such as events and animations. The ease with which these re-usable elements can be created based on custom graphical design elements makes them significantly different than traditional widgets that offer a fixed behavior and are difficult to create.

A component is created by selecting a Storyboard Designer group or control model element in the Storyboard editor or the Application View and right clicking and selecting Create New Component. A dialog will prompt for a component name, description, and additional animations and resources to include in the component. Components are saved by default in the project's `templates` and use the file extension `sbc` which stands for Storyboard Component.

Components located in the `templates` directory are automatically loaded into the Components View. To use a component simply drag the component into the Storyboard Editor into a visual design context. The first time that a component is used, all of the resources associated with the component (images, fonts, scripts and animations) will be extracted into the local project. Subsequent uses of the component will not overwrite the initial resources. If there is a name conflict with resources that are already in the project, then the default behavior is to use the existing project resources and not to overwrite the project resource with the template resource.

When a component is used within an application, a link is established between the component definition and each instance where that component is used within the application. This association allows components to be changed and for those changes to be applied as updates to any existing instance of the component.

Creation Guidelines and Conventions

Components provide a lot of freedom to users with regards to packaging re-usable content. However, there are some guidelines that if followed will reduce the potential for confusion when your component is applied to a number of projects.

Favor Groups over Controls

While it is possible to create components from a single control, this is generally only useful for the simplest of components. In order to manage interactive state, provide feedback and handle input it is better to use a group to provide a consistent namespace for controls and variables.

Use a Consistent Name Prefix

Components are intended to be created in the context of one project and then shared and used in other projects. However, since a component packages all of its images, fonts, scripts and animations with it, there is always the potential for name conflicts to occur. To minimize the change of conflict it is suggested that you name all of resources with a prefix that links them to component. For example Crank provided components use following naming convention:

- A prefix name is established using a prefix of `SBC` followed by the name of the component. For example `SBCPushButton`
- All image resources use the prefix name (e.g., `SBCPushButtonBevel.png`)

- All animation definitions use the prefix name (e.g., `SBCPushButtonFadeOut`)
- All Lua script files use the prefix name (e.g., `SBCPushButton.lua`)

Keep Lua Scripts Isolated

In order to avoid any conflict within the Lua namespace it is important to not only identify the Lua script file(s) used by the component but also to ensure that all of the Lua functions and variables used by the component do not conflict with anything else in the application. This means ensuring only a minimal set of callback functions are exposed and that the `local` keyword is rigorously applied.

Provide Change Notification via callback Variable

In many component situations it is desirable to remove the user from the mechanics of what a component is doing within the UI and to simply provide status information as a final result. For example a check-box button might have a selected/unselected state or a slider might have a percentage result that they indicate. In these cases it is a convention to use a variable associated with the group named `callback` that is typed as a Lua function variable. This allows the user to set different callback functions for different instances of the component but use the same controlling logic.

Editing Components and Propagating Changes

There are two ways to change and update a component after it has been created.

The first way is to simply use the component in an application, make the modifications that you want and then to save the component back as a new component, potentially overwriting the previous file. This is the same process used to create the initial component, the only difference being that you started the process by using a component to seed the initial functionality and behavior. This technique is most applicable when you want to introduce significant structural change to a component, for example adding or removing controls, render extensions or actions. In these situations the component you are creating is not really related any longer to any instances that they user may have created and would be difficult to synchronize while guaranteeing compatible behavior.

The second approach for updating a component is most suitable when you want to make cosmetic changes to the component and have those changes be applicable to any of the current instances of the component in the application. In this case you can edit the component file directly from the `templates` directory launching a Component Editor. When you do this, the component will automatically be opened and applied to a sandboxed Storyboard project containing a single screen and single layer. From within this sandboxed project, you can change the existing properties such as color, images or variable values. When the component is edited in this fashion, it will save a history of what properties have been changed along with a new internal revision number.

When components are changed using the Component Editor technique the changes that are recorded within the template can be applied to components that have been used in the current application. From the Component View select the component to update and right click and select the Update Instances option. This selection will scan the application for any uses of the component that are not already synchronized with the latest changes in the template and offer to synchronize the changes for you.

Chapter 24. Collaboration and Team Development

Traditional development techniques rely on a common source code repository that is revision controlled using tools such as SVN, Perforce, Clearcase, GIT or Mercurial. Storyboard Designer projects are designed to be directly integrated into this type of environment so that the UI can be shared and improved by many developers working in parallel.

Revision Control System Integration

In order to provide an integrated support for various revision control systems, Designer uses the *Eclipse Team Provider* plugins. Plugins are available for most revision control systems from marketplace.eclipse.org [<http://marketplace.eclipse.org>]. Here are links for several of the more common/popular plugins:

- Subversion svn Plugin: <http://marketplace.eclipse.org/content/subversive-svn-team-provider>
- Mercurial hg Plugin: <http://marketplace.eclipse.org/content/mercurialeclipse-was-hgeclipse>
- GIT egit Plugin: <http://marketplace.eclipse.org/content/egit-git-team-provider>
- Perforce p4 Plugin: http://www.perforce.com/product/components/eclipse_plugin

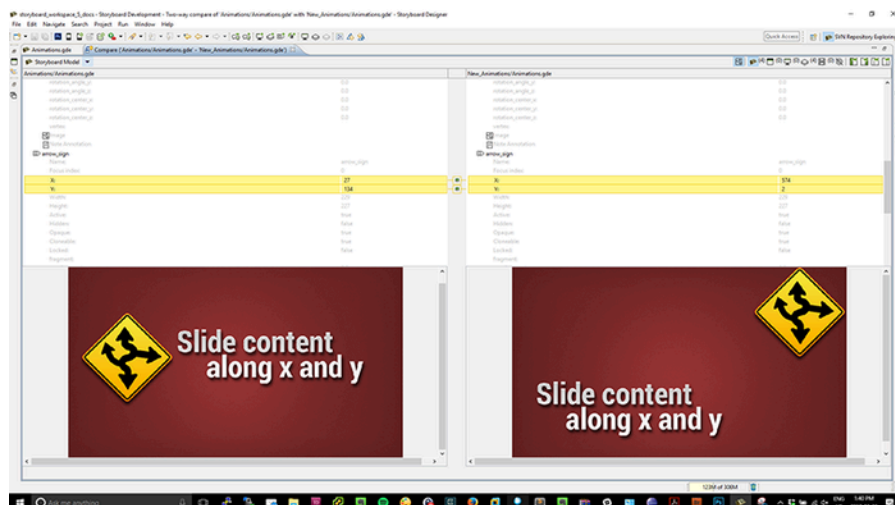
Comparing and Merging Model Files

The Storyboard Designer model file is a single model file. Conflicting changes to this model file can be visually inspected and merged from within Designer using the model comparison tools.

To compare two Designer files within the same project or file system workspace. Select both of the model files (e.g., `file1.gde` and `file2.gde`) in the Navigator view. Right click and select Compare With > Each Other from the menu.

To compare a Designer file that is in under revision control to a previous version, right-click on the file and select the menu entry Compare With. Different version control systems provide different specific terms, but . In the sub-menu you can select Latest from Repository or Revision if you want to compare with a specific version.

In either case, local comparisons or comparisons with versions from a revision control system, the comparison will open an editor that will highlight the differences in the model elements in the two files and allow each of the changes to be viewed in context and merged or discarded as may be required by the final design.

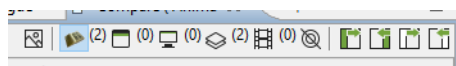


Triggering a comparison provides a hierarchical breakdown of the models' objects, with two sides representing the two files. Any differences between the two models will be highlighted in yellow. The two types of differences are property changes and additions/deletions.

Property changes show the value on both sides of the viewer. Changes are marked by a delta icon between the two tables in the two reference comparison or an arrow representing the direction of the change when three reference points are available. The three reference point comparison is usually only available when a source control system is available to provide the revision history.

For additions/deletions, the side representing the file that has the object will show it, and the other side will show empty space. Additions and deletions are marked by a + or - icon, and an arrow representing the direction of the change when three reference points are available.

In addition to changes, the comparator will also show any unchanged objects/properties, for reference and context. They are displayed in gray text, with no background color. In a three way comparison with an ancestor, it is possible that a conflicting change exists, where both sides have modified the same object/property from the original ancestor. These will be highlighted in red.



The toolbar contains actions and options to merge changes and switch the view. From left to right, here is a description of each one:

- **Toggle Graphical Compare:** Enables the graphical compare, which will appear on the bottom half of the screen and allow the user to visualize the changes on a model object.
- **All Changes Filter:** Displays all changes across all model objects.
- **Application Filter:** Displays changes on the application level. This includes application properties, and any application level variables or actions that have been added or deleted.
- **Screen Filter:** Displays changes on the screen level. This includes screen properties, layer instance properties and layer instance additions/deletions.
- **Layer Filter:** Displays changes on the layer level. This includes layer/control/render extension properties and control/render extension additions/deletions.

- Animation Filter: Displays any changes related to animations.
- Toggle Unchanged Properties: Shows or hides the unchanged objects/properties.
- Copy Left-Right/Copy Right-Left: Merges changes that have been selected in the viewer. If a model object is selected, any changes to its children will be merged.
- Copy All Left-Right/Copy All Right-Left: Merges all changes. In the two way case, this is non-destructive and will not delete any deletions, but will add the additions. In the three way case, this respects the direction of the changes.

After making changes, saving the comparator tab will appropriately save the changes to the model file(s). Exiting the comparator without saving will revert any applied changes. The global undo/redone functions are also available to revert and re-apply changes.

If you are working from a command line environment, or integrating Storyboard project merging into a continuous integration environment where file merges must happen automatically with a binary pass (merge successful) or fail (merge conflict), then Storyboard provides a command line merge tool for you to use: `${SB_DESIGNER_EXT} -nosplash -application com.crank.gdt.merge.gdemerge ${ancestor} ${current} ${other}`. Here the `${SB_DESIGNER_EXT}` represents the path to your Storyboard Designer executable and will vary based on your operating system and installation path. For a standard windows distribution you would find the Storyboard Designer binary in `C:\Program Files\Crank Software\Storyboard_Designer\Storyboard.exe` while on a MacOS system it may be `/Applications/Crank_Software/Storyboard_Designer/Storyboard.app/Contents/MacOS/Storyboard`.

The values of `${ancestor}` `${current}` `${other}` are the full paths to the files being compared and are designed to align with external source management tools such as subversion (svn) and git. The merged content will be stored into the `${current}` file by default but can be redirected by specifying `-output=newfilename`. As an example of how to use this merge command with a command line git client, put the following in your `.gitconfig` file: `[mergetool "mergegde"] cmd = \"C:\\Program Files\\Storyboard_Designer\\Storyboard.exe\" -nosplash -application com.crank.gdt.merge.gdemerge -quiet=1 $BASE $LOCAL $REMOTE trustExitCode = false`. Assuming that the mergegde is specified as the merge tool for all `*.gde` files then this will invoke the Storyboard specific merge for model files.

In addition to the merge tool built into Designer, there is a standalone comparator application that allows for an automated merge as well as a manual merge with a visual diff. The application will first attempt an automatic merge of the provided content. If that fails due to the discovery of a conflict, the visual diff will appear and allow the user to merge manually. The application is packaged in Storyboard alongside Designer, and makes use of the exact same command line options as above along with two new ones. Specifying `-headless` will suppress the visual diff regardless of the result and just attempt an automatic merge, exiting if a failure occurs. Specifying `-manual` will skip an automatic merge attempt and jump straight into the visual diff of the files. Similarly, this is what a mergetool entry would look like in `.gitconfig`: `[mergetool "mergegde"] cmd = \"C:\\Program Files\\Storyboard_Comparator\\StoryboardComparator.exe\" [-quiet] [-headless|manual] [output=/path/to/result.gde] $BASE $LOCAL $REMOTE`

Comparing and Merging Projects

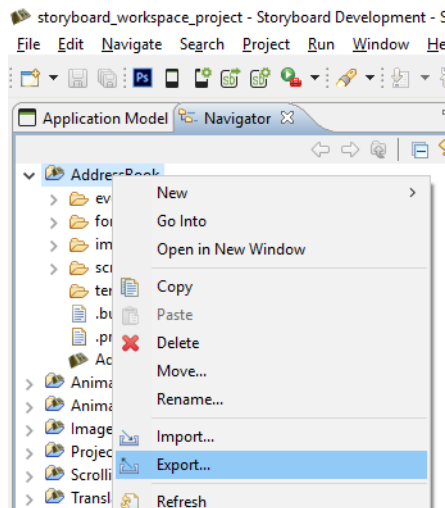
Entire projects and directory structures can be compared as easily as comparing single model files. This will allow developers to understand which resources such as images, fonts and script files have changed in addition to the changes to the model logic. The same as a model file comparison, this can be performed by right clicking on two projects or directories and selecting Compare With > Each Other from the menu.

When this comparison is done a hierarchy is presented indicating files that have changed, been added or removed from the source or the destination directory. Selecting any of these files will open a comparison editor that is appropriate for that file type.

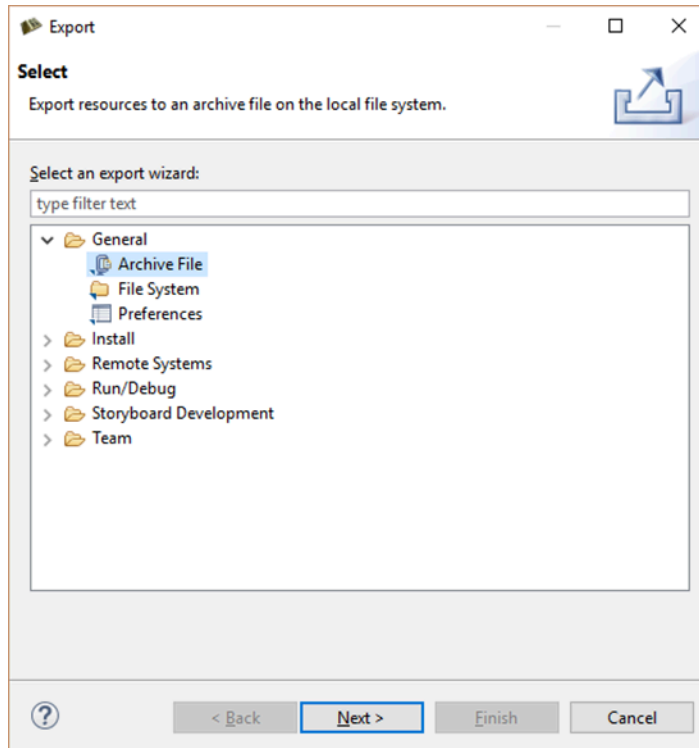
Exporting Storyboard Projects for Sharing

Storyboard Designer gives you the ability to export your project for either archiving, sharing or demo purposes. Here are a couple of easy steps showing you how to do that.

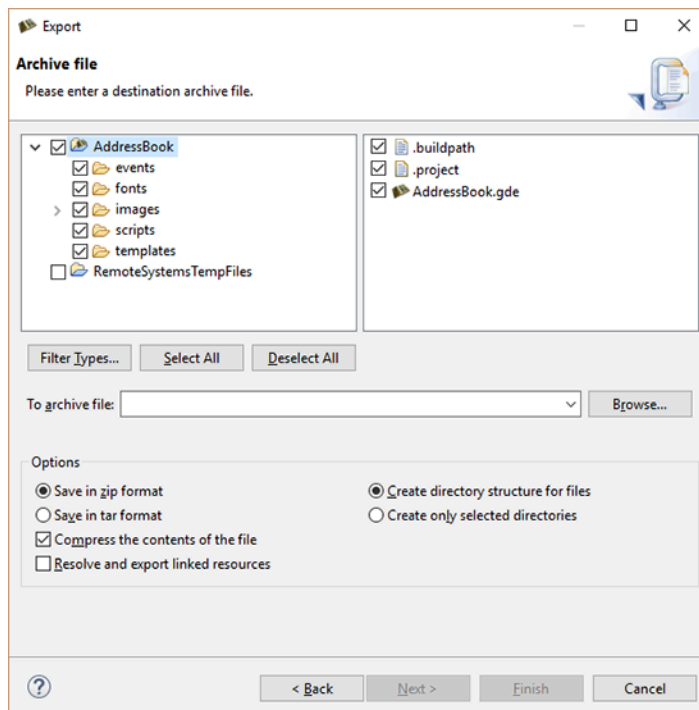
When your Storyboard Project is complete and you are ready to export, right click on the project folder and select Export.



Next you will be presented with the Export Selection dialogue. Expand General by clicking on the triangle to the left of the folder. Select Archive File and then click Next.



Next you will see the Export Archive file box. Here you will see all the folders and files that will be included in the archive you are about to create. Browse to the location where you want to save and then provide a name for your archive. Review and verify your Options and then click Finish.



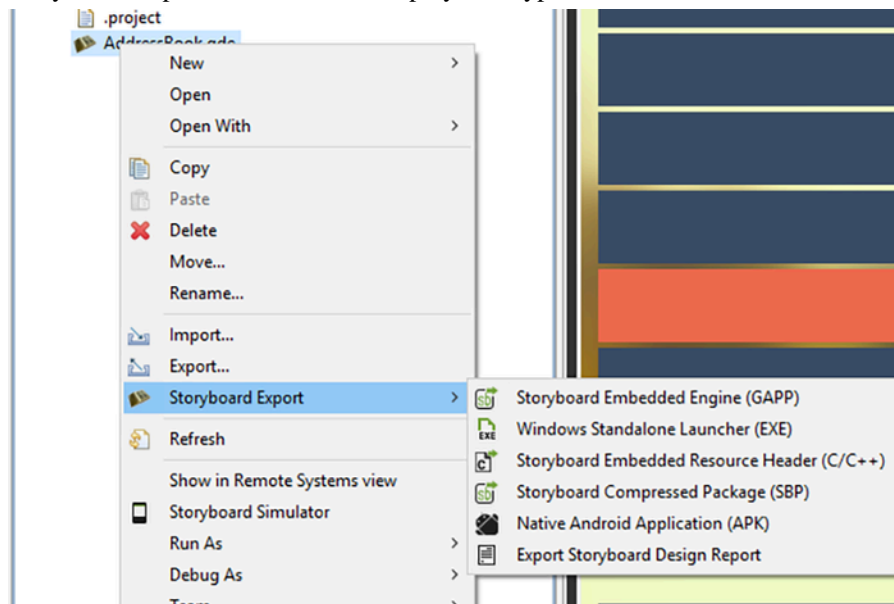
Chapter 25. Exporting and Running on your Embedded Target

A Storyboard application can be exported in several different ways. The output of the export process is what is considered a Storyboard Deployment Bundle. Depending on the bundle packaging these can be ran by sbengine, installed on an iOS/Android based mobile platform, executed on Windows as a normal application, or used by the user to compile into their RTOS application.

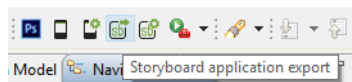
In the past deployment exports were done through a few different export wizards, but as of Storyboard 5.0 these have been consolidated into a single export configuration system. This export system allows the user to create, save, and share their application export parameters. These configurations can be customized through the Storyboard Application Export Configuration dialog.

Export Workflow

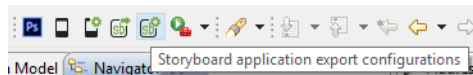
To export a Storyboard Deployment Bundle, right-click the Storyboard application file (*.gde) and select Storyboard Export, and choose the deployment type.



An export can also be launched by clicking on the Storyboard Application Export button in the toolbar. This button will launch the first export configuration found for the current application, or create one if none exist.

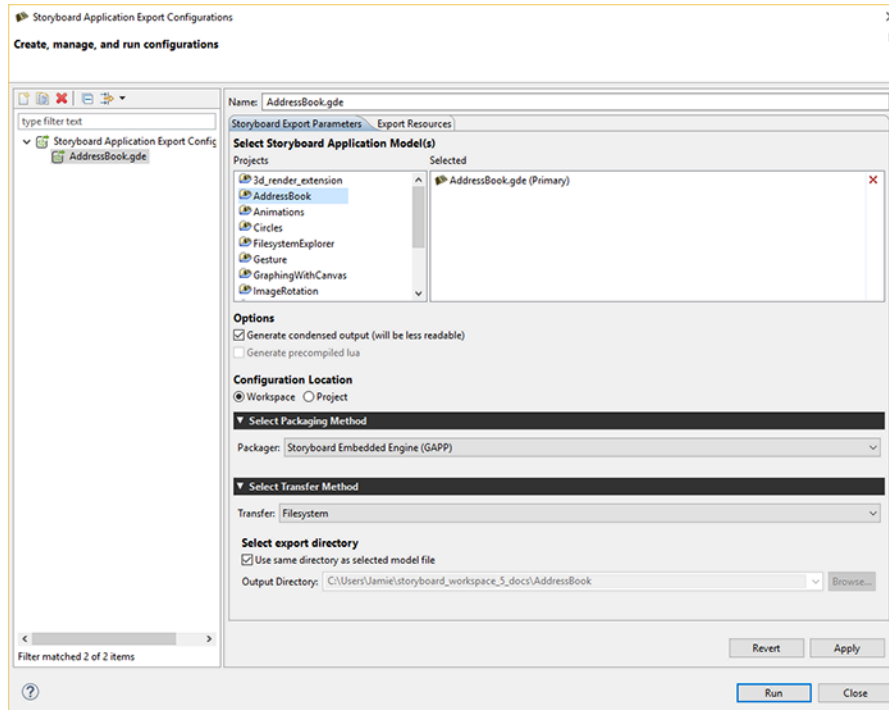


The Storyboard Export Configuration dialog will appear the first time an application is exported for deployment. This dialog allows you to customize how the application is exported and can be accessed at any time by clicking on the Storyboard Application Export Configuration button in the toolbar.



The export dialog is separated into sections that take you through the required elements of an export.

Exporting and Running on your Embedded Target



Application Model To Export

The first export parameter block is the selection of which application model file (*.gde) files to export. Selecting more than one will cause these files to be merged when exporting the deployment bundle.

Generic Model Export Options

The second export parameter block lists generic Storyboard Model export options. These are options that are independent of the specific packaging format but apply generically to the model.

The Generate condensed output option determines if extraneous character information is removed from the exported model file. The Generate precompiled Lua option will attempt to pre-compile the project's Lua scripts to bytecode prior to packaging. Only the top level files in the project's scripts directory will be compiled into platform independent bytecode. Compilation uses the same file names as the original files so this option is only available when the user is exporting to a directory other than the project directory.

Export Packaging

The third export parameter block determines how the exported model should be packaged. These packages and their options are described below.

Export Transfer

The final export parameter block is where the packaged content should be saved or transferred. Currently transfer methods include copy the package to the local filesystem or transferring it to a target over the network using scp. The transfer methods and their options are described below.

Once the user has finished customizing their export configuration they can click the Apply button to save the configuration and the export can be launched by click on the Run button. The application will then be exported according to the configuration.

Selecting Files For Export

All of the resources used by the application (images, fonts, scripts, translation files etc) must be available at runtime and included in the bundle packaging. The exporter will automatically select common resources but the packaged content can be customized using the Export Resources tab of the export wizard

The common resources extracted from the Storyboard application's project include:

- The `images` directory content. These are image assets that are required by the application.
- The `fonts` directory content. These are fonts that are required by the application
- The `scripts` directory content. These are usually Lua script files that provide glue logic for the application
- The `translations` directory content. These are usually translation files used by the application

Users may include additional content or exclude specific files as required by the application. The directory structure of the exported bundle package will reflect the same directory structure as used in the Designer project and should be maintained when the deployment bundle is moved to an embedded target.

Deployment Bundle Packages

Storyboard Embedded Engine (GAPP)

This will export the application as a GAPP file. The application resources will be transferred along with the GAPP file. This is the default package when exporting a new application.

Storyboard Compressed Package (SBP)

Export the application as a compressed package. The exported `.sbp` file will contain the application resources that were selected.

The following are parameters available for the Storyboard Compressed Packaged export

Encrypt exported package

The user can choose to encrypt the exported SBP package.

Native Android Application (APK)

Export the application as a Android Native Application. The exported `.apk` file will contain the application resources that were selected.

The following are parameters available for the Native Android Application export

Application Parameters

These parameters allow the user to change the application name, the APK file name, the application UID, and the application version information. Other parameters allow the user to change the application orientation, scaling, and whether the status bar is shown.

Application Name

The name that will appear on your device's UI for the app. This name doesn't have to be unique

	Android Package File Name	This is the name of the apk file that is exported. This name has no effect once your application is installed.
	Android Unique Application Identifier	This ID uniquely identifies your app on the device and in Google Play Store. It looks like a Java package name (ie. com.example.myapplication).
Runtime Parameters	This is where the user specifies the Storyboard Runtime to export in the APK. The user can also specify runtime options here.	
Manifest Parameters	Here the user can set their shared object list and add additional permissions to the Android manifest file.	
Icon Parameters	Here the user can set the icons for the application.	

Native iOS Application

Export the application as a iOS Application. The exported .app file will contain the application resources that were selected. This option is only available on MacOS development platforms and requires that the iOS mobile development environment be configured prior to export. This configuration is discussed in the chapter the section called “Exporting to iOS Devices”.

The following are parameters available for the Native iOS Application export

Application Parameters	These parameters allow the user to change the application name, the company identifier, the signing identity and provisioning profile. Other parameters allow the user to change the application orientation, scaling, whether the status bar is shown.
Runtime Parameters	This is where the user specifies the Storyboard Runtime to export in the app. The user can also specify runtime options here.
Icon Parameters	Here the user can set the icons and launch screens for the application. There are different images for different types of iOS devices, i.e. iPhone, iPhone retina, iPad, iPad retina.

Windows Standalone Launcher (EXE)

Export the application as a standalone Windows application. The exported .exe file will contain the application resources that were selected.

The following are parameters available for the Windows Standalone Launcher export

Windows Standalone Launcher Parameters:	The user specifies the Storyboard Runtime to export in the exe. The user can also specify runtime options here.
---	---

Storyboard Embedded Resource Header (C/C++)

Export the application as a set of C/C++ header files. This will export one or more header files that can be used in conjunction with the *Storyboard RTOS SDK* product which generally requires a direct compilation of the Storyboard application into the system image. There are a number of parameters available for this export, the full context of use for these options are described in the *Storyboard RTOS SDK* users guide.

The following are parameters available for the C/C++ Resource Header export

Draw images directly from code
storage (Flash/ROM)

This option relates to how the images used in the Storyboard application are going to be treated by the engine on the embedded system. Standard compressed image formats such as PNG or JPEG require decoding to a temporary buffer prior to display. This requires additional dynamic memory that may be beyond what the hardware platform has available. In order to support these smaller memory configuration systems with a graphically rich user interface, Storyboard Designer can pre-decode the images to a format that is directly compatible with the system's display output. When this occurs, the image data can stay resident in read-only memory and the Storyboard Engine can render it without incurring any additional dynamic memory overhead.

Export Model/Resources/Bitmap
Fonts

The C/C++ header export is split into three separate header file choices one for model, one for the image and file resources and one for the bitmap fonts (if they are being used). By default these files will all point to the same header file location, but they can be separated if that is more suitable for your development purposes. The paths specified are paths that are relative to the output location set in the Transfer section.

The bitmap font header selection is only required if the platform configuration does not include a dynamic font rendering engine, such as FreeType. In the absence of a font engine, which would use the TrueType font files directly, the fonts and glyphs must be pre-rendered for use by Storyboard's bitmap font engine. The number of glyphs generated is dependent on the content of the font files and the quality of the font rendering is controlled by the Alphamap Font selection. The more bits that are used, the smoother the font will render but the larger the resulting data will be

Export Transfer Methods

Filesystem

Copy the deployment bundle to a directory on the filesystem. The user can choose to export to the project directory or select another directory on the filesystem to copy to. This is the default transfer method when exporting a new application.

SCP Transfer

Copy the deployment bundle to a target via scp. The user needs to enter the target address (either hostname, IP address or mDNS service name eg: 'mydevboard.local' if supported), the username and password, and the target directory they want to copy to. Selecting the Post Transfer Script option will allow the user to execute a script on their target post launch. This could be used to launch the application they exported.

After running a export the scp parameters will be saved as a global workspace preset that could be used with future exports. Currently only 10 presets are saved at a time.

Exporting from the Command Line

It is also possible to perform headless exports of the Storyboard Design files to the Storyboard Embedded Engine files from a command line or scripting environment.

All of the Storyboard executables will be located in the `PATH_TO_INSTALL/Storyboard_Designer/storyboard` directories, though on each desktop platform they are named slightly differently

Windows:

```
....\Storyboard.exe -application com.crank.gdt.ui.gappexport  
    model=<PathToGDEFile[,PathToAdditionalGDEFile,...]>  
    output=<PathToGAPPFile>
```

Mac:

```
....\Storyboard.app/Contents/MacOS/Storyboard -application  
    com.crank.gdt.ui.gappexport  
    model=<PathToGDEFile[,PathToAdditionalGDEFile,...]>  
    output=<PathToGAPPFile>
```

Linux:

```
....\Storyboard -application com.crank.gdt.ui.gappexport  
    model=<PathToGDEFile[,PathToAdditionalGDEFile,...]>  
    output=<PathToGAPPFile>
```

Note: Storyboard requires a display, so to run in a true headless environment a Virtual frame buffer needs to be setup.

```
Xvfb :1 -ac -screen 0 1024x768x8 export DISPLAY=:1  
....\storyboard -application com.crank.gdt.ui.gappexport  
    model=<PathToGDEFile[,PathToAdditionalGDEFile,...]>  
    output=<PathToGAPPFile>
```

Where the `model` is the path to the Storyboard Designer model file. In the situation where multiple GDE model files are being joined together it is a comma separated list of model files where the first model file will be used for the start screen and the remaining models will be used for additional content. The `output` parameter specifies the filesystem path where the Storyboard Engine file will be created and the directory containing that file will be used to for the additional resource directories (scripts, images and fonts).

If the gde file that is being selected for export does not reside in the workspace that is going to be used by the command line exporter, then the export configuration must be stored in the project. This is an option that is available when modifying the export configuration. The following picture illustrates the selection that needs to be made:

Name: Bubblemark.gde-Exporter

Storyboard Export Parameters | Export Resources

Select Storyboard Application Model:

Bubblemark ... Primary Model: Bubblemark.gde

Options

☒ Generate condensed output (will be less readable)

Configuration Location

☐ Workspace ☒ Project

Select Packaging Method

Packager: Storyboard Embedded Engine (GAPP)

Select resource export configuration

Resource Export Configuration: Default

Select Transfer Method

Transfer: Filesystem

Select export directory

☒ Use same directory as selected model file

Output Directory: C:\Users\Rodney\Documents\Workspaces\storyboard-6.0\Bubblemark Browse...

Revert Apply

Note that the "Configuration Location" option is set to "Project". If the project resides in the same workspace that the command line exporter will be using, the configuration can be stored in either the workspace or the project

By default the exported gapp file will be compacted and not in a readable format. If a more readable format is desired, then the `readable` parameter can be passed along the command line such as:

```
.... /Storyboard -application com.crank.gdt.ui.gappexport  
readable  
model=<PathToGDEFile[,PathToAdditionalGDEFile,...]>  
output=<PathToGAPPFile>
```

Exporting from the Command Line using Export Configuration

It is also possible to perform a headless export using a pre-configured export configuration. After configuring the export in Designer a user can export this from the command line or scripting environment.

All of the Storyboard executables will be located in the `PATH_TO_INSTALL/Storyboard_Designer/storyboard` directories, though on each desktop platform they are named slightly differently

Windows:

```
....\Storyboard.exe -application com.crank.gdt.ui.configexport  
model=<PathToGDEFfile[,PathToAdditionalGDEFfile,...]>  
configuration=<ConfigurationName>
```

Mac:

```
....\Storyboard.app/Contents/MacOS/Storyboard -application  
com.crank.gdt.ui.configexport  
model=<PathToGDEFfile[,PathToAdditionalGDEFfile,...]>  
configuration=<ConfigurationName>
```

Linux:

```
....\Storyboard -application com.crank.gdt.ui.configexport  
model=<PathToGDEFfile[,PathToAdditionalGDEFfile,...]>  
configuration=<ConfigurationName>
```

Note: Storyboard requires a display, so to run in a true headless environment a Virtual frame buffer needs to be setup.

```
Xvfb :1 -ac -screen 0 1024x768x8 export DISPLAY=:1  
....\storyboard -application com.crank.gdt.ui.configexport  
model=<PathToGDEFfile[,PathToAdditionalGDEFfile,...]>  
configuration=<ConfigurationName>
```

Where the `model` is the full path to the Storyboard Designer model file. This model file will be used to search for a export configuration. The `configuration` parameter specifies the configuration name. If this is provided then only the configuration with that name will be used in the export.

Additional Options

`-data` Is used to specify the workspace to be used when performing the export. This workspace should not be opened by other instances of Storyboard. This is helpful when running the `configexport` option in order for Storyboard to find your export configurations in your project. It is specified as follows:

```
....\Storyboard.exe -application com.crank.gdt.ui.configexport  
-data <PathToWorkspace>  
model=<PathToGDEFfile[,PathToAdditionalGDEFfile,...]>
```

```
configuration=<ConfigurationName>
```

-consoleLog Is used to enable additional logging to the console.

-noExit Is used to prevent the application from closing automatically after performing the export. This is sometimes useful to keep the log console open post-mortem.

Setting up Storyboard Engine

An engine execution environment is provided for each supported operating system, architecture and rendering system. The target system should be configured with the Engine (sbengine) and plugins required for the target application. All plugins are loaded via the SB_PLUGINS environment variable and all libraries are loaded via the LD_LIBRARY_PATH environment variable.

For example assuming a Storyboard Engine installation located at /home/crank/linux-imx6yocto-armle-opengles_2.0-obj then you might configure the following environment variables:

```
export SB_PLUGINS=/home/crank/linux-imx6yocto-armle-opengles_2.0-obj/plugins
export LD_LIBRARY_PATH=/home/crank/linux-imx6yocto-armle-opengles_2.0-obj/lib
```

On Windows systems, there is no LD_LIBRARY_PATH so the PATH environment variable should be used instead. Similarly on MacOS, DYLD_LIBRARY_PATH should be used instead of LD_LIBRARY_PATH

Font Environment Variable

The Storyboard Engine, when used with the FreeType font engine, has environment variables for font options:

SB_FONT_HINT="normal"	This corresponds to the default hinting algorithm, optimized for standard gray-level rendering.
SB_FONT_HINT="light"	A lighter hinting algorithm for non-monochrome modes. Many generated glyphs are more fuzzy but better resemble its original shape. A bit like rendering on Mac OS X.
SB_FONT_CACHE_SIZE=[size]	This variable is used to set the size of the FreeType font face cache. By default the cache is disabled and a value of 0 will disable the cache. Any other number is the number of font faces to cache, using this cache can decrease memory mappings of font files.

QNX Screen Environment

The QNX screen composited window system provides the ability for multiple applications to render content to a display at the same time and to have that content merged together.

SB_SENSITIVITY=[0 1]	By default (0), the screen composition manager does not deliver input events to an application that has fully transparent pixel content. This corresponds to the default QNX screen input setting of SCREEN_SENSITIVITY_TEST. Setting the SB_SENSITIVITY option to 1 any touch input received on the applications display area will be delivered to the application regardless of if the pixels were transparent or not. This corresponds to the QNX screen input setting of SCREEN_SENSITIVITY_ALWAYS.
----------------------	---

Running Storyboard Engine

The Storyboard Engine executable (sbengine) is located in the bin folder of the Storyboard Engine directory structure. Now that the Storyboard Engine and Storyboard application (development bundle) are located on the embedded target and the Environmental Variables have been set, the Storyboard Engine can run a Storyboard application as follows:

```
sbengine thermostat.gapp
```

The Storyboard Engine is a self contained executable which loads plugins for added functionality. The Engine can be run as follows:

```
sbengine [-i] [-v] [-o] [storyboard application]
```

Table 25.1. Options

OPTION	DESCRIPTION
-i	Displays which version of sbengine and related libs are being used
-v	Verbosity, more v's means more verbose output.
-o	Plugin or manager options

Each plugin or manager defines its name and possible options.

As the verbosity level to Storyboard is increased, you will see more information about the execution of the runtime engine.

Target Specific Configurations

Linux x86, armle

FBDEV

This build renders directly to the Linux framebuffer device (/dev/fb0). No other Graphical User Interface should be running when Storyboard is started as it assumes control of the framebuffer device. This build also uses the FreeType library for font loading and rendering.

Libraries:

- libfreetype.so

TSLIB

For the ARM version a plugin is available which supports a touchscreen device through the use of tslib (libgre-plugin-tslib.so). This plugin will use the standard tslib environment variables in order to find and configure the touch device as follows:

```
export TSLIB_CONSOLEDDEVICE=none
export TSLIB_TSDEVICE=/dev/input/ts0
export TSLIB_CALIBFILE=/etc/pointercal
export TSLIB_CONFFILE=/etc/ts.conf
```

It is assumed that the touch device has been configured previously. In order to configure the touch device please run the `ts_calibrate` which is part of the `tslib` distribution or build for Linux systems.

Libraries:

- `libts.so` (only for `tslib` plugin)

Note

Storyboard requires a `libts-0.0.so.0` to be in the `lib` path to use the `tslib`-plugin. If the board has `libts-1.0.so.0` simply create a symlink for `libts-0.0.so.0` and point it at the `libts-1.0.so.0`.

```
ln -s /lib/libts-1.0.so.0 /lib/libts-0.0.so.0
```

MTDEV

Setting up multi-touch with Storyboard is relatively straight forward. In most cases you only need to pass the multi-touch device to Storyboard using the following option:

```
sbengine -omtdev,device=[path to touch device] your_app.gapp
```

Determining the Touch Device

Typically the touch device is labelled as `/dev/input/touch0`. You can easily verify by using the `cat` utility:

```
cat /dev/input/touch0
```

Executing the above command and touching the screen should present characters to the terminal verifying you have the right device.

Input Bounds Parameters

Usually the `sbengine` `mtdev` plugin queries and receives the correct information from the touch device. Sometimes, due to manufacturer's not following the correct specifications etc., the touch device does not get initialized correctly and the touch coordinates are off. In that case you need to use the `-omtdev,calibrate` option in conjunction with the `mtcalib` application to acquire the bounds parameters for screen calibration. To do that you will need to follow these steps:

1. Get the `mtcalib` sample from the Crank public SVN. Importing sample projects from Crank's public SVN can be found [here](#).
2. Run `sbengine` with the `mtcalib` sample:

```
sbengine -omtdev,calibrate mtcalib.gapp
```

3. Pass the bounds information that you acquired from the `mtcalib` application to your original `sbengine` commandline:

```
sbengine -omtdev,device=/dev/input/touch0,bounds=0:0:32000:32000  
your_app.gapp
```

Libraries:

- `libmtdev.so`

SBIO

If your application uses the Storyboard IO library then the Linux kernel must have SysV message queue support.

Microsoft WinCE, Compact7 win32, armle

Requirements:

Alpha blending must be compiled in to the target WinCE image

To utilize the -v verbosity options, a console must be compiled in to the target WinCE image

Use command line option to pass SB_PLUGINS directory since WinCE does not support environment variables. E.g.: sbengine -omodel_mgr.plugin_path="/Temp"

liblua.dll must reside in the same directory as sbengine.exe, due to the lack of a PATH environment variable

Libraries:

- libgwes.dll Must be built into target WinCE image

Yocto Jethro Linux kernel (3.14) OpenGL, FBDEV, armle

Using the Yocto Jethro Linux kernel (3.14) with the boundary devices branch for the nitrogen6x you might encounter flickering graphics.

Requirements:

```
echo 10 >/sys/devices/soc0/backlight_lvds0.17/backlight/backlight_lvds0.17/brightness
```

Chapter 26. Working with Storyboard Lite Platforms

What is Storyboard Lite

Storyboard Lite is a feature reduced version of the Storyboard Engine. It is designed to fit on to platforms that have smaller memory and storage offerings. Storyboard Designer can still be used to work with a Storyboard Lite platform as the main engine is the same as the engine that runs on desktop machines.

Storyboard Lite Installers

Each platform that supports the Storyboard Lite configuration has an installer associated with it. For more information on which installers are available, contact info@cranksoftware.com.

Design Considerations

One of the new features in Designer is the ability to choose a feature set to work with for the Designer project. When creating a project for Storyboard Lite, users are encouraged to select the "Storyboard Lite" feature set. This selection can be done on the third page of the "New Project" wizard. This will reduce the number of actions that you can choose from in the "New Action" dialog. Some of the actions that are removed are the Lua callback action and any of the advanced screen transition actions. These actions have been removed due to the amount of memory that they can potentially use during execution. To add dynamic capabilities to your application a new action called the "C Callback" action has been added. This action allows the user to specify a C function to be called at runtime in response to an event. For simulation purposes on desktop machines, the C callback plugin will invoke Lua functions from Lua scripts in the project that have the same name as C function that is being called.

Another design consideration is to avoid the usage of rotated or scaled images. Rotating or scaling an image will require more memory by the engine to draw the image at runtime.

Storyboard Designer now has a metrics view which can be used to estimate how much memory and storage are required by an application. Also, there is a Resource Export Configuration editor that can be used to control how resources are exported by designer. Using this editor it is possible to reduce the amount of memory needed by fonts and images by drawing them directly from storage on the device. This will however increase the amount of storage needed to store these resources.

Chapter 27. Working with Mobile Platforms: Android and iOS

Exporting to iOS Devices

When exporting to an iOS Device all related files and available plugins are packaged into a single application for the device. Since this is the case there is no need to set up environment variables or specify runtime options on the target, since this will all be done in Storyboard Designer when you export the application.

To configure an iOS device to run your apps you need to set up a few things from Apple first:

1. Xcode
2. iOS developer account
3. A code signing certificate
4. The device ID of the Apple products you will be running on
5. The identifiers of the applications you will be making
6. A provisioning profile

Xcode

Xcode no longer includes the command line tools and need to be downloaded separately from the app store.

iOS Developer Account

To obtain a developer account you will need to go to <https://developer.apple.com>, click on iOS Dev Center, then click register. After that follow their step by step instructions and you will be a registered iOS Developer.

Code Signing Certificate

Log into your iOS Development account and click on the iOS Provisioning Portal link at the right hand side of the page. In your Provisioning Portal, click Certificates in the left tab bar. If you do not have a certificate, there will be an option to submit one. To do this you will need to make a Signing request. You can do this by launching Keychain Access, located at /Applications/Utilities. Then go to Keychain Access > Certificate Assistant > Request a Certificate From a Certificate Authority. Enter your email address and your name, then select save to disk. Once that has been generated, go back into the certificates page of your provisioning profile and submit it for approval. After the certificate gets approved, download the file and open it, and it will be added to you keychain access.

Device IDs

Log into your iOS Development account and click on the iOS Provisioning Portal link at the right hand side of the page. In your Provisioning Portal, click Devices in the left tab bar. Click the add devices button

Application IDs

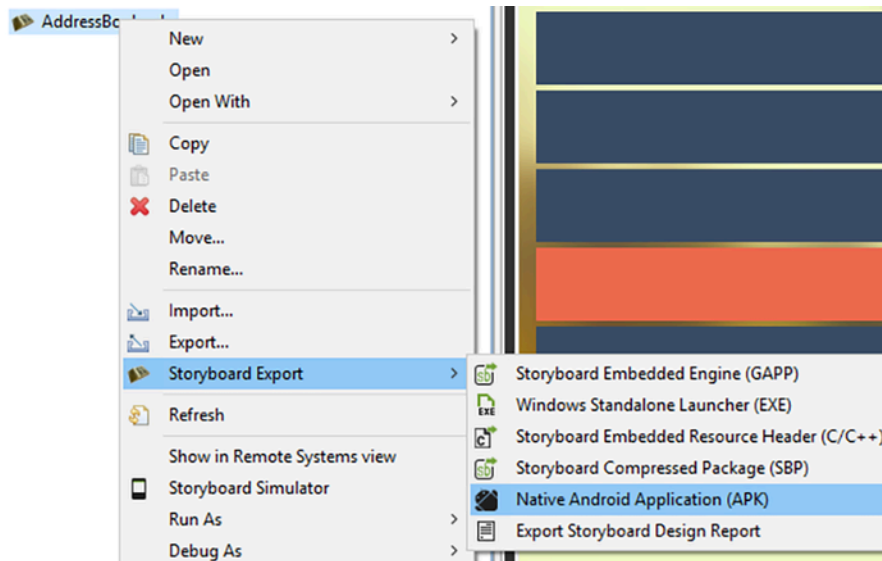
Provisioning Profile

Adding Extra Libraries for iOS

Exporting to Android Devices

Exporting as an Android application will create an Android package that is suitable for use on Android devices. The packages generated will only work on devices running *Android version 2.3.3 and above*.

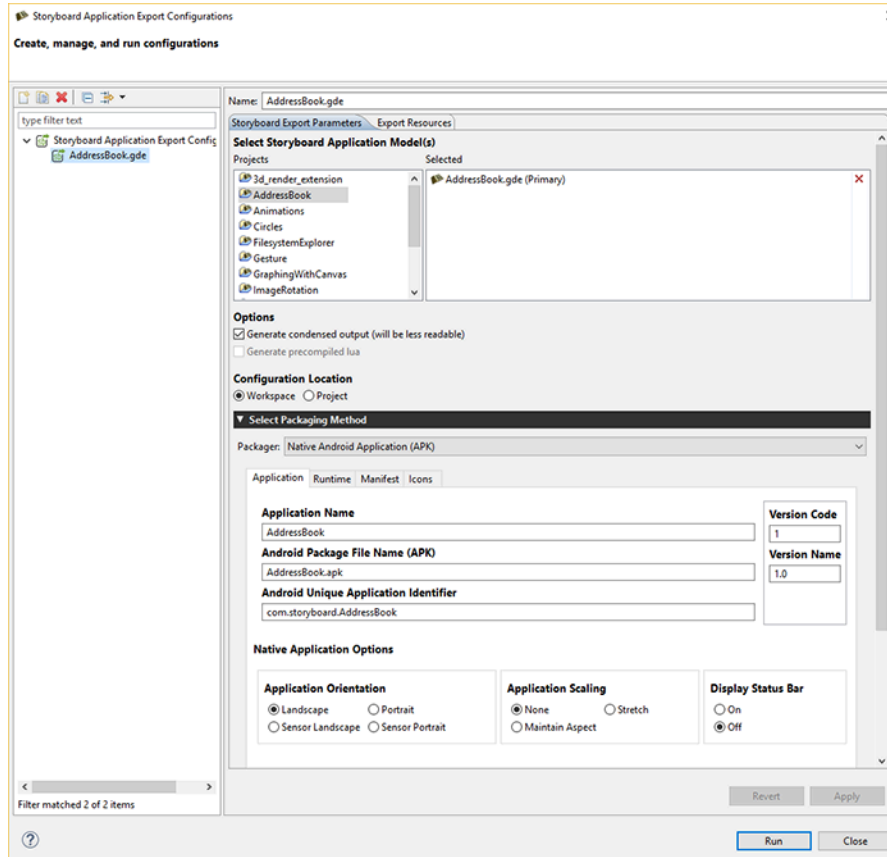
To create an Android package, right-click the project .gde File > Storyboard Export > Export as Native Android Application.



From the export file selection dialog, select the Storyboard application file (*.gde) that you want to export. Choose appropriate names for the application name, Android package file name, and the package name. Select the directory you want to export to and options for application orientation and fullscreen. For Android devices version 4.4 and newer, the fullscreen option uses Android's sticky immersion fullscreen.

Note

Currently, without rooting an Android device, there is no way to disable the bottom bar for some Android 3.0+ devices.



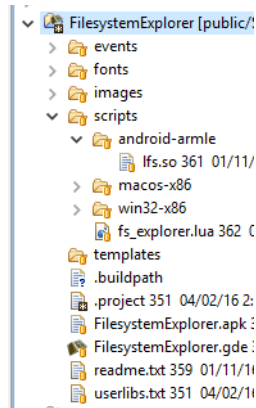
Select the Runtime tab to set Storyboard runtime options (see Storyboard Engine Options for list of available runtime options). Select the Manifest tab for advanced options to modify the default Permissions, Manifest file or Keystore settings (optional). Select the Icons tab for selecting the icons for the application. If you don't need to alter the settings, click Apply then Run to use the defaults.

By default the Android application package file (APK) will be created in the project's directory. To transfer this application package to an Android device, simply copy the package onto a USB or SD card.

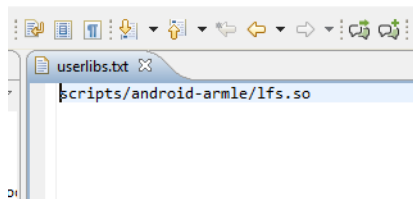
Adding Extra Libraries for Android

Sometimes a user creates a Storyboard app that requires a library that isn't included with the Storyboard Runtime. When exporting for Android we need to tell the exporter which libraries to preload. We do this by giving the exporter a text file with a list of libraries. Make sure that the paths to these libraries are relative to the Storyboard app's directory. As well, the order of the libraries in the list determine the order they get loaded in, therefore if one of the libraries has a dependency on another library make sure to have the dependent library higher in the list.

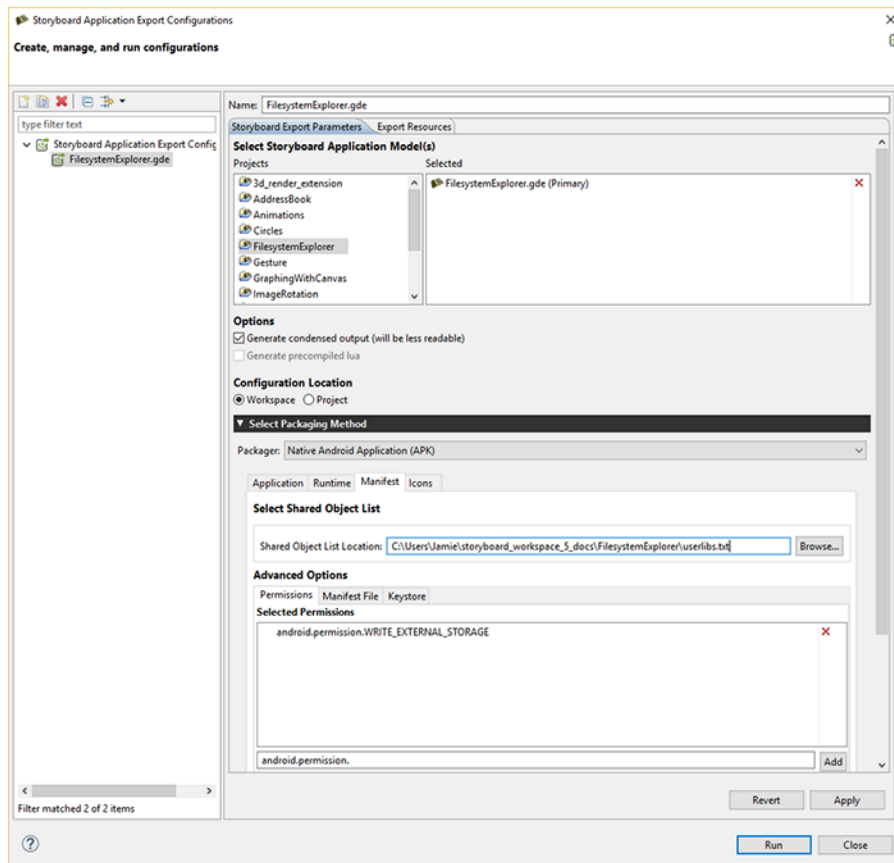
The example we'll use is modifying the FilesystemExplorer app from the Crank Software public repository. In order for this app to work on Android we'll need to include the LuaFileSystem module that's been compiled for Android (the lib's name is lfs.so). We'll add this file in scripts/android-armle.



Create a text file, which we'll call `user_libs.txt`, with the following contents:



When exporting the app make sure to include the path to this file and hit finish.



Now the app can make use of the functionality included in the new library.

Storyboard Lua Android Integration

On Android target platforms Storyboard provides an additional level of platform integration. In order to access the native Java service API on Android platforms Storyboard has incorporated the LuaJava [<http://www.keplerproject.org/luajava/>] module to provide a bridge from Storyboard Lua script functions to the Android Java API.

Access to the LuaJava bridge is through the `luajava` Lua variable. On non-Android platforms, this variable will not be defined and this can be used to provide alternate or simulated behavior.

```
function my_callback(mapargs)
    if(luajava == nil) then
        print("LuaJava bridge not available")
        return
    end

    -- LuaJava available for use ...
end
```

The general mapping of standard Lua/Java types such as strings and numbers is handled transparently so that Lua strings can be used in Java constructors and methods in the same way that the Java String class would normally be used and similarly for Lua numbers and vice/versa.

When a Lua variable is created that is a reference or proxy to a Java object, then access to the methods of that object are performed using the colon (`:`) notation with the Lua variable, `lua_variable:method_name()` notation. When accessing static member variables of an object, this can be performed using the traditional dot (`.`) notation `lua_variable.member_variable_name`. This is further demonstrated in the examples shown below.

In order to access a nested Java class for instantiation or binding, the dollar sign (`$`) must be used as a separator. For instance, if the Java class Bar is a nested class of Foo, then binding would work as follows: `luajava.bindClass("Foo$Bar")`. This is further demonstrated in the examples below.

A description of the complete Android Java API is beyond the scope of this document. For a complete coverage of the Android API refer to <http://developer.android.com/reference/packages.html>. Depending on the functionality that your application is going to access, there may be additional restrictions that must be explicitly declared in the `AndroidManifest.xml` file. Permissions can be added in the Advanced Options section when exporting your Android project. The `android:debuggable` option has been changed to false by default. To change this, you will need to use your own custom manifest file. Export your manifest file to view it by clicking the Export button under the Manifest File tab. You can make changes to this file and then select it as a custom manifest file when exporting to ensure the manifest file is setup the way you want it to be.

Within the Android environment the Storyboard Engine execution takes place outside of the main Android/Java event loop. When integrating with the Android API's developers should always consider that they are using the Android API as if they were executing in a background thread and act accordingly. This may require the creation of additional `Looper` message event handlers if callback event handlers are being used. For more information on Android process model and multi-threading considerations, refer to the Android documentation: <http://developer.android.com/guide/components/processes-and-threads.html>.

Android Lua Java API

The mapping of Lua referenced objects to Android Java objects is relatively straightforward. All of the API functionality is accessed via the `luajava` Lua global variable. This variable provides four functions

that can be used to access and manipulate standard Java objects and one variable that provides the Android Activity that is required.

`luajava.newInstance(className, ...)` This function creates a new Java object based on the fully qualified class name. Any additional parameters that are provided are passed through to the standard Java constructor.

The return value is a Lua variable that is a proxy to the Java object or nil if the class could not be instantiated.

```
-- Create an instance of a Java string tokenizer
local strTk = luajava.newInstance("java.util.
StringTokenizer", "a,b,c,d", ",")
while strTk.hasMoreTokens() do
    print(strTk.nextToken())
end

-- Create a new Android Intent object (unpopulated)
local intent = luajava.newInstance
("android.content.Intent")
```

`luajava.bindClass(className)` This function creates a reference to a Java class based on a fully qualified class name. This is different from `newInstance()` in that a new Java object is not created and the constructor is not invoked, but simply a reference to the class is returned. Use this when you need access to static fields or methods of a Java object.

The return value is a Lua variable that is a proxy to the Java Class object specified or nil if the class could not be found.

```
-- Get the current system time
local sys = luajava.bindClass("java.lang.System")
print ( sys.currentTimeMillis() )

-- Parse a string into an Android Uri
local uriClass = luajava.bindClass("android.net.Uri")
local phoneURI = uriClass.parse("tel:6135951999")
```

`luajava.new(classObject, ...)` This function is similar to the `newInstance()` function but rather than taking a fully qualified class name it takes an existing Class reference, generally obtained from calling `bindClass()`. Additional parameters can be passed to the Java constructor..

The return value is a Lua variable that is a proxy to the Java object or nil if the class could not be instantiated.

```
-- Create a new string instance
str = luajava.bindClass("java.lang.String")
strInstance = luajava.new(str)
```

`luajava.createProxy(interfaceNames, luaObject)`

If a Java API requires an interface to be implemented or provided as a set of callbacks, then it is where the `createProxy()` function can be used. The *interfaceNames* parameter is a comma separated list of fully qualified Java interfaces that will be implemented by the Lua variable *luaObject*. The names of the interface methods must be present in the `luaObject` variable.

The return value is a Lua variable that can be passed to any function or method that requires an implementation of that interface. If the creation of the proxy fails, then `nil` is returned.

```
-- Create a Lua variable with the same interface as an ActionListener
local button_cb = {}
function button_cb.actionPerformed(ev)
    -- I would do something interesting here ...
end

-- Map the Lua variable to the Java interface
buttonProxy = luajava.createProxy("java.awt.ActionListener", button_cb)

-- Use the newly created interface instance on a Java object
button = luajava.newInstance("java.awt.Button", "execute")
button:addActionListener(buttonProxy)
```

`luajava.nativeActivity()`

All significant interaction on an Android system involves working with an `Activity` (see <http://developer.android.com/reference/android/app/Activity.html>) Storyboard applications that are deployed to Android devices run as *native activities* which is a special class of the general `Activity` that allows those applications to interact directly with the graphics context and are generally C/C++ applications rather than pure Java applications.

The return value of this function is a Lua variable that is a proxy for the `NativeActivity` Java class used by this application or `nil` if the class could not be instantiated.

```
-- Start an activity specified by a previously created Intent object
local na = luajava.nativeActivity()
if(na ~= nil) then
    na:startActivity(intent)
else
    print("No Native Activity")
end
```

Storyboard Lua Android Example

This example demonstrates how a phone call could be invoked as part of a Lua callback. In order for this example to work, the `AndroidManifest.xml` file must be changed to give permission for

```
calls to be made:  %<uses-permission  android:name="android.permission.CAL-
L_PHONE"></uses-permission>

-- Log message routine to route diagnostic messages
local function lm(msg)
    print(msg)
end

-- Call a selected phone number using the Android API
-- Input is the string number value that is to be called
local function call_phone_number(number)
    if(luajava == nil) then
        lm("No luajava Lua object")
        return
    end

    local na = luajava.nativeActivity()
    if(na == nil) then
        lm("No native activity available")
        return
    end

    local uriClass = luajava.bindClass("android.net.Uri")
    if(uriClass == nil) then
        lm("No java.lang.String object")
        return
    end

    local phoneURI = uriClass:parse("tel:" .. tostring(number))
    if(phoneURI == nil) then
        lm("No java.net.URI object")
        return
    end

    local intentClass = luajava.bindClass("android.content.Intent")
    if(intentClass == nil) then
        lm("No intent class")
        return
    end

    local intent = luajava.newInstance("android.content.Intent",
    intentClass.ACTION_CALL, phoneURI)
    if(intent == nil) then
        lm("No intent object")
        return
    end

    lm("Calling " .. number)
    na.startActivity(intent)
end
```


This example demonstrates how to create a new instance of a nested inner class of a Java class. This example gets media metadata from the `android.provider.MediaStore.Audio.Media` class, which is a nested class of `android.provider.MediaStore.Audio`, which in turn is a nested class of `android.provider.MediaStore`.

```
-- In order to pass array's to any of the Android Java API's we must explicitly cr
-- a Java array from a Lua table and this function covers that work.
function make_array(dataClass, values)
    local arrayClass = luajava.bindClass("java.lang.reflect.Array")
    if(arrayClass == nil) then
        print("Can't get array class")
        return nil
    end

    local newTypedArray = arrayClass:newInstance(dataClass, #values)
    if(newTypedArray == nil) then
        print("Can't get array class")
        return nil
    end

    for i=1,#values do
        arrayClass:set(newTypedArray, i-1, values[i])
    end

    return newTypedArray
end

function get_album_files(album_id)
    if (luajava == nil) then
        return
    end

    if (luajava.bindClass == nil) then
        return
    end
    local na = luajava.nativeActivity()

    local mediastore = luajava.newInstance("android.provider.
MediaStore$Audio$Media")
    local externalURI = mediastore.EXTERNAL_CONTENT_URI
    local columns = {}
    columns[1] = mediastore.TITLE_KEY
    columns[2] = mediastore.DURATION
    columns[3] = mediastore.TITLE

    local stringClass = luajava.bindClass("java.lang.String")
    local array = make_array(stringClass, columns)
    local where = mediastore.ALBUM_KEY .. "=?"
    local what = {}
    what[1] = album_id
    local whatArray = make_array(stringClass, what)

    local cursor = na:managedQuery(externalURI, array, where, whatArray, nil);
    local res = cursor_to_table(cursor)
```

```
    return res  
end
```

Chapter 28. Sending and Receiving Data with Storyboard IO

The Storyboard IO API, formerly known as GREIO, provides a platform independent communication API that allows inter-task, inter-process and ip-network queued message passing. This is primarily used to allow external communication with a Storyboard application.

The API provides transport delivery guarantees for messages that are placed into the queue regardless of the implementation. The maximum transport size of a message and the total queue capacity varies slightly from implementation to implementation however a 2K message size should be considered a design limit with the practical implementation limit around a 4K message payload size.

Storyboard IO communicates events over a named uni-directional channel. By default, Storyboard applications have a receive channel, named after the deployment bundle file (i.e. [bundlename].gapp). This allows backend applications to send events to the Storyboard Application. A custom receive channel or tcp port number can be specified in the command option for sbengine. The Storyboard application can transmit events on one or more channels or tcp ip address ports. The backend software will need to open the channel and listen for events. See Storyboard IO API for more details.

Storyboard IO integration with the Storyboard Engine is implemented as a plugin. It is possible to create alternate Storyboard IO implementations that take advantage of custom communication facilities available on a platform. Additionally Storyboard IO has been extended to provide abstraction of tcp ip ports in the place of channels.

The Storyboard IO plugin provides a single communication channel which clients can use to inject events into the Storyboard application. These events will be queued and dispatched in the same manner as internally generated events.

Client applications can use Storyboard IO to create their own communication channels and then receive events from that channel from the Storyboard application or from any other Storyboard IO client.

Storyboard IO Transport protocols

The Storyboard IO API layers are implemented on top of native message passing and communication APIs. In addition to host operating system's TCP protocol support, the following protocols, dependent on the operating system, are supported:

Linux	SysV message queue
MacOS	SysV message queue
QNX	QNX POSIX message queue (mqqueue server or mq server/library)

Using the SBIO_MQ_PATH environment variable you can determine which message queue technology will be used. By default the standard mqqueue server and corresponding C library mq_* functions will be used.

If the SBIO_MQ_PATH environment variable is set to point at the libmq.so library (i.e. SBIO_MQ_PATH=/usr/lib/libmq.so) then the default binding for the message queue implementation can be changed to use the mq server and the corresponding mq library functions.

If the SBIO_MQ_PATH environment variable is used, then it must be used consistently with all Storyboard IO clients and servers that want to communicate with one another.

WinCE/WinCompact7

WinCE MSMQ

Win32

MS-Mailslots

RTOS (FreeRTOS, MQX,
ThreadX, etc)

In memory queue

Connecting to a Storyboard Application

In order to communicate with a running Storyboard application the external application must first attach to the application's Storyboard IO channel. By default this channel is named after the deployment bundle file (i.e. [bundlename].gapp), however the name of this channel can be customized by specifying `-ogreio,channel=newname` as an option to `sbengine`. Alternatively, on some platforms, it is also possible to set a GREIONAME environment variable to the channel name.

Once the channel name is determined, the connection to the channel can be established with the `gre_io_open()` function. This will connect to the channel and return a handle that can be used for future communication.

Once the application has determined that no further communication is necessary the channel should be closed via the `gre_io_close()` function. The `gre_io_close` should be called in a manner that is serialized with clients using the handle returned from `gre_io_open`. For threaded applications where a receive thread is running independently, it is always a good idea to shut down and close the channel from within the thread that is receive blocked on the channel by sending it a specific termination message.

Storyboard IO Over TCP

It is possible to send events over TCP as long as the `greio-tcp` library is available on your system. In order to completely disable TCP functionality from Storyboard Engine, you may remove the `greio-tcp` library from your device all together.

A TCP connection is specified by the format of the Storyboard IO channel name.

Receiving:

`tcp://<port or channel>`

Sending:

`tcp://<IP address>:<port or channel>`

The default IP address is 127.0.0.1 (localhost). The default port is 50000.

If using a port number, the valid range is 49152..65535. If a text channel name is used, Storyboard IO will hash the string into a port value within the valid range of 49152..65535.

The following are examples of valid GREIO TCP channel names.

Receiving:

- `tcp://` (Will use default port of 50000)

- `tcp://55555`
- `tcp://my_channel`

Sending:

- `tcp://` (IP 127.0.0.1 and port 50000)
- `tcp://55555` (IP 127.0.0.1)
- `tcp://my_channel` (IP 127.0.0.1)
- `tcp://192.168.1.101:51515`
- `tcp://192.168.1.101:my_channel`
- `tcp://dev-machine101:my_channel`

As shown above, if the network is configured to assign host names to machines on the network, these can be used in place of IPv4 addresses, provided that the machine has DNS capabilities.

Sending Events to a Storyboard Application

Storyboard events contain string based names and a variable data field. For this reason the event data must be serialized into a buffer for communication. The Storyboard IO API provides the functions needed to both serialize your data and send the event. The event you wish to send must first be serialized via a call to `gre_io_serialize()`. This will allocate a serialized data buffer for your event. The event can then be sent via the `gre_io_send()` function. Once the event has been sent the buffer can be reused or freed via a call to `gre_io_free_buffer()`.

Note

Serialized buffers can be reused multiple times. The `gre_io_serialize_buffer()` function will resize or reallocate the buffer if the data being serialized is larger than the existing buffer. This is designed to cut down on repetitive memory allocation and deallocation churn.

Data parameters must be sent in order of descending alignment requirements. Example: `4u1 4u1 2u1 1s0` is good, `2u1 4u1 4u1 1s0` is not

```
gre_io_t                *send_handle;
gre_io_serialized_data_t *nbuffer = NULL;
const char              *event_data = "my event data"

/*
 * Connect to a channel to send messages.
 */
send_handle = gre_io_open("my_channel", GRE_IO_TYPE_WRONLY);
if(send_handle == NULL) {
    printf("Can't open send handle [%s]\n", argv[1]);
    return 0;
}

/*
 * Alternatively to the above 'gre_io_open()' example,
```

```
* Connect to a tcp ip address port to send messages.
*/
send_handle = gre_io_open("tcp://192.168.1.101:my_channel", GRE_IO_TYPE_WRONLY);
if(send_handle == NULL) {
    printf("Can't open send handle [%s]\n", argv[1]);
    return 0;
}

/*
 * Send a named event containing no data payload
 */
nbuffer = gre_io_serialize(nbuffer, NULL,
    "my_event_name",
    NULL,
    NULL,
    0);

gre_io_send(send_handle, nbuffer);

/*
 * Send a named event with an additional string payload
 */
nbuffer = gre_io_serialize(nbuffer, NULL,
    "my_event_name",
    "ls0 data",
    event_data,
    strlen(event_data)+1);

gre_io_send(send_handle, nbuffer);
```

Event Naming Conventions

New events can be readily defined and are not required to contain a data payload. In this case their format string and data payload will be empty values. When creating new events, it is appropriate to namespace the event definitions so that the names of events do not collide. For example, the Storyboard framework reserves the name prefix of `gre.` for user interface events, and the timer functions all generate events that are prefixed with `timer.`

The use of events is closely coupled with the declaration and operation of actions. An action can only be invoked when an event matching the action definition is received. This results in a common design pattern where an action will perform sophisticated logic in an external script or program and then signal a completion action to run once the script work is complete.

Event > Action (script) > Work > Trigger Event > Action (completion)

Serialized Data and the Event Format String

Any data associated with an event as its payload must be a linearly contiguous block of memory. In order for clients receiving the event to decode the event data memory block back into structured content a key must be provided. The format string that is provided as part of the event is this decoding key.

The format string describes how the individual bytes of event data are to be grouped together as specific data types. For example the Lua script plugin can use the format string to convert the event data memory


block into Lua variables that conform to Lua's type system. Once converted, the symbolic name for the data, provided as part of the format string, can be used to reference that particular information. Other clients, such as C or C++ programs, may not need to interpret the data symbolically but may use a language specific mechanism to convert the memory block.

The format string is relatively straightforward to create and is a series of entries formatted as `[num-bytes][signed/unsigned][numelements][][name]`. For the standard C data types the number formatting would look like:

C/C++ Type	Format String	Data Size
int8_t	1s1	1 byte
uint8_t	1u1	1 byte
int16_t	2s1	2 bytes
uint16_t	2u1	2 bytes
int32_t	4s1	4 bytes
uint32_t	4u1	4 bytes
int64_t	8s1	8 bytes
uint64_t	8u1	8 bytes
float (IEEE754 float)	4f1	4 bytes
char *	1s0	Length of string including nul terminator

So, if you were transmitting the following C/C++ structure you would presume that the bitwise memory layout would be:

```
struct event_data {
    uint32_t a;
    int16_t b;
};
```




i
You would use a format string of 4s1 a 2u1 b to describe the event

You would use a format string of `4s1 a 2u1 b` to describe the event.

The symbolic field descriptions `a` and `b` are optional but highly recommended. They are used to give the data symbolic representation for clients that can't access the memory bytes directly (such as Lua). These symbolic field descriptions do not need to match the names of the structure member variables so an equally valid format string for the above structure might have been `4s1 angle 2u1 magnitude` if `angle` and `magnitude` were better symbolic names for what the data represents.

The format string provided describes the linear memory layout of the event data. Consequently it is important that the format string take into consideration any alignment or padding inserted when the memory block is created. Consider changing the order of the members in the sample structure:

```
struct event_data {
    uint16_t b;
    int32_t a;
};
```



i
4 byte types are aligned on 4 byte boundaries

without any additional guidance to tell it otherwise the C/C++ compiler is going to create storage for the structure such that members are aligned to boundaries that match their data types (ie 4 byte types are aligned on 4 byte boundaries). This can create holes in the memory layout.

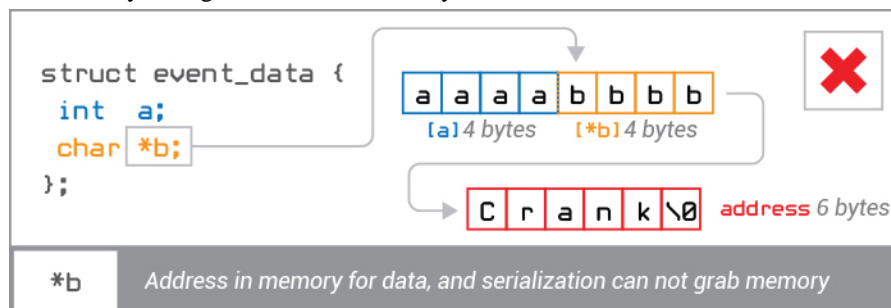
Here the 32 bit/4 byte member of the structure `a` comes after `b` but there are two additional bytes of padding inserted to ensure `a` starts on a 4 byte memory boundary. Since the format string must describe the linear memory layout for clients, we would have to change the format string to accomodate the extra padding inserted for alignment and the format string would be `2u1 b 2u1 pad 4s1 a`. It is always good practice to avoid wasting extra bytes on padding alignment, but Storyboard does not perform any sort of interpretation. In fact providing a format string that mis-aligns data can result in unpredictable behaviour.

Event data frequently will contain string information. Strings are simply an array of one byte values with a nul terminating character, often represented as a pointer to this memory (i.e. `char *`). All text in Storyboard is encoded using UTF-8 so this statement applies regardless of the text values being represented. If an event's data payload is composed of a single string, then the bytes of that string can be used directly as the block of memory:

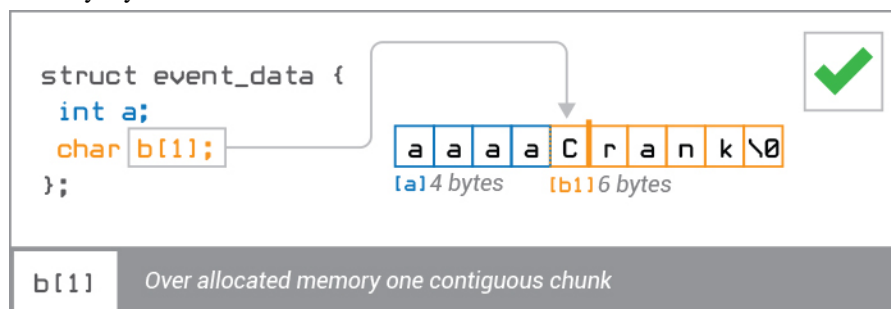
```
char * event_data = "Crank";
```

The `event_data` variable, as a pointer to memory, can be used directly and the format string used to represent it would be `1s0 msg`, where `msg` can be whatever symbolic name makes sense.

It is not possible to send C/C++ structures that contain strings as members if those variables are declared as pointers because the memory of the structure (including the string) is not linear and the event data must be a linearly contiguous block of memory.



However it is possible to include strings within structures by either fixing their size which will force their storage to be included as part of a structure block, i.e. `char msg[20]`, or if only a single string is being sent then the C/C++ idiom of overallocating the size of a structure can be used to force a linear memory layout:



The C/C++ code technique for using this would look something like:

```
struct event_data {  
    int      a; //Assume 32 bit integers  
    char     b[1];  
};
```



```
};

struct event_data *ed;

//Allocate the memory for the base structure and the string to follow it
ed = malloc(sizeof(*ed) + strlen("Crank"));

//Assign the values to the allocated structure
ed->a = 2018;
strcpy(ed->b, "Crank");           //nul character is accounted for by b[1]
```

In this case the data can now be described with the format string `4s1 a 1s0 b` where the `1s0` is shorthand for nul terminated strings and would be equivalent to saying `1s6` where 6 is the number of bytes in the string "Crank" plus the nul terminating character.

Setting Application Data

The Storyboard IO plugin provides the capability to set application variable values using the Storyboard IO API, allowing external client programs to change data dynamically.

Clients can use the `gre_io_add_mdata()` function to serialize each variable value that is to be set. As values are added to the serialized buffer, it will be grown until it reaches a maximum size for the transport, at which point the `gre_io_add_mdata()` function will return -1 indicating it is full. The data can be then sent by using the `gre_io_send_mdata()` function which will send the change request to the Storyboard IO plugin and set the appropriate values in the Storyboard application.

Note

Data must be sent in order of descending alignment requirements. Example: `4u1 4u1 2u1 1s0` is good, `2u1 4u1 4u1 1s0` is not

```
gre_io_t                *send_handle;
gre_io_serialized_data_t *md_buffer = NULL;
uint32_t                x;
char                    *ptr;
int                     ret;

/*
 * Connect to the application channel
 */
send_handle = gre_io_open("my_channel", GRE_IO_TYPE_WRONLY);
if(send_handle == NULL) {
    printf("Can't open send handle [%s]\n", argv[1]);
    return 0;
}

/*
 * Add some values to be set in the data manager
 */
ptr = "my string";
ret = gre_io_add_mdata(&md_buffer,
    "Test.String",
```

```
"ls0",
ptr, strlen(ptr)+1);

x = 1;
ret = gre_io_add_mdata(&md_buffer,
    "Test.Number",
    "4ul",
    &x, sizeof(uint32_t));

/*
 * Send the data to be set in the application.
 */
gre_io_send_mdata(send_handle, md_buffer);
```

Receiving Events from a Storyboard Application

In order to receive events from a Storyboard application, a client program must first create a receive communication channel using the `gre_io_open()` function. This function takes the name of a channel to create and the mode in which to open the channel, for reading or writing. Receiving clients must open it for reading. To receive events from a tcp socket, `gre_io_open()` needs to specify a tcp URI and optional channel name / port number. To specify the use of a tcp port, a tcp uri `"tcp://"` needs to prefix the optional Storyboard IO channel name when opening the read only channel. For example: `gre_io_open("tcp://my_channel", GRE_IO_TYPE_RDONLY)` Internally a channel name string is hashed into a tcp port number. The tcp sender should be configured with same channel name or port number.

Once the communication channel is created, then the client program then must call `gre_io_receive()` in order to receive and process events.

The client communication channel can be created in either a blocking or non-blocking mode. By default the `gre_io_receive()` function will not return unless there is an event available or an error has occurred.

Once an event has been received the data can be unserialized into its standard components using the `gre_io_unserialize()` function.

```
char    *name = (char *)arg;
gre_io_t *rhandle;
gre_io_serialized_data_t *buffer = NULL;
int ret;
char *revent_name;
char *revent_target;
char *revent_format;
uint8_t *revent_data;
int offset, i, rnbytes;

rhandle = gre_io_open(name, GRE_IO_TYPE_RDONLY);
if(rhandle == NULL) {
    printf("Can't open IO channel %s\n", name);
```

```
        return 0;
    }

    printf("Waiting on channel [%s]\n", name);
    while(1) {
        ret = gre_io_receive(rhandle, &buffer);
        if(ret < 0) {
            printf("Problem receiving data on channel [%s]\n", name);
            break;
        }

        rnbytes = gre_io_unserialize(buffer,
                                    &revent_target,
                                    &revent_name,
                                    &revent_format,
                                    (void **)&revent_data);
        printf("Event Received [%s] on channel [%s]:\n", revent_name, name);
        printf(" Event Target: [%s]\n", revent_target);
        printf(" Event Format: [%s]\n", revent_format);
        printf(" Event Data (%d bytes):\n", rnbytes);
    }

    gre_io_close(rhandle);
```

Debugging Storyboard IO

Debugging egress (Storyboard Engine sending to back-end)

Table 28.1. No events are being received

Possible Issue	Resolution
Receive channel is not open	Launch Storyboard engine with verbosity (-v) and you see the following warning on sending an event WARN [0.135]:Can't open greio channel channel_name , then the channel is not open. If receiver is in C code ensure the gre_io_open returned successfully. If communicating with another Storyboard application ensure the receiving application was launched with the expected channel name specified with command line parameter -ogreio,channel=channel_name
Channel name mismatch	If there is a mismatch on channel name you will not receive messages. One way to verify is to stop the receiver and use utility iorcv. Just launch iorcv with the same channel name as your receiver: ./iorcv [-s] channel_name
System Permissions	On some systems, system permissions can block the IO communications. Try running with administrator privileges, if the above steps have not resolved the

Possible Issue	Resolution
	issue. You can use iogen and iorcv to verify permissions.

Table 28.2. A particular event is not being received

Possible Issue	Resolution
event not sent	Launch Storyboard Engine with verbosity level 4 (-vvvv). Verify in the log that the action or Lua code to trigger the event was executed. If you see <code>WARN [0.135]:Can't open greio channel</code> and you have already verified channel is open, then verify the channel name is correct in send event call
Event not received	Most common scenario is a event name mismatch. Using iorcv you can verify the name that is being sent, ensure it matches what the receiver expects.

Table 28.3. Event data not correct

Possible Issue	Resolution
Format string not correct	If the data values are not received as expected it likely the format string either does not match the data. Ensure the format size matches the size of the data. If specifying a name to the parameter ensure the names are the same.
Data not aligned.	Data must be aligned to system data alignment. Particular problem sizes are 1u1, 1s1, 1s0 (strings). Data must be sent in order of descending alignment requirements. Example: 4u1 4u1 2u1 1s0 is good, 2u1 4u1 4u1 1s0 is not

Debugging ingress (Storyboard Engine receiving from backend)

Table 28.4. The trouble shooting steps for egress Storyboard apply in this scenario, so see the above steps.

Issue	Resolution
Event not received	Launch Storyboard Engine with verbosity level 4 (-vvvv) you will see the following event trace if successful. <pre> EVENT [111.811]:GREIO received event [MilleniumFalconSpeed] 2 bytes EVENT [111.811]:IO: Queue [1] MilleniumFalconSpeed EVENT [111.811]:IO: Dispatch [MilleniumFalconSpeed] ACTION [111.811]:ACTION: Invoke [MilleniumFalconSpeed]->[gra.lua] on app [app] If </pre>

Issue	Resolution
	the last two messages are not displayed Storyboard received the event but could not find a handler for the event.

Storyboard IO Utilities

Included with Storyboard are some command line utilities that can be useful tools when working with and configuring Storyboard IO. These utilities provide a thin layer on top of the Storyboard IO C API and can be used to verify that Storyboard IO is working properly on your platform.

iogen

The `iogen` utility is used to generate Storyboard IO events from the command line. The utility's command line arguments closely mirror the arguments that would be provided to the `gre_io_open` and `gre_io_send` Storyboard IO API functions. Running `iogen` without any parameters will show a usage message:

Note

Data must be sent in order of descending alignment requirements. Example: `4u1 4u1 2u1 1s0` is good, `2u1 4u1 4u1 1s0` is not

Usage:

```
./iogen channel_name [event_target event_name size data [size data]...]
```

For example:

Send a 'gre.quit' event to a client on channel 'sb'

```
./iogen sb no_target gre.quit
```

Send a 'gre.press' event (int button, timestamp, subtype, x, y, z, id, spare) @ 100,150 to a client on channel 'sb'

```
./iogen sb some_target gre.press 4u1:button 0 4u1:timestamp 0 2u1:sub-  
type 0 2s1:x 100 2s1:y 150 2s1:z 0 2s1:id 0 2s1:spare 0
```

Send a 'progress' event with an integer field 'percent' containing the value 50 on a channel 'sb'

```
./iogen sb no_target progress 4s1:percent 50
```

Send a 'greio.verbosity' event with an integer field 'verbosity' containing the desired level of engine debugging verbosity

```
./iogen sb no_target greio.verbosity 4s1:verbosity 4
```

Send a 'cluster_update' event with array data named 'tire_pressures' of 30,45,17,25 on channel 'sb' Note the leading ',' in the data, this is needed to differentiate small number from characters in a string. This can be omitted when sending numbers in anything other than the `1sN` format.

```
./iogen sb no_target cluster_update 1s4:tire_pressures ,30,45,17,25
```

Note

In this case the raw data representation of these numbers will be delivered as a string in the event payload. Use the `string.byte` function in Lua to decode the data into numerical format.

The `iogen` utility can also be used to set variables in a Storyboard application. To set a variable, the `event_target` parameter should contain the fully qualified path for the Storyboard variable and the `event_name` parameter should contain the SBIO event `greio.iodata_set`. For example:

Set an integer application variable 'progress' with a number (50)

```
./iogen sb progress greio.iodata_set 4s1 50
```

Set the text variable 'myvariable' on the control 'mycontrol' on the layer 'mylayer' with a string (Hello)

```
./iogen sb mylayer.mycontrol.myvariable greio.iodata_set 1s0 "Hello"
```

The definition and format of standard Storyboard events such as `gre.press` and `gre.release` can be found in the Storyboard header file `iodefs.h`.

iorcv

The `iorcv` utility is used to receive Storyboard IO events. This utility takes an input channel name as a command line parameter and prints the events it receives. Running `iogen` without any parameters provides a usage message:

Usage:

```
./iorcv [-s] channel_name
```

By default, `iorcv` will loop around receiving messages until the program is terminated. By specifying `-s` you can cause `iorcv` to exit once it has received a single message.

When a message is received, a summary of the event contents is printed to the output:

```
./iorcv my_channel
```

```
Waiting on channel [my_channel]
```

```
Event Received [my_event_name] on channel [my_channel]:
```

```
Event Target: [no_target]
```

```
Event Format: []
```

```
Event Data (0 bytes):
```

This would be the response to an event generated by `iogen` with the following arguments:

```
./iogen my_channel no_target my_event_name
```

```
Connecting to Storyboard IO channel [my_channel]
```

Chapter 29. Optimizing Your Storyboard Application

Measuring Performance

Using the Storyboard `logger` plugin it is possible to capture metrics detailing various aspects of a Storyboard applications performance. These metrics include screen, layer and control redraw times, action execution times and general event processing times. If a performance log file is captured as and saved with the file extension `.plog` (for performance log) then Storyboard Designer will automatically recognize it and open up a log file viewer that provides an organized display and base analysis of the performance events.

For more information on options to configure and control the performance monitoring of the engine, refer to the Logger plugin section of this document and the `gra.perf_state` action.

The Storyboard Embedded Engine runtime also provides a number of API functions that can be used at runtime to extract and display performance information.

<code>grd_fps</code> (string, 1s0)	The frame rate of display updates averaged over the last 5 seconds of display. This variable is only created and filled in if the <code>-os-screen_mgr, fps</code> option is passed to the Storyboard Engine.
------------------------------------	---

Storyboard display updates are entirely event driven, so unless the application that is being run is continuously changing content or generating redraw events such as is frequently done by benchmarking applications, this value may not reflect the true drawing performance of the system.

<code>gre.env("mem_stats")</code> (Lua)	On systems where this information is available this returns the amount of process and heap memory that the Storyboard Engine is using.
---	--

<code>collectgarbage("collect")</code> (Lua)	This is a Lua API call and will identify how much memory the Lua script interpreter is consuming. This will be a subset of the information returned by <code>gre.env()</code> .
--	---

This sample demonstrates how a periodic Lua script can be used to extract the local FPS The FPS value is stored as a string variable and may not exist until enough frames of data have been generated to derive a value.

```
-- Take a snapshot of the current execution metrics
function snapshot_metrics()
    local fps = gre.get_value("grd_fps") or 0
    local mem = gre.env("mem_stats")
    local lua_mem = collectgarbage("count")

    --Normalize, not all systems have all data
    if(fps ~= nil) then
        print(string.format("FPS   : %d", tonumber(fps)))
    end
    if(mem.process_used) then
        print(string.format("MEM   : %d", mem.process_used))
    end
end
```

```
end
if(mem.heap_used) then
    print(string.format(" HEAP: %d", mem.heap_used))
end
print(string.format(" LUA : %d", lua_mem))
end

-- Set up a periodic timer to snapshot execution metrics every 1s
function init_metric_snapshot(mapargs)
    gre.timer_set_interval(snapshot_metrics, 1000)
end
```

Action Execution Performance Considerations

All actions are executed within the context of an event delivery and as such their execution will have an impact on the overall throughput and responsiveness of the system. In particular with Lua scripts, it is important to limit the length of time that functions take to perform their work or to separate lengthy operations into separate tasks, threads or processes depending on the operating environment being used. The performance logs will provide a detailed account of how long various actions take to execute, but within your Lua scripts a profiling tool such as Lua Profile [<http://lua-users.org/wiki/ProfilingLuaCode>] can provide additional insight into your script execution.

The screen manager listens for data changes and checks the state of controls to determine when the display needs to be refreshed. In general practice the screen manager throttles data updates in a way that batches changes together to prevent visual flicker and excessive update of the display (in particular in situations where the display is not double buffered). However, there may be sequences of events or data changes that do not occur atomically and may result in excessive work to be performed and consume un-necessary CPU cycles. In these situations when multiple events are going to be changing data values, moving controls, or generating more events which would cause the display to be updated, and they are known to occur in a particular sequence, it is advisable to hold the screen manager updates until all changes have been made. Once modifications are complete the screen manager can be released and the display updated is needed. The actions are as follows:

1. `gra.screen.hold`
2. `gra.screen.release`

Choosing the Right Image Format(s) Bit Depth

When creating an application the developer must define the target system screen resolution and color depth. This color depth information is used internally to decide how to create and render display elements in an efficient manner. When adding images to the user interface it is always preferable to create them in the desired color depth. If the application will be running in 16bit color then the most efficient image to render will be a 16bit image. If alpha blending/transparency is not required when this image is rendered then it is advisable to create images in the application color depth or at least remove the alpha channel in the image.

Framerate (Frames Per Second)

Selecting an appropriate framerate for your screen transitions and animations will depend on your target system. You may think that selecting a higher framerate will make your animations run smoother, however, if your system can't keep pace with the selected framerate, Storyboard Engine will drop the frames it can't

display in a timely manner. This will result in the engine having to do more work to achieve a lower framerate than intended and will look worse than originally setting a lower framerate that the target could handle.

A framerate of 14 frames per second will look good for the majority of simple animations. The results may vary, though, depending on what is being animated, how long it is being animated for, and what the content beneath the animated element is composed of. The best plan is to evaluate your design and animations on your target hardware, and tune your settings appropriately.

Scaling Images

If you are only ever going to load an image once in your application don't scale the image, this is a performance hit at image render time. It's far better to use your favorite image editor and resize the image to exact size you intend to use it and turn the scale flag off.

Reducing Output Verbosity

Increasing the verbosity on `sbengine` is insightful when trying to track down behavioral issues and to gain a better understanding of the system behavior. However, don't forget to turn off the verbosity for release since the process of outputting diagnostic messages to a console or serial terminal can cause significant slowdown due to the limited bandwidth of the output devices.

Adjusting Engine Rendering Options

The Storyboard Engine provides a number of different global rendering defaults that can be adjusted via command line options at execution time.

If your application contains a number of rotated images, then the `-orender_mgr,quality` option can be used to trade between higher execution performance (0) and a better visual interpolation (2)

If your application is using an OpenGL renderer, then the `-orender_mgr,multisample` option can be adjusted to favour less GPU consumption with less anti-aliasing (0) or choose a smoother visual presentation but longer to render (4 or more).

Managing Resource Memory

By default `sbengine` uses as much memory as it requires to load all the assets that the application requires (images, fonts, scripts,...) but this can be tuned to save memory. Here are some options to help with this.

- Remove any unused plugins from the plugins directory if you are simply setting a directory for the `SB_PLUGINS` environment variable. The plugins that are available and being loaded will be shown by passing the `-i` option to the **sbengine** command line utility.
- Set the **resource_mgr** options for image and font cache to appropriate values. Remember the caches must be large enough to fit all the images and fonts for your most resource intensive screen.
- Use the Load Scaled flag in image render extension options if you are loading a scaled version of an image (e.g., an image thumbnails screen). If you are only ever loading the image once you should resize the image before deployment to avoid the runtime cost of image scaling.

OpenGL Scene Graph Optimization

The new OpenGL scene graph introduced in Storyboard 6.0 turns the old synchronous immediate mode OpenGL render manager into an asynchronous deferred OpenGL renderer. The new rendering paradigm allows the Engine to re-sequence, sort and combine GL calls in ways that are more efficient for both the CPU and GPU. For example, with the ability to sort, the depth buffer can be leveraged for all rendering elements on the current screen, which when combined with separate opaque and alpha operation ordering allows the engine to use hardware accelerated depth testing to greatly reduced overdraw on the GPU. This is functionality that should improve an application's performance without any change to the application itself.

The OpenGL scene graph also adds support for batching for some common elements and state transitions. This allows for previously separate elements with each their own expensive GL draw calls to be combined together based. This has the effect of greatly reduced the overall number of GL draw calls in particular. Currently fills can be batched, along with glyphs and images for which we leverage our image atlas, that allows us to access multiple images or glyphs from a single texture.

To maximize the benefit of batching, an application design should try to keep as many common classes of rendering elements together within the Z-order of the design to help improve the probability that they will be batched together. For example try to avoid a scenario where controls with single render extensions are organized (in a stacked Z order) as *image, fill, image, fill* a more ideal scenario would be (where possible): *image, image, fill, fill*.

In order to offer some additional configurability for design scenarios where the new rendering optimizations are not aligned with the application design, it is possible to control the batching and sorting behaviour with the engine options **-orender_mgr,no_batch** or **-orender_mgr,no_z_sort**.

The other configuration option that can play a significant role in rendering performance are the resource manager options that control the size of the font and image cache. The size of this cache plays a significant role in how much content can be batched together for draw operations. The **-oresource_mgr,image_block_size={-1|0|X}** and **-oresource_mgr,font_block_size={-1|0|X}** options can be use to configure the size and behaviour of the caches generally the larger the block sizes, the great chance things will be batched with a large potential gain on performance at the cost of a potential increase memory usage.

Chapter 30. Extending Storyboard Functionality

Storyboard, in its default configuration, provides enough standard functionality to satisfy most application development requirements. When additional functionality is required, there are several ways in which users can choose to extend the behavior of both Storyboard Designer and Engine.

- Custom Lua modules provide a way to integrate access to existing C/C++ libraries. The Lua extension interface is a well documented and standard way to add new functionality to the Lua scripting environment. Storyboard includes a sample demonstrating how this integration occurs in the Lua Custom Module sample. If event based communication using Storyboard IO is not suitable for the application, then this is a preferred route for accessing external data required by the user interface.
- The Storyboard SDK is an additional product that compliments the base Storyboard with the necessary API documentation and samples that allow users to create their own Storyboard Engine plugins. Storyboard plugins can be used to contribute new event sources, actions or render extensions. Using the Storyboard SDK one can also interact directly with the Engine's logging, performance metrics and data manager to perform various introspection activities at runtime.

User Defined Action Templates

When the Storyboard SDK is used to contribute new actions, it is desirable to have those actions available within the Storyboard Designer environment so that application designers can invoke the actions and configure their properties. To do this, an action template file describing the name and properties of the action can be created.

Action template files are added to individual projects and should be placed in the `templates` directory of the project and should use the file extension `sbat` (Storyboard Action Template)

The format of an action template file (`sbat`) file is as follows:

```
<actiontemplates>
<template name="NAME">
  <arguments>
    <element name="ARG_NAME" type="ARG_TYPE" />
    ... as many elements as there are arguments ...
  </arguments>
</template>
... as many templates as there are actions ...
</actiontemplates>
```

Where the fields `NAME`, `ARG_NAME` and `ARG_TYPE` are defined as

NAME This is the name of the action as it appears in the Storyboard Engine runtime (`gapp`) file.

ARG_NAME This is the name of an argument option as it appears in the Storyboard Engine runtime (`gapp`) file.

ARG_TYPE This is the type of the argument and can be one of the following:

boolean A boolean true/false value. An optional attribute, 'default' can be used to specify the default value.

color	A color value. An optional attribute, 'default' can be used to specify the default value as a hexadecimal value (0xRRGGBB).
file	A file string. An optional 'extensions' argument list contains a list of comma separated filter file extensions, without leading dots for example extensions="gif;png;bmp"
float	A numeric floating point value. An optional attribute, 'default', can be used to specify the default value. The range of the number can also be limited using the optional 'min' and 'max' attributes
integer	A numeric integer value. An optional attribute, 'default', can be used to specify the default value. The range of the number can also be limited using the optional 'min' and 'max' attributes
string	A text string value. An optional attribute, 'default' can be used to specify the default string value.

When the template file is placed in the `templates` directory, the new actions will be automatically loaded into Designer the next time the project is opened.

User Defined Render Extension Templates

When the Storyboard SDK is used to contribute new render extensions, it is desirable to have those render extensions available within the Storyboard Designer environment so that application designers can use the render extensions and configure their properties. To do this, an render extension template file describing the name and properties of the render extension can be created. It is not possible to present a visual display of the render extension within Storyboard Designer so a placeholder image is presented in its place.

Render extension template files are added to individual projects and should be placed in the `templates` directory of the project and should use the file extension `sbrt` (Storyboard Render Template)

The format of a render extension template file (`sbat`) file is as follows:

```
<rendertemplates>
<template name="NAME">
  <arguments>
    <element name="ARG_NAME" type="ARG_TYPE" />
    ... as many elements as there are arguments ...
  </arguments>
</template>
... as many templates as there are actions ...
</rendertemplates>
```

Where the fields `NAME`, `ARG_NAME` and `ARG_TYPE` are defined as

NAME This is the name of the render extension as it appears in the Storyboard Engine runtime (gapp) file.

ARG_NAME This is the name of an argument option as it appears in the Storyboard Engine runtime (gapp) file.

ARG_TYPE This is the type of the argument and can be one of the following:

boolean	A boolean true/false value. An optional attribute, 'default' can be used to specify the default value.
color	A color value. An optional attribute, 'default' can be used to specify the default value as a hexadecimal value (0xRRGGBB).
file	A file string. An optional 'extensions' argument list contains a list of comma separated filter file extensions, without leading dots for example extensions="gif;png;bmp"
float	A numeric floating point value. An optional attribute, 'default', can be used to specify the default value. The range of the number can also be limited using the optional 'min' and 'max' attributes
integer	A numeric integer value. An optional attribute, 'default', can be used to specify the default value. The range of the number can also be limited using the optional 'min' and 'max' attributes
string	A text string value. An optional attribute, 'default' can be used to specify the default string value.

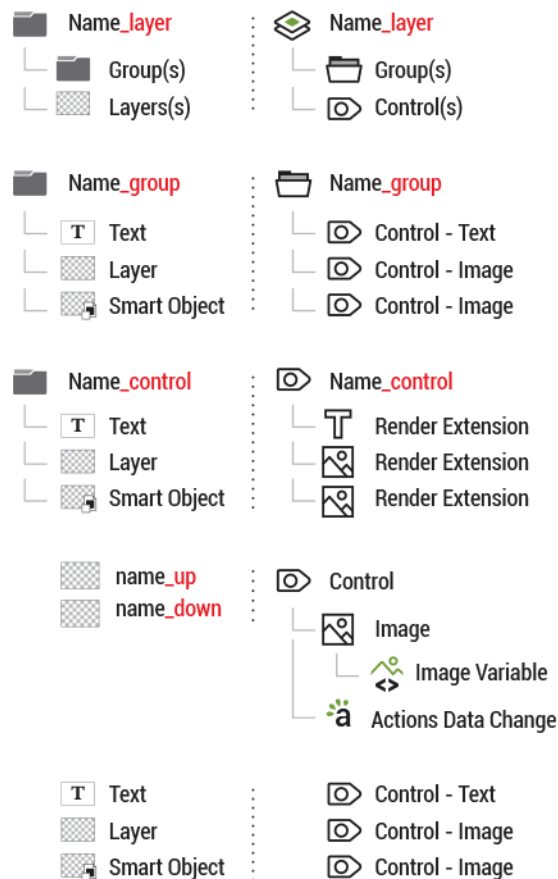
When the template file is placed in the `templates` directory, the new render extensions will be automatically loaded into Designer the next time the project is opened.

Chapter 31. Structuring Your Photoshop Import Content

PSD file Structure for Import into Storyboard

In Storyboard the appearance of the UI after using the Photoshop™ File Import feature will depend on properties and attributes that are set from Photoshop™. The content and organization of a Storyboard project can vary depending on the preparation of the Photoshop file prior to Import.

Photoshop to Storyboard



PSD File Requirements

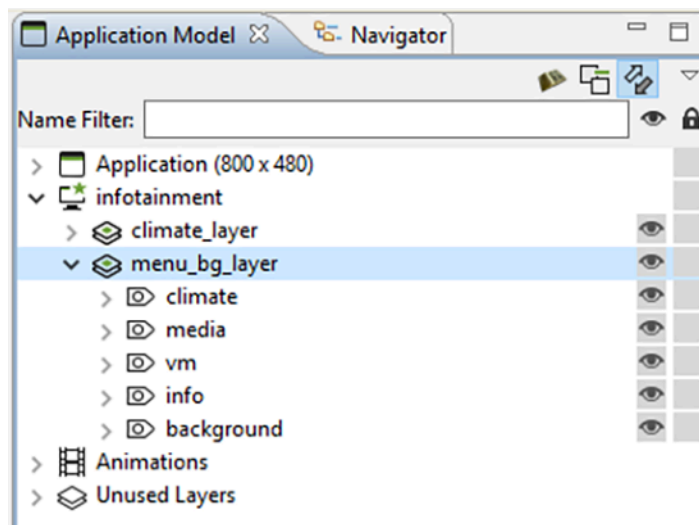
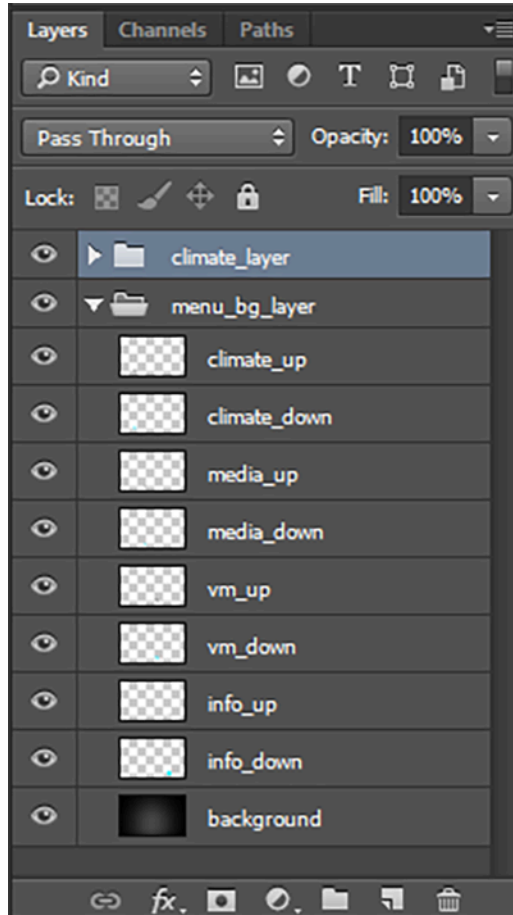
In order to import files from Photoshop™ to Storyboard, documents must be in a RGB color mode as an 8-bit file. Other formats are not supported.

Application Model Hierarchy

Understanding the Storyboard Application Model Hierarchy will help users decide how to name and organize content in Photoshop to better control how a Storyboard application comes together when using the PSD import feature.

Photoshop content names and organization will dictate the name of model elements like Screens, Layers, Groups, Controls and all of the image file names in the project image directory. How content is named can also control how it is placed within the Application Model.

Below is how the new project is laid out after the PSD file has been imported into Storyboard.

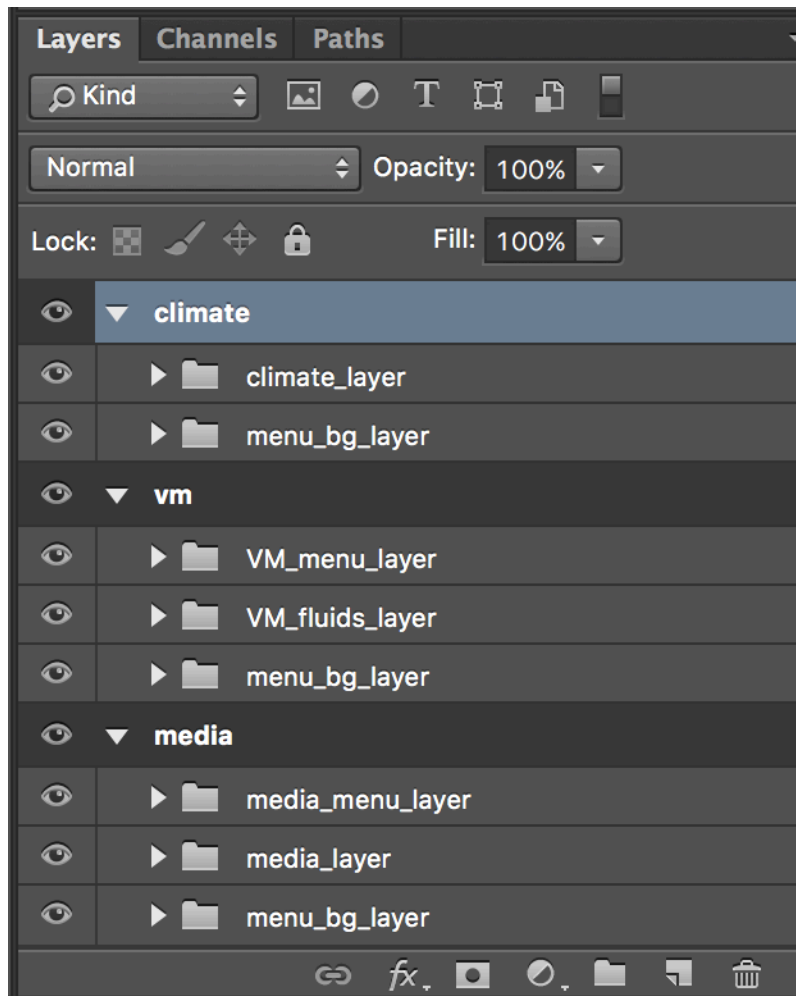


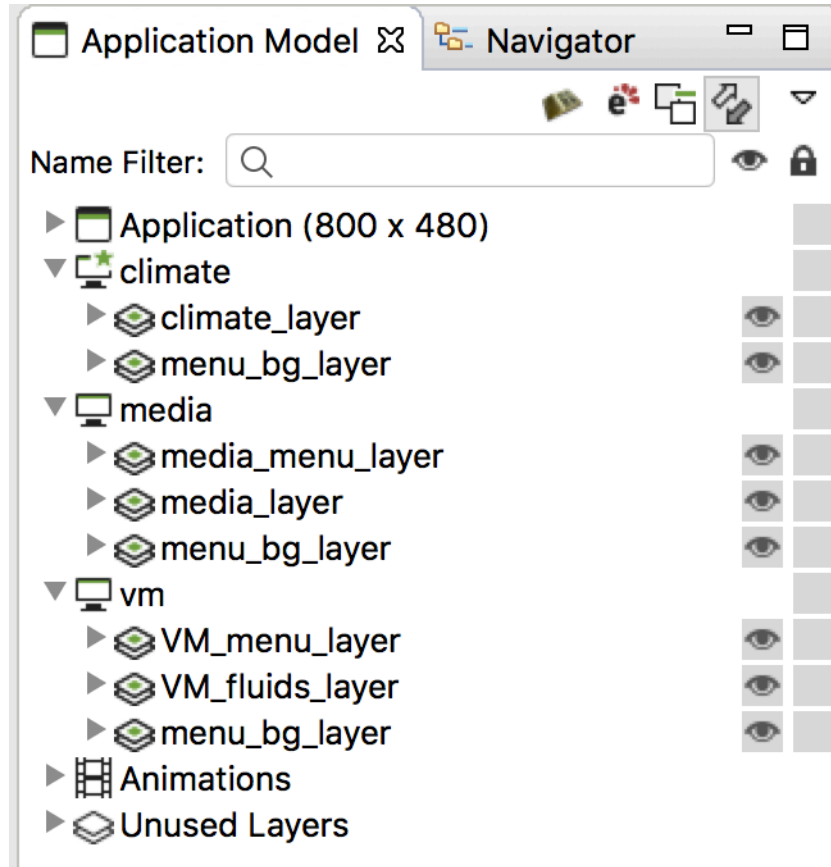
Artboards

Artboards in Photoshop can be used to restrict content to specific boundaries within a Photoshop file. When imported into Storyboard, an Artboard will become a screen in the resulting application and layer content defined within the Artboard will be applied to that screen. This feature can be used to re-organize several Photoshop files representing the screens of an application into one file that only needs to be imported once.

Unlike other Storyboard objects, there is no naming convention required for Artboards. All Artboards automatically be imported as screens with the name provided.

Below is how a new project is laid out after a PSD file with artboards has been imported into Storyboard.





Layer Effects and Blending Modes

Prior to importing a Photoshop file into Storyboard, properties that are unique to Photoshop like layer effects, blending modes, layer fill percentages or masks need to be rasterized or converted to smart objects in order to maintain the same appearance when they are brought into Storyboard.

By converting content with Photoshop specific rendering to smart objects, the editability within Photoshop still exists while the objects import to Storyboard as expected.

Naming Requirements

Model elements must be named using an alpha-numeric format. The naming of model elements can be done using combinations of “Aa-Zz” “0-9” and “_”, the underscore character. How content is named in Photoshop will be maintained when it is brought into Storyboard. When duplicate naming occurs, controls and files in the project image directory will then include numeric values with the name, e.g., “image.png” “image1.png” image2.png”

Naming Conventions

The importer will organize Photoshop content according to special naming conventions that have been assigned and place items within the Application Model accordingly. Important Naming Conventions:

name_layer

Photoshop group folders named with `_layer` at the end will import to Storyboard as a layer. Content within these group folders will import as project elements within that layer such as groups and controls. All Storyboard groups and controls can not exist outside of a Storyboard Layer.

name_group

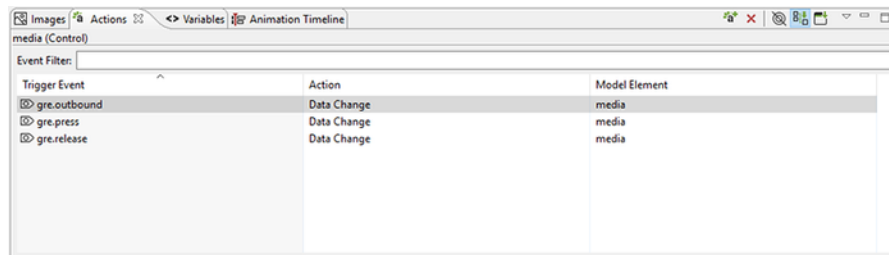
Photoshop group folders named with `_group` at the end will import to Storyboard as a group. Any content within these groups will import as controls within that group.

name_control

Photoshop groups folders named with `_control` at the end will import to Storyboard as a control. Content within these groups will import as text or image render extensions of that control.

name_up and name_down

Naming a PSD layer ending with `_up` directly above another layer with the same name ending with `_down` will create a control that behaves as a button. The control will be assigned events and actions and given an image variable that will work as an interactive button control upon import to Storyboard.



Common Photoshop Elements

Content from Photoshop like text layers, image layers, and smart objects will import to Storyboard as individual controls when they are not part of a group folder. Text layers from Photoshop have the option to import as controls with image or text render extensions.

Group folders and layers that are empty in Photoshop will be discarded, so importing an empty layer to Storyboard will not occur and blank images can not be created from Photoshop layers that have no pixels being drawn.

Notes created in the Photoshop file will be imported and applied as application level annotations in Storyboard.

Transparency

Photoshop layers that use a percentage of opacity will import to Storyboard with the same opacity. The value can be changed in the properties view under that render extension's alpha setting.

Naming Convention Deviation

Within the Photoshop Layer View, content that is not placed within a parent group folder using the `_layer` naming convention will be assigned to a Storyboard layer that's automatically generated. Storyboard Layers that are automatically generated will be assigned default names, “Layer” “Layer1” “Layer2” etc.

Sub Group Folders

Photoshop group folders that contain additional group folders need to follow the Application Model hierarchy. Layers, Groups, and Controls cannot contain model elements of the same type, e.g., Controls cannot contain other Controls.

Chapter 32. Re-Importing Photoshop and Updating Content

Re-Importing Photoshop Content

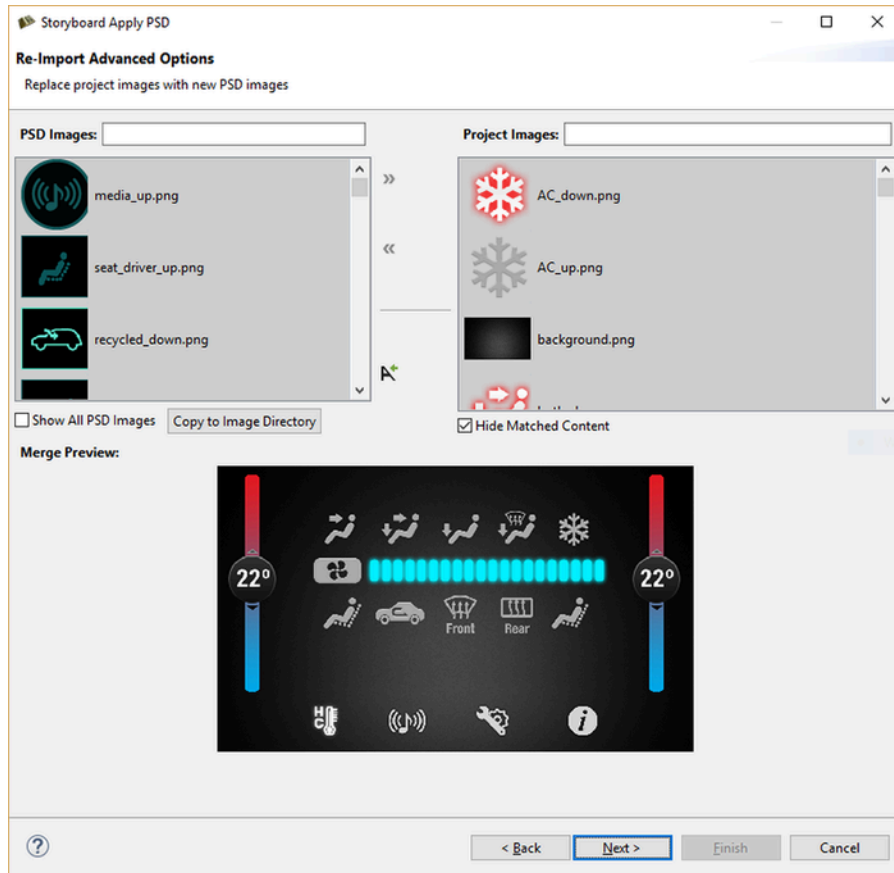
The Photoshop™ Re-Import feature enables the user to re-import a new or revised PSD file to replace existing images in a project's image directory. The re-import wizard is initiated from the main menu » File » Import

In the import dialog that appears, select the source PSD file to re-import and the destination Storyboard application where the new PSD image content will be placed. Then choose Next.

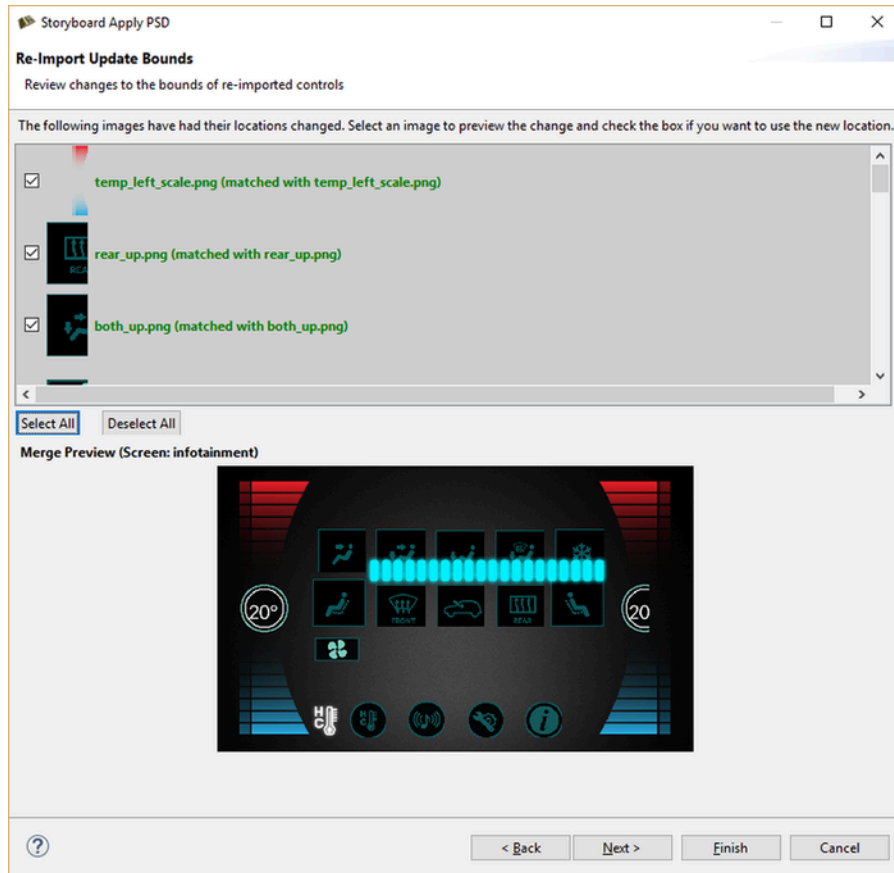
On the left of the dialog window is a list of image content from the PSD file. On the right is the content of the project's image directory. Selecting any PSD image will show candidate image match content from the PSD file. By clicking the “move right” arrow the selected PSD image is adopted as the new project image and replaces the model selection.

Below the separator in between the two viewers is the 'Match All Images' button. This will go through and automatically map any unique images of the same name. The name and extension must be identical for images to be mapped. If there is more than one identical match, the image won't be mapped.

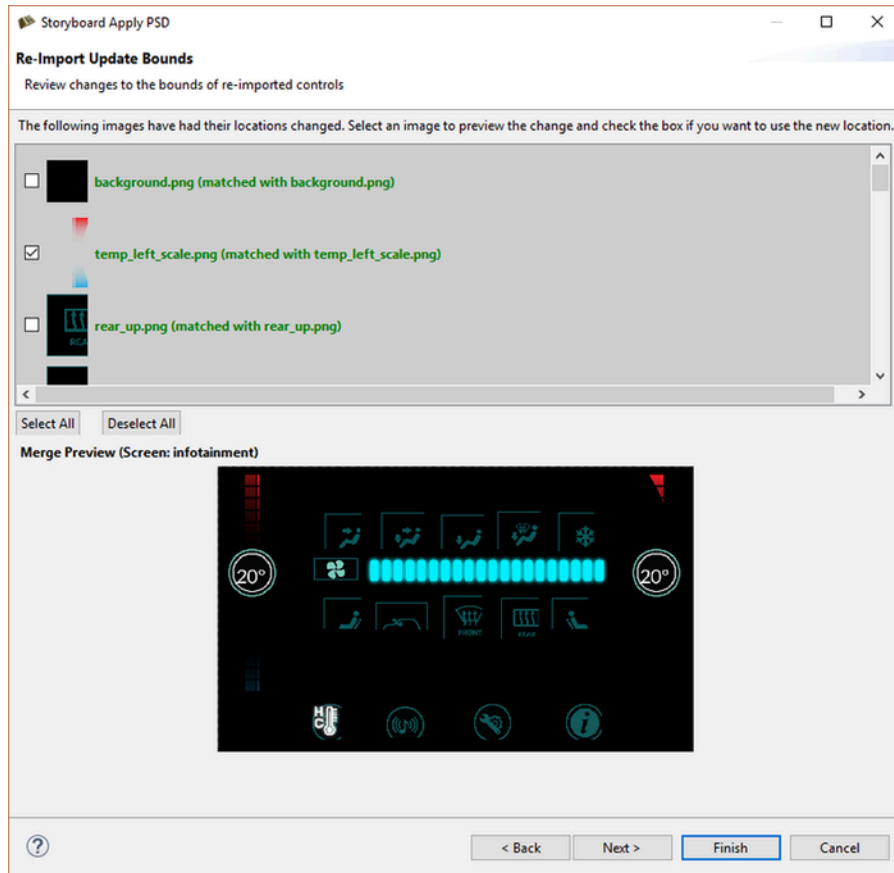
Matching and unmatching can be undone and redone by using the common keyboard shortcuts 'CTRL-Z' and 'CTRL-R'. If a PSD image does not have a project image with the same name it can still be added to the project image directory by clicking Copy to Image Directory button. Content that is being replaced will be shown in a preview below.



Choosing 'Next' brings you to a page that allows the user to update the bounds of any of the selected matches from the previous page. Any matches from the previous page that have changed location in the re-imported PSD file will appear in the list at the top. Selecting an item from this list shows the location change in the preview at the bottom, similar to the previous page. If you want to use the new location, check the box beside the match before hitting 'Finish'.



After updating the bounds, the next page allows the user to review any new images that have been added in the re-import. An image is considered new if it has no identical match in the current model, and it hasn't been mapped to anything in the first page. Selecting an item will preview it in the thumbnail below, similar to the previous page, and checking it will import the new control into the model.



After choosing to replace, copy or add at least one image, clicking the Finish button will show a prompt that gives the option to overwrite image files that are being updated or keep a copy of the older images that are being replaced. The Photoshop PSD Re-Import feature is meant to update existing content using consistent naming. Using the Import Photoshop PSD File is better for adding new content to an existing project.

Appendix A. Storyboard Lua API

Storyboard Lua API

gre.APP_ROOT

`gre.APP_ROOT`

This is a variable that is filled in by the Storyboard Engine to point at the filesystem path for the application's project root directory. This can be used in order to access resources in a project relative manner. This variable is only applicable for filesystem based configurations.

Historical code may use `gre.SCRIPT_ROOT .. "/.."` as a way of referring to files in a project relative manner, but `gre.APP_ROOT` is preferred as not all filesystems support relative path operations. Unix forward slash directory path separator conventions are preferred when working file filenames.

Example:

```
-- Open the french translation file in the translations folder
local translation_file = gre.APP_ROOT .. "/translations/french.csv"
local fp = io.open(translation_file, "r")
...
```

gre.SCRIPT_ROOT

`gre.SCRIPT_ROOT`

This is a variable that is filled in by the Storyboard Engine to contain the path to the application project's scripts directory.

The `gre.SCRIPT_ROOT` variable provides a convenient way to reference the Lua script related resources in a location independent, project relative, manner. This variable is only applicable for filesystem based configurations.

When used in conjunction with the `gre.env()` function, the `gre.SCRIPT_ROOT` can provide an effective way to configure the search path for extra Lua modules.

Example:

```
-- Add the scripts/<os>-<cpu> to the Lua file resolver for require statements
-- ie ....scripts/linux-x86
local target = gre.env({"target_os", "target_cpu"})
```



```
local extra_path = string.format("%s/%s-%s/?.lua;", gre.SCRIPT_ROOT,
target.target_os, target.target_cpu)
package.path = extra_path .. package.path
```

gre.PLUGIN_ROOT

`gre.PLUGIN_ROOT`

This is a variable that is filled in by the Storyboard Engine to contain the path to the Storyboard Engine's plugins directory. This is the directory that is specified to the `model_mgr` using the `-omodel_mgr,plugin_path` option or via the `SB_PLUGINSenvironment` variable. This path can be used to load application specific binaries that may be packaged with a particular Storyboard Engine. This variable is only applicable for systems that dynamically load plugins from a filesystem.

gre.LOG_ constants

`gre.LOG_ALWAYS gre.LOG_ERROR gre.LOG_WARNING gre.LOG_INFO gre.LOG_EVENT gre.LOG_ACTION gre.LOG_DIAG1 gre.LOG_DIAG2 gre.LOG_EVENT2 gre.LOG_TRACE1 gre.LOG_TRACE2`

<code>gre.LOG_ALWAYS</code>	-1	Always go to stdout
<code>gre.LOG_ERROR</code>	0	Errors, fatal and non-fatal
<code>gre.LOG_WARNING</code>	1	Warnings
<code>gre.LOG_INFO</code>	2	Information, one time, non-repetitive
<code>gre.LOG_EVENT1</code>	3	Event delivery, excluding motion/mtevent/redraws
<code>gre.LOG_ACTION</code>	4	Action execution
<code>gre.LOG_DIAG1</code>	5	Storyboard diagnostic informative
<code>gre.LOG_DIAG2</code>	6	Storyboard diagnostic detailed
<code>gre.LOG_EVENT2</code>	7	Motion/mtevent/redraw event delivery
<code>gre.LOG_TRACE1</code>	8	Storyboard minimal tracing
<code>gre.LOG_TRACE2</code>	9	Storyboard maximal tracing

gre.LEFT, gre.RIGHT, gre.CENTER, gre.TOP, gre.BOTTOM

`gre.LEFT gre.RIGHT gre.CENTER gre.TOP gre.BOTTOM`

These are definitions for horizontal and vertical values and are a shorthand for the following numeric values representing alignment:

<code>gre.LEFT</code>	1
<code>gre.CENTER</code>	2

gre.RIGHT	3
gre.TOP	1
gre.BOTTOM	3

gre.OPAQUE, gre.TRANSPARENT

gre.OPAQUE, gre.TRANSPARENT

These are definitions for the transparency (alpha) values and are a shorthand for the following numeric values representing extreme transparency:

gre.TRANSPARENT	0
gre.OPAQUE	255

gre.set_data

```
gre.set_data(  
    table  
)
```

Sets one or more items in the Storyboard application's data manager. Each index and value in the table passed in will be set in the data manager using the index name as the key.

Parameters:

table A table containing the variable to change as the key and the value to change it to as that key's value.

Example:

```
function lua_func( mapargs )  
    local data_table = {}  
    data_table["variable_name"] = "variable data"  
    gre.set_data( data_table )  
end
```

gre.get_data

```
gre.get_data(  
    key  
    [, key2, ...]  
)
```

Gets one or more values from the data manager. Each argument to the function is interpreted as a data manager key whose value should be extracted from the data manager. This function returns a table using

all the values as indexes and the corresponding value is the data returned from the data manager. A nil is returned for any values that do not match a key in the data manager.

Parameters:

key The key whose value should be extracted from the data manager.

Returns:

A table containing the passed in arguments as keys and the resulting data manager values as the values associated with those keys.

Example - Accessing Control Variables:

```
function get_data_func( mapargs )
    --When accessing control variables, use the following qualified
    --model path Layer.Control.Variable
    local data_table = gre.get_data("my_layer.my_control.variable_name")
    local value = data_table["my_layer.my_control.variable_name"]
    print("control_variable_name = " .. tostring(value))
end
```

Example - Accessing Control Width (Internal Variable):

```
function get_control_width( mapargs )
    --This will extract the width (grd_width) of the control
    --'my_control' on the layer 'my_layer'
    local data = gre.get_data("my_layer.my_control.grd_width")
    local value = data["my_layer.my_control.grd_width"]
    print("The width of the control is " .. tostring(value))
end
```

gre.set_value

```
gre.set_value(
    key,
    value
    [, key2, value2, ...]
)
```

Set a variable in the data manager to a particular value. This function is a convenience function on top of `gre.set_data` that allows the key and value to be passed as a set of arguments to the function instead of having to create a table containing the key/value pairs.

Parameters:

key A string value containing the key to be set with the next following value

value The value to be assigned to the preceding argument (key)

Example:

```
function lua_func( mapargs )
    -- Assign the string 'variable_data' to the application variable
    -- 'variable_name'
    -- This example is the same as gre.set_data()
    gre.set_value("variable_name", "variable_data")
end
```

gre.get_value

```
gre.get_value(
    key
    [, key2, ...]
)
```

Get the value of a variable from the data manager. This function is a convenience function on top of `gre.get_data` that allows the value to be returned directly to the caller instead of a single table return value. A nil is returned for any values that do not match a key in the data manager.

Parameters:

key The key whose value should be extracted from the data manager.

Returns:

The value associated with the data manager entry for the key, or nil if no entry exists. If multiple keys are specified, then multiple return values will be generated matching the argument order.

Example - Accessing Control Width:

```
function get_control_width( mapargs )
    -- This will extract the width of the control 'my_control'
    -- on the layer 'my_layer'
    -- This is the same example as gre.get_data()
    local value = gre.get_value("my_layer.my_control.grd_width")
    print("The width of the control is " .. tostring(value))
end
```

gre.resolve_data_key

```
gre.resolve_data_key(
    key1
    [, key2, ...]
```

```
)
```

This function allows Lua scripts to resolve Storyboard context variables to a fully qualified name based on the current execution context.

Parameters:

key1 ... One or more string arguments containing the variable to resolve.

Returns:

A table containing the arguments provided on input as keys with the values being the resolved data value.

Example:

```
-- Resolve the application my_var to a fully qualified name
local varname = "${app:my_var}"
local dv = gre.resolve_data_key(varname)
print("Full path for ${app:my_var} is " .. dv[varname])
```

gre.get_control_attrs

```
gre.get_control_attrs(
    control_name
    tags ...
)
```

Get properties for a control. Key name is the name of the control or a variable. Tags can be a list of the following values:

x, y, width, height, hidden, active, zindex, findex

A table with the results is returned.

Parameters:

control_name The model full path of the control to get information about
tags One or more tags as strings

Returns:

A table containing the tags as keys with the associated table value being the Storyboard value associated with that tag.

Example:

```
function check_if_hidden()
    local dk_data = {}
```

```
-- check if my_control is currently hidden
dk_data = gre.get_control_attrs("my_layer.my_control", "hidden")
if dk_data["hidden"] == 1 then
    print("my_control is currently hidden")
else
    print("my_control is currently visible")
end
end
```

gre.set_control_attrs

```
gre.set_control_attrs(
    control_name,
    tag_table
)
```

Set properties for a control. The `control_name` is the name of the control or a variable. The `tag_table` contains the tags and values for the properties to set.

`x, y, width, height, hidden, active, zindex, findex, effect`

In the case of the focus index (`findex`), the initial value set in Storyboard Designer must be non-zero in order for it to be changed dynamically at runtime

The `effect` tag is a special attribute that can be used with OpenGL rendering platforms to apply a custom shader directly as an effect on the control object

Parameters:

`control_name` The model full path of the control to change properties on
`tag_table` A table with tags as the keys and the new values stored as the table's key values

Examples:

```
function set_control_hidden()
    local dk_data = {}
    dk_data["hidden"] = 1
    gre.set_control_attrs("my_control", dk_data)
end
```

```
function set_control_blur_effect()
    local dk_data = {}
    local effect = {}

    effect["name"] = "blur"
    effect["passes"] = 3
    effect["radius"] = 1
    effect["composite"] = true
end
```

```
    dk_data["effect"] = effect
    gre.set_control_attrs("my_control", dk_data)
end
```

gre.get_table_attrs

```
gre.get_table_attrs(
    table_name,
    tags
)
```

Get properties for a table. Key name is the name of the control or a variable. Tags can be any of the control tags and any of the following values:

rows	The number of rows in the table
cols	The number of columns in the table
visible_rows	The number of visible rows in the table
visible_cols	The number of visible columns in the table
active_row	The active cell row
active_col	The active cell column
row	The row index of the upper left row
col	The column index of the upper left column
xoffset	The current scroll offset in the x direction
yoffset	The current scroll offset in the y direction
scroll_enabled	The current state of table content scrolling enablement. The value is 0 if scrolling is not enabled or 1 if it is.

Parameters:

table_name The model full path of the table to get information about
tags One or more tags as strings

Returns:

A table containing the tags as keys with the associated table value being the Storyboard value associated with that tag.

Example:

```
function check_if_hidden()
    local dk_data = {}
```

```
-- Get the active row/column
dk_data = gre.get_table_attrs("my_table", "active_row", "active_col")
print("Active Cell: " .. tostring(dk_data["active_row"]) .. ", "
      .. tostring(dk_data["active_col"]))
end
```

gre.set_table_attrs

```
gre.set_table_attrs(
    table_name,
    tag_table
)
```

Set properties for a table. The `table_name` is the name of the control or a variable. The `tag_table` contains the tags and values for the properties to set.

`x, y, width, height, hidden, active, rows, cols, xoffset, yoffset, scroll_enabled`

The scroll enablement value can only be set if the table had scrolling enabled in Designer.

Parameters:

`table_name` The model full path of the table to change properties on
`tag_table` A table with tags as the keys and the new values stored as the table's key values

Example:

```
function resize_table()
    local dk_data = {}
    dk_data["rows"] = 5
    dk_data["cols"] = 10
    gre.set_table_attrs("my_table", dk_data)
end
```

gre.get_table_cell_attrs

```
gre.get_table_cell_attrs(
    table_name,
    row,
    col,
    tags ...
)
```

Get properties for a table cell. `table_name` is the name of the table. Tags can be a list of the following values:

`x, y, width, height, hidden`

A table with the results is returned.

Parameters:

`table_name` The model full path of the table to get information about
`row` The row of the table to get information on
`col` The column of the table to get information on
`tags` One or more tags as strings

Returns:

A table containing the tags as keys with the associated table value being the Storyboard value associated with that tag.

Example:

```
function check_if_hidden()  
    local dk_data = {}  
    -- check if my_control is currently hidden  
    dk_data = gre.get_table_cell_attrs("my_table", 1, 1, "hidden")  
    if dk_data["hidden"] == 1 then  
        print("cell 1.1 of my_table is currently hidden")  
    else  
        print("cell 1.1 of my_table is currently visible")  
    end  
end
```

gre.get_group_attrs

```
gre.get_group_attrs(  
    group_name  
    tags ...  
)
```

Get properties for a group. Key name is the name of the group or a variable. Tags can be a list of the following values:

x, y, hidden

A table with the results is returned.

Parameters:

`group_name` The model full path of the control to get information about
`tags` One or more tags as strings

Returns:

A table containing the tags as keys with the associated table value being the Storyboard value associated with that tag.

Example:

```
function check_if_hidden()
    local dk_data = {}
    -- check if my_group is currently hidden
    dk_data = gre.get_group_attrs("my_layer.my_group", "hidden")
    if dk_data["hidden"] == 1 then
        print("my_control is currently hidden")
    else
        print("my_control is currently visible")
    end
end
```

gre.set_group_attrs

```
gre.set_group_attrs(
    group_name,
    tag_table
)
```

Set properties for a group. The `group_name` is the name of the group or a variable. The `tag_table` contains the tags and values for the properties to set.

x, y, hidden

Parameters:

`group_name` The model full path of the group to change properties on
`tag_table` A table with tags as the keys and the new values stored as the table's key values

Examples:

```
function set_control_hidden()
    local dk_data = {}
    dk_data["hidden"] = 1
    gre.set_control_attrs("my_layer.my_group", dk_data)
end
```

```
function set_control_blur_effect()
    local dk_data = {}
    local effect = {}

    effect["name"] = "blur"
    effect["passes"] = 3
    effect["radius"] = 1
    effect["composite"] = true
```

```
    dk_data["effect"] = effect
    gre.set_control_attrs("my_control", dk_data)
end
```

gre.get_layer_attrs

```
gre.get_layer_attrs(
    layer_name
    tags...
)
```

Get properties for a layer instance associated with a particular screen. The `layer_name` specifies either the fully qualified name of a layer instance using the `ScreenName.LayerName` naming convention or, if only the layer name is specified, the name will refer to a layer instance associated with the current screen

The tags are a list of string properties associated with the layer instance and can include one or more of the following values:

x, y, width, height, alpha, hidden, xoffset, yoffset, scroll_enabled, geometry, effect

A table containing the keys and their respective values is returned or nil if the layer can not be found.

Parameters:

`layer_name` The model full path of the layer to get information about
`tags` One or more tags as strings

Returns:

A table containing the tags as keys with the associated table value being the Storyboard value associated with that tag.

Example:

```
function check_if_hidden()
    -- check if my_layer is currently hidden
    local data = gre.get_layer_attrs("my_layer", "hidden")
    if data.hidden == 1 then
        print("my_layer is currently hidden")
    else
        print("my_layer is currently visible")
    end
end
```

gre.set_layer_attrs

```
gre.set_layer_attrs(
```

```
    layer_name,  
    tag_table  
)
```

Set properties for a layer instance associated with a particular screen. The `layer_name` specifies either the fully qualified name of a layer instance using the `ScreenName.LayerName` naming convention or, if only the layer name is specified, the name will refer to a layer instance associated with the current screen

`x, y, width, height, alpha, hidden, xoffset, yoffset, scroll_enabled, geometry, effect`

The scroll enablement value can only be set if the layer had scrolling enabled in Designer.

Parameters:

`layer_name` The model full path of the layer to change properties on
`tag_table` A table with tags as the keys and the new values stored as the table's key values

Example:

```
function set_layer_hidden()  
    local data = {}  
    data.hidden = 1  
    gre.set_layer_attrs("my_layer", data)  
end
```

Render Effects:

The attribute contains a table with the name and attributes for the specific render effect being applied. Currently the following effects are defined:

`blur, geometry`

blur

This effect will add a blur to the contents of the layer. The following blur attributes are defined:

Parameters:

`passes` This is a number value which is the number of blur passes
`composite` This is a boolean value. When true the blur will be applied to the final composition of this layer with the framebuffer content. If false the blur is only applied to the layer content.
`radius` This is a number value which defined the radius of the blur effect in pixels

Example:

```
function cbBlurEffect(mapargs)  
    local attrs = {}  
    local effect = {}
```

```
effect["name"] = "blur"
effect["passes"] = 2
effect["radius"] = 1
effect["composite"] = false
attrs["effect"] = effect
gre.set_layer_attrs("background.control1",attrs)
end
```

geometry

This effect will allow custom OpenGL ES geometry to be applied to the rendering of the control or layer. This includes custom vertices and UV coordinates. The following attributes are defined:

Parameters:

- width The viewport width for the content
- height The viewport height for the content
- type The type of primitive to render: fan | triangles
- nvert The number of vertex coordinates. Options are 2 (x,y) and 3 (x, y, z)
- nuv The number of UV coordinates. Options are 2 or 0
- data The table containing the vertex data

Example:

```
function cbSetGeometry(mapargs)
    local gdata = {}
    local dz = 0.0
    local offset = mapargs.offset
    local dz2 = -offset
    local w
    local h = 240

    dz2 = -offset
    w = 320 - offset
    gdata = {
        {x=w/2, y=h/2, z=dz2, u=0.5, v=0.5},

        {x=w, y=0, z=dz, u=1.0, v=0},
        {x=w/2, y=0, z=dz2, u=0.5, v=0},
        {x=0, y=0, z=dz, u=0, v=0},

        {x=0, y=h, z=dz, u=0, v=1.0},
        {x=w/2, y=h, z=dz2, u=0.5, v=1.0},
        {x=w, y=h, z=dz, u=1.0, v=1.0},

        {x=w, y=0, z=dz, u=1.0, v=0},
    }

    local attrs = {}
```

```
    attrs["geometry"] = {
        width = w,
        height = h,
        type = "fan",
        nvert = 3,
        nuv = 2,
        data = gdata
    }
    gre.set_layer_attrs("geometry.layer3",attrs)
end
```

gre.set_layer_attrs_global

```
gre.set_layer_attrs_global(
    layer_name,
    table
)
```

Set properties for a layer globally on all instances of the layer on all screens. The `layer_name` is the name of the layer. Table contains the tags and values for the properties to set.

alpha, hidden, active, x, y, width, height

Parameters:

`layer_name` The model full path of the layer to change properties on

`tag_table` A table with tags as the keys and the new values stored as the table's key values

gre.screen_attach_layer

```
gre.screen_attach_layer(
    layer_name,
    [zindex]
)
gre.screen_attach_layer(
    layer_name,
    screen_list,
    [zindex]
)
```

Attaches a layer to a screen dynamically creating a new layer instance. When used without a screen list this will attach the specified layer to all screens in the application. When used with a list of screens it will bind the specified layer to all of the screens named in the screen list.

If `zindex` is included, then the layer will be inserted at the specified `zindex` (0 = backmost). If `zindex` is not set, then the layer will be inserted at the top of the existing layer stack for the screen (equivalent to specifying a value greater than all of the layers in the project).

If the layer is already present on a screen, then that layers position in the screen z-order will be adjusted to the specified value. If the layer name used is a specific layer instance (ie screen_name.layer_name) then all of the properties of that layer instance (x, y, alpha, hidden) will also be copied to the new layer instance.

Parameters:

layer_name The name of the layer to attach to screens
screen_list A lua table containing the list of screen names to add the layer to
zindex The z-order (0 = backmost) to add the layer to the screen at.

gre.resize_control

```
gre.resize_control(  
    control_name,  
    dw,  
    dh,  
    w,  
    h,  
    [ah],  
    [av]  
)
```

Resize a control. The control_name is the name of the control or a variable. Setting dw and or dh will resize the control by the specified delta from its current size. The dw and dh values can be 0 to set an absolute position using the w and h values only.

Optional anchors can allow for automatic adjustment of the control's x and y positions to keep the control anchored in the desired position as the control's size change:

Horizontal:

- `grd.LEFT` anchor will not adjust the control's x position. This is the default behavior.
- `grd.CENTER` anchor will adjust the control's x position by half the difference in control width.
- `grd.RIGHT` anchor will adjust the control's x position by the difference in control width.

Vertical:

- `grd.TOP` anchor will not adjust the control's y position. This is the default behavior.
- `grd.CENTER` anchor will adjust the control's y position by half the difference in control height.
- `grd.BOTTOM` anchor will adjust the control's y position the difference in control height.

Parameters:

control_name The model full path of the control to move
dx A delta from the current x position or 0 to move using x
dy A delta from the current y position or 0 to move using y
x The x position to move to in absolute co-ordinates (omit to use dx)
y The y position to move to in absolute co-ordinates (omit to use dy)

ah The horizontal anchor, `gre.LEFT`, `gre.CENTER` or `gre.RIGHT` (Optional, default is `gre.LEFT`)
vh The vertical anchor, `gre.TOP`, `gre.CENTER` or `gre.BOTTOM` (Optional, default is `gre.TOP`)

gre.move_control

```
gre.move_control(  
    control_name,  
    dx,  
    dy,  
    x,  
    y,  
    [ah],  
    [av]  
)
```

Move a control to a new position. The `control_name` is the name of the control or a variable. Setting `dx` and or `dy` will move the control by the specified delta from its current position. The `dx` and `dy` values can be 0 to set an absolute position using the `x` and `y` values only.

Optional anchors allow for automatic adjustment of `x` and `y` values:

Horizontal:

- `grd.LEFT` anchor will not adjust the input. This is the default behavior.
- `grd.CENTER` anchor will adjust the `x` input by half the control width.
- `grd.RIGHT` anchor will adjust the `x` input by the full control width.

Vertical:

- `grd.TOP` anchor will not adjust the input. This is the default behavior.
- `grd.CENTER` anchor will adjust the `y` input by half the control height.
- `grd.BOTTOM` anchor will adjust the `y` input by the full control height.

Parameters:

<code>control_name</code>	The model full path of the control to move
<code>dx</code>	A delta from the current <code>x</code> position or 0 to move using <code>x</code>
<code>dy</code>	A delta from the current <code>y</code> position or 0 to move using <code>y</code>
<code>x</code>	The <code>x</code> position to move to in absolute co-ordinates (omit to use <code>dx</code>)
<code>y</code>	The <code>y</code> position to move to in absolute co-ordinates (omit to use <code>dy</code>)
<code>ah</code>	The horizontal anchor, <code>gre.LEFT</code> , <code>gre.CENTER</code> or <code>gre.RIGHT</code> (Optional, default is <code>gre.LEFT</code>)
<code>vh</code>	The vertical anchor, <code>gre.TOP</code> , <code>gre.CENTER</code> or <code>gre.BOTTOM</code> (Optional, default is <code>gre.TOP</code>)

gre.move_layer

```
gre.move_layer(  

```



```
    layer_name,  
    dx,  
    dy,  
    x,  
    y  
)
```

Move a layer to a new position. The `layer_name` is the name of the layer or a variable that is associated with the layer name. Setting `dx` or `dy` will move the layer by the specified delta from its current position. The `dx` and `dy` values can be 0 to set an absolute position using the `x` and `y` values only.

Parameters:

<code>layer_name</code>	The model full path of the layer to move
<code>dx</code>	A delta from the current x position or 0 to move using x
<code>dy</code>	A delta from the current y position or 0 to move using y
<code>x</code>	The x position to move to in absolute co-ordinates (omit to use dx)
<code>y</code>	The y position to move to in absolute co-ordinates (omit to use dy)

gre.set_focus

```
gre.set_focus(  
    fq  
)
```

Set the focus to a control as described by its fully qualified name. This function returns `true` if the control could be found and was focusable and focus could be set to it. If the control can be found but is not focusable then `false` will be returned. If the control can't be found then `nil` will be returned.

When the fully qualified name indicates the row and column of a table, then this function will set the active table cell rather than adjusting the control focus.

Parameters:

<code>fq</code>	The fully qualified name of the control to receive focus
-----------------	--

Returns:

`true` for success, `false` for failure or `nil` if an error occurs during processing

Example:

```
-- Set the focus to MyControl  
local didApply = gre.set_focus("MyLayer.MyControl")  
if(not didApply) then  
    print("Failed to set focus")  
end  
  
-- Set to focus to the second row item in a table
```

```
local didApply = gre.set_focus("MyLayer.MyTable.2.1")
if(not didApply) then
    print("Failed to set table cell focus")
end
```

gre.get_focus

```
gre.set_focus()
```

Get the fully qualified path to the control that is currently focused or nil if no control currently has focus

Returns:

A string with the fully qualified name of the control with focus or nil if no control focused.

gre.send_event

```
gre.send_event(
    event,
    [channel]
)
```

Send an event to the application or to a Storyboard IO channel. channel is an optional parameter and if channel is not passed then the channel will be chosen as follows:

If the environment variable GREIONAME is set then it will be used otherwise the default channel is used.

Parameters:

event A string containing the event to send

OR

A table containing the following fields:

name A string containing the event to send

target A string containing the object to target the event to (see Storyboard IO) (optional)

format A string format of the event data payload (optional)

data A table whose keys match up with the keys specified in the format (optional)

channel An optional Storyboard IO channel to send the event on, if not specified the event is added directly into the current Storyboard application event queue if neither the environment variable or global GREIONAME variable are set.

Returns:

true for success, false for failure and error message string, e.g. "Can't open greio channel my_channel"

Example:

```
-- Send to the event to the application :
local success, error = gre.send_event("my_event")
if(success == false) then
    print(error)
    --handle error
end

--To send the event to a Storyboard IO channel via parameters:
local success, error = gre.send_event("my_event", "io_channel_name")
if(success == false) then
    print(error)
    --handle error
end
```

gre.send_event_target

```
gre.send_event_target(
    event_name,
    target,
    [channel]
)
```

Send an event to a targeted model element (control, layer instance or screen) using the model's fully qualified path. The channel is an optional parameter.

Parameters:

event_name A string containing the event to send
target A string containing the object to target the event to (see Storyboard IO)
channel An optional Storyboard IO channel to send the event on, if not specified the event is added directly into the current Storyboard application event queue if neither the environment variable or global GREIONAME variable are set.

Returns:

true for success, false for failure and error message string, e.g. "Can't open greio channel my_channel"

Example:

```
-- Send an event directed at a particular control target:
gre.send_event_target("my_event", "my_layer.my_control")

-- Send an event directed at a particular control on the channel
-- "io_channel_name" and grab the status of the operation
```

```
local success, error = gre.send_event_target("my_event",
    "my_layer.my_control", "io_channel_name")
if(success == false) then
    print(error)
    --handle error
end
```

gre.send_event_data

```
gre.send_event_data(
    event_name,
    format_string,
    data,
    [channel]
)
```

Send an event with custom data to the application or to a Storyboard IO channel. The data parameter is a Lua table where the indexes match the values from the format string. channel is an optional parameter.

Special consideration is required for sending data that is to be formatted as an array (i.e., N[suf]M where M is greater than 0). In this case the data entry should be provided as a Lua table and not as a raw value parameter.

Data parameters must be sent in order of descending alignment requirements. Example: 4u1 4u1 2u1 1s0 is good, 2u1 4u1 4u1 1s0 is not

Parameters:

event_name A string containing the event to send
format_string A string format of the event data payload
data A table whose keys match up with the keys specified in the format_string
channel An optional Storyboard IO channel to send the event on, if not specified the event is added directly into the current Storyboard application event queue if neither the environment variable or global GREIONAME variable are set.

Returns:

true for success, false for failure and error message string, e.g. "Can't open greio channel my_channel"

Example:

```
-- Send a 'int32_update' event with a 32bit signed integer (int32_t)
-- payload to the 'controller' channel
function send_integer(value)
    local format = "4s1 value"
    local data = {}
    data["value"] = value
    local success, error = gre.send_event_data("int32_update", format,
        data, "controller")
    if(success == false) then
```

```
        print(error)
        --handle error
    end
end

-- Send a 'int16_update' event with two 16bit signed integers (int16_t)
-- payload to the 'controller' channel
function send_two_integers(value1, value2)
    local format = "2s1 first 2s1 second"
    local data = {}
    data["first"] = value1
    data["second"] = value2
    local success, error = gre.send_event_data("int16_update", format,
    data, "controller")
    if(success == false) then
        print(error)
        --handle error
    end
end

-- Send an 'array_update' event with an array of int32_t numbers (provided
-- as a table) to the client
function send_integer_array(values)
    -- Generate the format string dynamically based on the number of entries
    local count = #values
    local format = string.format("4s%d values", count)
    local data = {}
    data["values"] = values
    local success, error = gre.send_event_data("array_update", format,
    data, "controller")
    if(success == false) then
        print(error)
        --handle error
    end
end

send_integer(12)
send_two_integers(10, 20)
send_integer_array({10, 20, 30, 40})
```

gre.greio_disconnect

```
gre.greio_disconnect(
    channel,
    [is_receive_channel]
)
```

This function forces any cached Storyboard IO channel connections to the specified channel to be closed. Subsequent calls using the same channel name will re-establish the connection to the channel if required.

Parameters:

`channel` The channel that is to be disconnected.
`is_receiving` An optional boolean parameter.
 -**True** if closing a receiving channel.
 -**False** or no argument if closing a sending channel.

Example:

```
-- Send an event to a custom channel
gre.send_event("StoryboardRocks", "my_channel")
-- Close the cached connection to that channel
gre.greio_disconnect("my_channel")
```

gre.clone_object

```
gre.clone_object(
    reference_object_name,
    new_object_name,
    parent_name,
    data
)
```

Create a new control, table or group (`new_object_name`) within an existing parent (`parent_name`) by copying all of the properties of an existing object (`reference_object_name`). This new object will have all of the same actions, variables and it's current state will match the state of the reference object that is being copied. In the case of cloning groups, all of the reference group's controls will be cloned into the new group.

Currently only controls and groups are supported as source reference objects.

- Controls and tables can be cloned into either layer or group parents as long as no name conflict exists.
- Groups can be cloned into layer parents as long as no name conflict exists.

The data argument is a table of properties that match the properties for that type of object as described in the `gre.set_control_attrs`, `gre.set_table_attrs` or `gre.set_group_attrs` functions as applicable. For example, most objects support `x`, `y`, `hidden` properties

Parameters:

`reference_object_name` The name of the object that will be cloned. This may be a fully qualified name of a group or control
`new_object_name` The name for the new object, this must be a unique name in the parents namespace
`parent_name` The name of the layer or group to place this object within, this object must exist
`data` Optional: A table containing control attribute tags as the keys with new values to be applied.

Example:

```
function create_new_control()
    local data = {}
```

```
data["x"] = 10
data["y"] = 10
gre.clone_object("my_control", "my_new_control", "my_layer", data)
end
```

gre.delete_object

```
gre.delete_object(
    object_name,
)
```

Delete an object that has been cloned using `gre.clone_object` from the model. The object must be a control, table or a group.

Parameters:

`object_name` The fully qualified name of the object to delete

Example:

```
function delete_object()
    gre.delete_object("my_layer.my_object")
end
```

gre.clone_control

```
gre.clone_control(
    reference_control_name,
    new_control_name,
    layer_name,
    data
)
```

This is a function to clone a control. This function has been deprecated and has been replaced with the `gre.clone_object` Lua API function. See the `gre.clone_object` for usage and examples.

gre.delete_control

```
gre.delete_control(
    control_name
)
```

This is a function to delete a cloned control. This function has been deprecated and has been replaced with the `gre.delete_object` Lua API function. See the `gre.delete_object` for usage and examples.

gre.poly_string

```
gre.poly_string(  
    x_values,  
    y_values  
)  
or  
gre.poly_string(  
    {{x=, y=}, ...}  
)
```

This is a higher performance function for generating a polygon string based on a set of numeric data points maintained in Lua table arrays.

In the two argument form, the function receives as inputs two Lua tables whose content represents the numeric x and y data points to be converted to a string. The tables are 1 based arrays and must be of the same length.

In the single argument form, the function receives as input a single Lua table whose array content are tables with an "x" and "y" member value.

The string returned is designed to be compatible with the Storyboard polygon plugin and is in the form of X1:Y1 X1:Y2 ...

Parameters:

x_values,
y_values An table containing numeric data for the x and y points respectively.

{{x=, y=}} A table containing tables with x and y members specifying the x and y points.

Example:

```
-- Create a triangle polygon in a 100x100 square  
local x_points = { 0, 50, 100 }           -- Left, Middle, Right  
local y_points = { 100, 0, 100 }          -- Bottom, Top, Bottom  
local xy_string = gre.poly_string(x_points, y_points)  
print("X Y String: " .. xy_string)  
  
-- Create the same triangle, but with x,y member variables  
local xy_points = { {x=0,y=100}, {x=50,y=0}, {x=100,y=100} }  
local xy_string = gre.poly_string(xy_points)  
print("XY String: " .. xy_string)
```

gre.get_string_size

```
gre.get_string_size(  
    font,
```



```
    font_size,  
    string,  
    length,  
    width,  
    data  
)
```

Calculate the area in pixels which the given string will occupy on the screen.

This call can only be made from the main Lua action execution thread.

Parameters:

string	The string to render
font	The name of the font to render in
font_size	The size of the font to render in
string_length	The length of the string to render or 0 for all (optional)
width	A clipping width (in pixels) for the string, used to calculate how many characters fit
data	A table containing additional options. Valid options are: ["length"] - number of characters to process ["spacing"] - letter spacing parameter ["wrapping"] - boolean to activate wrapping logic

Returns:

A table containing the following entries:
"width" width in pixels of the widest line that will be rendered
"height" total string height in pixels
"n_lines" number of lines text occupies when rendered

gre.load_resource

```
gre.load_resource(  
    pool_name,  
    resource_name,  
    [pool parameters]  
)
```

This function will force the loading of a resource, such as an image or font, into the Storyboard application. This can be used in order to avoid load time delays that may be incurred as resources are lazy loaded into the application.

This call can only be made from the main Lua action execution thread.

Parameters:

pool_name	The name of the resource pool: image or font
resource_name	The name of the resource that is to be loaded

The optional parameters vary depending on the pool being specified may not be required:

image pool:
 w The width to cache the image at
 h The height to cache the image at

background Whether or not to load the image asynchronously the 'background'. Asynchronous loading capabilities are determined by the hardware rendering capabilities of the system and may require serialization with the main rendering thread for a complete load to occur.

font pool:

size The point size of the font to load (required)

antialias A flag indicating if anti aliasing is to be used

These options should be passed as a table as the third parameter to ensure that the loader receives the appropriate values.

On completion of a 'background' loaded resource, the following event is sent:

gre.resource_loaded 1s0 resource

Example:

```
-- Call this to pre-load the image and font into the cache
function on_app_init(mapargs)
    -- Call this to pre-load a font at a 24pt size
    local opt = {}
    opt.size = 24
    gre.load_resource("font", "fonts/DejaVu.ttf", opt)

    -- Call this to pre-load the image unscaled
    gre.load_resource("image", "images/tree.jpg")

    -- Call this to pre-load the image and scale it to 100x100
    local opt = {}
    opt.w = 100
    opt.h = 100
    gre.load_resource("image", "images/scaledtree.jpg", opt)

    -- Call this to pre-load the image and scale it to 100x100 asynchronously
    local opt = {}
    opt.w = 100
    opt.h = 100
    opt.background = 1
    gre.load_resource("image", "images/scaledtreebg.jpg", opt)
end
```

gre.dump_resource

```
gre.dump_resource(
    pool_name,
    resource_name
)
```

This function performs the opposite of the `gre.load_resource` call and removes a resource from the specified resource pool cache.

This call can only be made from the main Lua action execution thread.

Parameters:

pool_name The name of the resource pool: image or font
resource_name The name of the resource that is to be removed

Example:

```
-- Force the tree.jpg image out of the cache, image will reload as required
function flush_tree_image()
    gre.dump_resource("image", "images/tree.jpg")
    gre.dump_resource("font", "fonts/DejaVu.ttf:18")
end
```

gre.walk_pool

```
gre.walk_pool(
    pool_name
)
```

This function reports on the memory used by all of the resources loaded into a particular resource pool.

Parameters:

pool_name The resource pool whose content should be reported

Returns:

A table is returned with keys as the resources that are contained in the pool and values indicating the number of bytes that a particular resource is using within the system.

Example:

```
-- Display the content of the current image cache
function show_image_cache(mapargs)
    print("Images")
    local data = gre.walk_pool("image")
    for k,v in pairs(data) do
        print("  ".. tostring(k) .. "=" .. tostring(v))
    end
end
```

gre.load_image

```
gre.load_image(
    image_name,
```

```
    [optional table of parameters]
)
```

This function will force the loading of an image into the Storyboard application. This can be used in order to avoid load time delays that may be incurred as resources are lazy loaded into the application.

This is a convenience function for calling the more generic `gre.load_resource`

This call can only be made from the main Lua action execution thread.

Parameters:

`resource_name` The name of the resource that is to be loaded

The optional parameters are as follows:

`w` The width to cache the image at
`h` The height to cache the image at
`background` Whether or not to load the image asynchronously the 'background'. Asynchronous loading capabilities are determined by the hardware rendering capabilities of the system and may require serialization with the main rendering thread for a complete load to occur.

On completion of a 'background' loaded resource, the following event is sent:

`gre.resource_loaded` `1s0 resource`

gre.timer_set_timeout

```
gre.timer_set_timeout(  
    function,  
    timeout  
)
```

This function creates a one-shot timer which fires after "timeout" milliseconds and then executes "function"

Parameters:

`function` The function to be called when the timer fires
`timeout` The time in milliseconds before the timer should fire

Returns:

A piece of `lightuserdata` which serves as an identifier for the timer

Example:

```
local idval = {}  
function cb_func()  
    print("CB FUNC HAS BEEN CALLED")  
end
```

```
--Call cb_func after 1 second
function cb_set_timeout()
    idval = gre.timer_set_timeout(cb_func, 1000)
end
```

gre.timer_set_interval

```
gre.timer_set_interval(
    function,
    interval
)
```

This function creates a repeating timer which fires every "interval" milliseconds and then executes "function"

Parameters:

function	The function to be called when the timer fires
interval	The time in milliseconds of how often the timer should fire

Returns:

A piece of lightuserdata which serves as an identifier for the timer

Example:

```
local idval = {}
function cb_func()
    print("CB FUNC HAS BEEN CALLED")
end

--Call cb_func every 2 seconds
function cb_set_interval()
    idval = gre.timer_set_interval(cb_func, 2000)
end
```

gre.timer_clear_timeout

```
gre.timer_clear_timeout(
    id
)
```

This function stops an existing timer from firing

Parameters:

id	The lightuserdata representing the timer
----	--

Returns:

Nothing

Example:

```
local idval = {}
function cb_func()
    print("CB FUNC HAS BEEN CALLED")
end

--Call cb_func after 5 seconds
function cb_set_timeout()
    idval = gre.timer_set_timeout(cb_func, 2000)
end

function cb_clear_timeout()
    local data

    data = gre.timer_clear_timeout(idval)
end
```

gre.timer_clear_interval

```
gre.timer_clear_interval(
    id
)
```

This function stops an existing timer from firing

Parameters:

id The userdata representing the timer

Returns:

Nothing

Example:

```
local idval = {}
function cb_func()
    print("CB FUNC HAS BEEN CALLED")
end

--Call cb_func every 5 seconds
function cb_set_interval()
    idval = gre.timer_set_interval(cb_func, 2000)
```

```
end

function cb_clear_interval()
    local data

    data = gre.timer_clear_interval(idval)
end
```

gre.animation_create

```
gre.animation_create(fps, [auto_destroy], [end_callback])
```

Create a new animation at the desired frame rate (fps). The second parameter (optional), `auto_destroy`, tells if the animation should be released once completed. If you specify a value of 1 the animation will be released and the returned id is not valid once the animation has completed. The third parameter (optional) indicates a callback function to be invoked when the animation is complete.

Parameters:

- `fps` The animation frame rate
- `auto_destroy` Pass 1 in to release the animation once completed
- `end_callback` Provide a Lua function to be called in the animation

Returns

An animation id to be used on future animation calls, nil on failure.

Example:

```
function animation_create(mapargs, fps)
    local id
    id = gre.animation_create(fps)
end

--Example of an creating an animation with an animation complete callback
local animation_state = "STOPPED"
function animation_create(mapargs, fps)
    local id
    id = gre.animation_create(fps, 0, animation_complete)

    gre.animation_trigger(id, {context="my_layer.my_control",
    id="my_control_animation"})
    animation_state = "RUNNING"
end

--The callback's first argument will be the completed animation's id.
--When triggered with an animation instance id (e.g., "my_control_animation"),
--otherwise it will be the id returned from calling animation_create.
function animation_complete(id)
    animation_state = "COMPLETED"
end
```

gre.animation_add_step

```
gre.animation_add_step(id, data)
```

Add a step to a created animation. The id must be from a call to gre.animation_create. The data parameter defines the animation step values.

Parameters:

id The animation id

data A table of animation step values which can include:

key: The data key for the animation step to act upon

rate: The animation rate string: [linear | easein | easeout | easeinout | bounce]

duration: The length of the step (msec)

offset: The offset from animation start where this step begins (msec)

from: The value to start the animation at, if not specified the value is the current value of "key"

to: The end point for the animation

delta: The delta for the end of the animation from the start point. If both "to" and "delta" are given then the "to" value is used.

Example:

```
function create_animation(mapargs)
    local data = {}

    -- slide the x position 400 pixels over 2000 msec and auto-destroy
    -- it on completion
    id = gre.animation_create(60, 1)
    data["rate"] = "linear"
    data["duration"] = 2000
    data["offset"] = 0
    data["delta"] = 400
    data["key"] = "mylayer.mycontrol.grd_x"
    gre.animation_add_step(id, data)
end
```

gre.animation_destroy

```
gre.animation_destroy(id)
```

Destroy the animation associated with id.

Parameters:

id The animation to destroy

Example:


```
function create_animation(mapargs)
    local data = {}

    -- slide the x position 400 pixels over 2000 msec
    id = gre.animation_create(60)
    data["rate"] = "linear"
    data["duration"] = 2000
    data["offset"] = 0
    data["delta"] = 400
    data["key"] = "mylayer.mycontrol.grd_x"
    gre.animation_add_step(id, data)

    -- destroy it
    gre.animation_destroy(id)
end
```

gre.animation_trigger

```
gre.animation_trigger(animation_id, data)

gre.animation_trigger("animation_name")
```

Trigger an animation to run. If an `animation_id` is used to trigger the animation, then it must be the return value from `gre.animation_create()`. If a name is used to trigger an animation, then that name must be the name of the animation specified in Designer. This function can take an optional parameter, `data_table`. The `data_table` contains the tags and values for the extra arguments to set.

Parameters:

<code>animation_id</code>	The animation to trigger
<code>data</code>	A table containing the tags and values for the extra arguments to set
<code>id</code>	The animation id used in the case of multiple animations with the same name
<code>context</code>	The fully qualified name of an object in the model which will be used as the context for the animation

Example:

```
function create_animation(mapargs)
    local data = {}

    -- slide the x position 400 pixels over 2000 msec and auto-destroy
    -- it on completion
    id = gre.animation_create(60, 1)
    data["rate"] = "linear"
    data["duration"] = 2000
    data["offset"] = 0
    data["delta"] = 400
    data["key"] = "mylayer.mycontrol.grd_x"
    gre.animation_add_step(id, data)

    gre.animation_trigger(id)
```

```
end

--Example of using gre.animation_trigger passing animation names.
function cb_toggle_cur_5day()
    if cur_5day_toggle == false then
        gre.animation_trigger("show_5day")
    else
        gre.animation_trigger("hide_mon_to_fri")
    end
end

--Example of using gre.animation_trigger with context.
function cb_toggle_cur_5day()
    local data = {}

    data["context"] = "Layer1.mycontrol"
    gre.animation_trigger("show_5day", data)
end
```

gre.animation_stop

```
gre.animation_stop(animation_id, data)

gre.animation_stop("animation_name")
```

Stop an animation. If an `animation_id` is used to stop the animation, then it must be the return value from `gre.animation_create()`. If a name is used to stop an animation, then that name must be the name of the animation specified in Designer. This function can take an optional parameter, `data_table`. The `data_table` contains the tags and values for the extra arguments to set.

Parameters:

<code>animation_id</code>	The animation to stop
<code>data</code>	A table containing the tags and values for the extra arguments to set
<code>id</code>	The animation id used in the case of multiple animations with the same name
<code>context</code>	The fully qualified name of an object in the model which will be used as the context for the animation

Example:

```
function create_animation(mapargs)
    local data = {}

    -- slide the x position 400 pixels over 2000 msec and auto-destroy
    -- it on completion
    id = gre.animation_create(60, 1)
    data["rate"] = "linear"
    data["duration"] = 2000
    data["offset"] = 0
    data["delta"] = 400
    data["key"] = "mylayer.mycontrol.grd_x"
```

```
gre.animation_add_step(id, data)

gre.animation_trigger(id)

--do stuff

--Stop the animation
gre.animation_stop(id)
end

--Example of using gre.animation_stop passing animation names.
function cb_stop_cur_5day()
    if cur_5day_toggle == false then
        gre.animation_stop("show_5day")
    else
        gre.animation_stop("hide_mon_to_fri")
    end
end

--Example of using gre.animation_stop with context.
function cb_toggle_cur_5day()
    local data = {}

    data["context"] = "Layer1.mycontrol"
    gre.animation_stop("show_5day", data)
end
```

gre.animation_pause

```
gre.animation_pause(animation_id, data)

gre.animation_pause("animation_name")
```

Pause a running animation. If an `animation_id` is used to pause the animation, then it must be the return value from `gre.animation_create()`. If a name is used to pause an animation, then that name must be the name of the animation specified in Designer. This function can take an optional parameter, `data_table`. The `data_table` contains the tags and values for the extra arguments to set.

Parameters:

<code>animation_id</code>	The running animation to pause
<code>data</code>	A table containing the tags and values for the extra arguments to set
<code>id</code>	The animation id used in the case of multiple animations with the same name
<code>context</code>	The fully qualified name of an object in the model which will be used as the context for the animation

Example:

```
function pause_animation(id)
    --Pause the animation
    gre.animation_pause(id)
```

```
end

--Example of using gre.animation_pause passing animation names.
function cb_pause_cur_5day()
    if cur_5day_toggle == false then
        gre.animation_pause("show_5day")
    else
        gre.animation_pause("hide_mon_to_fri")
    end
end

--Example of using gre.animation_pause with context.
function cb_pause_cur_5day()
    local data = {}

    data["context"] = "Layer1.mycontrol"
    gre.animation_pause("show_5day", data)
end
```

gre.animation_resume

```
gre.animation_resume(animation_id, data)

gre.animation_resume("animation_name")
```

Pause a running animation. If an `animation_id` is used to pause the animation, then it must be the return value from `gre.animation_create()`. If a name is used to pause an animation, then that name must be the name of the animation specified in Designer. This function can take an optional parameter, `data_table`. The `data_table` contains the tags and values for the extra arguments to set.

Parameters:

<code>animation_id</code>	The running animation to pause
<code>data</code>	A table containing the tags and values for the extra arguments to set
<code>id</code>	The animation id used in the case of multiple animations with the same name
<code>context</code>	The fully qualified name of an object in the model which will be used as the context for the animation

Example:

```
function resume_animation(id)
    --Resume the animation
    gre.animation_resume(id)
end

--Example of using gre.animation_resume passing animation names.
function cb_resume_cur_5day()
    if cur_5day_toggle == false then
        gre.animation_resume("show_5day")
    else
        gre.animation_resume("hide_mon_to_fri")
    end
end
```

```
    end
end

--Example of using gre.animation_resume with context.
function cb_resume_cur_5day()
    local data = {}

    data["context"] = "Layer1.mycontrol"
    gre.animation_resume("show_5day", data)
end
```

gre.animation_create_tween

```
gre.animation_create_tween(name, tween_callback)
```

Create a new animation tweening (interpolation) function that can be used by both Lua and Animation Timeline animations. The name of the tweening function cannot collide with any existing animation tweening names. The callback function is a generic tween function that will be provided with specific parameters outlining the desired range of the values being tweened and the frame at which the values should be interpolated.

The tweening function is called with four parameters `elapsed`, `base`, `change`, `duration`. The `elapsed` value indicates the interpolation location and is in the range from `[0,duration]` so if one wanted to derive the percentage that the animation has completed, it would be `elapsed/duration`. The `base` value represents the numeric starting point of interpolation and the end point of the animation will be `base+change`. These four values can be applied to various transforms to return a single floating point value that represents the desired outcome of the interpolation function at that point in time. The output of the interpolation is clamped to the range `[base,base+change]` though this restriction may be lifted in future releases of Storyboard.

Parameters:

- `name` The name to use for the tween operation, it is case sensitive and must not collide with other tween names
- `tween_callback` The callback function to be invoked when an animation using the named tween function is invoked

Returns

- `true` on successful registration

Example:

```
-- Re-implemented linear tween
local function CustomTweenCB(elapsed, base, change, duration)
    return base + ((change * elapsed) / duration)
end

-- This is called at application initialization to register the custom
-- tween function
function CBRegisterTween(mapargs)
```

```
    gre.animation_create_tween("MyTween", CustomTweenCB)
end

-- Create an animation that fades out a layer and uses the custom
-- tween function
function CBAAnimation(mapargs)
    local id = gre.animation_create(60)

    local data = {}
    data["rate"] = "MyTween"
    data["duration"] = 2000
    data["offset"] = 0
    data["from"] = 255
    data["to"] = 0
    data["key"] = "MyScreen.MyLayer.grd_alpha"
    gre.animation_add_step(id, data)

    gre.animation_trigger(id)
end
```

gre.touch

```
gre.touch(
    x ,
    y,
    [channel]
)
```

Send a touch event to the application at the co-ordinates passed in through the parameters. `channel` is an optional parameter

Parameters:

<code>x</code>	The x position to simulate the touch event at
<code>y</code>	The y position to simulate the touch event at
<code>channel</code>	An optional Storyboard IO channel to send the event on, if not specified the event is added directly into the current Storyboard application event queue if neither the environment variable or global GREIONAME variable are set.

Returns:

true for success, false for failure and error message string, e.g. "Can't open greio channel my_channel"

Example:

```
function CBTouch()
    gre.touch(25, 50)
end

function CBSendTouch()
    local success, error = gre.touch(25, 50, "my_channel")
```

```
        if(success == false) then
            print(error)
            --handle error
        end
    end
end
```

gre.key_up

```
gre.key_up(
    code,
    [channel]
)
```

Send a key_up event to the application with the scancode passed in the parameters. channel is an optional parameter

Parameters:

code The UTF-8 character code to inject
channel An optional Storyboard IO channel to send the event on, if not specified the event is added directly into the current Storyboard application event queue if neither the environment variable or global GREIONAME variable are set.

Returns:

true for success, false for failure and error message string, e.g. "Can't open greio channel my_channel"

Example:

```
function CBSpaceUp()
    gre.key_up(0x20)
end

function CBSendSpaceUp()
    local success, error = gre.key_up(0x20, "my_channel")
    if(success == false) then
        print(error)
        --handle error
    end
end
```

gre.key_down

```
gre.key_down(
    code,
    [channel]
)
```

Send a key_down event to the application with the scancode passed in the parameters. channel is an optional parameter

Parameters:

code The UTF-8 character code to inject
channel An optional Storyboard IO channel to send the event on, if not specified the event is added directly into the current Storyboard application event queue if neither the environment variable or global GREIONAME variable are set.

Example:

```
function CBSpaceDown()  
    gre.key_down(0x20)  
end  
  
function CBSendSpaceDown()  
    local success, error = gre.key_down(0x20, "my_channel")  
    if(success == false) then  
        print(error)  
        --handle error  
    end  
end
```

gre.key_repeat

```
gre.key_repeat(  
    code,  
    [channel]  
)
```

Send a key_repeat event to the application with the scancode passed in the parameters. channel is an optional parameter

Parameters:

code The UTF-8 character code to inject
channel An optional Storyboard IO channel to send the event on, if not specified the event is added directly into the current Storyboard application event queue if neither the environment variable or global GREIONAME variable are set.

Returns:

true for success, false for failure and error message string, e.g. "Can't open greio channel my_channel"

Example:

```
function CBKeyRepeat()  
    gre.key_repeat(0x20)  
end  
  
function CBSendKeyRepeat()  
    local success, error = gre.key_repeat(0x20, "my_channel")  
    if(success == false) then
```



```
        --handle error
    end
end
```

gre.redraw

```
gre.redraw(
    x,
    y,
    width,
    height,
    [channel]
)
```

Force a screen redraw. channel is an optional parameter. Specifying a x,y,width,height of 0 will result in a full screen refresh occurring.

Parameters:

x	The x position of the redraw bounding box event
y	The y position of the redraw bounding box event
width	The width position of the redraw bounding box event
height	The height position of the redraw bounding box event
channel	An optional Storyboard IO channel to send the event on, if not specified the event is added directly into the current Storyboard application event queue if neither the environment variable or global GREIONAME variable are set.

Returns:

true for success, false for failure and error message string, e.g. "Can't open greio channel my_channel"

Example:

```
function CBRedraw()
    gre.redraw(25, 50, 100, 100)
end

function CBSendRedraw()
    local success, error = gre.redraw(25, 50, 100, 100, "my_channel")
    if(success == false) then
        print(error)
        --handle error
    end
end
```

gre.quit

```
gre.quit(
    [channel]
)
```

Send QUIT event to application to force shutdown. `channel` is an optional parameter.

Parameters:

`channel` An optional Storyboard IO channel to send the event on, if not specified the event is added directly into the current Storyboard application event queue if neither the environment variable or global GREIONAME variable are set.

Returns:

true for success, false for failure and error message string, e.g. "Can't open greio channel my_channel"

Example:

```
function CBQuit()
    gre.quit()
end

function CBSendQuit()
    local success, error = gre.quit("my_channel")
    if(success == false) then
        print(error)
        --handle error
    end
end

-- Send a quit message to the application
gre.quit()
```

gre.thread_create

```
gre.thread_create(func)
```

This function starts a new operating system thread of execution that is independent from Storyboard's main thread. The function provided as an argument indicates the starting context for this new thread of execution.

The Storyboard data and event (`get_data/set_data/send_event`) API are thread safe. However the execution of data changes outside of the main thread of execution can have a significant impact on performance of the application and the preferred way of synchronizing data obtained in a thread with the Storyboard UI thread is by using a Storyboard IO event and sending the data via `gre.send_event` or `gre.send_event_data`. There are no thread specific synchronization primitives, such as mutexes, for synchronizing Lua data access, serialize to the main thread using an event if this is a requirement.

In scenarios where a controlled shutdown and restart of a Storyboard application is required, separate threads of execution pose a synchronization challenge. In these situations all created thread(s) must have their execution interrupted and terminate in order for a clean shutdown to be observed. This can be accomplished nominally by intercepting the `gre.quit` event and then taking appropriate action to flag a shutdown variable or send an unblocking event.

This function is not available on all systems and is not available if `gre.thread_create` is set to nil.

Parameters:

func The Lua function to run in a separate thread of execution from the main UI thread.

Example:

```
-- Flag to indicate that we want our threads to quit executing
local quit_threads = false

-- Run a poll loop waiting for a file (a_file) to appear and
-- then send an event
function async_function()
    while(not quit_threads) do
        local fp = io.open("a_file")
        if(fp ~= nil) then
            fp:close()
            gre.send_event("file_created")
            return
        end
    end
end

-- Create the monitoring thread of execution
gre.thread_create(async_function)
```

gre.receive_event

```
gre.receive_event(
    channel
)
```

Receive an event from a Storyboard IO channel. This is a blocking call and works best when used on a separate Lua thread.

Parameters:

channel A Storyboard IO channel to receive the event on.

Returns:

event A table containing the name, target, format and a data table from a received event.

Example:

```
-- Receive a Storyboard IO event with data payload x, y, z:
ev = gre.receive_event("my_channel")

if ev ~= nil then
    print(ev.name)
```

```
    for k,v in pairs(ev.data) do
        print(tostring(k).. " " ..tostring(v))
    end
end

--To disconnect from my_channel:
gre.greio_disconnect("my_channel", true)
```

gre.env

```
gre.env(
    string_key
)
or
gre.env(
    table
)
```

Return information about the Storyboard runtime environment. The input can be either a single string containing the key to look up or a table of keys for variables to match. The following table describes the available keys:

version	The version of this engine as a string value. The format of the string is four version numbers separated by dots: major.minor.service.build.
target_os	The target operating system
target_cpu	The target processor
renderer	The name of the graphics rendering technology being used.
screen_width	The dimensions of the screen width
screen_height	The dimensions of the screen height
active_screen	The name of the currently active screen
render_caps	The rendering capabilities. Currently the only defined capability is "3d" if 3D rendering is supported
mem_stats	Platform memory statistics for the engine. The results are returned in a table with string keys, indicating the type of memory value being reported, and number values in bytes or nil if the system doesn't support the memory information request.
	The key <code>process_used</code> returns the number of bytes of memory that is being used by the Storyboard Engine process.
	The key <code>heap_used</code> returns the number of bytes of memory that is specifically being used by the heap. This number is usually smaller than the process memory and does not typically include large memory allocations for display framebuffers or memory mapped files.
greio_channel	The current application's IO channel name.

Parameters:

key A string or a table of strings to look up.

Returns:

If a single string is provided as an input argument, just a single data value for that argument is returned

If a table is provided as an input argument, then a table with key/value pairs corresponding to the keys of the input argument and the results they provide.

Example:

```
-- Get the target OS for dynamic module loading
local os = gre.env("target_os")
print("Running on target OS: " .. tostring(os))

-- Report on the Storyboard version and rendering technology
local info = gre.env({"version", "renderer"})
local msg = string.format("Storyboard version %s (%s renderer)", \
    info.version, info.renderer)
print(msg)
```

gre.log

```
gre.log(
    id,
    msg
)
```

Generate a log message that will be carried through the Storyboard engine logging system. This can be more effective than using standard Lua print statement because the output can be redirected to different outputs.

Parameters:

id	An integer value that matches the enumeration in >gre/gre.h< to prefix the log message with a code, or -1 to simply output the message. If non-negative value is used, then the output will be subjected to the verbosity logging filtering.
msg	The message to output to the logging system.

Returns:

Example:

```
-- These values are defined in gre for use with gre.log
--gre.LOG_ALWAYS   = -1,  -- Always go to stdout
--gre.LOG_ERROR    = 0,   -- Errors (fatal and non-fatal)
--gre.LOG_WARNING  = 1,   -- Warnings
--gre.LOG_INFO     = 2,   -- Information, one time, non-repetitive
--gre.LOG_EVENT1   = 3,   -- Event delivery, excluding motion/mtevent/redraws
--gre.LOG_ACTION   = 4,   -- Action execution
--gre.LOG_DIAG1    = 5,   -- Storyboard diagnostic informative
```

```
--gre.LOG_DIAG2    = 6,    -- Storyboard diagnostic detailed
--gre.LOG_EVENT2    = 7,    -- Motion/mtevent/redraw event delivery
--gre.LOG_TRACE1    = 8,    -- Storyboard minimal tracing
--gre.LOG_TRACE2    = 9     -- Storyboard maximum tracing

-- Generate an error message
gre.log(gre.LOG_ERROR, "This is an error message")

-- Generate an diagnostic information message
gre.log(gre.LOG_INFO, "This is an info  message")

-- Generate an unfiltered message
gre.log(gre.LOG_ALWAYS, "This message will always be shown")
```

gre.mstime

```
gre.mstime()
gre.mstime(app_relative)
```

Retrieve the current time in milliseconds in the default, no argument flavour. This call provides a higher resolution than the standard Lua `os.clock()` or `os.date()` functions.

When `true` is passed in as an argument, then the time returned will be relative to the application start time and aligned with the timestamps that are generated by the Storyboard logging API.

Returns:

The current time in milliseconds in a system specific manner (`gre.mstime()`) or the time in milliseconds since the start of the application (`gre.mstime(true)`)

Example:

```
-- Time an operation
local s = gre.mstime()
my_function()
local e = gre.mstime()
print("my_function took " .. tostring(e - s) .. "ms")

-- Determine how long from app start to this point
local delta = gre.mstime(true)
print(string.format("Application start to now: %d ms", delta))
```

gre.rgb

```
gre.rgb(
  r,
```

```
    g,  
    b,  
    [a]  
  )
```

Create a color number value from the individual red, green and blue color components. An optional fourth argument, alpha, may be provided.

Parameters:

r	The red color field as a value from 0-255
g	The green color field as a value from 0-255
b	The blue color field as a value from 0-255
a	An optional alpha value from 0-255

Returns:

A number value that represents the combined red, green, blue and alpha values

Example:

```
-- Create a grey single color value  
local grey = gre.rgb(100, 100, 100)  
  
-- Extract the red, green and blue values from the grey  
local r, g, b = gre.torgb(grey)
```

gre.torgb, gre.to_rgb

```
gre.torgb(  
  color  
)  
gre.to_rgb(  
  color  
)
```

Extract the red, green, blue and alpha components from a single color value.

Parameters:

value	The color value as a number
-------	-----------------------------

Returns:

The red, green, blue and alpha components as a multi-return value set

Example:

```
-- Create a grey single color value  
local grey = gre.rgb(100, 100, 100)  
  
-- Extract the red, green and blue values from the grey
```

```
local r, g, b = gre.to_rgb(grey)
```

gre.to_alignment

```
gre.to_alignment(  
    alignment  
)
```

Generate a numeric alignment value based on a symbolic, case sensitive, alignment string

Parameters:

alignment A string value of "left", "right", "center", "top", "bottom"

Returns:

A numeric value for alignment corresponding to `gre.LEFT`, `gre.RIGHT`, `gre.CENTER`, `gre.TOP`, `gre.BOTTOM`

Example:

```
-- Set an alignment variable to be left aligned  
local alignment = gre.to_alignment("left")  
gre.set_value("myLayer.myControl.horizontalAlignment", alignment)
```

gre.to_alpha

```
gre.to_alpha(  
    percent  
)
```

Generate a numeric alpha value in the range of 0 - 255 based on a percentage from 0 - 100

Parameters:

percent The percent of transparency (0 = transparent, 100 = opaque)

```
-- Create a 50% transparent value  
local alpha = gre.to_alpha(50)  
gre.set_value("myLayer.myControl.alpha", alpha)
```

gre.rtext_text_extent

```
gre.rtext_text_extent(text, target, table)
```

Gets the extent of a rich text field

Parameters:

text - the rich text string to extent

target - A string containing the parent control of the rich text text you are targeting.

This will target the first rich text text found on the control. If one is found then the extent calculation will use that rich text text's parameters.
table - a table containing optional parameters

Optional parameters:
(these will override any parameters found in the targeted rich text text)

size - the font size to use when calculating extent
line-height - the line height to use when calculating extent. This is a percentage of the font line height
max-width - the max width to use when calculating extent
regular-font - the regular font to use
bold-font - the bold font to use
italic-font - the italic font to use
italic-bold-font - the italic/bold font to use

Returns:
A table containing the rich text width ("width") and height ("height")

gre.perf_trace_point

```
gre.perf_trace_point(  
    trace_description  
)
```

Puts an entry in to the Storyboard performance log with the string that is provided as the description. The time that the trace point occurs at will also be a part of the output in the performance log

Parameters:
trace_description The description to show in the performance log
Example:

```
function CBAnimationDone()  
    print("Animation done")  
    gre.perf_trace_point("Lua Animation Finished")  
end
```

gre.perf_trace_duration

```
gre.perf_trace_duration(  
    trace_description,  
    time  
)
```

Puts an entry in to the Storyboard performance log with the string that is provided as the description and a time. The time that the trace point occurs at will also be a part of the output in the performance log

Parameters:

 trace_description The description to show in the performance log

 time The time to put as a value to show in the performance log

Example:

```
local timeSinceLastCall = 0
function CBTimerFired()
    print("Timer Fired")
    local currentTime = gre.mstime(true);
    if(timeSinceLastCall <= 0) then
        gre.perf_trace_duration("Timer last fired", currentTime)
    else
        gre.perf_trace_duration("Timer last fired", os.difftime(currentTime,timeSinceLastCall))
    end
    timeSinceLastCall = currentTime
end
```

Storyboard Lua Canvas API

The Storyboard canvas API provides an interface for clients to perform their own basic drawing operations into an in-memory buffer. In order to access the drawing operations first a canvas render extension must be created and given a unique name within the application. Once a canvas render extension is created, a Lua canvas object can be accessed using the `gre.get_canvas` call. In this documentation the object returned from the `gre.get_canvas` functions will be identified as a CANVAS object. Methods associated with the CANVAS object must be invoked using the Lua colon (:) notation for example `CANVAS:stroke_rect`

The Lua canvas API is dependent on both the Lua plugin (`libgre-plugin-lua`) and the canvas render extension plugin (`libgre-plugin-canvas`).

CANVAS **gre.get_canvas**(name);

name This is the name of the parameter;

gre.get_canvas

gre.get_canvas(name)

Get a canvas object handle by name

Returns:

 A CANVAS object that represents the canvas or nil if no canvas can be found.

CANVAS:get_dimensions

CANVAS:get_dimensions()

Return the width and height of a canvas object.

Returns:

A table containing two fields width and height

```
function PrintCanvasSize(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    print(string.format("Canvas is %d x %d", size.width, size.height));
end
```

CANVAS:fill

`CANVAS:fill(color)`

Flood fill the entire canvas with a specific color

Parameters:

color An RGB color value as an integer value.

```
-- Flood fill the canvas with a red background value
function FillWithRed(name)
    local canvas = gre.get_canvas(name)
    canvas:fill(0xff0000)
end
```

CANVAS:fill_rect

`CANVAS:fill_rect(x1, y1, x2, y2, color)`

Fill a rectangle defined by the bounding area of x1,y2 to x2,y2 with a specific color

Parameters:

x1 The x position of the first corner
y1 The y position of the first corner
x2 The x position of the second corner
y2 The y position of the second corner
color An RGB color value as an integer value.

```
-- Draw three colored bars across the canvas
function FillRGB(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local rw = size.width / 3
    canvas:fill_rect(0, 0, rw, size.height, 0xff0000)
    canvas:fill_rect(rw, 0, 2*rw, size.height, 0x00ff00)
```

```
        canvas:fill_rect(2*rw, 0, 3*rw, size.height, 0x0000ff)
    end
```

CANVAS:fill_poly

```
CANVAS:fill_poly(xytable, color)
```

Fill the content of a polygon through the points defined in the `xytable` with a specific color. The polygon must be a closed simple polygon.

Parameters:

`xytable` A Lua table `{{x=x1,y=y1},{x=x2,y=y2}...}`. Alternatively two tables of parameters containing an array of points may be provided as in `{x1,x2,...}`, `{y1,y2,...}` similar to the section called “`gre.poly_string`”

`color` An RGB color value as an integer value.

```
-- Fill a triangle using a polygon
function FillRedTrianglePoly(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local mid = size.width / 2

    -- Shrink the bounds to make the lines visible
    size.height = size.height - 2
    size.width = size.width - 2

    local pts = {}
    table.insert(pts, {x=2,y=2})
    table.insert(pts, {x=mid,y=size.height})
    table.insert(pts, {x=size.width,y=2})
    table.insert(pts, pts[1]) --Close the polygon
    canvas:fill_poly(pts, 0xff0000)
end
```

CANVAS:stroke_line

```
CANVAS:stroke_line(x1, y1, x2, y2, color)
```

Stroke a line between the points `x1,y1` to `x2,y2` with a specific color. The width of the line is the last value passed to `CANVAS:set_line_width` or 1 if no width has ever been specified.

Parameters:

`x1` The x position of the first corner
`y1` The y position of the first corner
`x2` The x position of the second corner
`y2` The y position of the second corner
`color` An RGB color value as an integer value.

```
-- Stroke a triangle with three different colored line segments
```

```
function StrokeRGB(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local mid = size.width / 2

    -- Shrink the bounds to make the lines visible
    size.height = size.height - 2
    size.width = size.width - 2

    canvas:stroke_line(2, 2, mid, size.height, 0xff0000)
    canvas:stroke_line(mid, size.height, size.width, 2, 0x00ff00)
    canvas:stroke_line(2, 2, size.width, 2, 0x0000ff)
end
```

CANVAS:stroke_rect

`CANVAS:stroke_rect(x1, y1, x2, y2, color)`

Stroke a rectangle outline defined by the bounding area of `x1,y1` to `x2,y2` with a specific color. The width of the outline is the last value passed to `CANVAS:set_line_width` or 1 if no width has ever been specified.

Parameters:

- `x1` The x position of the first corner
- `y1` The y position of the first corner
- `x2` The x position of the second corner
- `y2` The y position of the second corner
- `color` An RGB color value as an integer value.

```
-- Draw three colored outlines across the canvas
function StrokeRGB(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local rw = size.width / 3
    canvas:stroke_rect(0, 0, rw-1, size.height, 0xff0000)
    canvas:stroke_rect(rw, 0, 2*rw-1, size.height, 0x00ff00)
    canvas:stroke_rect(2*rw, 0, 3*rw-1, size.height, 0x0000ff)
end
```

CANVAS:stroke_poly

`CANVAS:stroke_poly(xytable, color)`

Stroke a polygon through the points defined in the `xytable` with a specific color. The width of the line is the last value passed to `CANVAS:set_line_width` or 1 if no width has ever been specified.

Parameters:

- `xytable` A Lua table `{{x=x1,y=y1},{x=x2,y=y2}...}`. Alternatively two tables of parameters containing an array of points may be provided as in `{x1,x2,...}, {y1,y2,...}` similar to the section called “`gre.poly_string`”
- `color` An RGB color value as an integer value.

```
-- Stroke a triangle using a polygon
function StrokeRedTrianglePoly(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local mid = size.width / 2

    -- Shrink the bounds to make the lines visible
    size.height = size.height - 2
    size.width = size.width - 2

    local pts = {}
    table.insert(pts, {x=2,y=2})
    table.insert(pts, {x=mid,y=size.height})
    table.insert(pts, {x=size.width,y=2})
    table.insert(pts, pts[1]) --Close the polygon
    canvas:stroke_poly(pts, 0xff0000)
end
```

CANVAS:clear_rect

```
CANVAS:clear_rect(x1, y1, x2, y2)
```

Make transparent a rectangle defined by the bounding area of `x1, y2` to `x2, y2`.

Parameters:

- `x1` The x position of the first corner
- `y1` The y position of the first corner
- `x2` The x position of the second corner
- `y2` The y position of the second corner

```
-- Fill a canvas with red and poke a transparent hole in the middle of it
function MakeTransparentHole(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local hole_width = size.width / 3
    local hole_height = size.height / 3

    canvas:fill(0xff0000)
    canvas:clear_rect(hole_width, hole_height, 2*hole_width, 2*hole_height)
end
```

CANVAS:set_pixel

```
CANVAS:set_pixel(x, y, clr)
```

Set the pixel value at `x, y` to the specified color. This is equivalent to drawing a 1x1 filled rectangle.

Parameters:

x The x position pixel
y The y position pixel
color An RGB color value as an integer value.

CANVAS:set_alpha

`CANVAS:set_alpha(value)`

Set the transparency level with which subsequent draw operations should be performed. The default value for alpha is 255 (fully opaque).

Parameters:

value An integer value from 0 (transparent) to 255 (opaque). Values outside this range will be clamped.

```
-- Draw three colored bars with different opacities across the canvas on
-- an orange background
function FillRGB(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local rw = size.width / 3
    canvas:fill(0xff8000)
    canvas:set_alpha(50)
    canvas:fill_rect(0, 0, rw, size.height, 0xff0000)
    canvas:set_alpha(150)
    canvas:fill_rect(rw, 0, 2*rw, size.height, 0x00ff00)
    canvas:set_alpha(255)
    canvas:fill_rect(2*rw, 0, 3*rw, size.height, 0x0000ff)
end
```

CANVAS:set_line_width

`CANVAS:set_line_width(value)`

Set the line width in pixels that all subsequent stroke operations should use. The default value for line width is 1.

Parameters:

value An integer value greater than 1 indicating the pixel width.

```
-- Draw three colored outlines width different widths across the canvas
function StrokeRGB(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local rw = size.width / 3
    canvas:set_line_width(5)
    canvas:stroke_rect(0, 0, rw-1, size.height, 0xff0000)
```

```
    canvas:set_line_width(3)
    canvas:stroke_rect(rw, 0, 2*rw-1, size.height, 0x00ff00)
    canvas:set_line_width(1)
    canvas:stroke_rect(2*rw, 0, 3*rw-1, size.height, 0x0000ff)
end
```

CANVAS:draw_image

`CANVAS:draw_image(name, attrs)`

Draw an image within the canvas directed by the user specified properties.

Parameters:

- `name` The project relative name of the image, same as used in the the section called “Image”
- `attrs` A table of properties containing information about how to draw the image. This table can contain the following keys:
 - `x` The x position of the upper left corner of the text (default 0)
 - `y` The y position of the upper left corner of the text (default 0)
 - `w` The width to scale the image to (default: natural width)
 - `h` The height to scale the image to (default: natural height)

```
-- Draw an image scaled to the canvas size
function DrawImage(name)
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local attrs = {}
    attrs.w = size.width
    attrs.h = size.height
    canvas:draw_image("images/logo.png", attrs)
end
```

CANVAS:draw_text

`CANVAS:draw_text(text, attrs)`

Draw an string within the canvas directed by the user specified properties.

Parameters:

- `text` The text string to display
- `attrs` A table of properties containing information about how to draw the text. This table can contain the following keys:
 - `font` The font to use to render the text (required, no default)
 - `x` The x position of the upper left corner of the image (default 0)
 - `y` The y position of the upper left corner of the image (default 0)
 - `size` The point size to render the text at (default 18)
 - `color` The color to render the text in (default black:0x000000)

```
-- Draw a hello world string centered on the canvas
```



```
function DrawCenteredText(name)
    local msg = "Hello World"
    local canvas = gre.get_canvas(name)
    local size = canvas:get_dimensions()
    local attrs = {}
    attrs.font = "fonts/RobotoBold.ttf"
    attrs.size = 24
    local strSize = gre.get_string_size(attrs.font, attrs.size, msg, 0)
    attrs.x = (size.width - strSize.width) / 2
    attrs.y = (size.height - strSize.height) / 2
    canvas:draw_text("Hello World", attrs)
end
```

Storyboard Lua DOM Module

The Storyboard `gredom` Lua module provides a limited access to the hierarchical model used by the Storyboard Engine. This functionality is provided in a separate plugin (`plugins/libgre-plugin-luagredom.so`) and should be available for any target platform that contains the Lua Storyboard plugin. Client can check for the existence of the DOM module by checking that the symbol `gredom` is not `nil`.

The DOM module provides two sets of function interfaces. The first set of functions are associated with the `gredom` namespace and are used to lookup or access an Lua object (table) that contains a special set of methods (metatable entries) that are used to extract additional information about the object. In this documentation the object returned from the `gredom` namespace functions will be referred to as a `DOMOBJECT` object. Methods associated with the `DOMOBJECT` object must be invoked using the Lua colon (`:`) notation for example `DOMOBJECT:get_name()`

gredom

gredom.get_application

```
gredom.get_application()
```

Get an object handle for the application root

Returns:

A `DOMOBJECT` object that represents the application.

gredom.get_object

```
gredom.get_object(fqn)
```

Get an object handle for the model object matching the specified fully qualified name.

Parameters:

`fqn` The fully qualified name of the model entry (screen, layer, control), or a short name to autosearch for a match

Returns:

A `DOMOBJECT` object that represents the named model object.

gredom.get_control

```
gredom.get_control(control)
```

Get a **CONTROL** object for the model object matching the fully qualified name. If the parameter is a **DOMOBJECT**, this function behaves as a cast.

Parameters:

control The fully qualified name of a control model object, or a **DOMOBJECT** representing a control.

Returns:

A **CONTROL** object that represents a control model object.

gredom.get_table

```
gredom.get_table(table)
```

Get a **TABLE** object for the model object matching the fully qualified name. If the parameter is a **DOMOBJECT**, this function behaves as a cast.

Parameters:

table The fully qualified name of a table model object, or a **DOMOBJECT** representing a table.

Returns:

A **TABLE** object that represents a table model object.

gredom.get_group

```
gredom.get_group(group)
```

Get a **GROUP** object for the model object matching the fully qualified name. If the parameter is a **DOMOBJECT**, this function behaves as a cast.

Parameters:

group The fully qualified name of a group model object, or a **DOMOBJECT** representing a group.

Returns:

A **GROUP** object that represents a group model object.

gredom.get_layer

```
gredom.get_layer(layer)
```

Get a **LAYER** object for the model object matching the fully qualified name. If the parameter is a **DOMOBJECT**, this function behaves as a cast.

Parameters:

layer The fully qualified name of a layer model object, or a **DOMOBJECT** representing a layer.

Returns:

A **LAYER** object that represents a layer model object.

gredom.get_layer_instance

```
gredom.get_layer_instance(layerinstance)
```

Get a LAYERINSTANCE object for the model object matching the fully qualified name. If the parameter is a DOMOBJECT, this function behaves as a cast.

Parameters:

layerinstance The fully qualified name of a layer instance model object, or a DOMOBJECT representing a layer instance.

Returns:

A LAYERINSTANCE object that represents a table model object.

gredom.get_screen

```
gredom.get_screen(screen)
```

Get a SCREEN object for the model object matching the fully qualified name. If the parameter is a DOMOBJECT, this function behaves as a cast.

Parameters:

screen The name of a screen model object, or a DOMOBJECT representing a screen.

Returns:

A SCREEN object that represents a screen model object.

DOMOBJECT

DOMOBJECT is the base class for Lua objects representing Storyboard model objects. All other types, CONTROL, TABLE, GROUP, LAYER, LAYERINSTANCE, SCREEN extend the base functionality. In Lua, this class is called *gredom#domobject*.

DOMOBJECT:get_name

```
DOMOBJECT:get_name()
```

Gets the name of the specified DOM Object

Returns:

The name of the specified DOMOBJECT

DOMOBJECT:get_type

```
DOMOBJECT:get_type()
```

Gets the Storyboard type of the specified DOMOBJECT. The type may be one of gredom.APP, gredom.SCREEN, gredom.LAYER, gredom.LAYER_INSTANCE, gredom.GROUP, gredom.CONTROL, gredom.TABLE, gredom.TEMPLATE.

Returns:

A the type of the specified DOMOBJECT.

DOMOBJECT:get_parents

```
DOMOBJECT:get_parents()
```

Gets the parent DOMOBJECT objects for the specified DOMOBJECT. An array of parents is returned because in some cases, such as for a layer, there may be more than one parent representation.

Returns:

An array table containing the parent DOMOBJECT entries.

DOMOBJECT:get_children

```
DOMOBJECT:get_children()
```

Gets the child DOMOBJECT objects for the specified DOMOBJECT. This function returns only the model objects and does not include the variables.

Returns:

An array table containing the child DOMOBJECT entries

DOMOBJECT:get_variables

```
DOMOBJECT:get_variables()
```

Gets variables associated with the specified DOMOBJECT

Returns:

An array table containing the variables associated with this object.

DOMOBJECT:get_value

```
DOMOBJECT:get_value(  
    key  
    [, key2, ...]  
)
```

Gets one or more values from the data manager. Each argument to the function should be a key relative to the DOMOBJECT. The keys are fully qualified before being used as a data manager key whose value should be extracted from the data manager. This function returns multiple values if multiple parameters are provided.

Parameters:

key A relative key, to be fully qualified in the context of this DOMOBJECT.

Returns:

The value associated with the data manager entry for the key, or nil if no entry exists. If multiple

keys are specified, then multiple return values will be generated matching the argument order.

DOMOBJECT:get_data

```
DOMOBJECT:get_data(  
    key  
    [, key2, ...]  
)
```

Gets one or more values from the data manager. Each argument to the function should be a key relative to the DOMOBJECT. The keys are fully qualified before being used as a data manager key whose value should be extracted from the data manager. This function returns a table using all the keys as indexes and the corresponding value is the data returned from the data manager. A nil is returned for any values that do not match a key in the data manager.

Parameters:

key A relative key, to be fully qualified in the context of this DOMOBJECT.

Returns:

A table containing the passed in arguments as keys and the resulting data manager values as the values associated with those keys.

DOMOBJECT:set_value

```
DOMOBJECT:set_value(  
    key,  
    value  
    [, key2, value2, ...]  
)
```

Sets one or more values in the data manager. Each key should be relative to the DOMOBJECT. The keys are fully qualified before being used as a data manager key.

Parameters:

key A relative key, to be fully qualified in the context of this DOMOBJECT
value The value to be assigned

DOMOBJECT:set_data

```
DOMOBJECT:set_data(  
    table  
)
```

Sets one or more items in the Storyboard application's data manager. Each key should be relative to the DOMOBJECT. The keys are fully qualified before being used as a data manager key.

Parameters:

table A table containing the variable to change as the relative key and the value to change it to as the fully qualified

DOMOBJECT:key

```
DOMOBJECT:key(  

```

```
    name  
)
```

Get a fully qualified key from a key that is relative to the DOMOBJECT. This function does not ensure that the fully qualified key is valid.

```
string.format("%s.%s", DOMOBJECT:get_name(), name)
```

Parameters:

name A string containing a relative key, to be fully qualified in the context of this DOMOBJECT

Returns:

A fully qualified string.

CONTROL

CONTROL is the class for Lua objects that represent Storyboard control model objects. CONTROL extends DOMOBJECT so objects of this type can also invoke all the functions of DOMOBJECT. In Lua, this class is called *gredom#control*.

CONTROL:get_x

```
CONTROL:get_x()
```

Returns:

The `grd_x` value of this control

CONTROL:get_y

```
CONTROL:get_y()
```

Get the value of `grd_y` for this control.

Returns:

The `grd_y` value of this control

CONTROL:get_width

```
CONTROL:get_width()
```

Get the value of `grd_width` for this control.

Returns:

The `grd_width` value of this control

CONTROL:get_height

```
CONTROL:get_height()
```

Get the value of `grd_height` for this control.

Returns:

The `grd_height` value of this control

CONTROL:get_position

`CONTROL:get_position()`

Get the value of `grd_x` and `grd_y` for this control.

Returns:

The `grd_x` value of this control

The `grd_y` value of this control

CONTROL:get_size

`CONTROL:get_size()`

Get the value of `grd_width` and `grd_height` for this control.

Returns:

The `grd_width` value of this control

The `grd_height` value of this control

CONTROL:get_bounds

`CONTROL:get_bounds()`

Get the value of `grd_x`, `grd_y`, `grd_width` and `grd_height` for this control.

Returns:

The `grd_x` value of this control

The `grd_y` value of this control

The `grd_width` value of this control

The `grd_height` value of this control

CONTROL:get_hidden

`CONTROL:get_hidden()`

Get the value of `grd_hidden` for this control.

Returns:

The `grd_hidden` value of this control

CONTROL:set_x

```
CONTROL:set_x(  
    x,  
    [anchor]  
)
```

Set the value of `grd_x` for this control to the provided value.

Optional anchors can allow for automatic adjustment of x values to keep the control anchored in the desired position as the control's size change:

Optional anchor allow for automatic adjustment of x value:

Horizontal:

- `grd.LEFT` anchor will not adjust the input. This is the default behavior.
- `grd.CENTER` anchor will adjust the x input by half the control width.
- `grd.RIGHT` anchor will adjust the x input by the full control width.

Parameters:

`x` An integer value to set to `grd_x` of this control
`anchor` Horizontal anchor: `gre.LEFT`, `gre.CENTER`, `gre.RIGHT` (Optional, default is `gre.LEFT`)

CONTROL:set_y

```
CONTROL:set_y(  
    y,  
    [anchor]  
)
```

Set the value of `grd_y` for this control to the provided value.

Optional anchor allow for automatic adjustment of y value:

Vertical:

- `grd.TOP` anchor will not adjust the input. This is the default behavior.
- `grd.CENTER` anchor will adjust the y input by half the control height.
- `grd.BOTTOM` anchor will adjust the y input by the full control height.

Parameters:

`y` An integer value to set to `grd_y` of this control
`anchor` Vertical anchor: `gre.TOP`, `gre.CENTER`, `gre.BOTTOM` (Optional, default is `gre.TOP`)

CONTROL:set_width

```
CONTROL:set_width(  
    width,  
    [anchor]  
)
```

Set the value of `grd_width` for this control to the provided value

Optional anchors can allow for automatic adjustment of the control's x to keep the control anchored in the desired position as the control's width change:

Horizontal:

- `grd.LEFT` anchor will not adjust the control's x position. This is the default behavior.

- `grd.CENTER` anchor will adjust the control's x position by half the difference in control width.
- `grd.RIGHT` anchor will adjust the control's x position by the difference in control width.

Parameters:

- `width` An integer value to set to `grd_width` of this control
- `anchor` Horizontal anchor: `gre.LEFT`, `gre.CENTER`, `gre.RIGHT` (Optional, default is `gre.LEFT`)

CONTROL:set_height

```
CONTROL:set_height(  
    height,  
    [anchor]  
)
```

Set the value of `grd_height` for this control to the provided value

Optional anchors can allow for automatic adjustment of the control's y to keep the control anchored in the desired position as the control's height change:

Vertical:

- `grd.TOP` anchor will not adjust the control's y position. This is the default behavior.
- `grd.CENTER` anchor will adjust the control's y position by half the difference in control height.
- `grd.BOTTOM` anchor will adjust the control's y position the difference in control height.

Parameters:

- `height` An integer value to set to `grd_height` of this control
- `anchor` Vertical anchor: `gre.TOP`, `gre.CENTER`, `gre.BOTTOM` (Optional, default is `gre.TOP`)

CONTROL:set_position

```
CONTROL:set_position(  
    x,  
    y,  
    [ah],  
    [av]  
)
```

Set the value of `grd_x` and `grd_y` for this control to the provided values

Optional anchor allow for automatic adjustment of y and values:

Horizontal:

- `grd.LEFT` anchor will not adjust the input. This is the default behavior.
- `grd.CENTER` anchor will adjust the x input by half the control width.
- `grd.RIGHT` anchor will adjust the x input by the full control width.

Vertical:

- `grd.TOP` anchor will not adjust the input. This is the default behavior.
- `grd.CENTER` anchor will adjust the y input by half the control height.
- `grd.BOTTOM` anchor will adjust the y input by the full control height.

Parameters:

- `x` An integer value to set to `grd_x` of this control
- `y` An integer value to set to `grd_y` of this control
- `ah` Horizontal anchor: `gre.LEFT`, `gre.CENTER`, `gre.RIGHT` (Optional, default is `gre.LEFT`)
- `av` Vertical anchor: `gre.TOP`, `gre.CENTER`, `gre.BOTTOM` (Optional, default is `gre.TOP`)

CONTROL:set_size

```
CONTROL:set_size(  
    width,  
    height,  
    [ah],  
    [av]  
)
```

Set the value of `grd_width` and `grd_height` for this control to the provided values

Optional anchors can allow for automatic adjustment of the control's x and y to keep the control anchored in the desired position as the control's size changes:

Horizontal:

- `grd.LEFT` anchor will not adjust the control's x position. This is the default behavior.
- `grd.CENTER` anchor will adjust the control's x position by half the difference in control width.
- `grd.RIGHT` anchor will adjust the control's x position by the difference in control width.

Vertical:

- `grd.TOP` anchor will not adjust the control's y position. This is the default behavior.
- `grd.CENTER` anchor will adjust the control's y position by half the difference in control height.
- `grd.BOTTOM` anchor will adjust the control's y position the difference in control height.

Parameters:

- `width` An integer value to set to `grd_width` of this control
- `height` An integer value to set to `grd_height` of this control
- `ah` Horizontal anchor: `gre.LEFT`, `gre.CENTER`, `gre.RIGHT` (Optional, default is `gre.LEFT`)
- `av` Vertical anchor: `gre.TOP`, `gre.CENTER`, `gre.BOTTOM` (Optional, default is `gre.TOP`)

CONTROL:set_bounds

```
CONTROL:set_bounds(  
    x,  
    y,
```

```
        width,  
        height  
    )
```

Set the value of `grd_x`, `grd_y`, `grd_width` and `grd_height` for this control to the provided values

Parameters:

- `x` An integer value to set to `grd_x` of this control
- `y` An integer value to set to `grd_y` of this control
- `width` An integer value to set to `grd_width` of this control
- `height` An integer value to set to `grd_height` of this control

CONTROL:set_hidden

```
CONTROL:set_hidden(  
    hidden  
)
```

Set the value of `grd_hidden` to the provided value.

Parameters:

- `hidden` An integer value to set to `grd_hidden` of this control

CONTROL:hide

```
CONTROL:hide()
```

Set the value of `grd_hidden` for this control to 1.

CONTROL:show

```
CONTROL:show()
```

Set the value of `grd_hidden` for this control to 0.

CONTROL:clone

```
CONTROL:clone(  
    new_object_name,  
    parent_name,  
    data  
)
```

Create a new control (`new_object_name`), within an existing parent layer or group (`parent_name`) by copying all of the properties of this control. This new object will have all of the same actions, variables and it's current state will match the state of this control.

- Controls and tables can be cloned into either layer or group parents as long as no name conflict exists.

The data argument is a table of properties that match the properties for that type of object as described in the `gre.set_control_attrs` or `gre.set_table_attrs` functions as applicable.

Parameters:

`new_object_name` The name for the new object, this must be a unique name in the parents namespace
`parent_name` The name of the layer or group to place this object within, this object must exist
`data` Optional: A table containing control attribute tags as the keys with new values to be applied.

Returns:

A **CONTROL** object representing the newly created control.

CONTROL:delete

```
CONTROL:delete()
```

Delete this control. This function only works on controls that were created as clones of other controls.

TABLE

TABLE is the class for Lua objects that represent Storyboard table model objects. TABLE extends **CONTROL** so objects of this type can also invoke all the functions of **CONTROL** and **DOMOBJECT**. In Lua, this class is called *gredom#tablecontrol*.

TABLE:get_rows

```
TABLE:get_rows()
```

Get the value of `grd_rows` for this table.

Returns:

The `grd_rows` value of this table

TABLE:get_cols

```
TABLE:get_cols()
```

TABLE:cell_key

```
TABLE:cell_key(  
    row,  
    col,  
    name  
)
```

Get a fully qualified key from a name that is relative to the **TABLE** object. This function does not ensure that the fully qualified key is valid. Equivalent to:

```
string.format("%s.%d.%d.%s", TABLE:get_name(), row, col, name)
```

Parameters:

`row` The row
`col` The column
`name` A string containing a relative key, to be fully qualified in the context of this **TABLE**

Returns:

A fully qualified string.

GROUP

GROUP is the class for Lua objects that represent Storyboard group model objects. GROUP extends DOMOBJECT so objects of this type can also invoke all the functions of DOMOBJECT. In Lua, this class is called *gredom#group*.

GROUP:get_x

```
GROUP:get_x( )
```

Get the value of `grd_x` for this group.

Returns:

The `grd_x` value of this group

GROUP:get_y

```
GROUP:get_y( )
```

Get the value of `grd_y` for this group.

Returns:

The `grd_y` value of this group

GROUP:get_hidden

```
GROUP:get_hidden( )
```

Get the value of `grd_hidden` for this group.

Returns:

The `grd_hidden` value of this group

GROUP:set_x

```
GROUP:set_x(  
    x  
)
```

Set the value of `grd_x` for this group.

Parameters:

x The `grd_x` value of this group

GROUP:set_y

```
GROUP:set_y(  
    y  
)
```

Set the value of `grd_y` for this group.

Parameters:

y The `grd_y` value of this group

GROUP:set_hidden

```
GROUP:set_hidden(  
    hidden  
)
```

Set the value of `grd_hidden` for this group.

Parameters:

hidden The `grd_hidden` value of this group

GROUP:hide

```
GROUP:hide()
```

Set the value of `grd_hidden` for this group to 1.

GROUP:show

```
GROUP:show()
```

Set the value of `grd_hidden` for this group to 0.

GROUP:clone

```
GROUP:clone(  
    name,  
    parent,  
    data  
)
```

Create a new group (`new_object_name`), within an existing parent layer (`parent_name`) by copying all of the properties and children of this group. This new object will have all of the same actions, variables and it's current state will match the state of this control. All of the reference group's controls will be cloned into the new group.

The `data` argument is a table of properties that match the properties for that type of object as described in the `gre.set_group_attrs` function.

Parameters:

<code>new_object_name</code>	The name for the new object, this must be a unique name in the parents namespace
<code>parent_name</code>	The name of the layer to place this object within, this object must exist
<code>data</code>	Optional: A table containing group attribute tags as the keys with new values to be applied.

Returns:

A GROUP object representing the newly created group.

GROUP:delete

```
GROUP:delete()
```

Delete this group. This function only works on group that were created as clones of other controls.

LAYERINSTANCE

LAYERINSTANCE is the class for Lua objects that represent Storyboard layer instance model objects. LAYERINSTANCE extends DOMOBJECT so objects of this type can also invoke all the functions of DOMOBJECT. In Lua, this class is called *gredom#layerinstance*.

LAYERINSTANCE:get_layer

`LAYERINSTANCE:get_layer()`

Get the LAYER for this layer instance's layer.

Returns:

a LAYER object representing this layer instance's layer

LAYERINSTANCE:get_x

`LAYERINSTANCE:get_x()`

Get the value of `grd_x` for this layer instance.

Returns:

The `grd_x` value of this layer instance

LAYERINSTANCE:get_y

`LAYERINSTANCE:get_y()`

Get the value of `grd_y` for this layer instance.

Returns:

The `grd_y` value of this layer instance

LAYERINSTANCE:get_width

`LAYERINSTANCE:get_width()`

Get the value of `grd_width` for the layer instance's layer.

Returns:

The `grd_width` value of this layer instance's layer

LAYERINSTANCE:get_height

`LAYERINSTANCE:get_height()`

Get the value of `grd_height` for this layer instance's layer.

Returns:

The `grd_height` value of this layer instance's layer

LAYERINSTANCE:get_position

```
LAYERINSTANCE:get_position()
```

Get the value of `grd_x` and `grd_y` for this layer instance.

Returns:

- The `grd_x` value of this layer instance
- The `grd_y` value of this layer instance

LAYERINSTANCE:get_size

```
LAYERINSTANCE:get_size()
```

Get the value of `grd_width` and `grd_height` for this layer instance's layer.

Returns:

- The `grd_width` value of this layer instance's layer
- The `grd_height` value of this layer instance's layer

LAYERINSTANCE:get_bounds

```
LAYERINSTANCE:get_bounds()
```

Get the value of `grd_x`, `grd_y`, `grd_width` and `grd_height` for this layer instance

Returns:

- The `grd_x` value of this layer instance
- The `grd_y` value of this layer instance
- The `grd_width` value of this layer instance's layer
- The `grd_height` value of this layer instance's layer

LAYERINSTANCE:get_alpha

```
LAYERINSTANCE:get_alpha()
```

Get the value of `grd_alpha` for this layer instance.

Returns:

- The `grd_alpha` value of this layer instance

LAYERINSTANCE:get_hidden

```
LAYERINSTANCE:get_hidden()
```

Get the value of `grd_hidden` for this layer instance.

Returns:

- The `grd_hidden` value of this layer instance

LAYERINSTANCE:set_x

```
LAYERINSTANCE:set_x(  
    x  
)
```

Set the value of `grd_x` for this layer instance to the provided value.

Parameters:

`x` An integer value to set to `grd_x` of this layer instance

LAYERINSTANCE:set_y

```
LAYERINSTANCE:set_y(  
    y  
)
```

Set the value of `grd_y` for this layer instance to the provided value.

Parameters:

`y` An integer value to set to `grd_y` of this layer instance

LAYERINSTANCE:set_width

```
LAYERINSTANCE:set_width(  
    width  
)
```

Set the value of `grd_width` for this layer instance to the provided value.

Parameters:

`width` An integer value to set to `grd_width` of this layer instance

LAYERINSTANCE:set_height

```
LAYERINSTANCE:set_height(  
    height  
)
```

Set the value of `grd_height` for this layer instance to the provided value.

Parameters:

`height` An integer value to set to `grd_height` of this layer instance

LAYERINSTANCE:set_position

```
LAYERINSTANCE:set_position(  
    x,  
    y  
)
```

Set the value of `grd_x` and `grd_y` for this layer instance to the provided values.

Parameters:

- `x` An integer value to set to `grd_x` of this layer instance
- `y` An integer value to set to `grd_y` of this layer instance

LAYERINSTANCE:set_size

```
LAYERINSTANCE:set_size(  
    width,  
    height  
)
```

Set the value of `grd_width` and `grd_height` for this layer instance's layer to the provided values.

Parameters:

- `width` An integer value to set to `grd_x` of this layer instance's layer
- `height` An integer value to set to `grd_height` of this layer instance's layer

LAYERINSTANCE:set_bounds

```
LAYERINSTANCE:set_bounds(  
    x,  
    y,  
    width,  
    height  
)
```

Set the value of `grd_x`, `grd_y`, `grd_width` and `grd_height` for this layer instance to the provided value.

Parameters:

- `x` An integer value to set to `grd_x` of this layer instance
- `y` An integer value to set to `grd_y` of this layer instance
- `width` An integer value to set to `grd_width` of this layer instance's layer
- `height` An integer value to set to `grd_height` of this layer instance's layer

LAYERINSTANCE:set_alpha

```
LAYERINSTANCE:set_alpha(  
    alpha  
)
```

Set the value of `grd_alpha` for this layer instance to the provided value.

Parameters:

- `alpha` An integer value to set to `grd_alpha` of this layer instance

LAYERINSTANCE:set_hidden

```
LAYERINSTANCE:set_hidden(  
    hidden
```

```
)
```

Set the value of `grd_hidden` for this layer instance to the provided value.

Parameters:

`hidden` An integer value to set to `grd_hidden` of this layer instance

LAYERINSTANCE:hide

```
LAYERINSTANCE:hide()
```

Set the value of `grd_hidden` for this layer instance to 1.

LAYERINSTANCE:show

```
LAYERINSTANCE:show()
```

Set the value of `grd_hidden` for this layer instance to 0.

Lua DOM Samples

```
-- Print a list of all of the user variables associated with a
-- specified control
function print_variables(control_name)
    -- Get the DOM object for the control name passed in
    local domObject = gredom.get_object(control_name)
    if(domObject == nil) then
        print("Can't find name for " .. tostring(control_name))
        return
    end

    -- Get the variables defined on this DOM object
    local vars = domObject:get_variables()
    if(vars == nil or #vars == 0) then
        print("No variables for " .. control_name)
    else
        print("Variables for " .. control_name)
        for i=1,#vars do
            print("# " .. tostring(vars[i]))
        end
    end
end

-- Print out all of the screens where this control's container layer is
-- being used
function print_used_on_screens(control_name)
    -- Get the DOM object for the control name passed in
    local domObject = gredom.get_object(control_name)
    if(domObject == nil) then
        print("Can't find name for " .. tostring(control_name))
        return
    end
end
```

```
-- Build up the full path to this object
-- Walk up the tree looking at all parents adding screens we find
local screen_list = {}
local parent_list = {}
table.insert(parent_list, domObject:get_parents())

local i = 1
while i <= #parent_list do
    local parents = parent_list[i]
    for p=1,#parents do
        -- If this was a screen, add it to our collection
        if parents[p]:get_type() == gredom.SCREEN then
            screen_list[parents[p]] = true
        else
            -- If this has parents of its own,
            -- then add them to the search list
            parents = parents[p]:get_parents()
            if (parents ~= nil and #parents > 0) then
                table.insert(parent_list, parents)
            end
        end
    end
    end
    i = i + 1
end

-- Print out all of the screens that we have identified
print(control_name .. " is used on the following screens:")
for screen,v in pairs(screen_list) do
    print("# " .. screen:get_name())
end

end

-- Invoke our DOM example functions with the context control
function CBDom(mapargs)
    print_used_on_screens(mapargs.context_control)
    print_variables(mapargs.context_control)
end
```

Appendix B. Storyboard IO API

Storyboard IO API

The details the functions available in the Storyboard IO library, `libgreio.a`, are also documented in the Storyboard IO header file `gre/greio.h`.

`gre_io_add_mdata`

```
int gre_io_add_mdata(  
    gre_io_serialized_data_t ** mbuffer,  
    const char * key_name,  
    const char * data_format,  
    const void * data,  
    int data_nbytes  
)
```

Add a data change key/value pair to a serialized buffer. This call can be used to serialize multiple data changes into a single Storyboard IO send operation to improve efficiency.

Once an multi-part data buffer is constructed, it can be sent using the `gre_io_send_mdata` function.

Parameters:

`buffer` The buffer containing the serialized data
`key_name` The data key which is to be set
`data_format` The format for the data to be set
`data` The data value to set
`data_nbytes` The number of bytes used for the data

Returns:

-1 on failure anything else is success

`gre_io_close`

```
void gre_io_close(  
    gre_io_t * handle  
)
```

Close an io connection. Any pending clients will return with an error on their pending actions. This call should be serialized with other clients that are referencing the same `gre_io_t` handle to avoid race conditions. A common technique for closing handles where a receiving thread may be blocked on the handle is to generate a termination event and have the receiving thread perform the close and channel cleanup.

Parameters:

handle A valid handle created with gre_io_open()

gre_io_free_buffer

```
void gre_io_free_buffer(  
    gre_io_serialized_data_t *  buffer  
)
```

This de-allocates the memory associated with a buffer created through the Storyboard IO API.

Parameters:

buffer The buffer whose memory is to be de-allocated

gre_io_grow_buffer

```
void gre_io_grow_buffer(  
    gre_io_t*                handle,  
    gre_io_serialized_data_t *  buffer  
)
```

This function attempts to expand the internal capacity of the Storyboard IO transport to ensure that the payload contained within serialized buffer can be transmitted.

Note

This call is not supported by all platforms and may fail if the transport buffer can not be resized.

Parameters:

handle The handle to the Storyboard IO channel to resize

buffer The buffer whose capacity is to be matched by the transport

Returns:

-1 on failure otherwise success

gre_io_open

```
gre_io_t* gre_io_open(  
    const char *  io_name,  
    int          flag,  
    ...  
)
```

Open a Storyboard IO communication channel using a named connection.

Parameters:

`io_name` The name of the io-channel to use

`flags` The mode you want to open the queue in

Flags define how the connection is opened. Possible flags are:

`GRE_IO_TYPE_RDONLY`: open read only, creating the channel if it doesn't exist

`GRE_IO_TYPE_XRONLY`: open for exclusive read, unlinking an existing channel and creating a new one

`GRE_IO_TYPE_WRONLY`: open write only

`GRE_IO_FLAG_NONBLOCK`: open non-blocking

Returns:

Returns a valid Storyboard IO handle or NULL if no channel can be created.

gre_io_receive

```
int gre_io_receive(  
    gre_io_t *                handle,  
    gre_io_serialized_data_t ** buffer  
)
```

Receive a serialized event from a channel. By default this call blocks until an event is received or until the channel is destroyed unless the `GRE_IO_FLAG_NONBLOCK` flag was passed to the `gre_io_open()` call.

In order to receive events, the handle must have been opened for reading using one of `GRE_IO_RDONLY` or `GRE_IO_XRONLY`.

Parameters:

`handle` A valid handle created with `gre_io_open()`

`buffer` A pointer to a serialized buffer pointer. If the buffer is NULL then a new buffer is allocated otherwise the buffer provided is used to store the received event.

Returns:

The size of the message received in bytes or -1 on failure.

When a data buffer is successfully received, the event specific content can be extracted by making a call to `gre_io_unserialize`. The values returned by the `unserialize` call will be pointers directly into the memory allocated to the data buffer. Clients can read and write to the extracted values (such as event name, event format and data payload) directly as long as the serialized buffer is not re-used at the same time. Any data from the event that needs to be maintained across calls to `gre_io_receive` must be copied by the user before the serialized buffer is re-used.

gre_io_send

```
int gre_io_send(  
    gre_io_t *                handle,  
    gre_io_serialized_data_t * buffer  
)
```

Send a serialized event buffer to a channel. In order to send events, the handle must have been opened for writing using `GRE_IO_WRONLY`.

Parameters:

handle A valid handle created with `gre_io_open()`
buffer A data buffer containing a serialized event

Returns:

-1 on failure otherwise success.

gre_io_send_mdata

```
int gre_io_send_mdata(  
    gre_io_t *          handle,  
    gre_io_serialized_data_t * md_buffer  
)
```

Send a serialized buffer of mdata (data manager key/value pairs) to the handle. The handle must have been opened for writing using `GRE_IO_WRONLY`.

Parameters:

handle A valid handle created with `gre_io_open()`
buffer A data buffer containing a serialized data

Returns:

-1 on failure anything else is success

gre_io_serialize

```
gre_io_serialized_data_t* gre_io_serialize(  
    gre_io_serialized_data_t * buffer,  
    const char *              event_target,  
    const char *              event_name,  
    const char *              event_format,  
    const void *              event_data,  
    int event_nbytes  
)
```

Serialize individual event items (see `gre/io_mgr.h`) into a single buffer for transmission using Storyboard IO.

Parameters:

buffer The buffer that will contain the serialized data or NULL if a new buffer should be allocated

event_target The name of the event target model element (screen,layer, control, render extension), or NULL to send to the default target (application level)
event_name The name of the event to send, or NULL to send an empty event
event_format The format description of the data (ex. "1s0 string") or NULL if no data is being sent
event_data A pointer do the data to transmit, or NULL if no data is transmitted
event_nbytes The number of data bytes to transmit, or NULL if no data is transmitted

Returns:

A buffer with the serialized data or NULL on error. It may be necessary for the internal buffer to be re-sized or re-allocated if the new data payload is larger than the previous one is being serialized.

Example:

```
nbuffer = gre_io_serialize(nbuffer, NULL, "cluster_update",
"2u1 speed 2u1 rpm 2u1 fuel 2u1 battery 2u1 oil 2u1 odometer 2u1 trip",
&event_data, sizeof(event_data));
if(!nbuffer) {
    fprintf(stderr, "Can't serialized data to buffer, exiting\n");
    break;
}
```

gre_io_size_buffer

```
gre_io_serialized_data_t* gre_io_size_buffer(
    gre_io_serialized_data_t *    buffer,
    int                            nbytes
)
```

This function ensures that the specified buffer has enough internal storage capacity for a payload of nbytes size. If the buffer is NULL or the existing capacity is not large enough then a new memory buffer will be assigned to the buffer object.

Data parameters must be sent in order of descending alignment requirements. Example: 4u1 4u1 2u1 1s0 is good, 2u1 4u1 4u1 1s0 is not

Parameters:

buffer The buffer to be sized, or NULL to allocate a new buffer
nbytes The number of bytes this buffer should be able to support

Returns:

layer_name

A buffer with room for a message nbytes in size or NULL if the space could not be allocated

gre_io_unserialize

```
int gre_io_unserialize(
    gre_io_serialized_data_t *    buffer,
```

```
char ** event_target,  
char ** event_name,  
char ** event_format,  
void ** event_data  
)
```

Transform a serialized buffer into individual event items (see `gre/io_mgr.h`). The pointers returned point back into the content of the serialized buffer so the buffer can't be de-allocated until clients are finished referencing the event items returned from this call.

Data parameters must be sent in order of descending alignment requirements. Example: 4u1 4u1 2u1 1s0 is good, 2u1 4u1 4u1 1s0 is not

Parameters:

buffer	The buffer containing the serialized data
event_target	Location to store the event target model element
event_name	Location to store the event name
event_format	Location to store the event format
event_data	Location to store the event data

Returns:

The number of bytes in the event_data structure

gre_io_zero_buffer

```
void gre_io_zero_buffer(  
    gre_io_serialized_data_t * buffer  
)
```

This clears the internal byte count of the buffer, but does not de-allocate the buffer's memory.

Use this function to reset a buffer in between multiple calls to `gre_io_serialize`

Parameters:

buffer	The buffer to have its byte count cleared
--------	---

gre_io_get_error_codes

```
void gre_io_get_error_codes(  
    gre_io_t *handle, gre_io_error_t *errorcodes  
)
```

This gets the error codes if greio experiences an error.

Parameters:

handle	A valid handle created with <code>gre_io_open()</code>
--------	--

errorcodes A valid `gre_io_error_t` structure to be filled out

gre_io_get_error_message

```
const char * gre_io_get_error_message(  
    gre_io_error_t *errorcodes  
)
```

This gets the error message for an error code returned by a call to `gre_io_get_error_codes`.

Parameters:

errorcodes A valid `gre_io_error_t` structure containing the errors returned from a call to `gre_io_get_error_codes`

Returns:

The error message.

Appendix C. Storyboard Engine and Plugin Options

Storyboard Engine Plugin Options

The Storyboard product ships with a standard set of plugins which add functionality to the system. Plugins are loaded based on the SB_PLUGINS environment variable. This variable can be a directory where all plugins are loaded from or a “;” separated list of plugins. Some plugins have options that can be passed via the command line to the plugin. To pass an option to a plugin use the -o option to sbengine in this format:

sbengine [[-oplugin_name,plugin_option[[,plugin_option2,...]]] **your_app.gapp**

Table C.1. Plugin Options

PLUGIN OPTION	DESCRIPTION
Plugin Name: io_mgr (-oio_mgr,...)	
no_compression =[size]	Disable event compression in the event queue. By default Storyboard will compress gre.motion and gre.mtevent events in order to reduce input lag and flooding of the event queue with motion events. Disable compression to receive a higher fidelity stream of motion events from Storyboard.
queue_size =[size]	4096 will limit the event queue size to a maximum of 4K for example. If the queue exceeds this size, events will be dropped and diagnostic messages will be logged regarding the dropped events. The default behavior is to have an unlimited event queue size.
Plugin Name: model_mgr (-omodel_mgr,...)	
block_sbio_sends [=0 1][integer]	By default (value of 0) Storyboard IO channels opened for writing will be opened with a non-blocking flag and if a queue gets full a warning will be generated and, if Lua is being used, an error will be returned. Setting this parameter to 1 will enable blocking behaviour when a queue is full which may have adverse effects on the usability and interactiveness of the UI but aligns with historical defaults of earlier versions of Storyboard.
fps [integer]	Limits the frame rate for all animations to a maximum fps specified (25 for example).
mem_stats =1	On platforms where process/task memory usage or heap allocator memory usage values are available, report them as performance log metrics. The value should be set to 1 to enable the reporting of the memory statistics at app initialization and at application termination. Future values are reserved.

PLUGIN OPTION	DESCRIPTION
	If the metrics plugin is present and this option is present and loaded then this option will also generate a cost estimate of the current application's in-memory structure cost which can be used within Storyboard Designer to estimate the RAM footprint of the application at design time.
plugin_path =[path]	Sets the plugin path to the specified directory (/temp for example). This setting overrides the SB_PLUGINS environment variable setting.
scroll_drag_px	The number of pixels that need to be moved before a scrollable container considers this a scroll operation (default 10).
scroll_bounce_ms	The number of milliseconds that the bounce operation on a scrollable container should animate once the bounce back is operation is started (default 200ms).
scroll_bounce_rate	The animation rate that should be used on a scrollable container for bounce back animations (default easein).
Plugin Name: font_mgr (-ofont_mgr,...)	
linegap =[0 1]	This option controls if the Storyboard Engine includes the line gap information (linegap=1 default) when positioning text for rendering or if it only includes the ascender information (linegap=0). The default value is to include the linegap information.
Plugin Name: render_mgr (-orender_mgr,...)	
x =[xpos]	This will position the application at the defined x-position.
y =[ypos]	This will position the application at the defined y-position.
sysdpi	This will enable the window upscaling performed by Windows according to the system DPI setting. PLATFORM: Windows, OpenGL, swrender, x86
dblbuffer	Enable double buffering (fullscreen redraws) PLATFORM: Linux, x86, armle, swrender
display =[index]	Connect to the given display index, this option is only available for the QNX Screen OpenGL and software render managers or the Linux i.MX6 OpenGL render manager where the value is the selected framebuffer index.
fb =[x]	This option pertains specifically to iMX6 hardware platforms. Starting at 0, x defines the framebuffer number to render to. PLATFORM: Linux, swrender

PLUGIN OPTION	DESCRIPTION
fullscreen	Run in fullscreen mode.
rotate =[90 180 270]	Rotate the application by the defined angle. PLATFORM: Windows CE 2013, macOS, Linux, QNX, x86, armle, OpenGL, swrender
rle_blit =[x]	Decode RLE compressed images directly to framebuffer surface memory in order to remove need for temporary decode buffer. Only available for 32 bits per pixel source images with No Scaling and No Rotation. This option pertains to swrender on RTExec platforms. Set x = 1 (default) to enable RLE blit; x = 0 is not enabled. PLATFORM: armle, RTExec, Storyboard Lite, swrender
quality =[0 1 2]	Visual fidelity of image rotation rendering. A value of 0 emphasizes speed at the cost of visually fidelity. A value of 2 provides the highest visual fidelity, with a potential impact on performance. The default value is set to 1.
dumpconfig	Dump the layer and graphics configuration information at startup. PLATFORM: Linux, x86, armle, swrender
layer =[index]	Set the directfb layer index that content will render to (default=0) PLATFORM: Linux, x86, armle, swrender
multisample =[value]	Indicates the degree of multisampling which affects the visual smoothness of edges. For example, a value of 4 would be 4x multisampling while 0 would be no multisampling. By default, this value is set dynamically based on the model content that is being interpreted in order to provide the best balance between high performance and high quality rendering. If your model contains 3D models, polygons or circles then multi-sampling is turned on to a level of 4, otherwise it is off. The command line option setting will override the default probing and ensure a fixed value. PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL
vbo	Enables the use of vertex buffer objects. PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL
scale =[aspect]	Scale the application to the physical display size. If aspect is passed the application will retain the proper aspect ratio when scaled.

PLUGIN OPTION	DESCRIPTION
	PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL
backbuffer	<p>Render the scene using a damage rectangle. On some OpenGL ES implementations this will give better performance but will use more memory as it has to allocate a separate display buffer.</p> <p>PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL</p>
npot	<p>Disable power-of-two texture allocations. By default the OpenGL ES API is queried to check for NPOT texture support. This option can be used to override this behavior and force support. NPOT textures will use less memory for image data.</p> <p>PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL</p>
fontsize=[size]	<p>Specify the size of the font texture sheet. Fonts are generated into sheets and the default size is 512x512. The number of glyphs put into the sheet is a function of the point size and the texture size. This option can be used to tune the number of available glyphs and the memory usage.</p> <p>PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL</p>
window_w=[w]	<p>Scale the application content and window to the specified width. This option is only valid on desktop systems which use a window manager. This option must be used along with 'window_h'</p> <p>PLATFORM: Windows, Windows CE 2013, macOS, Linux, QNX, x86, armle, OpenGL</p>
window_h=[h]	<p>Scale the application content and window to the specified height. This option is only valid on desktop systems which use a window manager. This option must be used along with 'window_w'</p> <p>PLATFORM: Windows, Windows CE 2013, macOS, Linux, QNX, x86, armle, OpenGL</p>
linejoin=[0 1]	<p>Set line join style for path drawing, drawing joins can have a performance impact on frame rate. 0=none, 1=round (default=1)</p> <p>PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL</p>
clipmode=[stencil scissor]	<p>Set the clipping mode to use, may have performance impacts. Each implementation defaults to the best performance.</p>

PLUGIN OPTION	DESCRIPTION
	PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL
error_event	An error event is generated for OpenGL render errors. Image and font errors will identify the image and font related to the error.
zorder=[z]	When using the QNX Screen engine this will position the application window at the defined Z index.
clearcolor=[color]	Specify the clear color to be used for the Window Surface. The color format is ARGB and specified in hexadecimal. The default clear color is 0x00000000. This option can be used for debugging purposes. PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL
image_block_size=[number]	This specifies the memory size of the shared blocks that are used within the image cache. These settings override the generic block_size setting that is used generically for the atlas and are bound by the -oresource_mgr,image setting. PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL
font_block_size=[number]	This specifies the memory size of the shared blocks that are used within the font glyph cache. These settings override the generic block_size setting that is used generically for the atlas and are bound by the -oresource_mgr,font setting. PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL
block_size=[-1 0 >0]	Specify how block size should be determined for growing the image atlas. For -1, a degenerate block size will be used, forcing block size to that of a single image. For 0, the block size is rounded up to the nearest power of two for an image's dimensions. For any value greater than zero, block allocations will be rounded up to the size specified. (default=1024) PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL
surface_cache=[-1(default) 0 >0]	Specify the surface cache control. For -1, the surface cache will grow unbounded (default). For 0, the surface cache will be disabled. For any value greater than zero, it will be used as the upper limit on the number of surfaces in the surface cache. PLATFORM: Windows CE 2013, macOS, Linux, QNX, armle, OpenGL

PLUGIN OPTION	DESCRIPTION
mainlayer =[number]	The main layer to use for rendering, defaults to layer 0 PLATFORM: Linux, QNX, armle
dumpcaps	Print the device capabilities and acceleration flags. PLATFORM: Windows CE 2013, armle
nohwcursor	Disables the HW cursor. PLATFORM: Windows CE 2013, armle
Plugin Name: resource_mgr (-oresource_mgr,...)	
error =[0 1]	When a resource error is encountered a diagnostic message is generated. This option controls if an event for the error is sent (1, default) or not sent (0). This option is currently only enabled for OpenGL render managers.
font =[number]	Similar to the <code>image</code> resource option, this value controls the number of bytes of memory allocated for the font glyph cache. After the cache is full, the behaviour of the allocations is controlled via the <code>font_oom</code> option. A value of 0 (default setting) indicates that the font glyph cache should be limited by the size of available system memory. Otherwise a positive value indicates the size of the cache in bytes and can be symbolically specified using K, M qualifiers such as 4K to indicate an font glyph cache of 4096 bytes or 1M to indicate 1048576 bytes of memory.
image =[number]	This will set the maximum number of bytes in the image cache. After the cache is full, the behaviour for allocations is controlled via the <code>image_oom</code> option. A value of 0 (default setting) indicates that the image cache should be limited by the size of available system memory. A value of -1 indicates no image cache and only one image at a time will be loaded and that image not cached. Otherwise a positive value indicates the size of the cache in bytes and can be symbolically specified using K, M qualifiers such as 4K to indicate an image cache of 4096 bytes or 1M to indicate 1048576 bytes of memory.
image_oom =[fail lru]	This sets the behaviour when the maximum value of the image cache is reached. If this is set to the value <code>fail</code> then when an allocation can't be fulfilled the request will fail. If this is set to the value <code>lru</code> then when an allocation can't be fulfilled the request will attempt to remove other entries from the cache until it can either fit the request or it fails. The default behaviour is <code>lru</code> .

PLUGIN OPTION	DESCRIPTION
Plugin Name: screen_mgr (-oscreen_mgr ,...)	
dl =[1]	Disable the use of a particular hardware layer.
fps	Display the frames per second of the display updates in the Storyboard Engine console output and also set the <code>grd_fps</code> variable with the calculated value.
overlay	Allow the application to be run as an overlay and show the content below. This only functions on particular render managers and if the application has a transparent background.
redraw_complete	Generate a completed event for every screen update.
swcursor	Enables the rendering of a software cursor.
Plugin Name: capture_playback (-ocapture_playback ,...) / Shared Library: libgre-plugin-capture-playback	
capture_events =[eventname]	Capture option to define a custom list of events to capture. Multiple events can be specified by using the ":" character as a delimiter.
file =[filename]	<p>If "capture" is specified as the mode it indicates the contents of the file specified will be overwritten with the new event stream.</p> <p>If "playback" is specified then the contents of the file specified will be used as an event stream source.</p>
loop =[count]	A playback option that indicates how many times the playback should iterate through its content. The default is to playback the content once, otherwise if a <code>count</code> is specified the plugin will replay the full content count times.
mode =[capture playback]	<p>Specifies the behavior of the plugin for either playback or capture. If capture is specified then input events (<code>gre.press</code>, <code>gre.release</code>, <code>gre.motion</code>, <code>gre.key-press</code> etc) will be logged and stored in an output file.</p> <p>If playback is specified then the contents of an input file are read and the input events are injected back into the application.</p>
no_defaults	<p>Capture option to specify that there are no default events set to be captured. By default we capture the following events:</p> <ol style="list-style-type: none"> 1 <code>gre.press</code> 2 <code>gre.release</code> 3 <code>gre.keyup</code>

PLUGIN OPTION	DESCRIPTION
	<p>4 gre.keydown</p> <p>5 gre.keyup</p> <p>6 gre.keyrepeat</p> <p>7 gre.motion</p> <p>8 gre.mtevent</p>
quit_playback	Playback option that indicates if a <code>gre.quit</code> event should be automatically generated after playback is complete. The default is to not generate a quit event.
verbosity	Indicates that the plugin should log events that it is either capturing or playing back to the standard output.
Plugin Name: ccallback (-ocallback,...)	
path	Point the ccallback plugin to a dynamic library (dll/so). Only applies to dynamically linked engines. On statically linked engines, this is not required.
Plugin Name: dev-input (-odev-input,...) / Shared Library: libgre-plugin-dev-input	
kbd =[device]	<p>The name of the keyboard device, for example /dev/input/event1</p> <p>One of either the mouse or kbd arguments must be passed to enable this plugin. There are no default bindings so the full path to the desired input device must be specified.</p>
mouse =[device]	<p>The name of the mouse device, for example /dev/input/event0</p> <p>One of either the mouse or kbd arguments must be passed to enable this plugin. There are no default bindings so the full path to the desired input device must be specified.</p>
Plugin Name: gesture (-ogesture,...) / Shared Library: libgre-plugin-gesture	
file =[filename]	filename is a text file containing custom gesture definitions. When the gesture plugin is in 'auto' mode, the gesture plugin will determine if the application uses any of the custom gesture events and enable single touch events.
mode =[disabled single multi]	<p>By default, sbengine will search your application for use of any gesture events and gestures will be enabled as required. This behaviour can be overridden with any of the following mode options:</p> <p>disabled will not generate any gesture events.</p>

PLUGIN OPTION	DESCRIPTION
	<p><code>single</code> allows for generation of single touch gesture events.</p> <p><code>multi</code> allows for generation of both single and multi-touch gesture events.</p>
threshold =[level]	Specifying a level allows the user to configure the sensitivity level which determines if we translate a motion into a gesture event. By default the threshold is set to 100.
Plugin Name: <code>gfi-input (-ogfi-input,...)</code> / Shared Library: <code>libgre-plugin-gfi-input</code>	
mouse =[device]	<p>The name of the mouse device, for example <code>/dev/pci/mouse0</code></p> <p>PLATFORM: QNX</p>
rotate =[90 270]	<p>If specified, this indicates that the input co-ordinates should be rotated by 90 or 270 degrees.</p> <p>PLATFORM: QNX</p>
	By default the input system used the <code>gfi</code> interface based on the <code>devi</code> drivers. The <code>devi</code> driver must be run with the <code>-P</code> option. If you pass the <code>mouse</code> option then the mouse/touchscreen is used in raw mode.
Plugin Name: <code>greio (-ogreio,...)</code> / Shared Library: <code>libgre-plugin-greio</code>	
channel =[name]	<p>The value specifies the name that the applications Storyboard IO channel will use. This name can then be used by <code>gre_io_open</code> or <code>iogen</code> clients to send events to the application.</p> <p>In the case of a TCP channel, [name] needs to have a URI of "tcp://" followed by optional <ip address> and optional <channel> or <port number></p>
queue	This flag indicates if the events on the Storyboard IO channel should be asynchronously queued into the application's message queue or if a new event should be added only after the last event has been processed. The default is to only have one event being processed by the application at a time.
Plugin Name: <code>logger (-ologger,...)</code> / Shared Library: <code>libgre-plugin-logger</code>	
buffer =[bytes]	This option will buffer all logging output to an allocated in-memory buffer and only flush the output when the buffer content is full. The number of bytes allocated for the buffer are provided by the option to the argument. If the buffer size is 0 or invalid, then 4K will be allocated for the buffer.
data	This option enables the logging of data change events as they occur. The data key that has been changed is displayed to standard output.

PLUGIN OPTION	DESCRIPTION
event =[event_name]	This option will enable the generation of a custom Storyboard event any time that an ERROR message is detected by the Storyboard Engine logging system. When the error is detected, an event "event_name" will be sent to the application and it will contain a payload of "1s0 msg" where the "msg" is the diagnostic string that would have been logged.
filter =[keyword]	This option enables filtering of events based on the keyword provided. Multiple keywords can be specified. Filtered events are delimited by a ':'. sbengine -ologger,io,filter=gre.press:gre.release Each keyword can also be negated by the '^' symbol. Therefore we could ignore all motion events by passing in the following command: -ologger,io,filter=^gre.motion Filtering applies to the 'io' and 'data' options. If neither 'io' or 'data' options are specified, then this option does nothing.
io =[level]	This option enables the logging of IO events in variable levels of verbosity. If no option is specified then the level will default to 1 and the event name and its size in bytes will be displayed. If 2 is specified then the format is displayed. If 3 is specified then the data payload will be dumped to the standard output in both hexadecimal and character formats.
metrics_event_prefix =[prefix]	This option specifies the prefix to use in the metrics event names. This defaults to metrics.
metrics_output =[0 1 2 3]	This option tells the metrics portion of the logger plugin where to out the metrics data. 0 just updates the data, and this is the default, 1 is used to output the data to the log, 2 outputs the data to the performance log, if the perf option is also specified, and 3 will output the data to both the log and the performance log if the perf option is specified.
output =[filename]	<p>This option specifies a path in the file system to direct the Storyboard standard output to. The directory path to the file must already exist.</p> <p>+ At the beginning of the file name will append to the log file, otherwise the file will be overridden on each invocation of sbengine.</p> <p>%D in the filename will be replaced by a date stamp with YYYY-MM-DD format.</p> <p>%T in the filename will be replaced by a 24h time stamp with HHMMSS format.</p> <p>Both %D and %T may be used on the same filename.</p>

PLUGIN OPTION	DESCRIPTION
	A valid command would be: -ologger,output=/logs/log-%D-%T.txt , provided the /logs/ directory existed prior to runtime.
perf	This option enables the logging of performance data to the standard output (or file if <code>perf_file</code> is used). If a value of 0 is specified to the <code>perf</code> option then performance logging is enabled, but the capture of data is not immediately started and can be toggled using the <code>gra.perf_state</code> action. If the value is set to 1 or is not specified, then performance data will be immediately captured. For example -ologger,perf=1 will enable performance logging with the immediate capture of performance metrics.
perf_buffer =[bytes]	This option will buffer all performance logging output to an allocated in-memory buffer and only flush the output when the buffer content is full. The number of bytes allocated for the buffer are provided by the option to the argument. If the buffer size is 0 or invalid, then 4K will be allocated for the buffer.
perf_event_prefix =[prefix]	This option specifies the prefix to use in the perf event name. This defaults to <code>perf</code> .
perf_file =[filename]	<p>This option specifies a path in the file system to direct the performance data output to. The directory path to the file must already exist and the contents of the file will be overwritten on each invocation of <code>sbengine</code>.</p> <p>+ At the beginning of the file name will append to the log file, otherwise the file will be overridden on each invocation of <code>sbengine</code>.</p> <p>%D in the filename will be replaced by a date stamp with YYYY-MM-DD format.</p> <p>%T in the filename will be replaced by a 24h time stamp with HHMMSS format.</p> <p>Both %D and %T may be used on the same filename.</p> <p>A valid command would be: -ologger,perf_file=/logs/perf-%D-%T.plog, provided the /logs/ directory existed prior to runtime.</p>
perf_filter =[keyword]	This option enables filtering of perf types based on the keyword provided. The <code>sb_Event</code> type cannot be filtered out. This is because Storyboard engine is event based, and everything that happens is tied back to the event that kicked things off. Multiple keywords can be specified. Filtered types are de-

PLUGIN OPTION	DESCRIPTION
	limited by a ':'. -ologger,perf,perf_filter=sb_RenderExtension Each keyword can also be negated by the '^' symbol. Therefore we could ignore all render extension types by passing in the following command: -ologger,perf,perf_filter=^sb_RenderExtension Filtering applies to the perf option.
poll_metrics_interval =[ms]	This option sets up a timer to fire at the interval specified by ms. When the timer fires the system metrics for memory, CPU and FPS will be updated. See metrics_output to control where the metrics output will go.
retailmsg	The logger will output using RETAILMSG instead of sending output to stdio. PLATFORM: Windows CE 2013
	If performance logging is enabled then the output is a set of comma separated values (CSV) with the following fields: PERF, application time, type, operation, name, duration
application time	This is the time that the performance event was finished relative to the start time of the application in milliseconds.
type	This is the type of performance operation that was recorded as a broad classifier.
operation	This is a sub-classification of the type used for additional tracing granularity.
name	This is an identifier that can be used, along with the type and operation fields, to identify the context of the performance operation being performed.
duration	This is the duration of the operation in milliseconds.
slogger =[opcode]	This option specifies that sbengine should use the QNX system logging infrastructure. if no opcode is given, sbengine will provide an appropriate opcode. The opcode is a combination of a major and minor code. Create the opcode using the <code>_SLOG_SET_CODE(major, minor)</code> macro that's defined in <code>sys/slog.h</code> . This option is only available for systems running QNX.
sbio =[channel]	This option specifies that sbengine should use the SBIO channel specified as the log content destination output. The log data will be logged with an event <code>gre.log</code> and a simple format string <code>1s0 msg</code> .

PLUGIN OPTION	DESCRIPTION
	If the channel is not available at start up, an error will be generated and the logging will revert to the default logging output. If a channel becomes available then the logging will switch back to that channel. If a channel becomes full because no receiver is pulling the log messages, then this will block the execution of the Storyboard Engine until content can be read.
Plugin Name: lua (-olua,...) / Shared Library: libgre-plugin-lua	
gc =[0 1]	This option is used to minimize the runtime memory footprint of the Lua script engine by invoking the Lua garbage collector after every Lua action. By default this option is set to 0 indicating that garbage collection will occur at the natural points specified by Lua's <code>collectgarbage</code> option. If the value is set to 1, then garbage collection is run after every Storyboard Lua action invocation, reducing the active runtime memory footprint with a slight cost to execution performance.
hold =[0 1 2]	This option controls how Lua posts data manager change notifications. By default all changes are held until the end of script execution (1), except whenever a local event is sent through the <code>gre.set_event</code> call. In this case the data manager will be released to process the notifications, the <code>send_event</code> call will be issued, and then the data manager will again be held from making data change notifications. If 0 is specified, change notifications and events are triggered as soon as changes are made using <code>gre.set_data</code> calls. If 2 is specified then the data manager is held from emitting change notifications until the end of script execution even if a <code>gre.send_event</code> call is made.
Plugin Name: model3d (-omodel3d,...) / Shared Library: libgre-plugin-model3d	
novbo	Disable the use of vertex buffer objects, by default Vertex buffer objects are used for rendering.
Plugin Name: mtdev (-omtdev,...) / Shared Library: libgre-plugin-mtdev	
bounds =[X1:Y1:X2:Y2]	Where X1,Y1 is the top-left corner and X2,Y2 is the bottom-right corner of the touch screen device. On some screens, X1 may be less than X2 and Y1 may be less than Y2.
calibrate	Puts mtdev into raw mode to be used in conjunction with the mtcilib application (See Crank Public SVN) to acquire the bounds parameters for screen calibration. Use the -omtdev,bounds option to pass the parameters to sbengine.
device =[path to touch device]	Plugin for Linux Multi-Touch Protocol to be used with kernels supporting multi-touch events.

PLUGIN OPTION	DESCRIPTION
points =[integer]	This is the number of multitouch fingers that is supported. Events will only be generated for this number of fingers in contact with the screen, the default is 5.
rotate =[0 90 180 270]	Clockwise rotation of the touch input coordinates. This option is not often required. By default, mt-dev plugin input will be rotated depending on the render manager configuration. The input transformation will be calculated by combining the specified rotation value and the render manager rotation.
threshold =[integer]	This is the number of pixels a touch point has to move in order to generate a motion event, the default value is 1.
Plugin Name: tslib (-otslib,...) / Shared Library: libgre-plugin-tslib	
calibrate	Put tslib into raw mode which is used for calibration. If you do not have the following <code>tslib</code> variables setup the plugin will not load or function properly. <code>TSLIB_CONSOLEDEVICE</code> <code>TSLIB_TSDEVICE</code> <code>TSLIB_CALIBFILE</code> <code>TSLIB_CONFFILE</code>
motion =[integer]	The number of consecutive motion events to compress. Can be useful on a device which delivers a high rate of motion events. Default is to not compress.
pressure =[integer]	Set the pressure value which corresponds to a press. The default is any value greater than 0 is a press.

Appendix D. Standard Event Definitions

Standard Event Definitions

Storyboard supports a list of standard events. These events are all prefixed with `gre.` and can be used by your application.

System Events

`gre.init`

The system has been initialized and is ready. This is the first event set in the system.

Data:

No data payload

`gre.quit`

The system is being shutdown.

Data:

No data payload

`gre.redraw`

An area of the screen has been damaged (visible data has changed). A redraw event may not cause actual screen drawing if the control which has changed is hidden or offscreen.

Data:

```
int32_t    x
int32_t    y
int32_t    width
int32_t    height
```

If the values are all 0 then the entire screen has been damaged

`gre.rendermgr.error`

This event will be generated when an error has occurred with a OpenGL ES 2.0 runtime using the "error_event" option.

Format:

4s1 code 1s0 msg

code :The code is the error code that is returned from the GL framework on the API call `glGetError()`

msg :The msg is a human readable diagnostic message about the context of the error and any associated resources involved. For example: problems loading font or image resource textures will identify the image and font related to the error, other API calls will be identified by context of execution (ie GL function name, shader compilation).

Pointer Events

The following events are generated in response to a device such as a mouse or a touchscreen. These events are targeted at specific controls based upon the controls location and sensitivity.

gre.press

A mouse/touchscreen has been pressed.

Data

uint32_t	button
uint32_t	timestamp
int16_t	subtype
int16_t	x
int16_t	y
int16_t	z
int16_t	id
int16_t	spare

Where:

button	GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left GR_EVENT_BTN_MIDDLE - 0x0002 GR_EVENT_BTN_RIGHT - 0x0004
timestamp	This is an event timestamp in milliseconds since application start
subtype	GR_EVENT_RELEASE_IN GR_EVENT_RELEASE_OUT
z	This parameter is dependent on the availability of z- co-ordinate information
id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

gre.motion

A touch contact has moved.

Data

```
uint32_t    button
uint32_t    timestamp
int16_t     subtype
int16_t     x
int16_t     y
int16_t     z
int16_t     id
int16_t     spare
```

Where:

```
button      GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left
            GR_EVENT_BTN_MIDDLE - 0x0002
            GR_EVENT_BTN_RIGHT - 0x0004

timestamp    This is an event timestamp in milliseconds since application start

subtype      GR_EVENT_RELEASE_IN
            GR_EVENT_RELEASE_OUT

z            This parameter is dependent on the availability of z- co-ordinate information

id           This parameter is used to track multi-touch presses as they come in

spare        This is padding and should be 0
```

gre.release

A mouse/touchscreen has been released.

Data

```
uint32_t    button
uint32_t    timestamp
int16_t     subtype
int16_t     x
int16_t     y
int16_t     z
int16_t     id
int16_t     spare
```

Where:

```
button      GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left
            GR_EVENT_BTN_MIDDLE - 0x0002
            GR_EVENT_BTN_RIGHT - 0x0004

timestamp    This is an event timestamp in milliseconds since application start

subtype      GR_EVENT_RELEASE_IN
            GR_EVENT_RELEASE_OUT

z            This parameter is dependent on the availability of z- co-ordinate information
```

id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

gre.touch

If a mouse/touchscreen presses and then releases on the same control then a touch event will be generated. This is useful for activating button style elements. If the release is found to intersect a different control then a touch event is not generated.

Note

This event is synthetically generated by the framework based on incoming `gre.press` and `gre.release` events. Event redirectors should generally not include this event in their list of redirection events.

Data

uint32_t	button
uint32_t	timestamp
int16_t	subtype
int16_t	x
int16_t	y
int16_t	z
int16_t	id
int16_t	spare

Where:

button	GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left GR_EVENT_BTN_MIDDLE - 0x0002 GR_EVENT_BTN_RIGHT - 0x0004
timestamp	This is an event timestamp in milliseconds since application start
subtype	GR_EVENT_RELEASE_IN GR_EVENT_RELEASE_OUT
z	This parameter is dependent on the availability of z- co-ordinate information
id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

gre.mtevent

A touchscreen contact has changed and there is information for more than one touch contact. A `gre.mtevent` will only be delivered to the application and screen. Storyboard Engine will divide the event into discrete `gre.mtpress`, `gre.mtmotion` and `gre.mtrelease` events which are then delivered to the appropriate model elements.

Data

```
uint32_t      timestamp
int16_t       npoints
int16_t[npoints] x
int16_t[npoints] y
int16_t[npoints] z
int16_t[npoints] id
int16_t[npoints] state
int16_t[npoints] spare
```

Where:

timestamp	This is an event timestamp in milliseconds since application start
npoints	This is the number of points included in this event and may not correlate to the number of fingers currently in contact with the touchscreen.
x	This is an array of x values for npoints touch contacts
y	This is an array of y values for npoints touch contacts
z	This is an array of z values for npoints touch contacts. This parameter is dependent on the availability of z-co-ordinate information
id	This is an array of id values for npoints touch contacts. This parameter is used to track multi-touch presses as they come in
state	This is an array of state values for npoints touch contacts. GR_PTR_STATE_PRESS - 0x01: as a finger makes contact with the touchscreen. GR_PTR_STATE_MOTION - 0x02: as a finger moves on the touchscreen. GR_PTR_STATE_RELEASE - 0x03: as a finger is lifted from the touchscreen.
spare	This is padding and should be 0

gre.mtpress

A touchscreen has been pressed. This event is emitted where are two or more contact points.

Data

```
uint32_t      button
uint32_t      timestamp
int16_t       subtype
int16_t       x
int16_t       y
int16_t       z
int16_t       id
int16_t       spare
```

Where:

button	GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left
timestamp	This is an event timestamp in milliseconds since application start

subtype	GR_EVENT_RELEASE_IN GR_EVENT_RELEASE_OUT
z	This parameter is dependent on the availability of z- co-ordinate information
id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

gre.mtmotion

A touch contact has moved. This event is emitted when there are two or more contact points.

Data

uint32_t	button
uint32_t	timestamp
int16_t	subtype
int16_t	x
int16_t	y
int16_t	z
int16_t	id
int16_t	spare

Where:

button	GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left
timestamp	This is an event timestamp in milliseconds since application start
subtype	GR_EVENT_RELEASE_IN GR_EVENT_RELEASE_OUT
z	This parameter is dependent on the availability of z- co-ordinate information
id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

gre.mtrelease

A touchscreen has been released. This event is emitted when there are two or more contact points.

Data

uint32_t	button
uint32_t	timestamp
int16_t	subtype
int16_t	x
int16_t	y
int16_t	z
int16_t	id
int16_t	spare

Where:

button	GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left
timestamp	This is an event timestamp in milliseconds since application start
subtype	GR_EVENT_RELEASE_IN GR_EVENT_RELEASE_OUT
z	This parameter is dependent on the availability of z- co-ordinate information
id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

gre.inbound

A mouse/touch has entered a control (if dragging a pointer or finger) which has no other mouse/touch points in the control boundary. This event is generated once the coordinates enter a control boundary. If mouse motion events are not supported on the platform then this event will not be generated.

Note

Control groups can not receive inbound events.

Data

uint32_t	button
uint32_t	timestamp
int16_t	subtype
int16_t	x
int16_t	y
int16_t	z
int16_t	id
int16_t	spare

Where:

button	GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left GR_EVENT_BTN_MIDDLE - 0x0002 GR_EVENT_BTN_RIGHT - 0x0004
timestamp	This is an event timestamp in milliseconds since application start
subtype	GR_EVENT_RELEASE_IN GR_EVENT_RELEASE_OUT
z	This parameter is dependent on the availability of z- co-ordinate information
id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

gre.outbound

A mouse/touch has left a control (if dragging a pointer or finger) and there are no remaining fingers in the control boundary. This event is generated once the coordinates leave a control boundary. If mouse motion events are not supported on the platform then this event will not be generated.

Note

Control groups can not receive outbound events.

Data

uint32_t	button
uint32_t	timestamp
int16_t	subtype
int16_t	x
int16_t	y
int16_t	z
int16_t	id
int16_t	spare

Where:

button	GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left GR_EVENT_BTN_MIDDLE - 0x0002 GR_EVENT_BTN_RIGHT - 0x0004
timestamp	This is an event timestamp in milliseconds since application start
subtype	GR_EVENT_RELEASE_IN GR_EVENT_RELEASE_OUT
z	This parameter is dependent on the availability of z- co-ordinate information
id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

gre.mtinbound

A mouse/touch has entered a control (if dragging a pointer or finger) while other mouse/touch points are currently in the control boundary. This event is generated once the coordinates enter a control boundary. If mouse motion events are not supported on the platform then this event will not be generated.

Note

Control groups can not receive outbound events.

Data

uint32_t	button
uint32_t	timestamp
int16_t	subtype

```

int16_t    x
int16_t    y
int16_t    z
int16_t    id
int16_t    spare

```

Where:

button	GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left GR_EVENT_BTN_MIDDLE - 0x0002 GR_EVENT_BTN_RIGHT - 0x0004
timestamp	This is an event timestamp in milliseconds since application start
subtype	GR_EVENT_RELEASE_IN GR_EVENT_RELEASE_OUT
z	This parameter is dependent on the availability of z- co-ordinate information
id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

gre.mtoutbound

A mouse/touch has left a control (if dragging a pointer or finger) while other mouse/touch points remain in the control bounds. This event is generated once the coordinates leave a control boundary. If mouse motion events are not supported on the platform then this event will not be generated.

Note

Control groups can not receive outbound events.

Data

```

uint32_t    button
uint32_t    timestamp
int16_t     subtype
int16_t     x
int16_t     y
int16_t     z
int16_t     id
int16_t     spare

```

Where:

button	GR_EVENT_BTN_LEFT - 0x0001: if this is a touchscreen then the button is always left GR_EVENT_BTN_MIDDLE - 0x0002 GR_EVENT_BTN_RIGHT - 0x0004
timestamp	This is an event timestamp in milliseconds since application start
subtype	GR_EVENT_RELEASE_IN

	GR_EVENT_RELEASE_OUT
z	This parameter is dependent on the availability of z- co-ordinate information
id	This parameter is used to track multi-touch presses as they come in
spare	This is padding and should be 0

Keyboard Events

The following events are generated if a keyboard is present and supported by the render manager.

gre.keydown

A key is in the pressed state

Note

gre.keydown events with control as a destination require focus.

Data

uint32_t	code
uint32_t	modifiers

Where:

code	This is the UTF-8 key value
modifiers	A set of modifiers applied to the key

GR_EVENT_KEYMOD_ALT
GR_EVENT_KEYMOD_CTRL
GR_EVENT_KEYMOD_SHIFT

gre.keyup

A key which was previously pressed has been released

Data

uint32_t	code
uint32_t	modifiers

Where:

code	This is the UTF-8 key value
modifiers	A set of modifiers applied to the key

GR_EVENT_KEYMOD_ALT

GR_EVENT_KEYMOD_CTRL
GR_EVENT_KEYMOD_SHIFT

Screen Manager Events

The following events are generated by the Screen Manager during screen transitions. These events are generated in the following order:

`gre.screenshow.pre` delivered to target (end) screen
`gre.screenhide.pre` delivered to source (start) screen
`gre.screenshow.post` delivered to target (end) screen
`gre.screenhide.post` delivered to source (start) screen

`gre.screenshow.pre`

A screen is being shown. This event is triggered before the screen is shown and signifies that a transition may be starting

Data:

`char *name` The name of the screen which is being shown

`gre.screenshow.post`

A screen has been shown. This event is triggered after the screen is shown and signifies that a transition has ended.

Data:

`char *name` The name of the screen which has been shown

`gre.screenhide.pre`

A screen is being hidden. This event is triggered before the screen is hidden and signifies that a transition may be starting.

Data:

`char *name` The name of the screen which is being hidden

`gre.screenhide.post`

A screen has been hidden. This event is triggered after the screen is hidden and signifies that a transition has ended.

Data:

`char *name` The name of the screen which has been hidden

Focus Events

The following events are generated on a change of control focus. If there is no focusable control on the current screen then these events will not be generated. These events are targeted at the currently or last focused control. When focus shifts from one control to another the lost focus event is sent first followed by the got focus event.

gre.gotfocus

A control has received focus, delivered to the control that received the focus.

No data payload.

gre.lostfocus

A control has lost focus, delivered to the control that has lost the focus.

No data payload.

Table Events

The following events are generated by a Table control. If no table control is present then these events will not be generated.

gre.table.viewport

A table has been resized via the table resize action. This event notifies the system of the new table size and visible area.

Data:

```
uint32_t    top_row
uint32_t    left_col
uint32_t    bot_row;
uint32_t    right_col;
char        *table
```

Where:

top_row	The top row that is visible
left_col	The left column that is visible
bot_row	The bottom row that is visible
right_col	The right column that is visible
table	The name of the table whose viewport changed to cause this event

gre.cell.gotfocus

A table cell has received focus and is the currently active cell. This is delivered to the control template with the *cell* focus information.

Data:

```
uint32_t    row;
uint32_t    col;
char *table
```

Where:

row The row that received focus

col The column that received focus

table The name of the table where the cell focus changed

gre.cell.lostfocus

A table cell has lost focus and is no longer the active cell. This is delivered to the control template with the *cell* focus information

Data:

```
uint32_t    row;
uint32_t    col;
char *table
```

Where:

row The row that received focus

col The column that received focus

table The name of the table where the cell focus changed

Table Scroll Events

The following events are only generated when the "Enable list scrolling behavior" option is checked in the Table properties.

gre.table.drag_start

This event is generated when a user begins dragging a scrolling table.

Data:

 The name of the object being scrolled

gre.table.drag_stop

This event is generated when a user stops dragging a scrolling table.

Data:

 The name of the object being scrolled

gre.table.scroll_trigger

This event is generated before the `table.drag_stop` event when it is detected that automatic scrolling will occur.

Data:

The name of the object being scrolled

gre.table.scroll_start

This event is generated when the scroll animation begins.

Data:

The name of the object being scrolled

gre.table.scroll_stop

This event is generated when the scroll animation completes.

Data:

The name of the object being scrolled

gre.table.scroll_cancel

This event is generated when the scroll animation is interrupted.

Data:

The name of the object being scrolled

gre.table.scroll_complete

This event is generated when the object being scrolled has stopped moving naturally after a drag or an animated scroll

Data:

The name of the object being scrolled

Layer Scroll Events

The following events are only generated when the "Enable layer scrolling behavior" option is checked in the Layer Scrolling properties.

gre.drag.start

This event is generated when a user begins dragging a scrolling layer.

Data:

The name of the object being scrolled

gre.drag.stop

This event is generated when a user stops dragging a scrolling layer.

Data:

The name of the object being scrolled

gre.scroll.trigger

This event is generated before the drag.stop event when it is detected that automatic scrolling will occur.

Data:

The name of the object being scrolled

gre.scroll.start

This event is generated when the scroll animation begins.

Data:

The name of the object being scrolled

gre.scroll.stop

This event is generated when the scroll animation completes.

Data:

The name of the object being scrolled

gre.scroll.cancel

This event is generated when the scroll animation is interrupted.

Data:

The name of the object being scrolled

gre.scroll.complete

This event is generated when the object being scrolled has stopped moving naturally after a drag or an animated scroll

Data:

The name of the object being scrolled

Mobile Events (Android and iOS)

The following events are only generated when running on Android and iOS.

gre.mobile.on_pause

The application has become inactive. The application will not be rendering to the screen after this event is received.

Data:

No data payload

gre.mobile.on_resume

The application has become active. The application will be rendering to the screen after this event is received.

Data:

No data payload

gre.mobile.on_background

The application has lost focus.

Data:

No data payload

Android Events

The following event is only generated when running on Android.

android.onBack

The back button on the Android application has been pressed.

Data:

No data payload

Windows Embedded Compact 2013 (WEC2013) Events

Limited gestures support has been added to the winevent plugin for the Windows Embedded Compact 2013 platform. This support has been added via the Storyboard Engine winevent plugin (libgre-plugin-winevent.dll) and this plugin must be included in Storyboard Engine distribution.

The gesture support is designed to make visible the internal Windows gesture events and payloads that are generated from the underlying system as described in this document: <https://msdn.microsoft.com/en-us/library/ee503599.aspx>

As of the Storyboard 4.2 release, only the GID_PAN and GID_SCROLL sub-category of WM_GESTURE gesture events are translated into corresponding Storyboard events. These events will only be generated on WEC2013 hardware platforms where the BSP has been configured with gesture event support and the touchscreen driver configured to enable such event generation.

In order to add application support for receiving these win.gesture events, the events must be added to the Storyboard Designer application. The events can be added in the same manner as any other user defined events are added at the point where they are used to trigger an action within the "New Action" dialog.

win.gesture.pinch

This event is generated in response to the Windows WM_GESTURE:GID_PAN event. The event data is taken directly from the Windows event.

Data: (4s1 x 4s1 y 4s1 spread)

```
int32_t x
int32_t y
int32_t spread
```

win.gesture.[up|down|left|right|unknown]

This event is generated in response to the Windows MW_GESTURE:GID_SCROLL event. The event data is taken directly from the Windows event.

Data: (4s1 velocity 4s1 angle)

```
int32_t velocity
int32_t angle
```

Plugin Specific Event Definitions

The following events are generated by optional Storyboard plugins.

Timer Events

These events are generated by the `libgre-timer` plugin

timer.[name] Timer Events

Timer events are generated as a result of a timer action. See documentation on the `gra.timer` action for further information about configuring timers.

The timer event name will be formatted as `timer.[name]` where `name` is the value set as the name of the timer when the action was defined.

Animation Events

These events are generated by the `libgre-animation` plugin

gre.animate.complete.[name]

Animation complete events are generated as a result of an animation action. See documentation on the `gra.animation` action for further information about configuring animations.

When an animation is completed, an animation complete event will be fired. The event name will be formatted as `gre.animate.complete.[name]` where `name` is the value set as the name of the animation when the action was defined.

gre.animate.stop.[name]

Animation stop events are generated as a result of an animation stop action. See documentation on the `gra.animate.stop` action for further information on terminating animations.

When an animation is stopped, an animation stop event will be fired. The event name will be formatted as `gre.animate.stop.[name]` where `name` is the value set as the name of the animation when the action was defined.

Data: (1s0 id)

Gesture Events

These events are generated by the `libgre-gesture` plugin

`gre.gesture.up`

Data:

```
char *gesture_num
int32_t time
```

`gre.gesture.down`

Data:

```
char *gesture_num
int32_t time
```

`gre.gesture.left`

Data:

```
char *gesture_num
int32_t time
```

`gre.gesture.right`

Data:

```
char *gesture_num
int32_t time
```

Screen Display Capture (ScreenDump) Events

These events are generated by the `libgre-screen-dump` plugin

`gre.screendump.complete`

A screen dump action has completed.

gre.screendump.failed

Data:

`char *message`

Screen Event Capture/Playback Events

These events are generated by the `libgre-animation` plugin

gre.capture.started

This event will be generated when the capture-playback plugin begins an event capture session.

gre.capture.stopped

This event will be generated when the capture-playback plugin event capture session is stopped.

gre.playback.started

This event will be generated when the capture-playback plugin begins an event playback session.

gre.playback.complete

This event will be generated when the capture-playback plugin playback is finished by reaching the end of the playback session.

gre.playback.stopped

This event will be generated when the capture-playback plugin playback session is stopped.

Media Events

gre.media.exit

The media backend application has exited.

Data:

No data payload

gre.media.timeupdate

Emitted when the time has been updated.

Data: "4u1 time_elapsed 1s0 channel_name"

```
unsigned time_elapsed
char channel_name[MAX_CHANNEL_NAME_LEN + 1]
```

Where:

time_elapsed The time that has elapsed during play back

channel_name The name of the channel that this time event occurred on

gre.media.durationupdate

Emitted when a change in the duration of the media is detected.

Data: "4u1 total_time 1s0 channel_name"

```
unsigned total_time
char channel_name[MAX_CHANNEL_NAME_LEN + 1]
```

Where:

total_time The total duration of the current media file

channel_name The name of the channel that this time event occurred on

gre.media.statechange

Emitted when the player has changed state, between a paused and playing state.

Data: "1s33 channel_name 1s0 state"

```
char channel_name[MAX_CHANNEL_NAME_LEN + 1]
char state[1]
```

Where:

channel_name The name of the channel that is changing state

state The new state: "paused" | "playing"

gre.media.complete

Triggered when the named media has played to the end and stopped playing

Data: "1s33 channel 1s0 name"

```
char channel_name[MAX_CHANNEL_NAME_LEN + 1]
char media_name[1]
```

Where:

channel_name The name of the channel that has completed playback

media_name The name of the media stream that completed playback

gre.media.error

Triggered when there was an error playing the media source.

Data: "1s33 channel_name 1s0 error_msg"

```
char channel_name[MAX_CHANNEL_NAME_LEN + 1]
char error_msg[1]
```

Where:

channel_name The name of the channel that received an error

error_msg[1] The error message

Logger Events

gre.%perf_prefix%.start

Turn on performance logging. Note that %perf_prefix% defaults to "perf", but can be set through the logger plugin options.

Data: "1s0 options" The options to start the performance logging with. This can be omitted. Only the performance logging options will be recognized

gre.%perf_prefix%.stop

Turn off performance logging. Note that %perf_prefix% defaults to "perf", but can be set through the logger plugin options.

Data:

No data payload

gre.%perf_prefix%.set.options

Set the options to use for performance logging. Note that %perf_prefix% defaults to "perf", but can be set through the logger plugin options.

Data: "1s0 options" The options to use with performance logging. Only the performance logging options will be recognized

Appendix E. Standard Action Definitions

Built-in Action Definitions

Storyboard supports a number of standard actions which are built-in to the framework. These actions are all prefixed with `gra.` and can be incorporated into your application design without any plugin dependency.

gra.screen

Cause a screen transition to occur by replacing the current screen with the new one.

The action arguments are:

`screen` The name of the screen to transition to.

gra.screen.fade

Causes a screen transition to occur by fading the old screen into the new one.

The action arguments are:

`screen` The name of the screen to transition to.

`rate` Animation rate string that defines how the alpha value will change over the transition:

`linear`
`easein`
`easeout`
`easeinout`
`bounce`

`fps` The frames per second to use for the transition

`duration` The duration of the transition in milliseconds

gra.screen.hold

Hold all screen updates. While held a screen will not redraw.

gra.screen.release

Release a held screen. If a screen was damaged during the period of time that the screen was being held, then a redraw action will be triggered.

gra.sendevent

Send an event to the application's input event queue. This action is equivalent to injecting an event via Storyboard IO or using the Lua `gre.send_event()` API

The action arguments are:

event The name of the event to send

gra.datachange

Change or create a variable value in the data manager.

The action arguments are key/value pairs such that the key is the fully qualified model path for the variable and the value is the new value to assign to that variable. For more information on creating the model paths, see the Data Variables section of this document.

gra.screen.focus.set

Set the focus to a specific control.

The action arguments are:

index The focus index to set the focus to. If the control name is to be used instead of specific focus index value then this value should be set to -1 to avoid confusion about the intent of the action.

control The name of a control to set the focus to.

Only one of either the index or the control need to be provided for this action. If both arguments are set, then the index value will be used.

In order for this action to complete successfully, the control specified must be set as focusable. For more information about making controls focusable and the focus operation in Storyboard, refer to the Focus section in the Execution Pipeline part of this document.

gra.screen.focus.next

Move the current focus to the next focusable control.

The action arguments are:

min The minimum focus index to move to or -1 to have no minimum value

max The maximum focus index to move to or -1 to have no maximum value

In order for this action to complete successfully, there must be a control whose focus index lies between the min and max values. For more information about making controls focusable and the focus operation in Storyboard, refer to the Focus section in the Execution Pipeline part of this document.

gra.screen.focus.prev

Move the current focus to the previous focusable control.

The action arguments are:

min The minimum focus index to move to or -1 to have no minimum value

max The maximum focus index to move to or -1 to have no maximum value

In order for this action to complete successfully, there must be a control whose focus index lies between the min and max values. For more information about making controls focusable and the focus operation in Storyboard, refer to the Focus section in the Execution Pipeline part of this document.

gra.screen.focus.direction

Move the current focus to the next control in a direction.

The action arguments are:

min	The minimum focus index to move to or -1 to have no minimum value
max	The maximum focus index to move to or -1 to have no maximum value
direction	The direction to search for the next focusable control
	up
	down
	left
	right

In order for this action to complete successfully, there must be a control whose focus index lies between the min and max values. For more information about making controls focusable and the focus operation in Storyboard, refer to the Focus section in the Execution Pipeline part of this document.

gra.table.scroll

Scroll the content of one or more tables.

Note

Only one table scroll animation can be run at a time. Storyboard Engine will cancel the currently running table scroll animation by snapping to the end of the animation and then it will trigger the new table scroll animation.

The action arguments are:

control	The name of the table control to scroll. May be a comma separated list if multiple tables are specified.
row	The absolute 1 based row to start the scroll from. The default, 0, indicates that scrolling should start from the current row
col	The absolute 1 based column to start the scroll from. The default, 0, indicates that scrolling should start from the current column
delta_row	The number of rows to move. A positive value moves the table <i>down</i> a negative value moves the table <i>up</i>
delta_col	The number of columns to move. A positive value moves the table <i>right</i> a negative value moves the table <i>left</i>
fps	The frames per second rate at which to scroll the table.
duration	The duration in milliseconds to run the scroll over.

gra.table.resize

Set the number of rows and columns for a table. This action does not resize the control, simply the number of cells contained within the virtual table.

The action arguments are:

control	The name of the table control to resize.
rows	The number of rows for the table. Specifying 0 will leave the current number of rows unchanged.
columns	The number of columns for the table. Specifying 0 will leave the current number of columns unchanged.

When the table is resized, a `gre.table.resize` event will be emitted.

gra.table.navigate

Navigates the cells of a table, sets the active cell which in turn generates the cell focus events. If the new active cell is not visible the table will be scrolled in order to show this cell.

The action arguments are:

control	The name of the table control to scroll. May be a comma separated list if multiple tables are specified.
fps	The frames per second rate at which to scroll the table, 0 performs an immediate scroll.
duration	The duration in milliseconds to run the scroll over, 0 scrolls it immediately.
direction	The type of navigation to perform
set	Sets the active row and column to what is specified in the row/col parameters. The option only ensures that the cell is visible and does not guarantee the cell will be at the top of the visible list.
next	Move to the next cell, scroll by column then by row
prev	Move to the previous cell, scroll by column then by row
up	Move to the cell above the current one
down	Move to the cell below the current one
left	Move to the cell to the left of the current one
right	Move to the cell to the right of the current one
home	Move to the first cell in the table at row,column 1,1
end	Move to the last cell in the table
row	The row to navigate to. This is only used if <code>direction</code> is assigned <code>set</code>
col	The column to navigate to. This is only used if <code>direction</code> is assigned <code>set</code>

gra.scroll.stop

Stop the scrolling momentum of an object (layer or table)

The action arguments are:

object The object whose scrolling should stop. This should be the fully qualified name of a layer instance, or the name of a table.

gra.log

Use the GRE logging mechanism to output a message.

The action argument is the string message that should be output.

gra.resource.dump_def

Remove a resource which is managed by the resource manager.

The action arguments are:

pool The pool name containing the resource to dump

ref The name of the resource to dump

The currently defined resource pools are `image` containing all of the images and `font` containing all of the fonts associated with the Storyboard application.

gra.playback

This action is used to begin or to stop playback of a capture-playback file using the capture-playback plugin. The plugin currently only supports the playback of one file at a time.

The action arguments are:

Playback True to begin playback

False to stop playback

Filename The name of the file to playback

Loop The number of times to loop playback

Quit Send a quit message when finished

gra.capture

This action is used to begin or to stop capture of a capture-playback file using the capture-playback plugin. Only one capture-playback file may be captured at a time.

The action arguments are:

Capture True to begin capture

False to stop capture

Filename The name of the file to capture the events

Plugin Action Definitions

The following actions are only available when optional Storyboard plugins has been loaded.

gra.lua

Cause a Lua script function to execute.

Plugin `libgre-plugin-lua.so`

Options: `script` The name of the Lua function to invoke

Additional arguments can be passed to the function by providing additional key/value pairs to the action. The key/value pairs are provided to the Lua function as values in the argument table.

For example to call the Lua function `myfunction` with an extra argument, `firstargument`, that corresponds to the value of the application variable `myvar` you would simply add a new entry to the parameter list.

The corresponding call to the Lua function would fill the entry into the argument table such that:

```
function myfunction(mapargs)
    print("The value is: " .. tostring(mapargs.firstargument))
end
```

would print out the value of `${app:myvar}`.

gra.ccallback

Cause a C function to execute. This action is only available on Storyboard Lite platforms. The functionality can be simulated in Designer though through Lua. The C Callback plugin will search for a Lua function that has the same name as the C function.

Plugin `libgre-plugin-ccallback.a`

Options: `function` The name of the C function to invoke

gra.animate

Start an animation. Animations are started based on their name. Each animation can have an optional identifier (id) which is used to ensure that animations run in an exclusive manner. If an existing animation is running that uses the same identifier, then that animation is stopped before this animation is started. The data argument is as follows:

Plugin `libgre-plugin-animation.so`

Options: `name` The animation name to start

id An optional instance id to be associated with the animation. Animation identifiers can be used with different animations to ensure that only one animation of the set is running at a time.

When the animation stops it will emit a notification event in the form of `gre.animate.complete.[name]`. This event will be delivered within the context of the `gre.animate` action and will be delivered to the object which invoked the action.

gra.animate.stop

Stop an animation. If you stop an animation only by name then all running animations with that name will stop and emit a complete event. If you stop an animation by id then only that specific animation will stop and emit a complete event. The data argument is as follows:

Plugin `libgre-plugin-animation.so`

Options: **name** The animation name to stop

id An optional instance id associated with the name

When the animation stops it will emit a notification event in the form of `gre.animate.complete.[name]`

gra.audio

Start or stop the asynchronous playback of a WAV audio file. The data argument is as follows:

Plugin: `libgre-plugin-audio.so`

Options: **filename** A filename to play, or empty to stop the current playback.

gra.greio

Send a new event over a Storyboard IO channel. This action is best suited for name only event delivery. It is possible to use this action for simple single data item payloads (ie a single string or a single numeric value) but it is not recommended. The Lua API provides a more comprehensive Storyboard IO event delivery mechanism via the `gre.send_event` or `gre.send_event_data` function calls.

The action parameters below map directly to the Storyboard IO C API and a more comprehensive explanation of their use can be found there and in the Storyboard architecture and Working with Events sections of this document.

Plugin: `libgre-plugin-greio.so`

Options: **name** Storyboard IO channel name to send the event to (required)

event The name of the event to generate (required)

target The fully qualified name of the target for the event as a string. (optional)

format The format of the event data (optional). Only a single data format element can be provided in this action (ie `1s0` or `4s1`)

data The data payload for the event (optional). This data must match the type associated with the format string to be properly interpreted.

gra.perf_state

Control the capture of performance data

Plugin: `libgre-plugin-logger.so`

Options: `state` Turn off (0) or on (1) performance data capture

In order for this action to be used, the `libgre-plugin-logger.so` must have been started with performance logging enabled, but not necessarily to have it start capturing the performance data. For example `sbengine -ologger,perf=0` will enable performance logging but not start capturing events at startup while `sbengine -ologging,perf=1` will enable performance logging and immediately start capturing events.

gra.redirect

Redirect all events to another Storyboard IO channel.

Plugin: `libgre-plugin-redirect.so`

Options: `channel` Storyboard IO channel name to send the events to

gra.screen.path

Causes a screen transition to occur by sliding the old screen out and/or the new one in from a given direction.

Plugin: `libgre-plugin-screen-path.so`

Options: `screen` The screen to transition to

`rate` Animation rate string that defines how the screen(s) will move over time

`linear`
`easein`
`easeout`
`easeinout`
`bounce`

`fps` Number of frames per second

`duration` Length of the transition in milliseconds

`direction` The direction to transition from

`left`
`right`
`top`
`bottom`

`layers` The layers to transition, all of the layers or just the layers that are different between source and destination.

`all`
`delta`

`moving` The screen(s) to animate with the desired path transition.

both
new only
old only

gra.screen.scale

Causes a screen transition to occur by scaling the old screen into the new one.

Plugin: `libgre-plugin-screen-scale.so`

Options: `screen` The screen to transition to

`rate` Animation rate string that defines how the alpha values will change

linear
easein
easeout
easeinout
bounce

`fps` Number of frames per second

`duration` Length of the transition in milliseconds

`layers` The layers to transition, all of the layers or just the layers that are different between source and destination.

all
delta

gra.screen.glswitch

Causes a screen transition to occur by using 3D to switch the old screen into the new one.

Plugin: `libgre-plugin-screen-3d.so`

Options: `screen` The screen to transition to

`rate` Animation rate string that defines how the alpha values will change

linear
easein
easeout
easeinout
bounce

`fps` Number of frames per second

`duration` Length of the transition in milliseconds

`layers` The layers to transition, all of the layers or just the layers that are different between source and destination.

all	Move all layers between source and destination screens
-----	--

delta	Collapse common topmost and bottom most visible layers and only move the layers in between that are different between screens.
-------	--

gra.screen.glrotate

Causes a screen transition to occur by using 3D to rotate the old screen in the x-axis into the new one.

Plugin: `libgre-plugin-screen-3d.so`

Options: `screen` The screen to transition to

`rate` Animation rate string that defines how the alpha values will change

 linear
 easein
 easeout
 easeinout
 bounce

`fps` Number of frames per second

`duration` Length of the transition in milliseconds

`direction` The direction to transition from

 left
 right
 top
 bottom

`layers` The layers to transition, all of the layers or just the layers that are different between source and destination.

all	Move all layers between source and destination screens
delta	Collapse common topmost and bottom most visible layers and only move the layers in between that are different between screens.

gra.screen.gflip

Causes a screen transition to occur by using 3D to switch the old screen into the new one.

Plugin: `libgre-plugin-screen-3d.so`

Options: `screen` The screen to transition to

`rate` Animation rate string that defines how the alpha values will change

 linear
 easein
 easeout

	easeinout bounce
fps	Number of frames per second
duration	Length of the transition in milliseconds
layers	The layers to transition, all of the layers or just the layers that are different between source and destination.

all	Move all layers between source and destination screens
delta	Collapse common topmost and bottom most visible layers and only move the layers in between that are different between screens.

gra.screen.gldoors

Causes a screen transition to occur by using 3D to switch the old screen into the new one using a door opening animation.

Plugin: libgre-plugin-screen-3d.so

Options:	screen	The screen to transition to
	rate	Animation rate string that defines how the alpha values will change
		linear easein easeout easeinout bounce
	fps	Number of frames per second
	duration	Length of the transition in milliseconds
	direction	The direction to transition from
		left right top bottom
	layers	The layers to transition, all of the layers or just the layers that are different between source and destination.

all	Move all layers between source and destination screens
delta	Collapse common topmost and bottom most visible layers and only move the layers in between that are different between screens.

gra.screen.gltip

Causes a screen transition to occur by using 3D to switch the old screen into the new one by tipping the display forward.

Plugin: `libgre-plugin-screen-3d.so`

Options: `screen` The screen to transition to

`rate` Animation rate string that defines how the alpha values will change

linear
easein
easeout
easeinout
bounce

`fps` Number of frames per second

`duration` Length of the transition in milliseconds

`direction` The direction to transition from

left
right
top
bottom

`layers` The layers to transition, all of the layers or just the layers that are different between source and destination.

all	Move all layers between source and destination screens
delta	Collapse common topmost and bottom most visible layers and only move the layers in between that are different between screens.

gra.screen.glcube

Causes a screen transition to occur by using 3D to switch the old screen into the new one using a cube animation.

Plugin: `libgre-plugin-screen-3d.so`

Options: `screen` The screen to transition to

`rate` Animation rate string that defines how the alpha values will change

linear
easein
easeout
easeinout
bounce

fps	Number of frames per second
duration	Length of the transition in milliseconds
direction	The direction to transition from left right top bottom
layers	The layers to transition, all of the layers or just the layers that are different between source and destination.
all	Move all layers between source and destination screens
delta	Collapse common topmost and bottom most visible layers and only move the layers in between that are different between screens.

gra.screen.rotate

Causes a screen transition where the new screen is rotated.

Plugin: `libgre-plugin-screen-rotate.so`

Options: `screen` The screen to transition to
`rotation` The rotation value to apply to the new screen.

gra.screendump

Dump the contents of the screen to an image file.

Plugin: `libgre-plugin-screen-dump.so`

Options: `filename` The filename of the image file to create. The directory path to the filename must exist and the filename will be overwritten if it is. The filename must end with either a `.bmp` extension to generate BMP formatted images or `.tga` to generate TGA formatted images

gra.system

Execute a system command. System commands will be dependent on the underlying operating system. The result of the command will be logged to standard output.

Plugin: `libgre-plugin-system.so`

Options: `command` A system command to execute in text form.

gra.timer

Start, stop and control a timer.

Plugin:	libgre-plugin-timer.so								
Options:	<table> <tr> <td>name</td><td>The name to use to identify this timer (required)</td></tr> <tr> <td>rtime</td><td>The time delay in milliseconds <i>relative</i> to the action invocation. Specify a value of 0 to stop an existing timer.</td></tr> <tr> <td>repeat</td><td>The number of milliseconds to delay after the timer first fires, used to provide a stable repeat timer. Specify 0 for a one shot timer.</td></tr> <tr> <td>count</td><td>The number of times that the timer should repeat before automatically stopping, assuming that the timer is not a one shot timer. Specify -1 to allow an unlimited number of repeat firings</td></tr> </table>	name	The name to use to identify this timer (required)	rtime	The time delay in milliseconds <i>relative</i> to the action invocation. Specify a value of 0 to stop an existing timer.	repeat	The number of milliseconds to delay after the timer first fires, used to provide a stable repeat timer. Specify 0 for a one shot timer.	count	The number of times that the timer should repeat before automatically stopping, assuming that the timer is not a one shot timer. Specify -1 to allow an unlimited number of repeat firings
name	The name to use to identify this timer (required)								
rtime	The time delay in milliseconds <i>relative</i> to the action invocation. Specify a value of 0 to stop an existing timer.								
repeat	The number of milliseconds to delay after the timer first fires, used to provide a stable repeat timer. Specify 0 for a one shot timer.								
count	The number of times that the timer should repeat before automatically stopping, assuming that the timer is not a one shot timer. Specify -1 to allow an unlimited number of repeat firings								

rtime must be specified and a value of 0 for rtime and repeat indicates that the timer should stop firing.

For example, to start a timer that fires in 1s from the event and then every 500ms afterwards:

```
<action ... type="gra.timer" data="name=MyTimer,rtime=1000,repeat=500" />
```

Then to stop the timer:

```
<action ... type="gra.timer" data="name=MyTimer,rtime=0,repeat=0" />
```

An event will be generated each time that the timer fires and will be named timer.<name> so for the examples above, the event would be generated would be timer.MyTimer.

gra.timer.stop

Stop a timer.

Plugin: libgre-plugin-timer.so

Options: name The name to use to identify this timer (required)

For example, to start a timer that fires in 1s from the event and then every 500ms afterwards:

```
<action ... type="gra.timer" data="name=MyTimer,rtime=1000,repeat=500" />
```

Then to stop the timer:

```
<action ... type="gra.timer.stop" data="name=MyTimer" />
```

Media Actions

The following actions can be used to control the media playback. Note that all actions, with the exception of the connect and disconnect actions, take a “channel_name” argument. This is used to target a specific playback channel. For example if a video is started with “gra.media.new.video” with “channel_name=video1” then any subsequent action which wants to act on this video, such as play/pause, must set the channel “video1”.

gra.media.connect

Tells the plugin to try and connect to the media backend.

There are no action arguments for this action

gra.media.disconnect

Tells the plugin to disconnect from the media backend.

There are no action arguments for this action.

gra.media.new.audio

Tells the plugin to play a new audio file.

The action arguments are:

channel_name	The channel name the new video is to be played on
media_name	The name of the media to play, full path to an audio file
volume	The initial volume value to play the media at. The value should be between 0 and 100.
update_interval	The number of milliseconds to wait in between update messages
emit_time_events	A value that is set to 1 to emit time update events, 0 otherwise
extra_data	Any extra data that should be passed to the backend, can be NULL

gra.media.new.video

Tells the plugin to play a new video file.

The action arguments are:

channel_name	The channel name the new video is to be played on
media_name	The name of the media to play, full path to a video file
volume	The initial volume that the media should be played at
object_name	The name of the external object to display content on. This is necessary when using an external render extension to display the content, please refer to the external render extension documentation
external_buffer_name	The name of the render extension to display content on. This is necessary when using an external render extension to display the content, please refer to the external render extension documentation
update_interval	The number of milliseconds to wait in between update messages
emit_time_events	A value that is set to 1 to emit time update events, 0 otherwise
output_width	The width of the video
output_height	The height of the video
output_depth	The output depth of the video in bytes per pixel. 16bit = 2, 24bit = 3, 32bit = 4

extra_data	Any extra data that should be passed to the backend, can be NULL. See each backend for a description of this data
------------	---

gra.media.volume

Triggers a change in the playback volume.

The action arguments are:

channel_name	The channel name to change the volume on
volume	The value to change the volume to, a number between 0 and 100
emit_volume_event	A value that is set to 1 if an event should be emitted or 0 otherwise

gra.media.seek

Triggers a change to the current playback position of the media that is playing.

The action arguments are:

channel_name	The channel name to change the seek position on
seek_num	The new seek position for the media file
emit_state_event	A value that is set to 1 if an event should be emitted or 0 otherwise

gra.media.stop

Changes the media playback state to stopped.

The action arguments are:

channel_name	The channel name to change the state on
emit_state_event	A value that is set to 1 if an event should be emitted or 0 otherwise

gra.media.resume

Changes the media playback state from paused to playing.

The action arguments are:

channel_name	The channel name to change the state on
emit_state_event	A value that is set to 1 if an event should be emitted or 0 otherwise

gra.media.pause

Changes the media playback state from playing to paused.

The action arguments are:

channel_name	The channel name to change the state on
emit_state_event	A value that is set to 1 if an event should be emitted or 0 otherwise

Appendix F. Standard Render Extension Definitions

Common Render Extension Options

The follow is a list of common options across many render extensions

X	The x position of the render extension relative to the control (number, optional, default: 0)
Y	The y position of the render extension relative to the controls (number, optional, default: 0)
Width	The width of the render extension, if it is set to -1 then it will set to the size of the object being rendered (number, optional, default: width of object it is attached to)
Height	The height of the render extension, if -1 then it will set to the size of the object being rendered (number, optional, default: height of object it is attached to)
Alpha	The alpha or transparency value for this render extension (number, optional, 0-255, default: 255 (opaque))
Rotation	The rotation for the item in degrees, (0-360) (number, optional, default: 0)
Vertical Alignment	The vertical alignment within the control (number, optional, default: 0). This alignment value is a numeric value corresponding to the following:

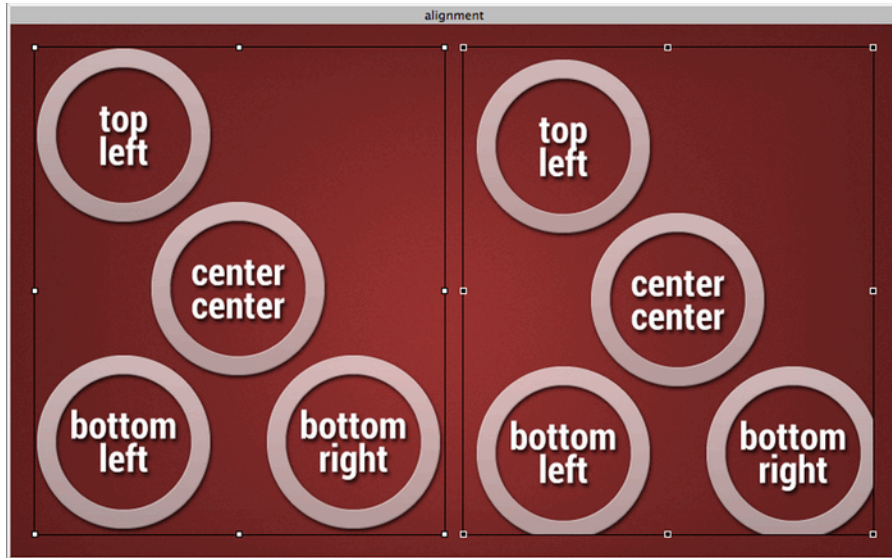
0	top (default)
1	top
2	center
3	bottom

Horizontal Alignment	The horizontal alignment within the control (number, optional, default: 0). This alignment value is a numeric value corresponding to the following:
----------------------	---

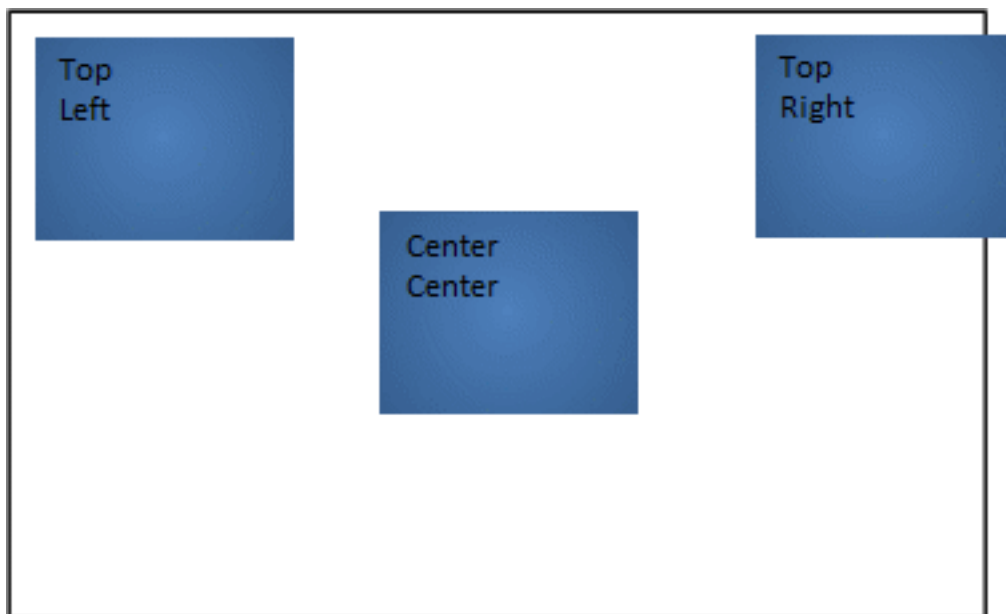
0	left (default)
1	left
2	center
3	right

A render extension can have a vertical and horizontal alignment. This alignment is based on the control area and the render extension position and size. The following describes the effects of these parameters on alignment.

Width and height set, position set to (0,0)



Width and height set, position set to (5,5). The position functions as an offset to the alignment



Canvas

The canvas render extension provides a surface for custom drawing using the the section called “Storyboard Lua Canvas API”. The Plotting Sample demonstrates how to use the Lua canvas API and the canvas render extension together.

The canvas render extension uses many standard options but also includes:

Name The canvas name to use with the `gre.get_canvas`

Circle and Arc

The circle and arc render extensions provide a quick way to draw filled or outlined circles and ellipses in addition to arc segments. The Circles Sample demonstrates how the properties of this render extension can be configured.

The circle render extension uses many standard options but also includes:

Style	This is the style of the circle or arc and can be filled or stroked. If it is stroked then the line width option can be set to determine the arc fill size
Color	Color to use to stroke or fill the circle or arc.
Start/End Angle	This option is available only for arcs and determines the start and end angle in degrees that the arc should sweep through.

External Buffer

The external buffer render extension creates a buffer for other system applications or tasks to render into, things such as video players and web browsers. This extension is only available when the external plugin has been loaded. The Media Sample demonstrates how to use an external buffer to render content from a video player.

The external buffer render extension uses many standard options but also includes:

Name	The name of the external render extension. This information should be provided by the external render extension application provider and is used to allow the application to send update messages to the Storyboard Engine.
Object	This is the path to a shared memory object which is created by the external application and is loaded by the Storyboard Engine. This information should be provided by the external render extension application provider.

Fill

The fill render extension draws a filled rectangle to the screen.

The fill render extension uses many standard options but also includes:

Color	Color to use for item (number RGB format, optional)
-------	---

Image

The image render extension draws an image to the screen. Images can be scaled and tiled. When scaled, the image can also be set to maintain the aspect ratio. Many of the Samples showcase image property manipulation.

The image render extension uses many standard options but also includes:

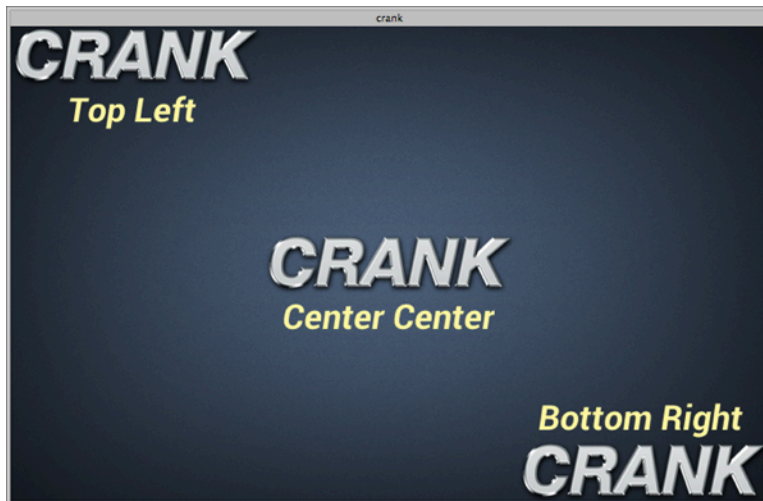
Name	The name of the image to use (string, optional)
Scale	Enable scaling of the image if set to 1. The image will be loaded at full resolution and then scaled when rendered. If width and height are not set the image is scaled to the control size (number, optional)

Load Scaled	Enable scaling on load of the image if set to 1. This will load the image at the specified size and scale during the image decode. If width and height are not set the image is scaled to the control size (number, optional)
Tile	Tile the image, if width and height are not set the image is tiled to the control size (number, optional)
Aspect	If scaling maintain the images aspect ratio (number, optional)
Center Rotation	If this value is turned on then any rotation applied to the image will happen around the center of the image and the values of Center X , Center Y will be ignored.
Center X, Y	These values are only applied when the Center Rotation option is disabled and they specify the location of the rotation center point as a value relative to the control's upper left corner as 0,0 increasing as you go right and down. So to rotate around the center of a control whose width, height was 10, 20 you could specify a center point of X = 5 and Y = 10

Non-scaled, Scaled and Tiled Image Sample



Image Alignment Sample



Polygon

The polygon render extension draws a filled (convex) polygon to the screen. This extension is only available when the polygon plugin has been loaded. The Trend Sample demonstrates how to use a polygon with dynamic point generation to render a trend graph.

The polygon render extension uses many standard options but also includes:

Points a list of points for the polygon (string). The list of points are x,y values that are space separated, for example: 10,20 5,5 0,0.

Style The style of polygon to render (string). The style corresponds to one of the following values:

Dash ("dash")	Draw a dashed line using Fill (color) and Dash Pattern attributes
Fill ("fill")	Draw a filled polygon using Fill (color) attribute
Fill & Stroke ("filloutline")	Draw a filled polygon using Fill (color) attribute, then outlined with Outline (color) attribute
Line ("line")	Draw a line using Fill (color) attribute
Loop ("loop")	Draw a line (with connected ends) using Fill (color) attribute

Line Width ("linewidth") Changes width of the outline (stroke) drawn (number)

Dash Pattern ("dash") Defines the dash size and the space size between each dash, only used with "dash" style (string). For example a long line short space might look like "10 5 10 5"

Anti-alias ("antialias") Indicates if the polygon line drawn should be anti-aliased (1) or not (0). Single width outlined polygons are not anti-aliased on OpenGL ES 2.0 platforms. The `multisample` option for the OpenGL ES 2.0 Storyboard runtime controls polygon anti-aliasing.

Fill ("color") The color used to fill the polygon.

Outline ("outlinecolor") The color used to outline (stroke) a filled polygon i.e. if the style is Fill & Stroke

Rectangle

The rectangle render extension draws a single pixel outline rectangle to the screen.

The rectangle render extension uses many standard options but also includes:

Color Color to use for item (number RGB format, optional)

Text

The text render extension draws a string to the screen. Strings can be wrapped on word boundaries and also rotated orthogonally. The following shows the effects of rotation on strings.

The text render extension uses many standard options but also includes:

Text The text string to display (string, optional)

Font The font to use (string, optional)

Size The point size of the string (number, optional)

Underline	Specifies if the string show display an underline (1) or not display an underline (0 default).
Overflow	Defines behaviour when text overflows available horizontal space. Options are do nothing ("none" default), wrap at word boundaries ("wrap"), wrap at character boundaries ("char") or end text with an ellipsis ("ellipsis").

3D Model

The 3D model render extension renders a 3D model into the control. Currently models in Wavefront Object (.obj) format are supported. This extension is only available when using OpenGL or OpenGL ES 2.0 based render managers, and requires the model3d plugin to be loaded.

The coordinate system in the render extension is the default OpenGL default coordinate system, with positive x to the right, positive y up, and positive z towards the viewer. The camera position defaults to (0, 0, 0), with the view direction along the negative z axis.

A Phong reflection model is implemented. A directional light source is present with white light coming from the (0, 1, 1) direction. The Phong model makes use of three terms:

- **Ambient** - The color of the material in the absence of direct light. The material will never appear darker than the ambient color.
- **Diffuse** - The color of light reflected from the material.
- **Specular** - The color of the highlights from the material. The specular exponent controls how large the highlight is.

For more details on the Phong reflection model refer to Phong Reflection Model [http://en.wikipedia.org/wiki/Phong_reflection_model] or to any book on computer graphics.

Rotations for the model are defined using Euler angles, with rotations applied around the z (psi), y (theta) and then x (phi) axes.

An OBJ file defines vertices and faces, and optionally normals and texture coordinates. If normals are not present, they will be calculated according to the convention that vertices in a face are specified in counter-clockwise order. If texture coordinates are not present, the model will not be rendered using a texture. Faces may be grouped together, and each group may be rendered with a different material.

Each OBJ file may also specify a Material (.mtl) file which allows for the material properties of the model to be specified. The following properties in a material file are currently supported:

- **d** - The transparency (alpha) of the material.
- **Ka** - The ambient lighting component of the material.
- **Kd** - The diffuse lighting component of the material.
- **Ks** - The specular lighting component of the material.
- **Ns** - The specular lighting exponent of the material.
- **map_Kd** - The texture specifying the diffuse color of the material. If the texture can be loaded, it will be used rather than the Kd parameter to when calculating the diffuse color.

If a material file is not present, the object will be rendered with a white color.

The 3D model render extension uses many standard options but also includes:

Filename	The name of the model to load.
Camera Position X/Y/Z	The X/Y/Z position of the camera.
Azimuth	The rotation of the camera around the y axis in degrees.
Elevation	The rotation of the camera around the x axis in degrees.
Field Of View	The field of view the camera in degrees. The field of view specifies how much of visual sphere is mapped to the control. A larger field of view is equivalent to using a wide-angle lens on a camera, and a smaller field of view is equivalent to using a zoom lens.
Model X/Y/Z	The X/Y/Z position of the model.
Orientation Phi	The rotation of the model around the x axis in degrees.
Orientation Theta	The rotation of the model around the y axis in degrees.
Orientation PSI	The rotation of the model around the z axis in degrees.

Appendix G. Storyboard Engine Public API

Users that are writing their own plugin or users who are working with a Storyboard RTExec or Storyboard Lite platform should familiarize themselves with this section as it talks about how the Storyboard engine is initialized. If you are running on a system such as Linux or QNX, then the information in this section is not needed.

The Storyboard Engine is initialized in two steps:

1. The first step is to create the engine's application handle via the `gr_application_create_args` function. This function initializes the core framework and plugins and loads the UI model and verifies its integrity.
2. Once the UI model is loaded then the main UI event loop can be started using the `gr_application_run` function along with the application handle returned from `gr_application_create_args`. The `gr_application_run` function will not return until the application terminates.

The Integration API is thread safe so it is possible to terminate the application, by calling `gr_application_quit`, in another thread. In general the `gr_application_create` and `gr_application_run` functions should be called from the same thread context since the initialization of some of the engine's sub-systems may create thread local data required for execution.

When the application terminates, and returns from `gr_application_run`, then the application handle should be released by calling `gr_application_free`. Once an application handle has been released it is no longer valid to be used.

The Engine provides execution tracing functionality through the `gr_application_debug` interface. This function takes an optional application handle and an integer command value. Additional arguments may be required based on the type of command and are documented within the `<gre/gre.h>` header. This function is most often used to provide additional diagnostics. For example invoking:

```
gr_application_debug(NULL, GR_DEBUG_CMD_VERBOSITY, GR_LOG_TRACE2);
```

will enable logging of all significant operations within the engine.

`gr_application_create_args`

```
#include <gre/gre.h>
gr_application_t * gr_application_create_args(
    const char *bundle,
    int flags,
    char * const *options,
    int option_count
);
```

The `gr_application_create_args` function creates the application handle for the Storyboard Engine. The application handle can be used to further configure the engine or to start its execution.

Parameters:

bundle A pointer to the model data. This may be a filename or a string based on the `flags` parameter.
flags One or more `GR_APP_*` flags that control how the application handle is created
options A pointer to an array of option/value pairs used to initialize the engine or `NULL` if no options are provided.
option_count The number of options in the options array, must be even (0, 2, 4, ...)

Returns:

An application handle on success or `NULL` on failure. On platforms where `errno` is available, it will contain failure details.

The `gr_application_create_args` function will load a model, initialize the engine's plugin subsystem and then return an application handle. The model can be loaded from a file if the flags are `GR_APP_LOAD_FILE` or the model can be loaded from an in-memory block if the flags are `GR_APP_LOAD_STRING`. Once the model is loaded and plugins are initialized control returns to the caller.

gr_application_create

```
#include <gre/gre.h>
gr_application_t * gr_application_create(
    const char *bundle
);
```

The `gr_application_create` function creates the application handle for the Storyboard Engine. The application handle can be used to further configure the engine or to start its execution.

Parameters:

bundle A pointer to the model data. This must be a filename.

Returns:

An application handle on success or `NULL` on failure. On platforms where `errno` is available, it will contain failure details.

The `gr_application_create` will load a model, initialize the engine's plugin subsystem and then return an application handle. Once the model is loaded and plugins initialized control returns to the caller.

gr_application_free

```
#include <gre/gre.h>
gr_application_t * gr_application_free(
    gr_application_t * handle
);
```

The `gr_application_free` Destroy and free an application handle. This will stop the execution of the application and free any resources that are associated with it.

Parameters:

`handle` A pointer to an application handle created using `gr_application_create_args`.

Returns:

A zero on success and a non-zero status code on failure.

The `gr_application_free` will destroy a created application handle and stop execution of the application. Any resources associated with the handle will be freed.

gr_application_run

```
#include <gre/gre.h>
int gr_application_run(
    gr_application_t * handle
);
```

The `gr_application_run` function creates the application handle for the Storyboard Engine. The application handle can be used to further configure the engine or to start its execution.

Parameters:

`handle` A pointer to an application handle created using `gr_application_create_args`.

Returns:

A zero on success and a non-zero status code on failure.

The `gr_application_run` will start the execution of the Storyboard Engine's main event loop. This call will only return when the engine has terminated.

gr_application_quit

```
#include <gre/gre.h>
int gr_application_quit(
    gr_application_t *handle
);
```

The `gr_application_quit` function provides a method of asynchronously terminating an executing application handle that has been started by `gr_application_run` from a separate thread.

Parameters:

handle A pointer to an application handle created using `gr_application_create_args`.

Returns:

Various values based on the command as listed in the header file `<gre/gre.h>`

The `gr_application_quit` is a method with which to terminate a handle started by `gr_application_run`. It is an asynchronous call, and will not wait for `gr_application_run` to complete.

gr_application_debug

```
#include <gre/gre.h>
int gr_application_debug(
    gr_application_t *handle,
    unsigned command,
    ...
);
```

The `gr_application_debug` function provides an interface to some of the debugging and introspection facilities of the Storyboard Engine.

Parameters:

handle A pointer to an application handle created using `gr_application_create_args`. Not all commands will require an application handle.
command A `GR_DEBUG_*` command code.

Returns:

Various values based on the command as listed in the header file `<gre/gre.h>`

The `gr_application_debug` is most frequently used to control the verbosity level of the application using the `GR_DEBUG_CMD_VERBOSITY` command and it can also be used to dump out the system configuration details when used with the `GR_DEBUG_CMD_SYSTEM_INFO`.

gr_app_log

```
#include <gre/gre.h>
void gr_app_log(
    gr_application_t *app,
    int level,
    const char *format,
    ...
);
```

The `gr_app_log` function provides an interface to integrate logging into user created threads and plugins.

Parameters:

`app` A pointer to an application handle created using `gr_application_create_args`.
`level` A value indicating the severity of the message, one of `GR_LOG_*`
`format` A 'printf' style format string

Returns:

Nothing

The `gr_app_log` is a method of integrating custom debug logging that obeys the the `GR_DEBUG_CMD_VERBOSITY` command and verbosity level.

gr_application_set_data

```
#include <gre/gre.h>
int gr_application_set_data(
    gr_application_t *app,
    const char      *key,
    gr_data_format_t fmt,
    gr_data_union_t *data
);
```

The `gr_application_set_data` function provides an interface to set a Storyboard data variable. Copies the data into the corresponding data manager key, creating the key if needed.

Parameters:

`app` A pointer to an application handle created using `gr_application_create_args`.
`key` The data key variable to set
`fmt` The format describing the data argument: `GR_DATA_FORMAT_(1s0|1u1|1s1|2u1|...)`
1s0 (string) | 4u1 (uint32_t) | 4s1 (int32_t) ...
`data` A pointer to the data to set, this is not a transfer of memory ownership.

Returns:

Zero on success, -1 on failure.

The `gr_application_set_data` is a method of setting, creating, and changing data variables which are useable by the Storyboard engine.

gr_application_set_data_variable

```
#include <gre/gre.h>
int gr_application_set_data_variable(
```

```
gr_application_t *app,  
const char      *fqn,  
const char      *variable,  
gr_data_format_t fmt,  
gr_data_union_t *data  
);
```

The `gr_application_set_data_variable` function provides an interface to set a Storyboard data variable. Combines the `fqn` and `variable` to create a key in the form of "`fqn.variable`". Copies the data into the corresponding data manager key, creating the key if needed.

Parameters:

`app` A pointer to an application handle created using `gr_application_create_args`.
`fqn` The fully qualified name of the object, should be NULL for app level variables
`variable` The name of the variable to set
`fmt` The format describing the data argument: `GR_DATA_FORMAT_(1s0|1u1|1s1|2u1|...)`
1s0 (string) | 4u1 (uint32_t) | 4s1 (int32_t) ...
`data` A pointer to the data to set, this is not a transfer of memory ownership.

Returns:

Zero on success, -1 on failure.

The `gr_application_set_data` is a method of setting, creating, and changing data variables which are useable by the Storyboard engine.

gr_application_get_data

```
#include <gre/gre.h>  
int gr_application_get_data(  
    gr_application_t *app,  
    const char      *key,  
    gr_data_format_t *fmt,  
    gr_wrapped_data_t *data  
);
```

The `gr_application_get_data` function provides an interface to get Storyboard data variables associated with a key, and their formats.

Parameters:

`app` A pointer to an application handle created using `gr_application_create_args`.
`key` The data key variable to get
`fmt` The format describing the data argument: `GR_DATA_FORMAT_(1s0|1u1|1s1|2u1|...)`
1s0 (string) | 4u1 (uint32_t) | 4s1 (int32_t) ...
Can be `GR_DATA_FORMAT_UNKNOWN` and the original format will be used.
`data` Location to store the returned data, this is a copy of the requested data and its format

Returns:

Zero on success, -1 on failure.

The `gr_application_get_data` function is a method of retrieving the data values and formats for any keys associated Storyboard engine variables.

gr_application_get_data_variable

```
#include <gre/gre.h>
int gr_application_get_data(
    gr_application_t *app,
    const char      *fqn,
    const char      *variable,
    gr_data_format_t fmt,
    gr_wrapped_data_t *data
);
```

The `gr_application_get_data_variable` function provides an interface to get Storyboard data variables associated with a key, and their formats. Combines the `fqn` and `variable` to create a key in the form of "`fqn.variable`".

Parameters:

`app` A pointer to an application handle created using `gr_application_create_args`.
`fqn` The fully qualified name name of the object, should be NULL for app level variables
`variable` The name of the variable to get
`fmt` The format describing the data argument: `GR_DATA_FORMAT_(1s0|1u1|1s1|2u1|...)`
1s0 (string) | 4u1 (uint32_t) | 4s1 (int32_t) ...
Can be `GR_DATA_FORMAT_UNKNOWN` and the original format will be used.
`data` Location to store the returned data, this is a copy of the requested data and it's format

Returns:

Zero on success, -1 on failure.

The `gr_application_get_data` function is a method of retrieving the data values and formats for any keys associated Storyboard engine variables.

gr_application_add_event_listener

```
#include <gre/gre.h>
gr_application_event_listener_t gr_application_add_event_listener(
    gr_application_t *app,
    const char *event,
    gr_event_listener_t handler,
    void *data
```

```
);
```

The `gr_application_add_event_listener` function provides an interface associate a call-back/handler to associate with a Storyboard event.

Parameters:

- `app` A pointer to an application handle created using `gr_application_create_args`.
- `event` The event to add a listener for
- `handler` The handler to call for this event
- `data` User defined data which is passed to the handler

Returns:

A pointer to the created listener.

The `gr_application_add_event_listener` function is the method used to associate a call-back/handler to execute with a Storyboard engine event. When the event occurs the handler will execute in the main Storyboard execution thread

gr_application_rem_event_listener

```
#include <gre/gre.h>
int gr_application_rem_event_listener(
    gr_application_t *app,
    gr_application_event_listener_t *listener
);
```

The `gr_application_rem_event_listener` function provides an interface to free a user created `gr_application_event_listener_t`

Parameters:

- `app` A pointer to an application handle created using `gr_application_create_args`.
- `listener` A pointer to a `gr_application_event_listener_t` created by `gr_application_add_event_listener`

Returns:

Zero on success, -1 on failure.

The `gr_application_rem_event_listener` is the method used to free a `gr_application_event_listener_t` created by `gr_application_add_event_listener`. Any handler associated with this listener will no longer execute

gr_application_send_event

```
#include <gre/gre.h>
int gr_application_send_event(
    gr_application_t *app,
    const char *event_target,
    const char *event_name,
    const char *event_format,
    const void *event_data,
    int event_nbytes
);
```

The `gr_application_send_event` function provides an interface to inject an event into the application event queue.

Parameters:

- `app` A pointer to an application handle created using `gr_application_create_args`.
- `event_target` The name of the event target model element (screen, layer, control, render extension), or NULL to send to the default target (application level)
- `event_name` The name of the event to send, must not be NULL
- `event_format` The format of the data, or NULL if no data is transmitted
- `event_data` A pointer to the data to transmit, or NULL if no data is transmitted
- `event_bytes` The number of data bytes to transmit, or NULL if no data is transmitted

Returns:

Zero on success, -1 on failure.

The `gr_application_send_event` is the method used to inject an event into the application event queue. The event will be processed after all events previously in the queue have been processed.

gr_context_get_application

```
#include <gre/gre.h>
gr_application_t *gr_context_get_application(
    gr_action_context_t *app
);
```

The `gr_context_get_application` function is used to obtain the application handle from an action context.

Parameters:

- `action_context` The action context to obtain the application handle from

Returns:

The application handle that is tied to the action context

The `gr_context_get_application` is used to obtain the application handle from an action context

gr_context_max_fqn

```
#include <gre/gre.h>
int gr_context_max_fqn(
    gr_action_context_t *action_context
);
```

The `gr_context_max_fqn` function estimates the maximum length a fully qualified name contained in an action context.

Parameters:

`action_context` The action context to estimate the fully qualified name for

Returns:

The estimated length for the maximum fully qualified name

The `gr_context_max_fqn` is used to estimate the maximum storage space needed to the longest fully qualified name for the action context

gr_context_get_row

```
#include <gre/gre.h>
int gr_context_get_row(
    gr_action_context_t *action_context
);
```

The `gr_context_get_row` returns the row number of a table cell, if the action's context is a table cell.

Parameters:

`action_context` The action context to get the row information from

Returns:

The row number if control context for the action context is a table template, -1 otherwise

The `gr_context_get_row` is used to get the row number of the table cell from the action context, if the control context of the action context is a table cell. If the control context of the action context is not a table cell, this function returns -1.

gr_context_get_column

```
#include <gre/gre.h>
int gr_context_get_column(
    gr_action_context_t *action_context
);
```

```
);
```

The `gr_context_get_column` returns the column number of a table cell, if the action's context is a table cell.

Parameters:

`action_context` The action context to get the row information from

Returns:

The column number if control context for the action context is a table template, -1 otherwise

The `gr_context_get_column` is used to get the column number of the table cell from the action context, if the control context of the action context is a table cell. If the control context of the action context is not a table cell, this function returns -1.

gr_context_get_control

```
#include <gre/gre.h>
int gr_context_get_control(
    gr_action_context_t *action_context,
    char *buffer,
    int buffer_len
);
```

The `gr_context_get_control` function stores the fully qualified name of the control in the buffer provided up to `buffer_len` characters.

Parameters:

`action_context` The action context to get the fully qualified name from

`buffer` The character buffer to store the controls fully qualified name in, the first character may be set to null, if there is no control in the action context

`buffer_len` The storage space that the buffer has to store the name, see function `gr_context_max_fqn`

Returns:

The length of the control name if there is a control in the context, -1 otherwise

The `gr_context_get_control` function will build the fully qualified name for the control context based on the action context and store it in the buffer that is provided, up to `buffer_len` bytes. It will return the length of the fully qualified name. If there is no control context in the action context, the function will return -1 and set the first character in buffer to NULL.

gr_context_get_group


```
#include <gre/gre.h>
int gr_context_get_group(
    gr_action_context_t *action_context,
    char *buffer,
    int buffer_len
);
```

The `gr_context_get_group` function stores the fully qualified name of the group in the buffer provided up to `buffer_len` characters.

Parameters:

`action_context` The action context to get the fully qualified name from
`buffer` The character buffer to store the groups fully qualified name in, the first character may be set to null, if there is no group in the action context
`buffer_len` The storage space that the buffer has to store the name, see function `gr_context_max_fqn`

Returns:

The length of the group name if there is a group in the context, -1 otherwise

The `gr_context_get_group` function will build the fully qualified name for the group context based on the action context and store it in the buffer that is provided, up to `buffer_len` bytes. It will return the length of the fully qualified name. If there is no group context in the action context, the function will return -1 and set the first character in buffer to NULL.

gr_context_get_layer

```
#include <gre/gre.h>
int gr_context_get_layer(
    gr_action_context_t *action_context,
    char *buffer,
    int buffer_len
);
```

The `gr_context_get_layer` function stores the fully qualified name of the layer in the buffer provided up to `buffer_len` characters.

Parameters:

`action_context` The action context to get the fully qualified name from
`buffer` The character buffer to store the layers fully qualified name in, the first character may be set to null, if there is no layer in the action context
`buffer_len` The storage space that the buffer has to store the name, see function `gr_context_max_fqn`

Returns:

The length of the layer name if there is a layer in the context, -1 otherwise

The `gr_context_get_layer` function will build the fully qualified name for the layer context based on the action context and store it in the buffer that is provided, up to `buffer_len` bytes. It will return the length of the fully qualified name. If there is no layer context in the action context, the function will return -1 and set the first character in buffer to NULL.

gr_context_get_screen

```
#include <gre/gre.h>
int gr_context_get_screen(
    gr_action_context_t *action_context,
    char *buffer,
    int buffer_len
);
```

The `gr_context_get_screen` function stores the fully qualified name of the screen in the buffer provided up to `buffer_len` characters.

Parameters:

`action_context` The action context to get the fully qualified name from
`buffer` The character buffer to store the screens fully qualified name in, the first character may be set to null, if there is no screen in the action context
`buffer_len` The storage space that the buffer has to store the name, see function `gr_context_max_fqn`

Returns:

The length of the screen name if there is a screen in the context, -1 otherwise

The `gr_context_get_screen` function will build the fully qualified name for the screen context based on the action context and store it in the buffer that is provided, up to `buffer_len` bytes. It will return the length of the fully qualified name. If there is no screen context in the action context, the function will return -1 and set the first character in buffer to NULL.

gr_context_get_fqn

```
#include <gre/gre.h>
int gr_context_get_fqn(
    gr_action_context_t *action_context,
    char *buffer,
    int buffer_len
);
```

The `gr_context_get_fqn` function stores the fully qualified name of the target model object in the buffer provided up to `buffer_len` characters.

Parameters:

action_context The action context to get the fully qualified name from
buffer The character buffer to store the fully qualified name in, the first character may be set to null, if there is no target in the action context
buffer_len The storage space that the buffer has to store the name, see function `gr_context_max_fqn`

Returns:

The length of the fully qualified name if there is a target model object in the context, -1 otherwise

The `gr_context_get_fqn` function will build the fully qualified name for the target model object based on the action context and store it in the buffer that is provided, up to `buffer_len` bytes. It will return the length of the fully qualified name. If there is no target model object in the action context, the function will return -1 and set the first character in buffer to NULL.

gr_context_get_event_name

```
#include <gre/gre.h>
char *gr_context_get_event_name(
    gr_action_context_t *action_context
);
```

The `gr_context_get_event_name` function returns the name of the event that the action is associated with.

Parameters:

action_context The action context to get the fully qualified name from

Returns:

The name of the event if there is an event name in the context, null otherwise

The `gr_context_get_event_name` function will return the name of the event that occurred that caused the action to be invoked.

gr_context_get_event_data

```
#include <gre/gre.h>
void *gr_context_get_event_data(
    gr_action_context_t *action_context,
    int *nbytes
);
```

The `gr_context_get_event_data` function returns the data payload of the event that the action is associated with.

Parameters:

action_context The action context to get the fully qualified name from
nbytes A point to an integer to store the size of the data in

Returns:

The data payload of the event if there is an event name in the context, null otherwise

The `gr_context_get_event_data` function will return the data payload of the event that occurred that caused the action to be invoked. It will store the size of the data payload in the `nbytes` parameter that is passed in.

Part I. Storyboard Tutorials

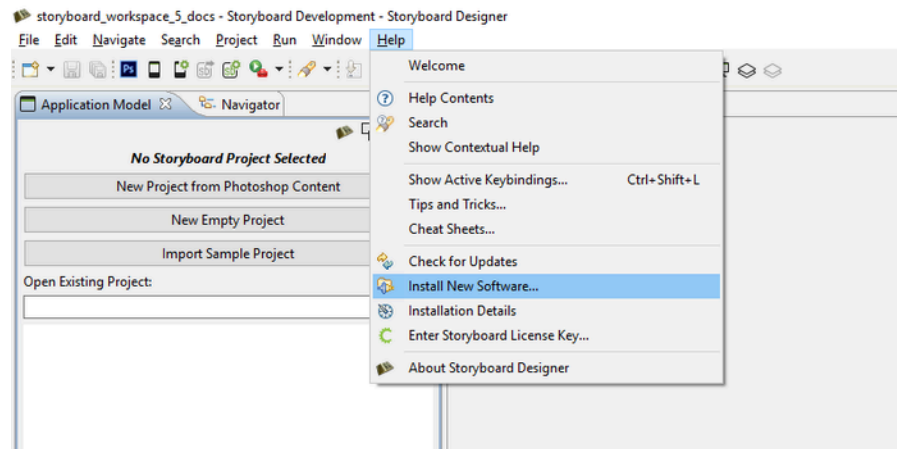
Table of Contents

33. Importing Sample Projects from Crank's Public SVN	381
34. Working with Multiple Application Design Files	386
Creating a Project	386
Resolving Conflicts	388
35. Creating a 3D Model Application	389
New Project	389
3D Model Control	389
Resize Model	391

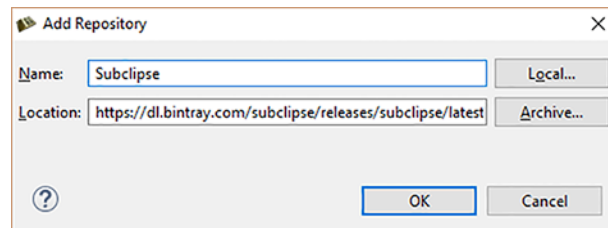
Chapter 33. Importing Sample Projects from Crank's Public SVN

Installing Subclipse and connecting to the Crank Software public repository

Before we are able to start using the demos from the public Crank code repository we will first need to install a SVN client. To do so in Storyboard Designer we go to Help > Install New Software.

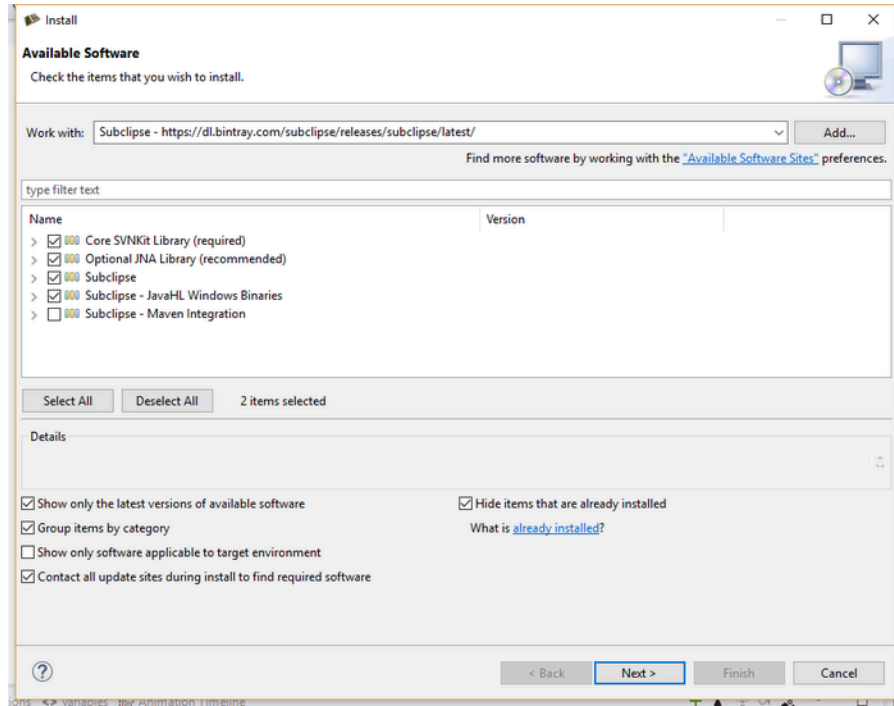


Here at Crank we use Subclipse. Click on Add and for Name you can enter anything you please. To keep it simple we will use Subclipse. For Location enter <https://dl.bintray.com/subclipse/releases/subclipse/latest/> Click OK.

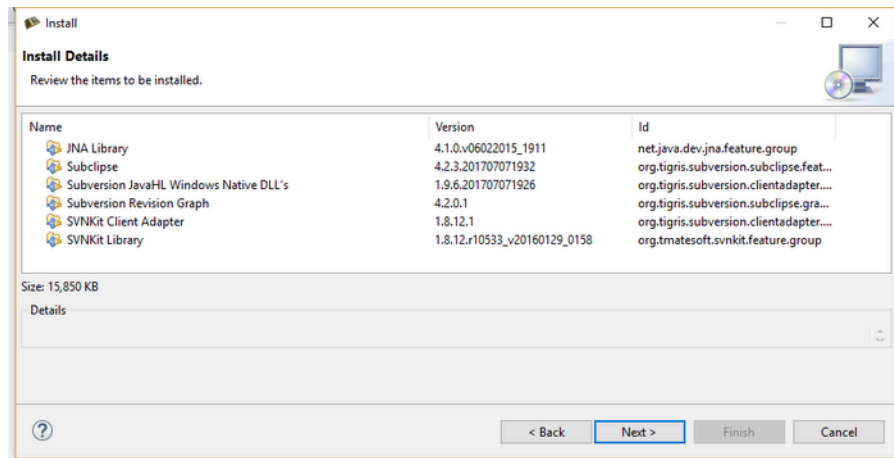


After you click OK you will be presented with a list of software to install. There is no need for Maven so you can uncheck it. Click Next.

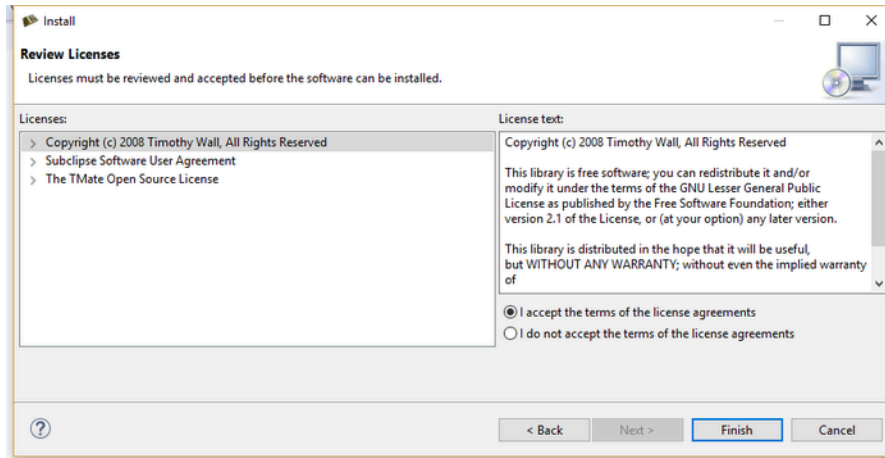
Importing Sample Projects from Crank's Public SVN



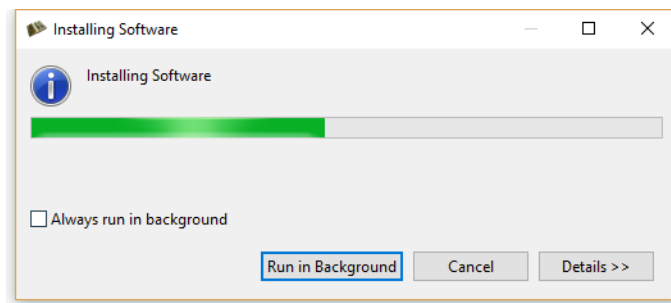
Details of the software to be installed. Click Next.



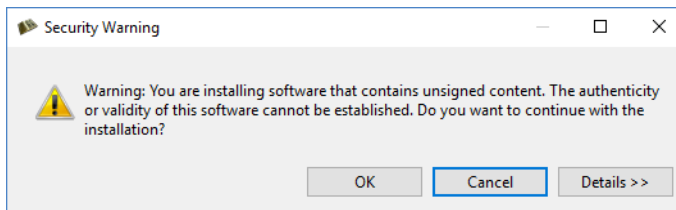
Accept license agreements. Click Finish.



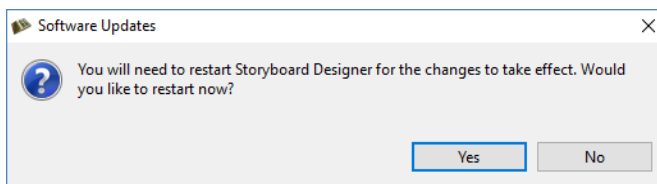
Software installation progress.



Generic warning that the software being installed is not signed. Click OK.

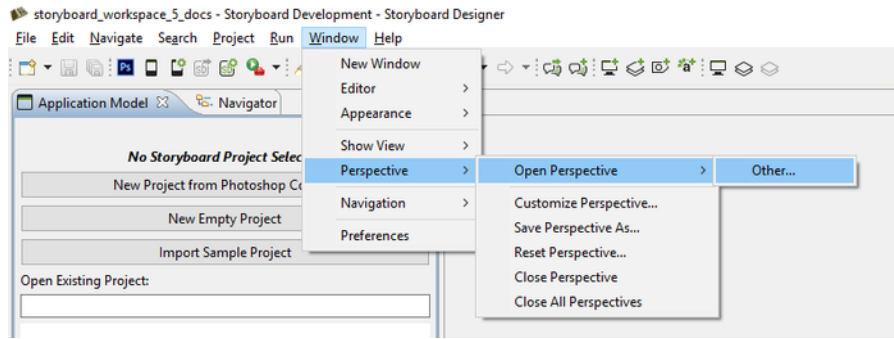


After the Subclipse SVN client is installed you will need to restart Storyboard Designer. Click Restart Now.

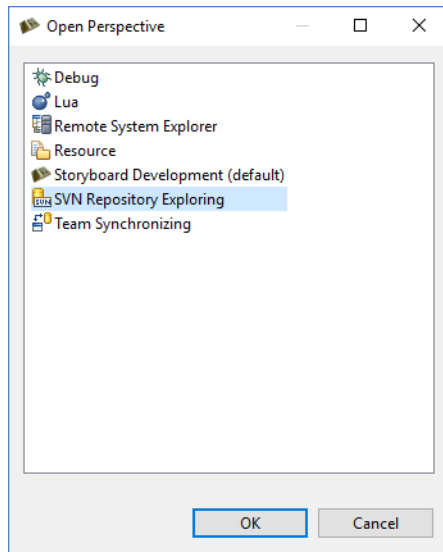


Now that Subclipse is installed we need to go to that perspective to add a repository.

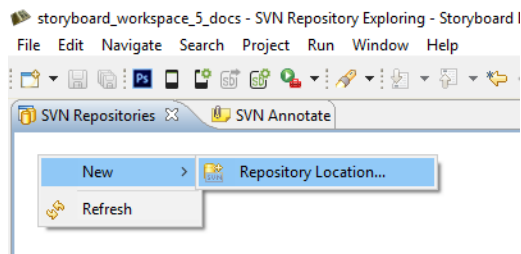
Importing Sample Projects from Crank's Public SVN



Select SVN Repository Exploring. Click OK.



To add a repository we simply right click in the SVN perspective window and select New > Repository Location.

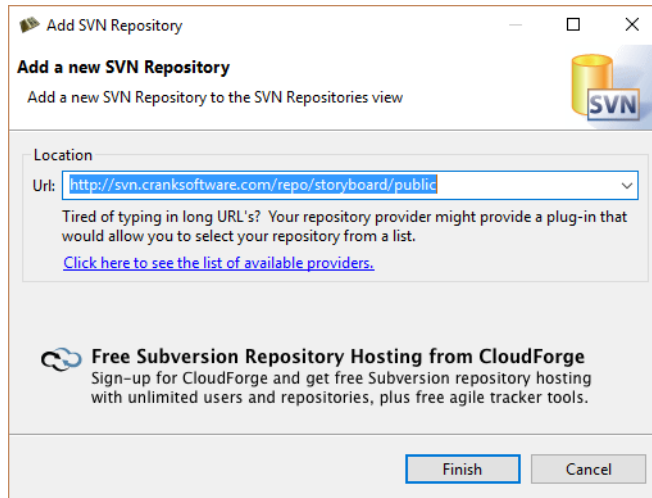


Enter the Crank public code repository URL <http://svn.cranksoftware.com/repo/storyboard/public> Click Finish. When you are prompted for login credentials use:

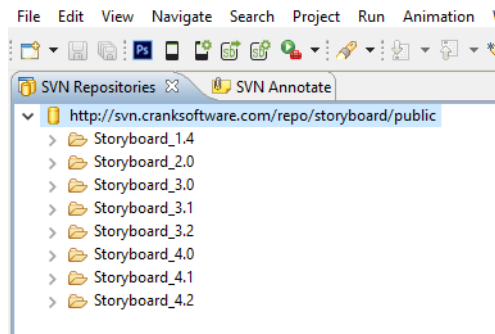
Username **storyboard**

Password **crankrocks**

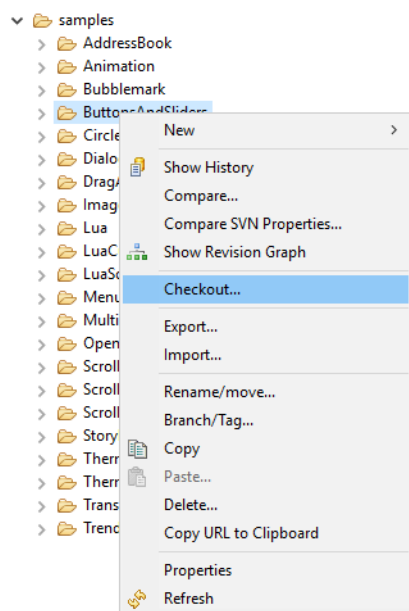
Importing Sample Projects from Crank's Public SVN



You are now connected to the Crank Public Repository. By expanding the directories you can see the different demos available for checkout.



To checkout from the repository you right click on the demo and select Checkout. Once completed click on the Storyboard Development tab to see the application in your workspace.



Chapter 34. Working with Multiple Application Design Files

Storyboard's collaborative features help multiple users develop applications faster. You can merge multiple files (*.gde) together and produce a single output during runtime. This tutorial explains how to create a project with multiple application files.

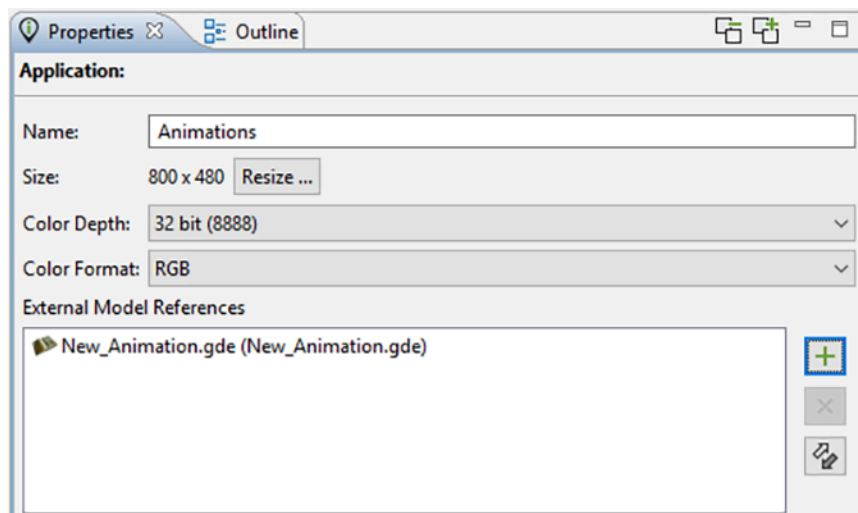
Creating a Project

A project with multiple application files starts the same as a single application file project. Select File > New > Storyboard Application or create a new project using the Photoshop Import feature.

A project can accommodate multiple stand-alone applications that share project images and script resources. You can add an additional application file to an existing project in multiple ways:

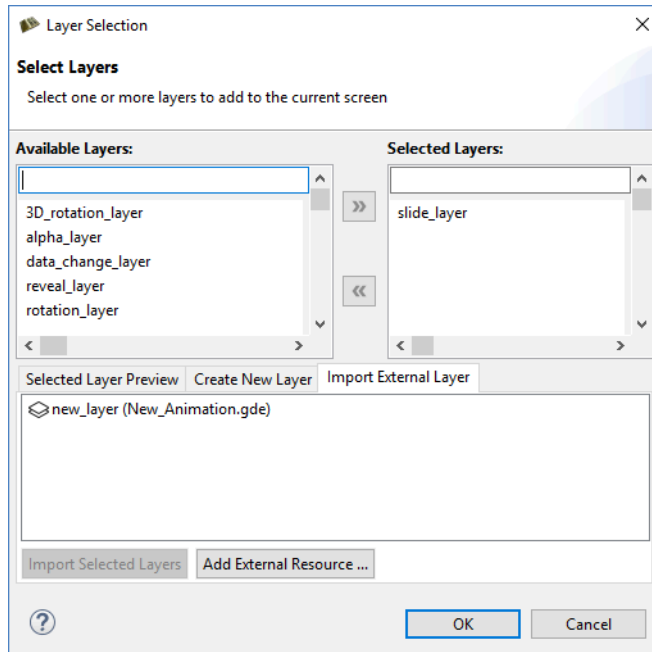
- Create a new file within an existing project by selecting File > New > Storyboard Application and choose to create a New Model in Existing Project
- Create a new file using the Photoshop Import feature and select the option to import .psd Into an Existing Project as a New File.
- In the navigator view select and copy an existing application, then paste and rename the copied application file.

For a multiple application project to function as a whole, application files need to reference one another. Select the application from the Application Model View and in the Properties View, select Add external model resource button. In the next dialog select the .gde file that was created in Step 2 and press OK.

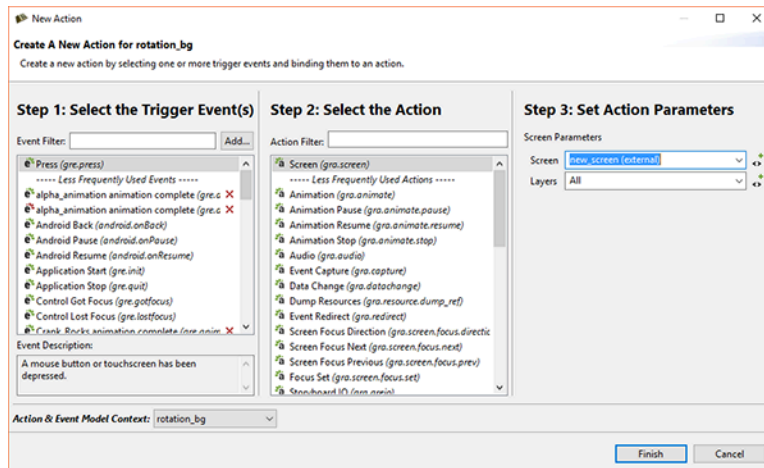


To add layers from an external model to a local application:

- Select the Import External Layer tab and then select Import Selected Layers.
- Choose the layer(s) to add to the current screen. After adding an external layer, Storyboard will recognize the external content and incorporate it to function like any other layer.

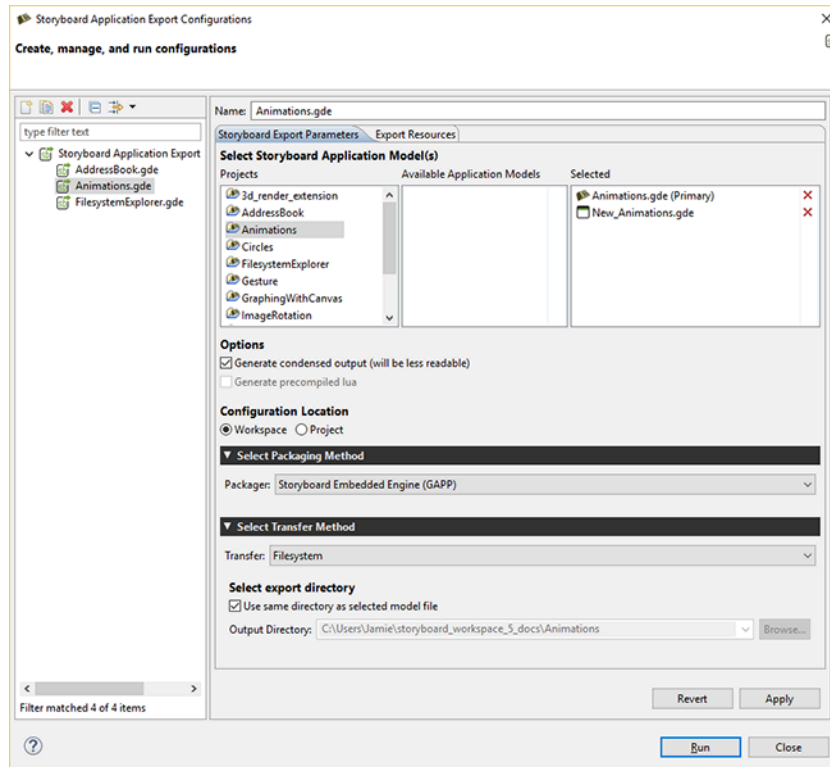


To create a transition from a source application screen to an external application screen, add actions to an application that perform a screen change.



Before launching a multi-file application with the simulator, the external .gde files have to be referenced in the Simulator Configurations dialog.

To manage the runtime configurations, select Run > Storyboard Simulator Configurations. A list of available models that can be included in the runtime export used with Storyboard Engine is in the selected project folder. To apply changes, select Apply and then Run.



If no conflicts occur within the selected applications, they are merged and converted into a single unified application at runtime. If conflicts exist, they must be resolved before the application can merge.

Resolving Conflicts

The application properties page provides an action to synchronize source content with referenced external content. Any differences are flagged as a conflict and the user is prompted to resolve the conflict based on the issue. Conflict types include:

Layers: If two or more layers have the same name their content needs to be identical.

Variables: Any application/global variables with the same name must have values that are the same.

Animations: If two or more animations use the same name then the animation needs to be identical.

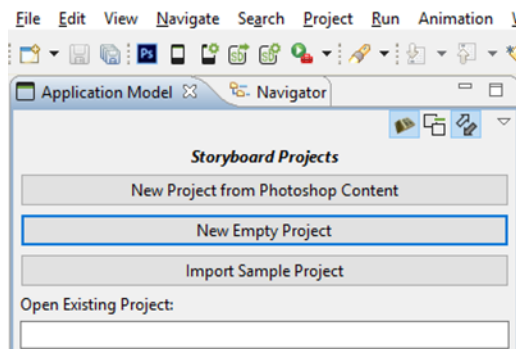
Screens: Screens from all applications are compared. Two or more screens with the same name prompt the user to resolve differences between the two.

Chapter 35. Creating a 3D Model Application

This quick tutorial will show how to use the 3D Model Control with a new project

New Project

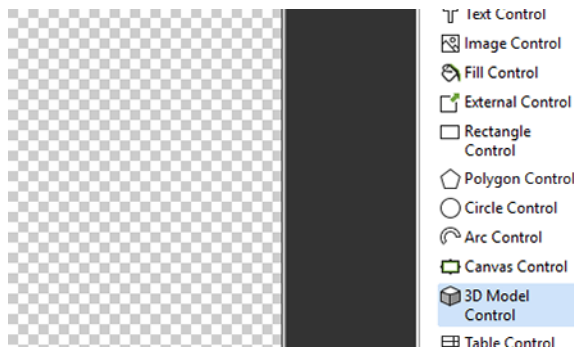
Create a new project by either File > New > Storyboard Application or selecting New Empty Project from the Application View.



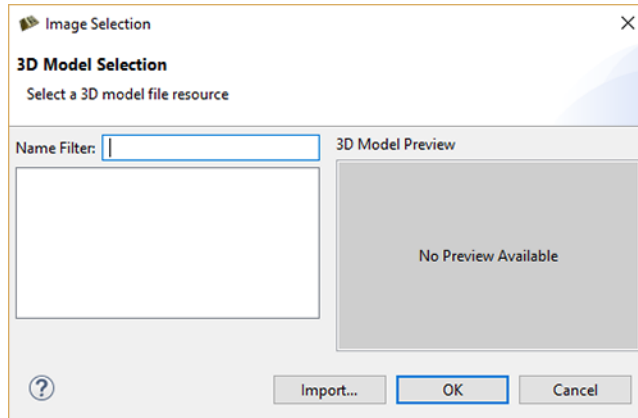
Give your Project a name and click Finish.

3D Model Control

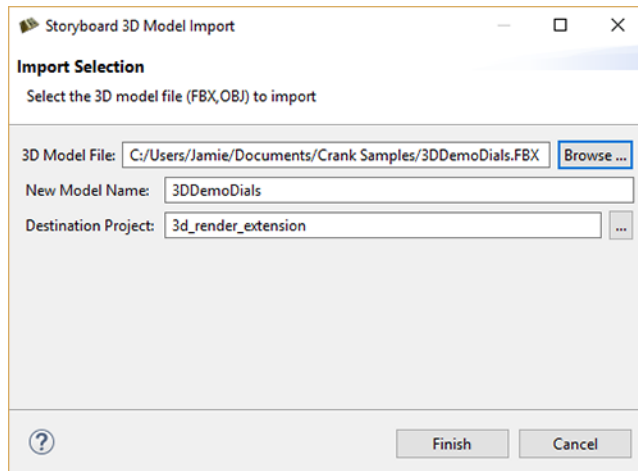
Once the empty project loads add a 3D Model Control to the screen.



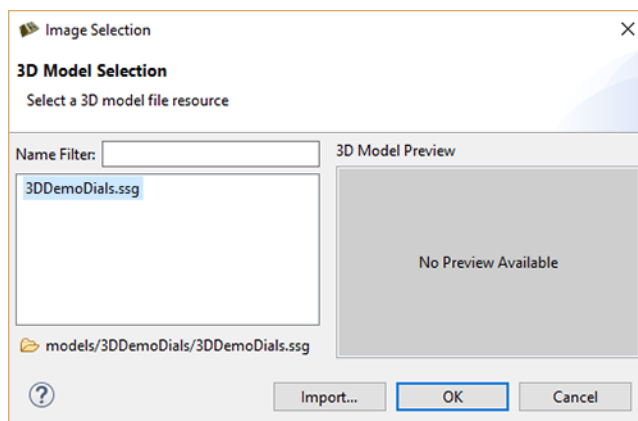
The 3D Model Selection dialog will pop up enabling you can navigate to the 3D model you want to load.



Click on the Import button to select your FBX or OBJ model.

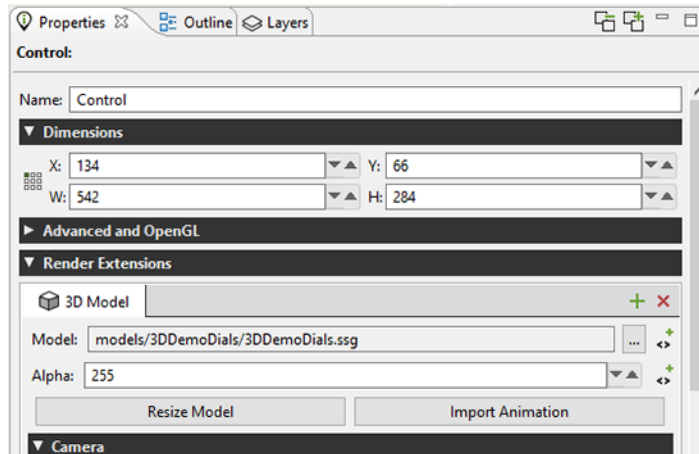


The 3D model is then converted to our .ssg format and placed in a newly created models directory at the root of your project. You can now choose the model to be used in your project.



Resize Model

Once the model loads it might not be immediately visible. In that case you can make use of the Resize Model option in the properties view of the 3D model Control. This option modifies some of the camera coordinates to make the model visible.



The 3D model should now be visible after using the Resize Model option.



Further manipulation of the 3D model properties may be needed in order to place the model in the desired position.

Part II. Storyboard Demo Images

Table of Contents

36. NXP	395
i.MX 6QuadPlus	395
Demo Details	395
i.MX 6UltraLite	395
Demo Details	395
i.MX 6ULL	396
Demo Details	396
i.MX 6DualLite	396
Demo Details	396
i.MX 6SoloX	396
Demo Details	396
Toradex i.MX 7Dual SoM	397
Demo Details	397
Flashing the Image	397
U-Boot	398
Copying Image to an SD Card - Linux	398
Copying Image to an SD Card - Windows	398
Running the Image	398
Turn On or Off the Console	398
37. STMicroelectronics	400
STM32F429	400
Demo Details	400
STM32F439	400
Demo Details	400
STM32 F7	400
Demo Details	400
Required Hardware and Software	401
Package Contents	401
STM32 ST-LINK Utility	402
Running the Demo	404
Importing the Demo into Storyboard Designer	405
Creating Your Own Demo	405
38. Microchip	406
SAMA5D2	406
Demo Details	406
SAMA5D2 with 7 inch Display	406
Demo Details	406
Copying Image to an SD Card - Linux	407
Creating an Image - Windows	407
Running the Image	407
39. Renesas	408
RZ/A1	408
Demo Details	408
Flashing the Image	408
40. Linux	409
TI AM355 Starter Kit	409
Step 1: Importing A Storyboard Sample	409
Step 2: Exporting A Storyboard Application	411
Step 3: Selecting The Storyboard Embedded Engine	413
Step 4: Configuring The Target Platform	413
Step 5: Running The Storyboard Application	413

Raspberry Pi 414

Chapter 36. NXP

Experience multi-market demo applications built with Storyboard on your favorite NXP platform. Crank Software demo images showcase Storyboard's design and development capabilities and the application potential of popular platforms. Download [http://www.cranksoftware.com/demo_image/nxp] the demo image that corresponds with your hardware, which includes everything you need to get and up and running in minutes.

i.MX 6QuadPlus

Demo Details

Image Name: CrankSoftware_demo_imx6qpsabresd.sdcard

Board: i.MX 6QuadPlus SABRE

Display: HDMI with Capacitive Touch

Rendering Technology: OpenGL ES 2.0

Demo Resolution: 1280 x 720

Operating System: Linux

This demo image includes the following demos:

- IoT - Smart Home
- Movie Kiosk
- Automotive 3D Instrument Cluster

i.MX 6UltraLite

Demo Details

Image Name: CrankSoftware_demo_imx6ulevk.sdcard

Board: i.MX 6UltraLite Evaluation Kit

Display: 4.3" LCD8000-43T

Rendering Technology: fbdev

Demo Resolution: 480 x 272

Operating System: Linux

This demo image includes the following demos:

- Medical
- White Goods - Washing Machine

- Home Automation

i.MX 6ULL

Demo Details

Image Name: CrankSoftware_demo_imx6ullevk.sdcard

Board: i.MX 6ULL Evaluation Kit

Display: 4.3" LCD8000-43T

Rendering Technology: fbdev

Demo Resolution: 480 x 272

Operating System: Linux

This demo image includes the following demos:

- Medical
- White Goods - Washing Machine
- Home Automation

i.MX 6DualLite

Demo Details

Image Name: CrankSoftware_demo_imx6dlsabresd.sdcard

Board: i.MX 6 SABRE Platform for Smart Devices

Display: LVDS (on board)

Rendering Technology: OpenGL ES 2.0

Demo Resolution: 1024 x 768

Operating System: Linux

This demo image includes the following demos:

- IoT - Smart Home
- Movie Kiosk
- Medical

i.MX 6SoloX

Demo Details

Image Name: CrankSoftware_demo_imx6xsabresd.sdcard

Board: i.MX 6SoloX SABRE

Display: LVDS connector - Pairs with MCIMX-LVDS1 LCD display board

Rendering Technology: OpenGL ES 2.0

Demo Resolution: 1024 x 768

Operating System: Linux

This demo image includes the following demos:

- IoT - Smart Home
- Movie Kiosk
- White Goods

Toradex i.MX 7Dual SoM

Demo Details

Image Name: CrankSoftware_demo_colibri_imx7.sdcard

Board: Toradex Colibri iMX7

Display: EDT 7" TFT WVGA with Touch

Rendering Technology: Linux framebuffer (fbdev)

Demo Resolution: 800 x 480

Operating System: Embedded Linux (Yocto)

This demo image includes the following demos:

- Home Automation
- White Goods
- Medical

Flashing the Image

To flash the Colibri iMX7 module a running U-Boot is required. Boot the module to the U-Boot prompt and insert the SD card (See Section 22.6 or 22.7 on Copying Image to an SD Card) and enter:

- run setupdate

then to update all components enter:

- run update

Once the update is complete the board will reboot into the Storyboard Demo Launcher.

U-Boot

If no U-Boot is present on the board additional steps are required. Please refer to the instructions on Toradex's website to get U-Boot on the board.

<http://developer.toradex.com/knowledge-base/flashing-linux-on-imx6-modules>

Copying Image to an SD Card - Linux

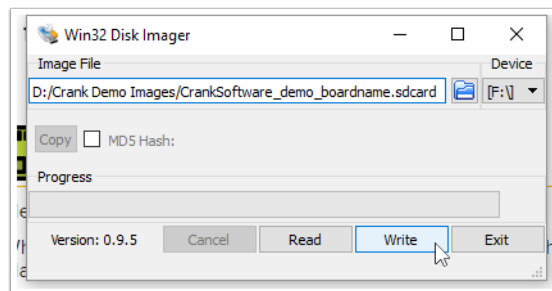
To copy the demo image to an SD card under Linux you first need to find the name of the attached SD card located under `/dev/`. You can then use the `dd` utility to copy the demo image to a 4Gb (or greater) SD card.

```
dd if="CrankSoftware_demo_boardname.sdcard" of="/dev/sdX" bs=1M
```

Copying Image to an SD Card - Windows

To copy the demo image to an SD card under Windows you can use the Win32 Disk Imager utility.

1. Specify the image file to copy
2. Specify the SD card device
3. Click the Write button



Running the Image

Each demo image contains all of the software needed to boot and run the card on the supported devices. Simply place the card in the appropriate SD card slot and power on the board. The cards will auto boot into the Crank Software Demo Launcher.

Turn On or Off the Console

By default the Linux console is disabled to improve performance and to prevent any display artifacts between application transitions. To re-enable the console do the following:

1. Connect a serial cable board
2. Interrupt the U-Boot startup sequence
3. `env edit console`
4. Switch `ttymxc1` to `ttymxc0`

5. `env save`

To disable the console, repeat the above steps but instead switch `ttymxc0` to `ttymxc1`.

Chapter 37. STMicroelectronics

Experience multi-market demo applications built with Storyboard on your favorite STMicroelectronics platform. Crank Software demo images showcase Storyboard's design and development capabilities and the application potential of popular platforms. Download [http://www.cranksoftware.com/demo_image/stm] the demo image that corresponds with your hardware, which includes everything you need to get and up and running in minutes.

STM32F429

Demo Details

Board: STM32429I-EVAL

Display: TFT LCD with resistive touchscreen

Rendering Technology: Chrom-ART Graphic Acceleration

Demo Resolution: 480 x 272

Operating System: FreeRTOS

This demo image includes the following demos:

- IoT - Home Automation
- White Goods
- Medical

STM32F439

Demo Details

Board: STM32439I-EVAL

Display: VGA TFT LCD with capacitive touchscreen

Rendering Technology: Chrom-ART Graphic Acceleration

Demo Resolution: 640 x 480

Operating System: FreeRTOS

This demo image includes the following demos:

- IoT - Home Automation

STM32 F7

Demo Details

Board: STM32F7 Discovery Kit (32F746GDiscovery)

Display: TFT LCD with capacitive touchscreen

Rendering Technology: Chrom-ART Graphic Acceleration

Demo Resolution: 480 x 272

Operating System: FreeRTOS

This demo image includes the following demos:

- IoT - Home Automation
- White Goods
- Medical

Required Hardware and Software

The STM32 demo images require the following:

- Storyboard Evaluation [<https://www.cranksoftware.com/free-trial>]
- STM32F746G-Discovery (“target platform”) or
- STM32F439I (“target platform”) or
- STM32F429I (“target platform”)
- STM32 ST-LINK Utility [<http://www.st.com/web/en/catalog/tools/PF258168>] used to download the demo image to the platform
- Micro SD Card (“sdcard”) with a FAT filesystem

Package Contents

This STM32 demo images contain:

- Storyboard_STM32F746G_Discovery.bin
 - Image to be downloaded to the target platform using the STM32 ST-LINK Utility
- Storyboard_STM324x9I__480_272_EVAL.bin
 - Image to be downloaded to the target platform using the STM32 ST-LINK Utility
- Storyboard_STM324x9I__640x480_EVAL.bin
 - Image to be downloaded to the target platform using the STM32 ST-LINK Utility
- demos/crank
 - Sample company information Storyboard application for the STM32F746G-Discovery or STM32F429

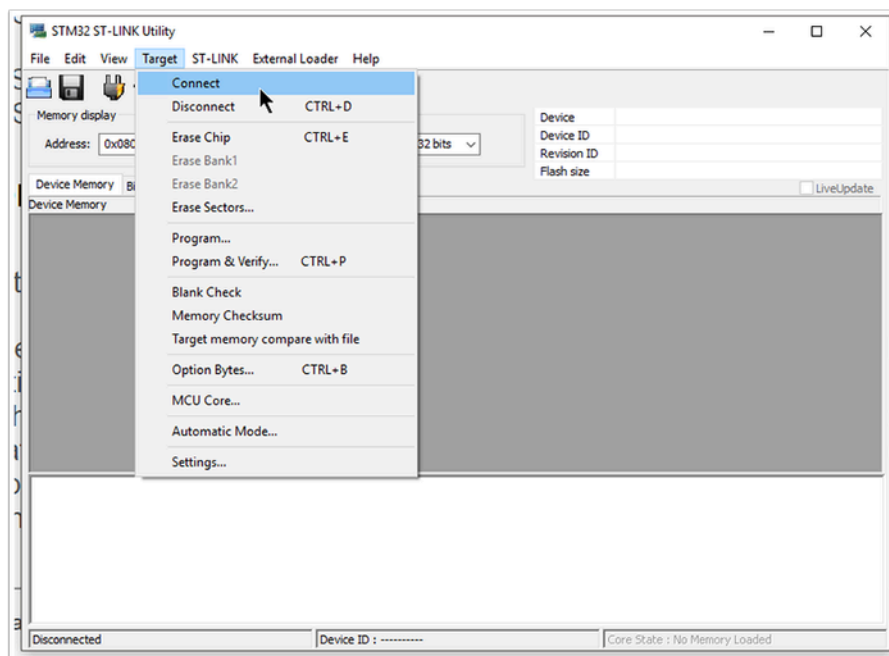
- demos/fbake
 - Sample white goods Storyboard application for the STM32F746G-Discovery or STM32F429
- demos/HomeAutomation-640
 - Sample home automation 640 x 480 Storyboard application for the STM32F439
- demos/HomeAutomation-480
 - Sample home automation 480 x 272 Storyboard application for the STM32F746GDiscovery or STM32F429
- demos/medical
 - Sample medical Storyboard application for the STM32F746G-Discovery or STM32F429
- demos/SB_App.txt
 - Storyboard application definition file

STM32 ST-LINK Utility

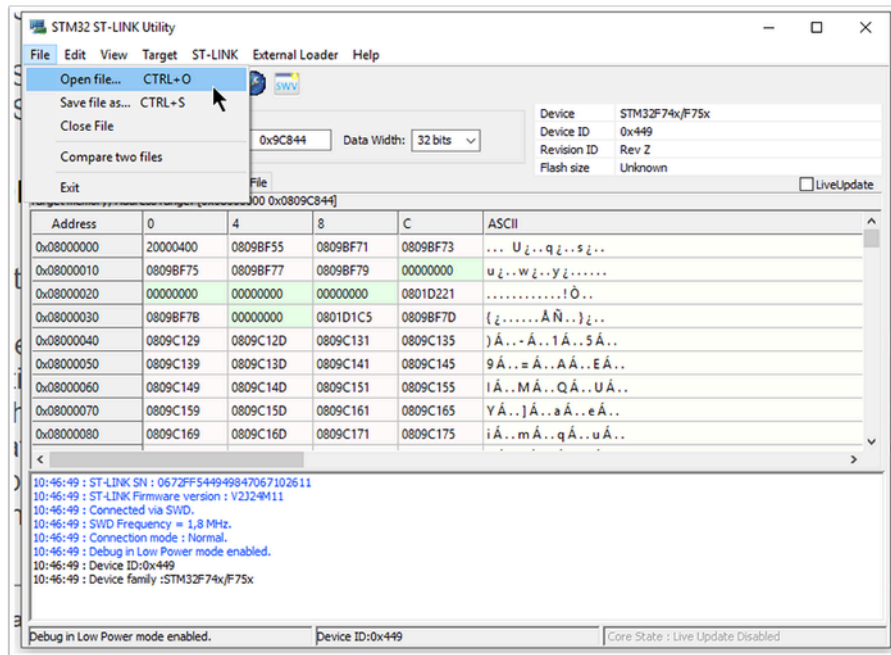
In order to use the STM32 demo images the following steps must be performed. These steps are specific to the STM32F746G_Discovery but use the correct .bin file for the STM32 target platform being used.

Write the “Storyboard_STM32F746G_Discovery.bin” file to the target platform using the STM32 ST-LINK Utility.

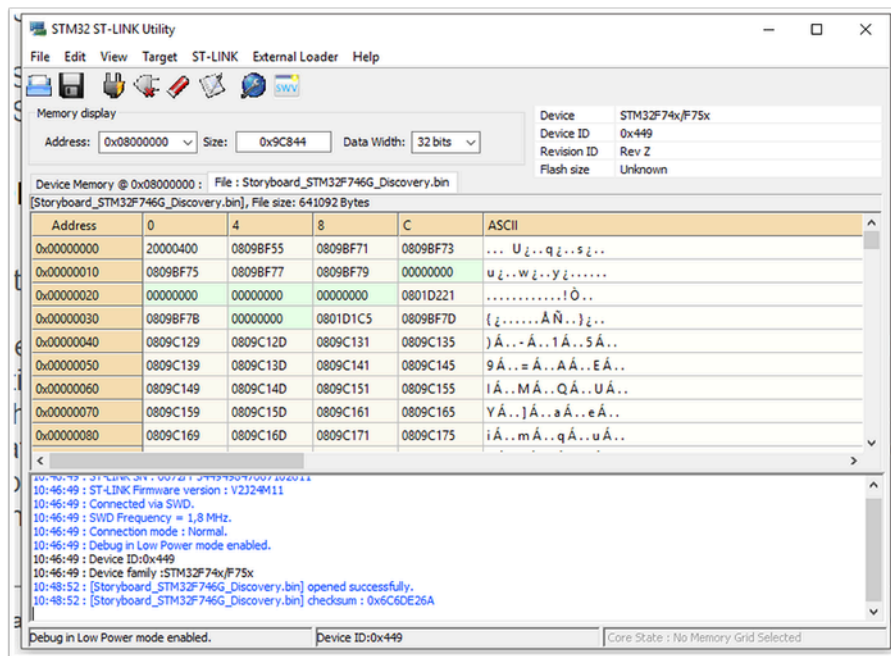
1. Connect to the target platform (STM32F746G_Discovery) Storyboard Evaluation: STM32 Storyboard



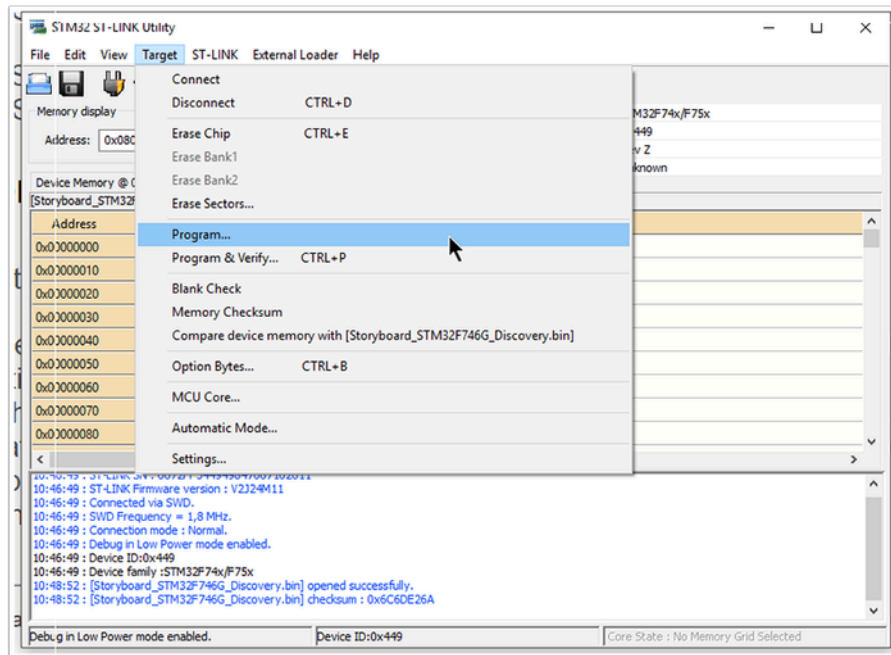
2. Browse to the Storyboard_STM32F746G_Discovery.bin file



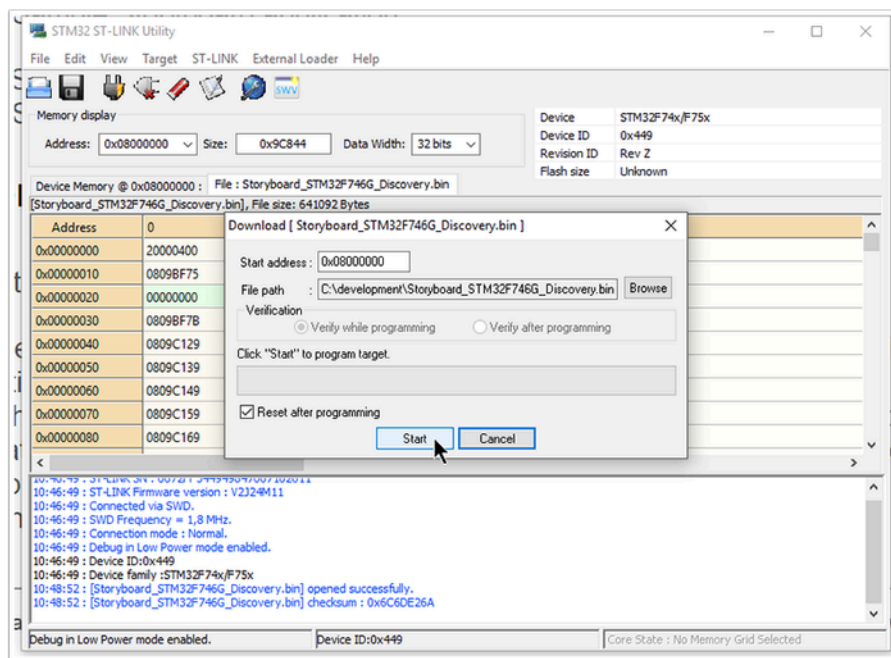
3. The Storyboard_STM32F746G_Discovery.bin file is now loaded in the utility



4. Program the Storyboard_STM32F746G_Discovery.bin file to the target platform



5. Start the programming process



Running the Demo

Now that the STM32 board has been flashed, the Storyboard demos can be run.

1. Mount the SD card and copy the contents of the evaluation “demos” directory to the SD card. Do not create the “demos” directory on the card, only copy the HomeAutomation-480 directory and SB_App.txt file.

2. Unmount the SD card and place the card back into the target platform.
3. Reboot the target platform.

At this point the demo should load and be interactive via the touchscreen.

Importing the Demo into Storyboard Designer

To import the HomeAutomation-480 demo into Storyboard Designer:

1. Start Storyboard Designer
2. Select “File” > “Import”
3. From the dialog select “General” > “Existing Projects into Workspace”
4. Browse to the “demos” directory and select “Open”
5. The Projects area should now list the HomeAutomation demo, select Finish
6. You now will have the HomeAutomation demo in your workspace

Creating Your Own Demo

Follow these steps to create your own demo in Storyboard Designer for the STM32 target platform:

1. Create a new application with a screen resolution of 480x272.
2. Add content to your application
3. Simulate and test your application on the desktop to verify functionality
4. Insert the SD card in your system and mount it
5. Export your Storyboard application directly to the SD card
 - a. Right click on the Application file (.gde)
 - b. Select “Storyboard Export”
 - c. Select “Export Storyboard Embedded Engine”
 - d. Under “Select export directory” deselect the “Use same directory as selected model file” checkbox and select “Browse”
 - e. Browse to the SD card and create a new directory for your application
 - f. Select “Finish”
6. Edit the “SB_App.txt” file on the SD card and change the name of the application to load to be your new application, including directory name and gapp file name.
7. Place the SD card into the target platform and reboot

Chapter 38. Microchip

Experience multi-market demo applications built with Storyboard on your favorite Microchip platform. Crank Software demo images showcase Storyboard's design and development capabilities and the application potential of popular platforms. Download [http://www.cranksoftware.com/demo_image/microchip] the demo image that corresponds with your hardware, which includes everything you need to get up and running in minutes.

SAMA5D2

Demo Details

Image Name: CrankSoftware_Demo_sama5d2xplained.sdcard

Board: SAMA5D2 Xplained

Display: PDA TM4300B

Rendering Technology: Framebuffer

Demo Resolution: 480 x 272

Operating System: Linux

This demo image includes the following demos:

- IoT - Home Automation
- White Goods
- Medical

SAMA5D2 with 7 inch Display

Demo Details

Image Name: CrankSoftware_Demo_sama5d2xplained_7.sdcard

Board: SAMA5D2 Xplained

Rendering Technology: Framebuffer

Demo Resolution: 800 x 480

Operating System: Linux

This demo image includes the following demos:

- IoT - Home Automation
- White Goods
- Medical

Copying Image to an SD Card - Linux

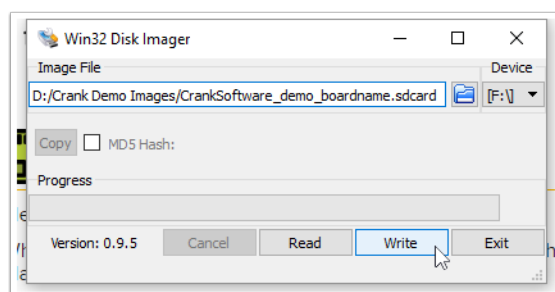
To copy the demo image to an SD card under Linux you first need to find the name of the attached SD card located under `/dev/`. You can then use the `dd` utility to copy the demo image to a 4Gb (or greater) SD card.

```
dd if="CrankSoftware_demo_boardname.sdcard" of="/dev/sdX" bs=1M
```

Creating an Image - Windows

To copy the demo image to an SD card under Windows you can use the Win32 Disk Imager utility.

1. Specify the image file to copy
2. Specify the SD card device
3. Click the Write button



Running the Image

Each demo image contains all of the software needed to boot and run the card on the supported devices. Simply place the card in the appropriate SD card slot and power on the board. The cards will auto boot into the Crank Software Demo Launcher.

Chapter 39. Renesas

Experience multi-market demo applications built with Storyboard on your favorite Renesas platform. Crank Software demo images showcase Storyboard's design and development capabilities and the application potential of popular platforms. Download [http://www.cranksoftware.com/demo_image/renesas] the demo image that corresponds with your hardware, which includes everything you need to get and up and running in minutes.

RZ/A1

Demo Details

Processor: ARM Cortex-A9

CPU Speed: 400 Mhz

Rendering Technology: Software

Operating System: Linux

This demo image includes the following demos:

- Home Automation
- White Goods - Washing Machine

Flashing the Image

To flash the Storyboard demo to the RZ/A1 you will need to do the following:

1. Connect the J-Link to the RZ/A1 board and your computer (USB)
2. Power on the RZ/A1 board
3. Unpack the archive to a directory on your computer
4. Open the RZA1_demo directory
5. Run the FlashHomeAutomationDemo.bat or FlashWashingMachineDemo.bat script
6. When flashed, the RZ/A1 will reboot and run the Storyboard demo

Chapter 40. Linux

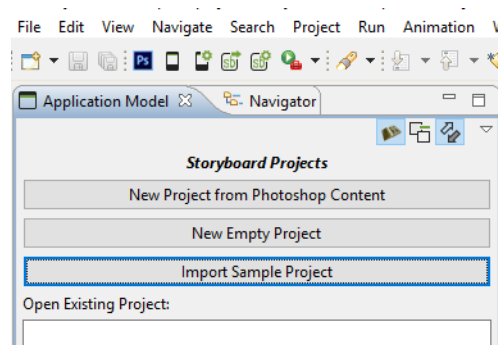
TI AM355 Starter Kit

The AM335x Starter Kit (EVM-SK) provides a stable and affordable platform to quickly start evaluation of Sitara™ ARM® Cortex™-A8 AM335x Processors (AM3352, AM3354, AM3356, AM3358) and accelerate development for smart appliance, industrial and networking applications. It is a low-cost development platform based on the ARM Cortex-A8 processor that is integrated with options such as Dual Gigabit Ethernet, DDR3 and LCD touch screen.

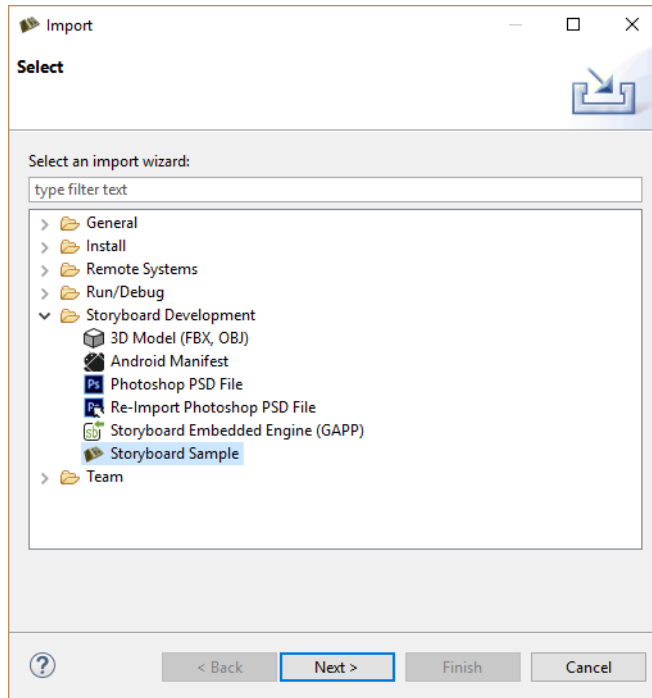
The following steps describe how to take a Storyboard sample and place it down on the TI AM335 board. It is assumed that the TI AM335 board has been setup correctly running Linux and that it is connected via a serial cable to either a laptop or desktop computer.

Step 1: Importing A Storyboard Sample

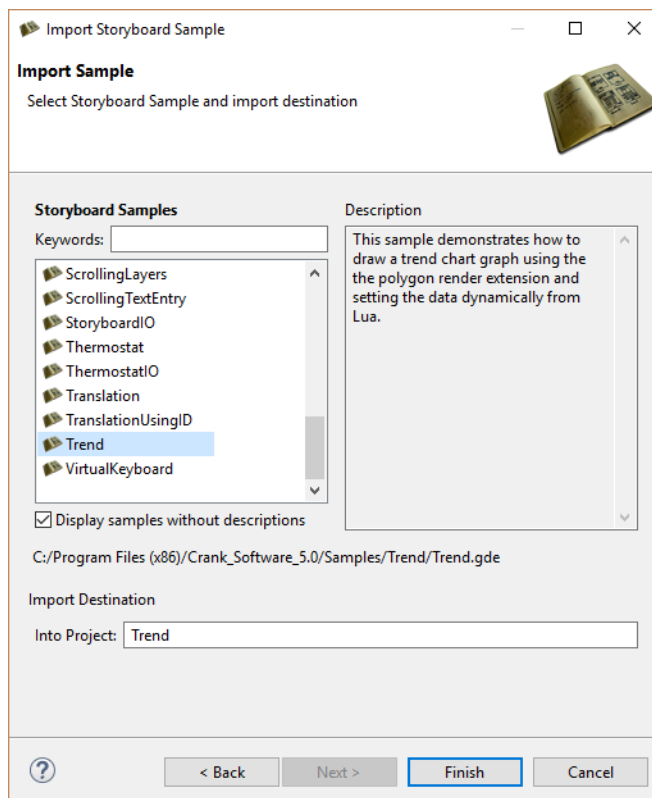
To import a Storyboard sample the user can right-click within the Navigator view and select Import or by selecting Import Sample Project from the Application Model view.



In the Select dialog expand the Storyboard Development folder and select Storyboard Sample. (This dialog box will be skipped if you selected Import Sample Project from the Application Model view.)

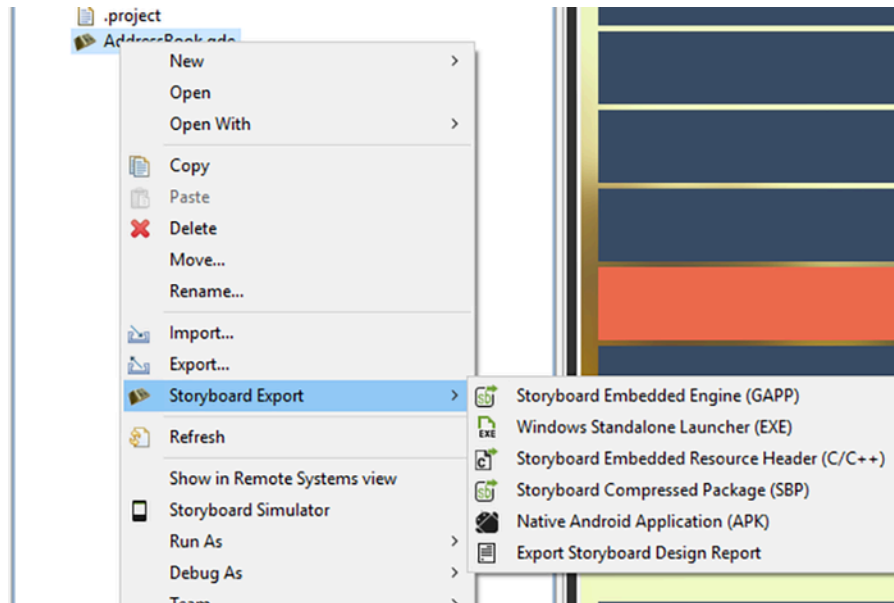


In the Import Sample dialog any sample can be used but for the purposes of this example the Trend sample has been selected.

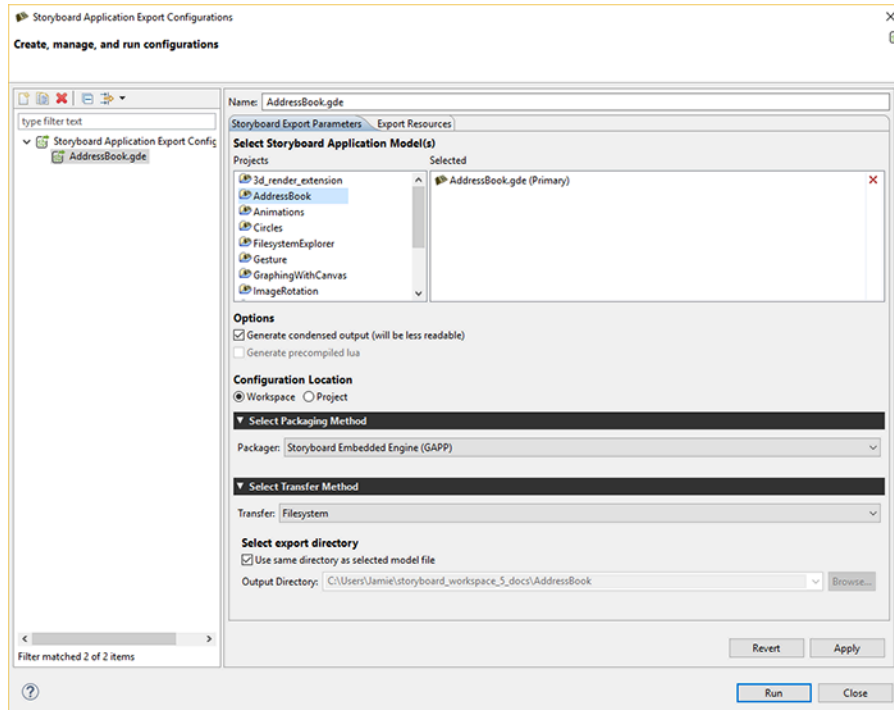


Step 2: Exporting A Storyboard Application

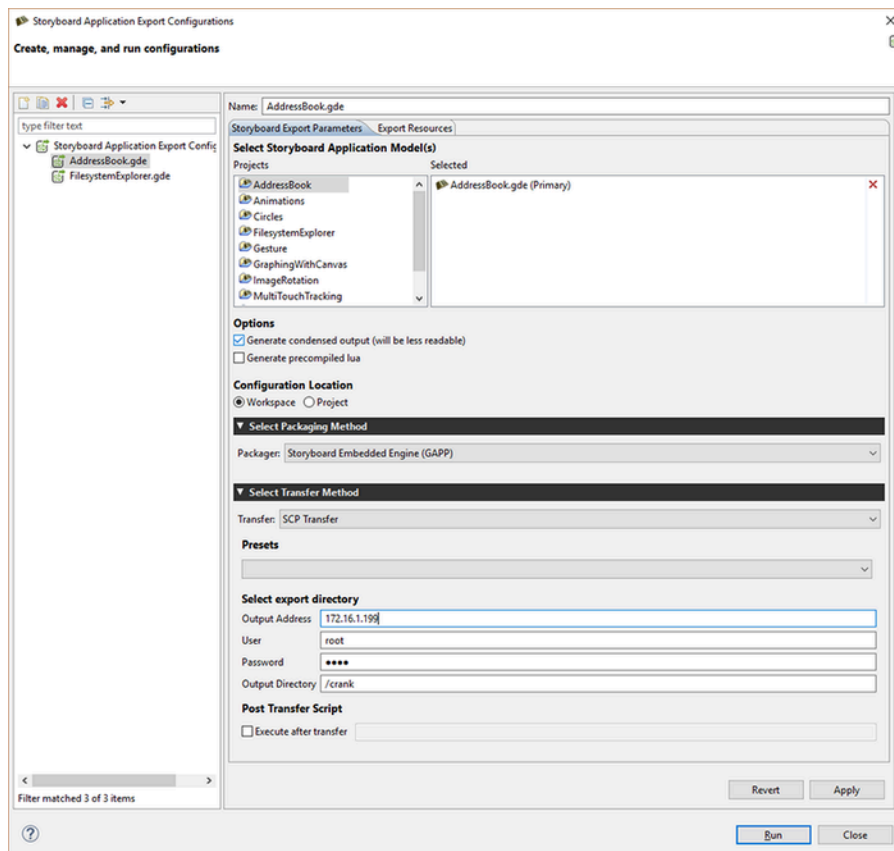
Once a Storyboard application is complete and ready to be placed on a target platform, it needs to be exported from Storyboard Designer in a format that the Storyboard Embedded Engine can use. Right-click the Storyboard application .gde file, located in the project folder in the Navigation View, and select Storyboard Export > Export Storyboard Embedded Engine.



The Export Selection dialog is used to tell Storyboard Designer where to export the selected Storyboard Application. Using the Select Transfer Method you can select if you want to export locally or remotely. Selecting Filesystem and leaving the "Use same directory as selected model file" option checked will place the data bundle for Storyboard Embedded Engine into the application's project directory. Deselecting the option enables the data bundle to be placed in another location (e.g., USB drive, NFS mount, etc).



Selecting SCP Transfer enables you to remotely connect to your target platform and transfer via SCP.



Step 3: Selecting The Storyboard Embedded Engine

The Storyboard Embedded Engine is the optimized runtime component that resides on the target platform that interprets the data bundle to display the Storyboard application. The Storyboard Embedded Engine is categorized by operating system, system architecture, and rendering technology.

All the supported Storyboard Embedded Engines are shipped with Storyboard and are located under the Storyboard_Engine directory.

Note

For the purpose of this example, the Storyboard application and Storyboard Embedded Engine will be exported to a USB drive.

To run the Storyboard application on the TI AM335 a Linux, Armle, OpenGL ES runtime is required. Copy the linux-tiam335x-armle-opengles_2.0-obj Storyboard Embedded Engine and place it on the USB drive along with the Storyboard application that was just exported.

linux-imx6yocto-armle-fbdev-obj	2017-02-21 4:19 PM	File folder
linux-imx6yocto-armle-opengles_2.0-obj	2017-02-21 4:19 PM	File folder
linux-imx53yocto-armle-fbdev-obj	2017-02-21 4:19 PM	File folder
linux-rzg1m-armle-opengles_2.0-waylan...	2017-02-21 4:19 PM	File folder
linux-sama5d-armle-fbdev-obj	2017-02-21 4:19 PM	File folder
linux-tiam335x-armle-fbdev-obj	2017-02-21 4:19 PM	File folder
linux-tiam335x-armle-opengles_2.0-obj	2017-02-21 4:19 PM	File folder
linux-tiam437x-armle-fbdev-obj	2017-02-21 4:19 PM	File folder
linux-tiam437x-armle-opengles_2.0-obj	2017-02-21 4:19 PM	File folder
linux-tij6-armle-fbdev-obj	2017-02-21 4:19 PM	File folder
linux-tij6-armle-opengles_2.0-obj	2017-02-21 4:19 PM	File folder

Step 4: Configuring The Target Platform

The USB drive with the Storyboard application along with the correct Storyboard Embedded Engine can now be ejected from the laptop/desktop and connected to the TI AM335.

The following commands are to be executed within the serial terminal connected to the TI AM335.

Login to the TI AM 335 and mount the USB drive:

```
login: root
```

```
mount /dev/sda1 /mnt/usb
```

Now, the Storyboard SB_PLUGINS specific environmental variable and an addition to the LD_LIBRARY_PATH environment variable need to be made:

```
export SB_PLUGINS=/mnt/usb/linux-tiam335x-armle-opengles_2.0-obj/plugins
```

```
export LD_LIBRARY_PATH=/mnt/usb/linux-codesourcery-armle-opengles_2.0-obj/lib:$LD_LIBRARY_PATH
```

Step 5: Running The Storyboard Application

With the needed environmental variables now set, the next step is to run the Storyboard application by passing it to the Storyboard Embedded Engine.

```
/mnt/usb/linux-codesourcery-armle-opengles_2.0-obj/bin/sbengine /mnt/usb/storyboard_export/trend.gapp
```

Raspberry Pi

The easiest way to get the Raspberry Pi up and running with a Storyboard application is by using the following steps:

1. Follow the steps in the "Getting Started with NOOBS" video from the Raspberry Pi website to get the Pi up and running with the Raspbian OS. NOTE: Configure Raspbian to boot to the console not the windowing system.

<https://www.raspberrypi.org/help/noobs-setup/>

2. Use the linux-raspberry-armle-opengles_2.0-obj runtime located in the Storyboard_Engine dir of your SB installation
3. Place the SB Pi runtime on the hardware.
4. Before executing the Storyboard runtime, the following environmental variables need to be setup. This example assumes the runtime and application were placed in a /crank dir

```
export SB_PLUGINS=/crank/linux-raspberry-armle-opengles_2.0-obj/plugins
```

```
export LD_LIBRARY_PATH=/crank/linux-raspberry-armle-opengles_2.0-obj/lib:
$LD_LIBRARY_PATH
```

5. Execute the runtime with mouse and keyboard support:

```
/crank/linux-raspberry-armle-opengles_2.0-obj/bin/sbengine -vv -oscreen_mgr,swcursor -odev-input,mouse=/dev/input/event0,kbd=/dev/input/event1 /crank/your_sb_app.gapp
```

Part III. Release Notes

Table of Contents

41. Release Notes 6.2	417
New Features and Functionality	417
Designer	418
Engine	418
Lua Scripting	419
Behavioural Changes and Deprecations	419
Known Issues	419
42. Release Notes 6.1	421
New Features and Functionality	421
Storyboard Designer	421
Storyboard Engine	422
Lua Scripting	422
Behavioural Changes and Deprecations	422
43. Release Notes 6.0	423
New Features and Functionality	423
General	424
Storyboard Designer	424
Storyboard Engine	424
Behavioural Changes and Deprecations	425

Chapter 41. Release Notes 6.2

New Features and Functionality

- Storyboard Engine ported to ITRON operating system.
- Storyboard IO over TCP/IP has been introduced as a native functionality available on target platforms where TCP/IP connectivity is part of the standard configuration. This means that you can now send and receive Storyboard events over the network.
- The Lua action has been enhanced to better support Lua object oriented programming techniques. Lua object modules can be referenced directly now as callbacks further encapsulating the context of execution and removing the need for some boilerplate glue logic.
- Fill and rectangle render extensions now have a corner radius property for rounded rectangles. This feature can be used to help reduce the memory or storage footprint by replacing images that may have been used for the same visual effect.
- For some shadowed scenarios where a rounded rectangle may not offer enough visual appeal nine-slice/nine-patch images can be used. Nine-patch images can now be rendered directly from flash and rotated using no additional memory for a scalable UI component and can be compressed as RLE for additional memory savings.
- Storyboard Designer's now includes the ability to turn performance logging and execution tracing on and off from the IO Connector. When coupled with the TCP/IP Storyboard transport this creates some very powerful remote analysis possibilities.
- The performance log viewer includes event data plotting and visualization tools that allow you to look at the data gathered in performance trace logs and put it into context with triggering events and screens of execution.
 - Redraw time, CPU consumption and Memory Usage information is plotted on platforms where this information is available.
 - Individual animation variables can be plotted as they change over time.
 - Specific problem solving tasks are made easier with pre-configured data plots.
 - Statistical plotting functionality allows users to create their own plots by extracting metrics that are relevant for their areas of investigation.
- A performance Lua API has been added that allows user applications to inject data markers and trace points into the performance log. Coupled with the plotting and visualization this can put user code execution in context of other metrics being considered.
- New algorithms for software rendering scaling and rendering operations resulted in a 50% reduction of dynamic memory consumption when these operations are used independently.
- The compare and merge operations on Storyboard Designer projects are more consistent and three way comparisons with source control systems (git/svn) are faster.

Designer

- The Find References action is now partnered with a find that will search for the selection within other files of the project. Useful for finding model and image references in Lua files.
- Component creation will now allow you to convert the original component source to a component itself. Copying and pasting a component within a project will maintain that component's relationship as a component for future updates.
- The animation preview now supports looping of the animation that is being previewed and limits the preview to the frames that will be shown by the runtime.
- All animation step variable keys will be included in the refactoring operations that occur when the names of variables or model objects are changed.
- Drag and Drop of model content now maintains consistent z-order.
- Direct inline text editing uses the correct font and display.
- Photoshop import and re-import does a more accurate job representing multi-line text and text with alignment attributes where it is possible to do so.
- The C/C++ Event Header can now be built from the command line.

Engine

- Layer and table scrolling can now be dynamically enabled/disabled through the new `scroll_enabled` attribute or `grd_scroll_enabled` variable.
- Table resize operations will no longer reset the current scroll offset unless the scroll offset would result in the data not being visible.
- Scrolling operations are smoother and snapping takes advantage of the direction to ensure that snapping operations occur to the right context boundary.
- OpenGL rendering quality improvements:
 - Alpha layer blending blends smoothly over the full alpha value range.
 - 3D model camera settings (Z in particular) properly show the model.
 - Single pixel offset incorrect for certain rectangles.
 - Custom shaders with UV wrap don't need `-oresource_mgr,image_block_size=-1`
 - Removed custom shader requirement for `-oresource_mgr,image=-1,font=-1` option
 - Proper format support for the `GL_BGRA_EXT`
- Logging and verbosity can now be redirected over an SBIO channel as events.
- Performance logging filename options now support appending, date and time stamps.
- The fbdev software render manager allows the selection of framebuffers to render to.
- Android and Windows will correctly restore themselves after coming back from sleep.

Lua Scripting

- A new Lua API has been added to allow controls to be resized or moved in a relative sense using the same anchoring functionality available in Designer when working interactively. Scale controls with a single command without having to hit x/y/width/height values independently.

Behavioural Changes and Deprecations

- A new font-manager option, `-ofont_mgr.linegap` has been introduced to disable the linegap offset that Storyboard Engine inserts with certain fonts (notably Noto). This option is disabled by default to maintain compatibility with the current Storyboard Engine, but will be enabled by default in the next major version of Storyboard.
- A new model-manager options, `-omodel_mgr.block_sbio_sends` has been introduced that will adjust the blocking behaviour of Storyboard Engine's SBIO send operations. The default configuration is to not block sending calls from the UI as that has the effect of potentially deadlocking the UI, however clients can revert the behaviour to the older behaviour by indicating that sends should be blocked.
- Multi-touch event compression enabled by default. This option should have no ill effect on clients but an option to remove all input event compression added

Known Issues

- OpenGL rendering clients should see an improvement in visual quality for the 6.2 release but that improvement comes at the cost of more GLSL shader operations.
- Some clients may observe a longer load time or a slight degradation in performance depending on the sophistication of their GPU technology.
- Dynamic injection is used on the shaders and patterns such as:
 - `gl_FragColor.r = grd_c_r;`
 - `gl_FragColor.g = grd_c_g;`
 - `gl_FragColor.b = grd_c_b;`
 - `gl_FragColor.a = col.a;`
- May require conversion to the style of:
 - `gl_FragColor = vec4(grd_c_r, grd_c_g, grd_c_b, col.a);`
- Direct editing of clipped, multi-line, wrapped or ellipsis text will not show the cursor in the correct location.
- Starting and stopping and appending results to the performance log may produce unusual display plots.
- Animated GIFs consume memory for each frame that is rendered and the frames can't be incrementally loaded making them poor choices for limited resource targets.
- The Metrics View does not report on all static consumers of memory including the display framebuffers, canvas and external render extensions and rich text.

- Debugging Lua on Linux systems may not work through the one click launcher, but the debugger connection may need to be launched before the main application.
- Some exceptionally large Lua files may identify as corrupt and not show properly in the outline.

Chapter 42. Release Notes 6.1

New Features and Functionality

- Text shaping for complex scripts has been added as a new plugin and is supported on MacOS, QNX and most Linux platforms that use FreeType as their font manager. This allows complex text scripts such as Thai to be properly rendered without the use of third party commercial integrations to Storyboard Engine:
 - https://en.wikipedia.org/wiki/Complex_text_layout
- A new text overflow option that enables individual character wrapping has been added to the Storyboard text render extension.
- The Performance Log Editor (for .plog files) has received a makeover. It is faster to load, handles large log files more capably and re-organizes the information presented to allow you to dig into issues more effectively. It's just way better!
- Support for 9-patch (9-slice) images that can be loaded directly from flash further reducing the dynamic memory requirements for scalable UI's on embedded platforms.
- A number of enhancements have been made to for deep embedded C/C++ application developers using the C Callbacks action feature and the Storyboard API.
 - The export is available directly in the context of the standard export dialog, not just as a standalone exporter.
 - Model objects such as controls and layers that are going to be used within C/C++ code can be given symbolic names in their properties that can be used to isolate changes to C/C++ code as the application UI is refactored. Look for this new label in the properties panel.
 - The Storyboard API has been enhanced with new accessor functions that make it easier to read, write and manipulate variable values from the Storyboard Engine. We also took the opportunity to clarify the previous API's with new data types that add absolute clarity to how memory and resources are to be managed through this API.

Storyboard Designer

- Exporting has gotten a whole lot wiser and faster too! Exporting the Storyboard model and resources to a filesystem or through an SCP network transfer now has the option of only pushing changed content which significantly improves round trip time for large projects. No more checking slack while your project downloads and launches!
- The Copy Path command can be customized in the preferences to use single quotes rather than double quotes to match project coding standards. The command has been made accessible as a keyboard shortcut (Ctrl+Alt+C Windows/Linux or Cmd+Option+C Mac) and if you have multiple objects selected will create paths for each entry onto the clipboard.
- Launching the Storyboard Simulator was always accessible from within a model editing environment via Ctrl+L keyboard shortcut, but that has been expanded to a new shortcut available when working in Lua files: Ctrl+Alt+L on Windows/Linux and Ctrl+Cmd+L on Mac

- Photoshop file import will generate font variables that are more aligned with the original font names rather than the name of the substituted font.
- Resources added to the export configuration can be removed rather than just excluded.
- Resizing an application will now propagate changes to all of the Design States.
- Import time for 3D model files in OBJ format has been reduced significantly.
- Reduced typing input delays that occurred on certain user interactions.

Storyboard Engine

- Layer variable resolution (`${layer}`) has been extended to a few locations where it was previously not available, namely scrolling, animations and the Lua API.
- Order of magnitude performance improvements for non-latin, unshaped, text rendering.
- Invalid UTF-8 character content for text content no longer causes rendering delays.
- Storyboard Virtual Filesystem (SBVFS) now supports multiple filesystem roots providing more flexibility for resource storage and optimizing development time using minimal flash updates.

Lua Scripting

- The `gre.get_string_size` API now allows you to get the bounds of wrapped text using the same breaking and wrapping algorithms that the render extension uses. Use this new API instead of hand-crafting your own text breaking algorithm in Lua!
- Performance improvements align the costs for `set_data`, `set_value`, `set_control_attrs`
- The GRE DOM interface (`libgre-plugin-luagredom` plugin) has been enhanced with additional API exposing common operations (`set/get position`, `bounds`, `visibility`).
- Inline editor documentation for GRE DOM and GRE Canvas functions.
- Code completion shows all functions available from files in the scripts directory.

Behavioural Changes and Deprecations

- Variables bound to the previous text wrapping option (`wrap`) will be automatically upgraded to the new `overflow` option introduced in Storyboard 5.3. Older runtime files (GAPP) will continue to work as expected but new files generated starting with 6.1 will only use the `overflow` option in the export. If wrapping was dynamically controlled as a variable then the model load into Designer will unbind this variable since the `wrap` property is no longer supported using the same variable parameters.

Chapter 43. Release Notes 6.0

New Features and Functionality

- Standalone Storyboard application file compare/merge utility included with Storyboard to support continuous integration and automated build workflows.
- Dark UI theme support for the Storyboard Designer development environment.
- The new Resource Export configuration editor allows fine grained control over how resources are exported and managed for embedded targets. Used with the new Project Metrics view, this replaces the previous export control in the Simulator/Export dialog.
- The new Project Metrics view allows application developers to understand the full memory and storage costs associated with their application as it is being designed. The Project Metrics works hand in hand with the new Resource Export configuration editor to show the different costs associated with various target deployment scenarios ranging from SW rendering/RT Exec to OpenGL/RTOS deployments. This is a powerful tool for the specialized activity of embedded application deployment.
- Design States offer a brand new way for application developers to see how their application will look in different scenarios. A perfect compliment to the existing screen-based organization states of an application's user interface, design states show you how a particular screen will look at different times based on changes caused by animations or external data. A great tool to visualize effects that occur in multiple screen contexts!
- Storyboard has revamped its translation and internationalization functionality to better address the needs of application developers and align with industry standard practices. The new Text Translation view shows what text content has been translated, what that translation value is and includes hints related to the correct visibility of the text. To extend translation to internationalization and some light theming support, users can now add additional variables such as fonts, point sizes and control metrics as attributes to be changed when languages change. Storyboard Engine supports this new functionality via a Lua module that allows fast and easy application language customization using the same translation text sources as in Designer.
- Storyboard 6.0 introduces an optimized OpenGL scene graph renderer. This is a complete re-engineering of the Storyboard Engine rendering pipeline to interface more optimally with high-performance 3D GPU technology in more configurations on more hardware platforms. In some scenarios, performance improves by up to 150%!
- Storyboard Lite Engine is a memory footprint reduced version of the Storyboard Engine specifically targeted at resource (memory and flash limited) constrained embedded target hardware platforms. In addition, the standard performance optimization tuning that occurs with each Storyboard Engine release, this configuration has been created with an eye to optimizing execution for low memory systems. A direct to C callback API allows developers to simulate environments with Lua and replace that functionality with compiled C code on target. The Resource Export configuration and the Project Metrics view have been specifically organized to give developers the insight they need to tune a Storyboard Lite Engine deployment.
- A feature scalable user interface has been introduced to Storyboard Designer to provide a common UI development tool that easily adapts itself to enable only those features available on your target platform. The feature sets allows developers to create rich multi-touch or gesture-based applications that leverage advanced 3D graphics and multimedia capabilities of OpenGL Storyboard Engine platforms in the same

environment as they would create svelte and beautiful user interfaces for the more resource-constrained Storyboard Lite Engine platforms. One platform environment supports it all!

General

- New Scrolling Functionality
 - New scroll stop/cancel action
 - New auto scroll detected trigger event
 - Unified table cell and pixel gesture-based scrolling for tables
 - Alignment option for table and layer scrolling
 - Animation engine based auto scroll
 - Improved experience for overlaid scrolling table and layers
- Named data changes allow many variables to be set at once in a re-usable manner
- Text and Rich Text recognize U+200B character as a zero width break character
- New C Function callback action and API for use with Storyboard Lite Engine
- Exported C/C++ Headers contain model file and build details

Storyboard Designer

- Performance logs now contain build and release information
- Performance logs now contain animation trace information
- Search for and count image references in model from the Images view
- Actions/Variables/Render Extensions are now grouped in the Application view
- Hotkey bindings for fill horizontal, vertical, container commands
- Copy/Paste of controls and groups no longer insert an automatic position offset
- New model elements are automatically selected as they are created
- Unlocking a child control/group will prompt to unlock the parent layer if locked
- Storyboard Designer renders previously unsupported OTF/TTF fonts (NotoSansCJK)
- IO Connector event generation view can be opened to a floating window
- Command line utility for resizing Storyboard applications

Storyboard Engine

Lua API Enhancements

- Callback context contains FQN of what is invoking the call in `active_context`

- Storyboard IO channel name available using `gre.env("greio_channel")`
- New gre API constants for alignment TOP CENTER BOTTOM LEFT RIGHT
- Symbolic API for alignment `gre.get_alignment()`
- Symbolic API for alpha as a percentage `gre.get_alpha()`
- Inbound and outbound events are now generated on multi-touch interfaces
- Inbound and outbound events are now delivered to all children/layers affected
- Screen Path screen transition arguments for top/bottom directions have been changed to match the documentation, they were previously reversed.
- Events bound to variables will now be delivered to the layer of the context if one is applicable.
- New Storyboard Engine API
 - Function callback for easier diagnostic logging
 - Enhanced model API for use with C Callback action.
- Image Load Manager
 - An interface has been created to allow plugins to add their own discrete image format loaders or to override the internal image loaders.
- Font Manager Plugin
 - Select TrueType or Bitmap or Monotype rendering using the same engine
 - Dumping all font resources (`gre.dump_resource`) will now completely drop the font from memory instead of holding a cached reference. This reduces memory overall, but can have an impact on performance if the font needs to be reloaded.
- Comprehensive and efficient 16 bit display rendering support
 - Design in 32bit with target content export in 16bit for storage savings
 - Mixed mode 32bit backing and 16bit display configurations available
- Memory Management
 - Discontinuous heap management and engine specific memory management
 - Model memory footprint was reduced by an average of ~16%, and as high as 42% reduction in some areas

Behavioural Changes and Deprecations

- The multi-touch events `mtXmove`, `mtXpinch` and `mtXrotate` have been removed and replaced with `mt-move`, `mtpinch` and `mtrotate` events containing a data payload.
- The Lua argument context no longer references `context_target`, that value has been replaced with a new variable `active_context` that is the FQN of the object that is invoking the Lua action.

- The second argument to Lua callback functions was deprecated in Storyboard Engine 2.x and has now been removed entirely.
- When using C/C++ resource exports, all Lua files would be loaded. Now only those files in the scripts directory itself are loaded.
- Storyboard Engine command line option changes:
 - -orender_mgr,rotated=[90|180|270] changed to -orender_mgr,rotate=[90|180|270]
 - The display render manager option now compliments the fb option to help specifically identify which display index should be used.
- Storyboard Engine linux-*-fbdev has been replaced by the generic linux-*-swrender
- Storyboard Engine win32-x86-win32 has been replaced by win32-x86-swrender
- When using an animated GIF, an unbounded resource manager image cache configuration must be used
- Storyboard Designer no longer supports the import and conversion of Storyboard model files prior to version 4.x. These model files can be converted with any earlier version of Storyboard Designer ranging from 4.0 - 5.3. Contact support@cranksoftware.com for more assistance if required.

Part IV. Licensing

Table of Contents

44. Storyboard Licensing 429

Chapter 44. Storyboard Licensing

Storyboard includes several third party royalty free open source software components. A description of the third party components, their licenses and the context of their use within Storyboard is located on the Storyboard Product Licensing [<http://www.cranksoftware.com/product-licensing>] page located at <http://www.cranksoftware.com/product-licensing>.