Crank Storyboard Suite

Crank Storyboard Suite

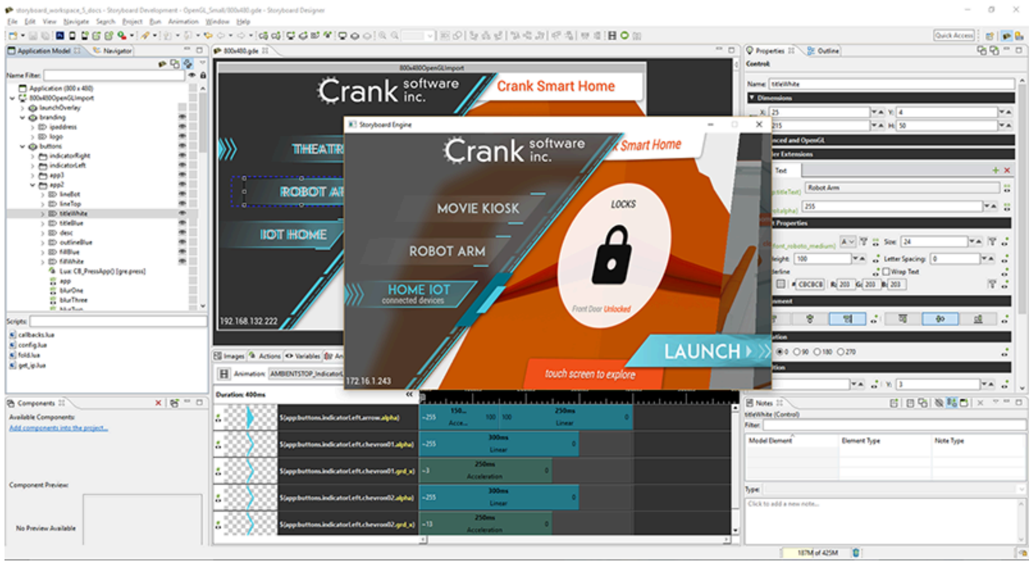
Table of Contents

List of Tables

Chapter 1. What Is Storyboard Suite

A New Way to Build Embedded User Interfaces

Storyboard Suite 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 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 Suite 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 Suite 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

Projects created with older versions of Storyboard Designer can be used directly with newer versions of Storyboard Designer, but may require additional import steps to update to the newer version. When migrating to new versions of Storyboard Suite save and archive a snapshot of the project if you plan to update it with an older Storyboard Designer environment.

- Storyboard Designer 5.x imports Storyboard Designer projects from Storyboard 4.x. To work with earlier projects within Storyboard Designer 5.x, use any Storyboard Designer 4.x to first convert the project before using it with Storyboard Designer 5.x.
- Storyboard Designer version 4.x and earlier imports Storyboard Designer projects from Storyboard 1.x, 2.x and 3.x based projects and converts them automatically to the new design file format established with Storyboard 4.0.

To use a Storyboard project from an earlier release than it was created (e.g., using a version 4.x project with a version 3.x 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

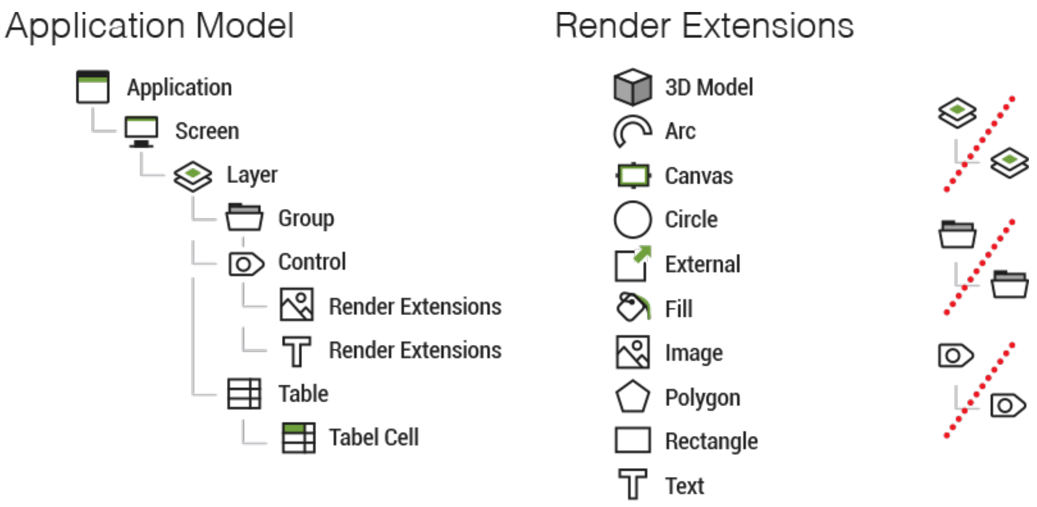
(*.gapp) from the design file (*.gde). The Storyboard Engine file contains only the information required to run the application and does not contain all of the design material (such as notes and unused layers) so that content will be lost. Features that are not present or used by the older Storyboard 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.

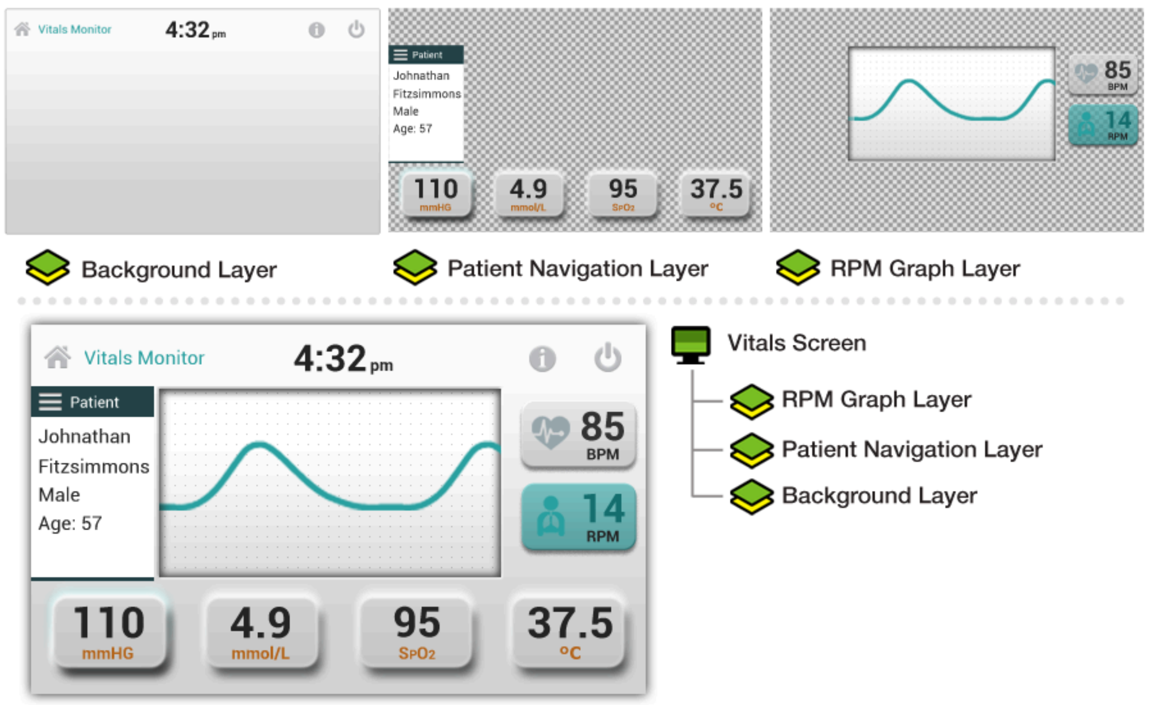
Application Model Hierarchy

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



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 17, *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.

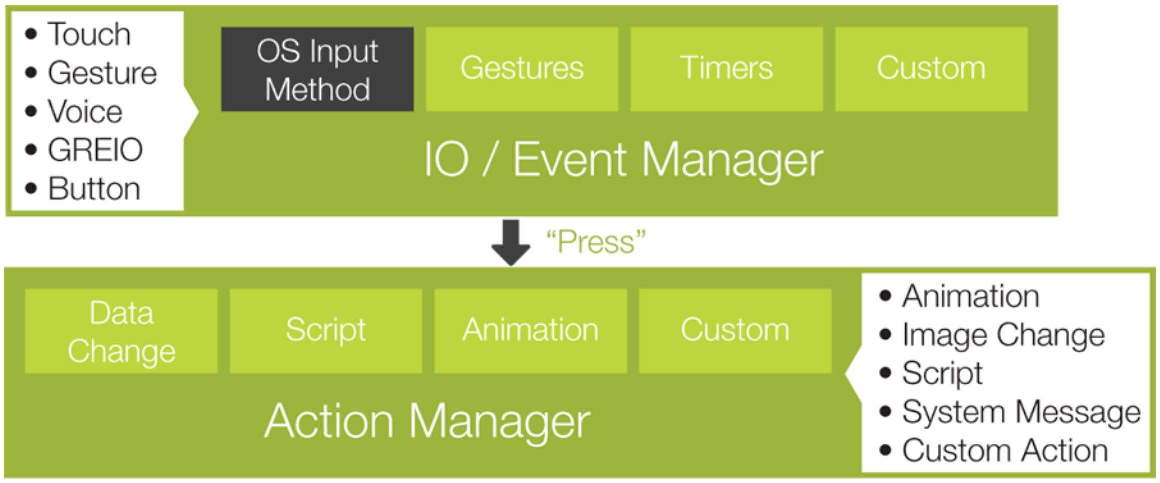
Event Driven Interaction Model

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.

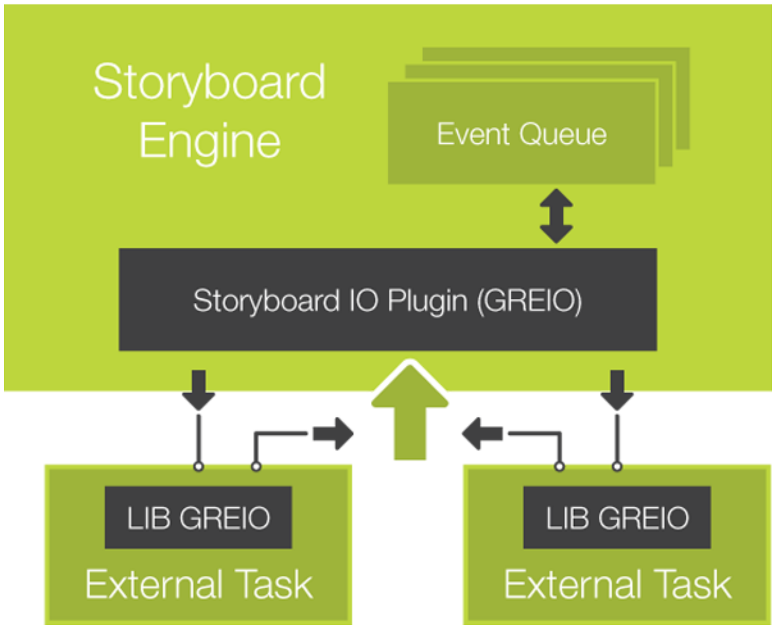
The event model also ensures a complete decoupling of the UI from the implementation and is what makes desktop simulation such a practical reality. Simulated data can be easily injected in environments where the real target hardware may not be ready, prohibitively expensive, or just plain awkward to 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 26, *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 actionsactions, 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.

varname	This identifies a variable, `varname`, as an application level variable
screen_name.varname	This identifies a variable, `varname`, as associated with the screen `screen_name`

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

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.

${app:varname}	Refers to the application variable `varname`.
${screen:varname}	Refers to the screen variable `varname`.
${layer:varname}	Refers to the layer variable `varname`.
${group:varname}	Refers to the group variable `varname`.
${control:varname}	Refers to the control variable `varname`.

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:

${app:varname}	`varname`
${screen:varname}	`MainScreen.varname`
${layer:varname}	`ALayer.varname`
${group:varname}	`ALayer.AGroup.varname`
${control:varname}	`ALayer.AGroup.AControl.varname`

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:

${cell:app:varname}	`varname.5.1`
${cell:group:varname}	`ALayer.AGroup.varname.5.1`

${cell:control:varname}	`ALayer.AGroup.ATable.varname.5.1`

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 variables

The following values can be queried and changed through normal data management channels. The position variables are relative to the screen.

`grd_x`	4s1	The layer instance's x position relative to the screen
`grd_y`	4s1	The layer instance's y position relative to the screen
`grd_xoffset`	4s1	The x pixel offset that will be used to determine the origin of the layer instance
`grd_yoffset`	4s1	The y pixel offset that will be used to determine the origin of the layer instance

Note

Any change to the following values affects all layers.

`grd_width`	4s1	The layer's width
`grd_height`	4s1	The layer's height
`grd_alpha`	1u1	The layer's transparency value. The values range from 255 (opaque) to 0 (transparent)
`grd_hidden`	1u1	The layer's visibility. A value of 0 states that the layer and all of its controls are visible and a value of 1 hides the layer and all of its controls

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`	lu1	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.

`grd_rows`	4s1	The number of rows in the table
`grd_cols`	4s1	The number of columns in the table
`grd_visible_rows`	4s1	The number of visible rows in the table
`grd_visible_cols`	4s1	The number of visible columns in the table
`grd_active_row`	4s1	The row index of the currently active cell
`grd_active_col`	4s1	The column index of the currently active cell
`grd_row`	4s1	The table’s current top left row
`grd_col`	4s1	The table’s current top left column
`grd_xoffset`	4s1	The x pixel offset that will be used to determine the origin of the 1,1 table cell
`grd_yoffset`	4s1	The y pixel offset that will be used to determine the origin of the 1,1 table cell

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/].

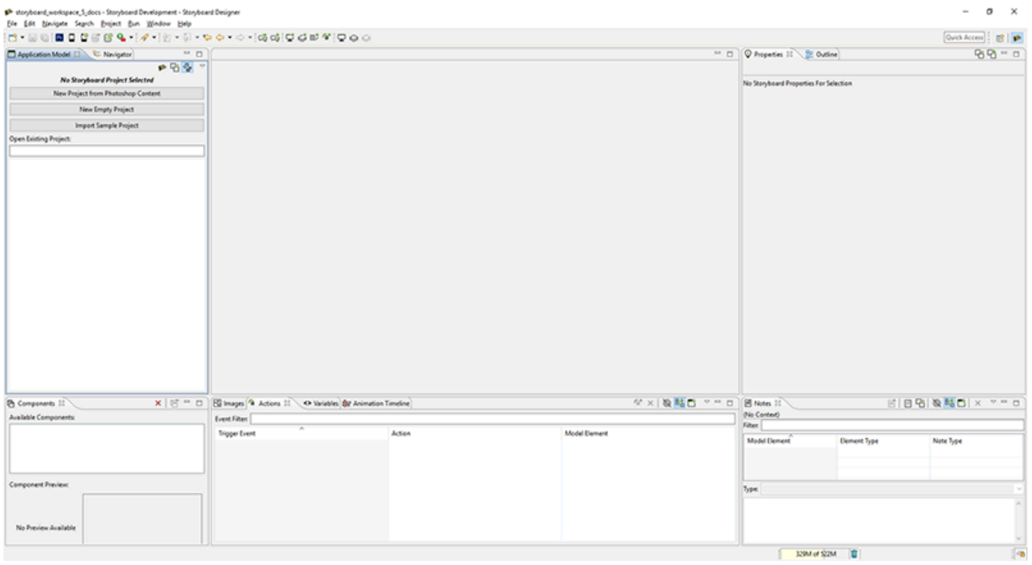
Embedded software developers typically work in C or C++ development environments. Storyboard Designer integrates into the CDT, the most common Eclipse based embedded development environment, so application user interface development can easily be done in parallel with other embedded software development.

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.



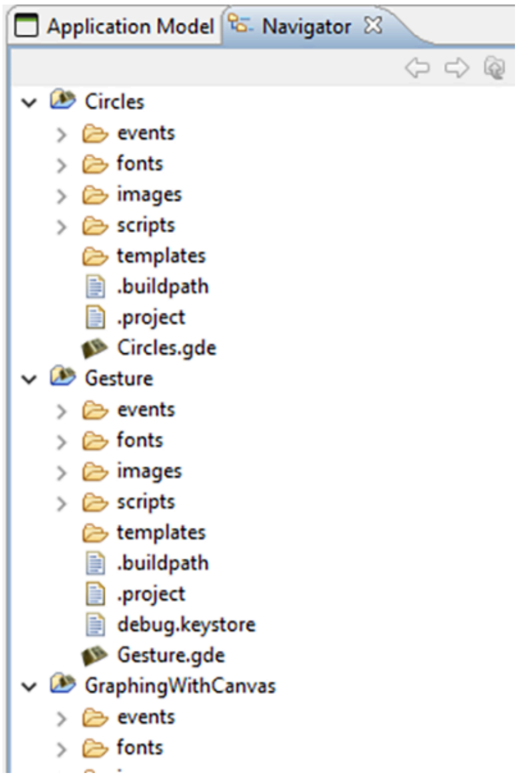
For more details on configuring the workbench refer to Help > Help Contents > Workbench User's Guide

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.

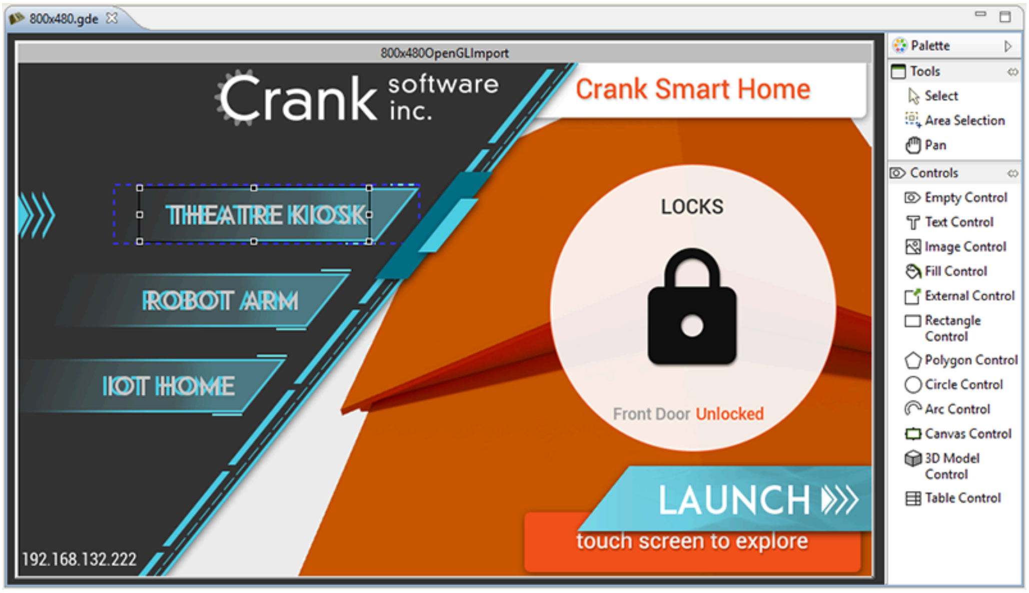
As changes are made in the filesystem, the workspace should refresh automatically and the changes are reflected in the Storyboard Designer user interface. An automatic refresh may be delayed due to system activity and can be forced at any time by selecting a project or file in the Navigator view and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

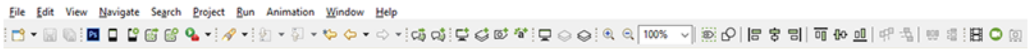
The right-hand side the editor contains a fly-out palette toolbar that provides the basic visual design elements for the application; screens, layers, and controls. These can be selected and dropped onto the editor to start building up the application.

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	This controls the current zoom level of the display. This value can also be adjusted by CTRL+MouseScroll or COMMAND+Scroll using a wheel mouse or touchpad.
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	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. This functionality is useful to minimize the amount of control overlap that occurs so as to prevent excessive redraw damage areas.

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.

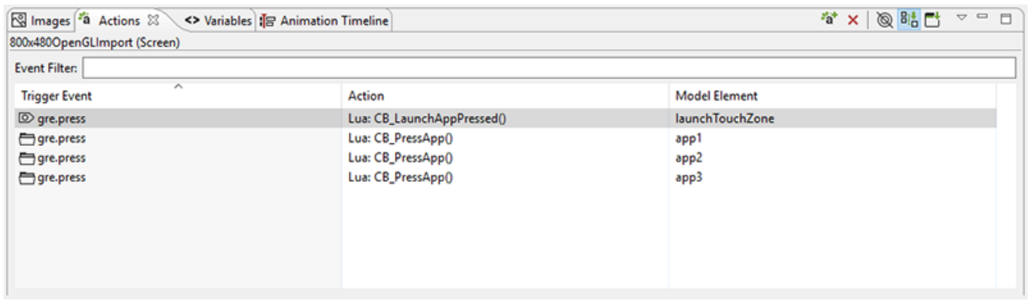
3D Model	When the direct edit mode is activated, a set of xyz axes will appear in the bottom left corner of the control. At that point the user can use their mouse to move the model. Holding the shift key allows the user to rotate the model

	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 mouse out of the control bounds.

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.



800x480OpenGLImport (Screen)

Event Filter:

Trigger Event	Action	Model Element
gre.press	Lua: CB_LaunchAppPressed()	launchTouchZone
gre.press	Lua: CB_PressApp()	app1
gre.press	Lua: CB_PressApp()	app2
gre.press	Lua: CB_PressApp()	app3

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:

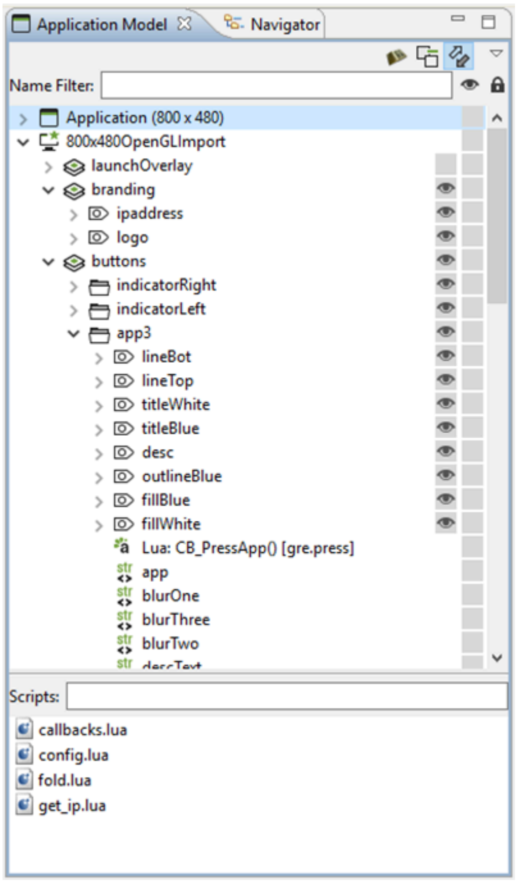
Selection Only	This shows only the actions associated directly with the selected model object.
Sub Hierarchy	This shows the actions of the selected model object and all of its child model objects.
Application Hierarchy	This shows all of the actions in the project.

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.

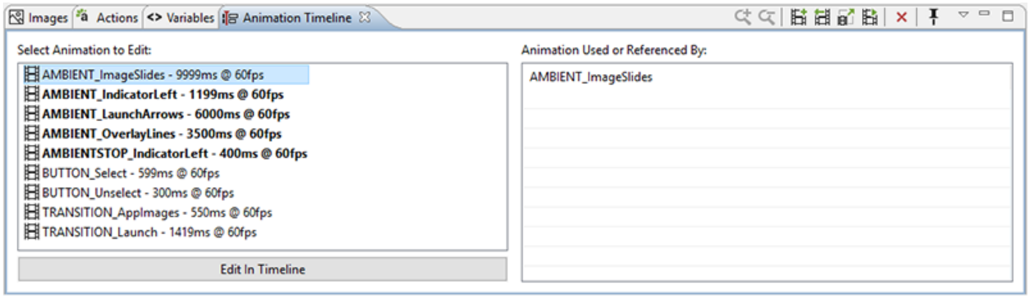
Since layers are displayed as layer instances, the tree will show layer and control content several times in the tree. If the Link with Editor toolbar option is enabled, when a model object is selected in the Application Model view the editor will automatically scroll to present the appropriate context 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.



Select Animation to Edit:

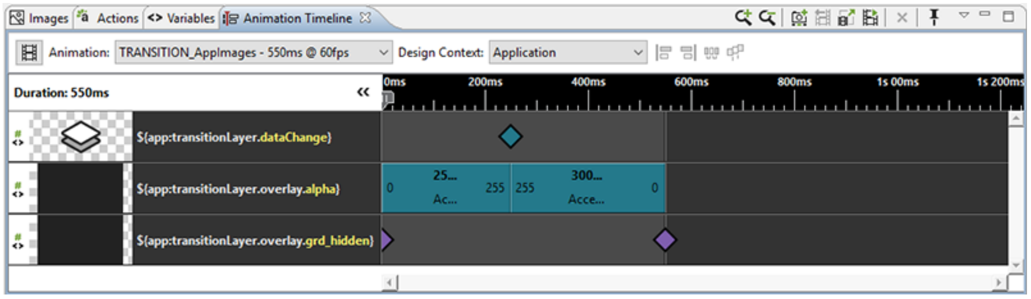
- AMBIENT_ImageSlides - 9999ms @ 60fps
- **AMBIENT_IndicatorLeft - 1199ms @ 60fps**
- **AMBIENT_LaunchArrows - 6000ms @ 60fps**
- **AMBIENT_OverlayLines - 3500ms @ 60fps**
- **AMBIENTSTOP_IndicatorLeft - 400ms @ 60fps**
- BUTTON_Select - 599ms @ 60fps
- BUTTON_Unselect - 300ms @ 60fps
- TRANSITION_AppImages - 550ms @ 60fps
- TRANSITION_Launch - 1419ms @ 60fps

Animation Used or Referenced By:

AMBIENT_ImageSlides

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.



Animation: TRANSITION_AppImages - 550ms @ 60fps
Design Context: Application
Duration: 550ms

${app:transitionLayer.dataChange}
${app:transitionLayer.overlay.alpha}
${app:transitionLayer.overlay.grd_hidden}

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.

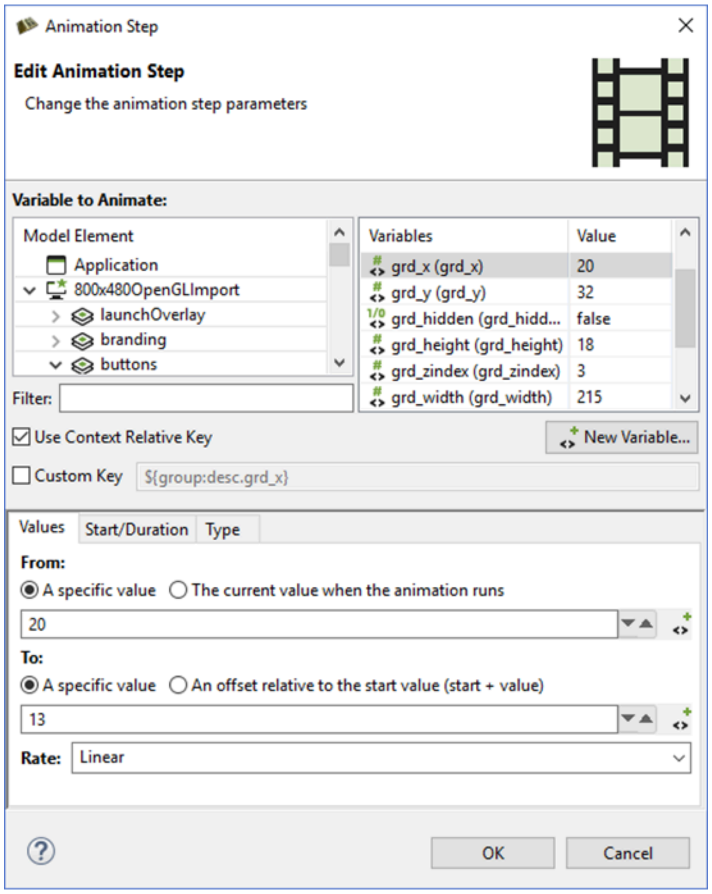
To easily line up the animation steps in the timeline, select multiple animation steps and use the align left, align right, distribute, or pack tools.

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.



Animation Step

Edit Animation Step

Change the animation step parameters

Variable to Animate:

Model Element
Application
800x480OpenGLImport
launchOverlay
branding
buttons

Variables	Value
grd_x (grd_x)	20
grd_y (grd_y)	32
grd_hidden (grd_hidd...	false
grd_height (grd_height)	18
grd_zindex (grd_zindex)	3
grd_width (grd_width)	215

Filter:

☑ Use Context Relative Key

☐ Custom Key ${group:desc.grd_x}

From:

● A specific value ○ The current value when the animation runs

20

To:

● A specific value ○ An offset relative to the start value (start + value)

13

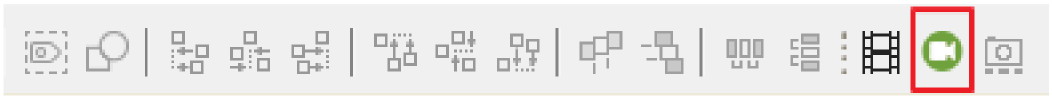
Rate: Linear

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.

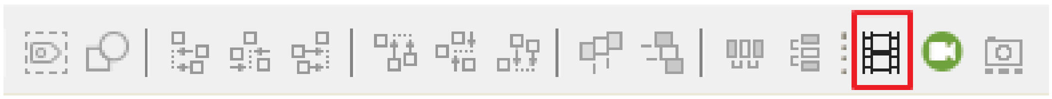
While recording, you have the ability to take snapshots. Snapshots cause subsequent steps created by changes during the recording to be offset by the duration of the previous step. This way, you can create a sequence of consecutive animation steps rather than have them execute simultaneously. The snapshot action is located right next to the recording action, and will only be enabled if you are currently recording.

Note

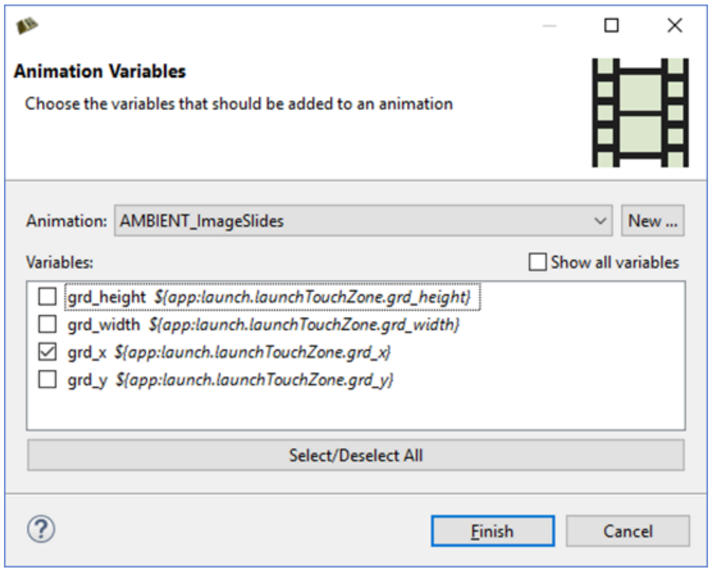
When using the animation recorder, any structural changes made to the model will be automatically reverted and lost at the end of the recording. The model recording only tracks changes to existing model properties and variables, creating variables automatically for any static elements that change.

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.



Animation Variables

Choose the variables that should be added to an animation

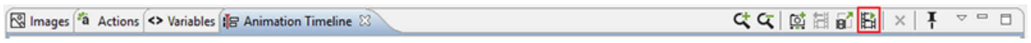
Animation: AMBIENT_ImageSlides

Variables:

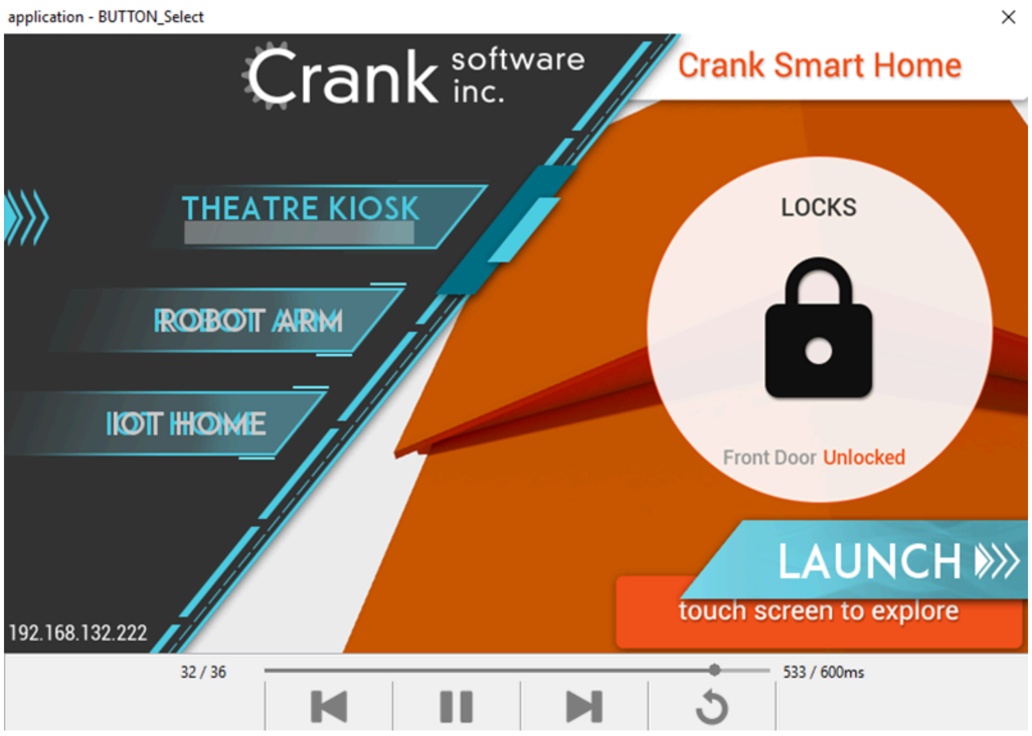
- [ ] grd_height *${app:launch.launchTouchZone.grd_height}*
- [ ] grd_width *${app:launch.launchTouchZone.grd_width}*
- [x] grd_x *${app:launch.launchTouchZone.grd_x}*
- [ ] grd_y *${app:launch.launchTouchZone.grd_y}*

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



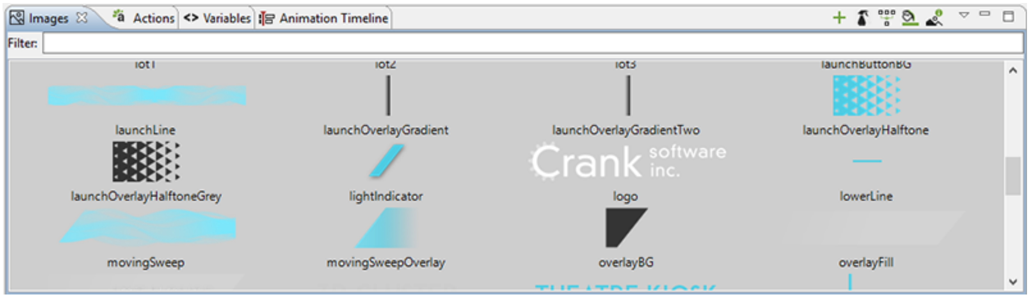
Preview the currently selected animation by clicking the animation preview action. An uneditable version of the application is displayed in a dialog and runs the animation. The bottom bar contains actions that control the preview. Play/Pause will stop and restart the animation from the current frame. 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.

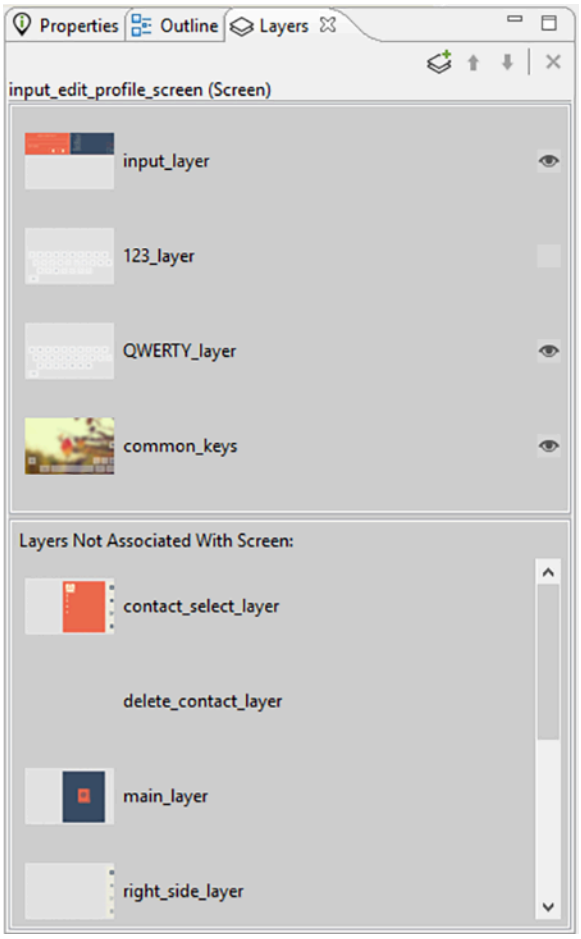
All files contained within the `images` folder are processed as images, regardless of their file extension. If an unrecognized image file is encountered, by default it will not be displayed. This behavior can be adjusted by deselecting `Show Only Images` in the view's drop-down menu. This menu also provides 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.



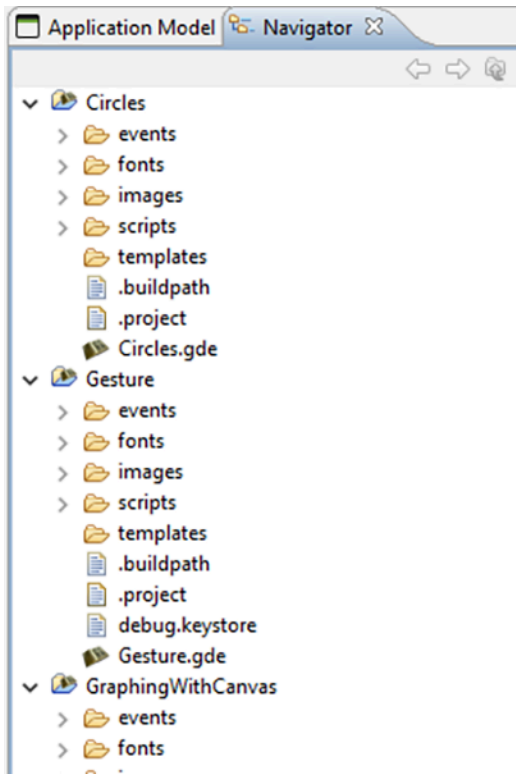
Layers are categorized as either being included on the current screen or not. Layers can be added or removed from the current screen by dragging and dropping them either into or out of the current screen area, or by using the delete key.

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.

Navigator View

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

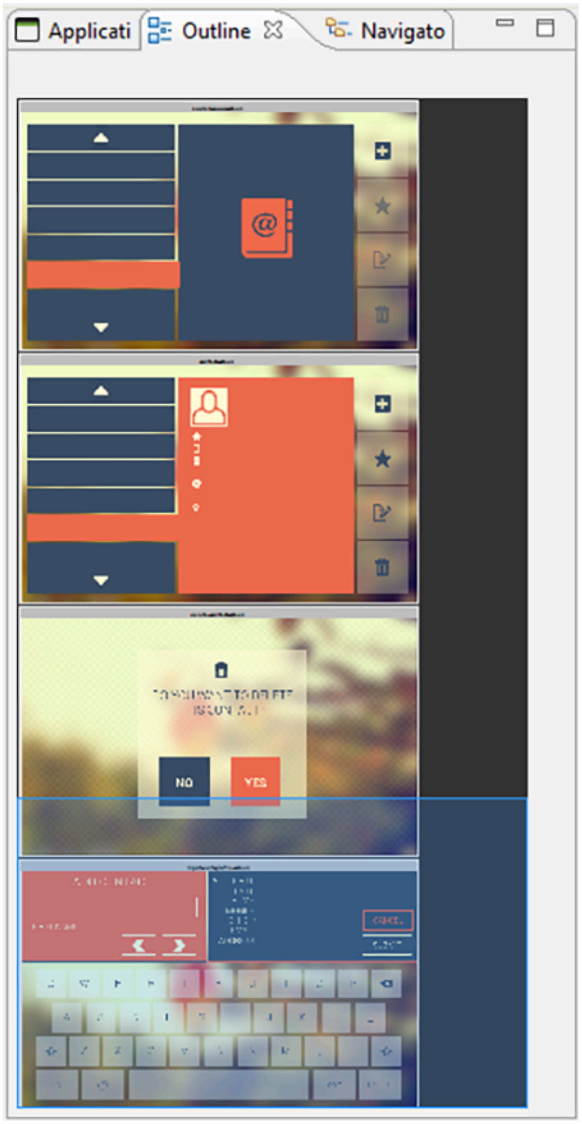


The Navigator view only displays content that has been imported into the workspace and starts with the top level Storyboard Project directories that have been created. Since the workspace allows multiple Storyboard projects to be shown, it is possible to work on multiple projects concurrently all within 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.

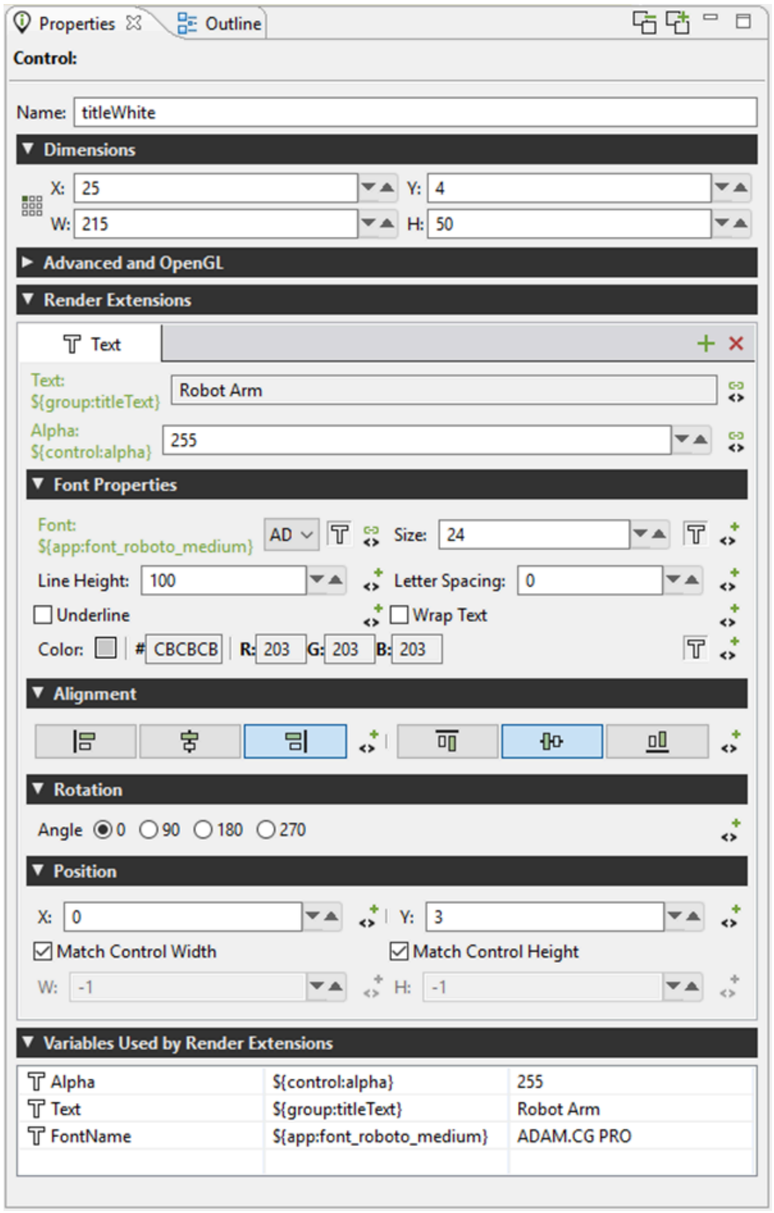
- Fill hides content. If a fill masks on top of other content it can be an ineffective use of processing resources on target devices.

- 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.



Properties | Outline

Control:

Name: titleWhite

▼ Dimensions

X: 25 Y: 4
W: 215 H: 50

► Advanced and OpenGL

▼ Render Extensions

Text

Text: ${group:titleText} — Robot Arm
Alpha: ${control:alpha} — 255

▼ Font Properties

Font: ${app:font_roboto_medium} — AD — Size: 24
Line Height: 100 — Letter Spacing: 0
☐ Underline — ☐ Wrap Text
Color: # CBCBCB R: 203 G: 203 B: 203

▼ Alignment

▼ Rotation

Angle [✓ 0] 90 180 270

▼ Position

X: 0 Y: 3
☑ Match Control Width ☑ Match Control Height
W: -1 H: -1

▼ Variables Used by Render Extensions

Alpha	${control:alpha}	255
Text	${group:titleText}	Robot Arm
FontName	${app:font_roboto_medium}	ADAM.CG PRO

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 it's 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.

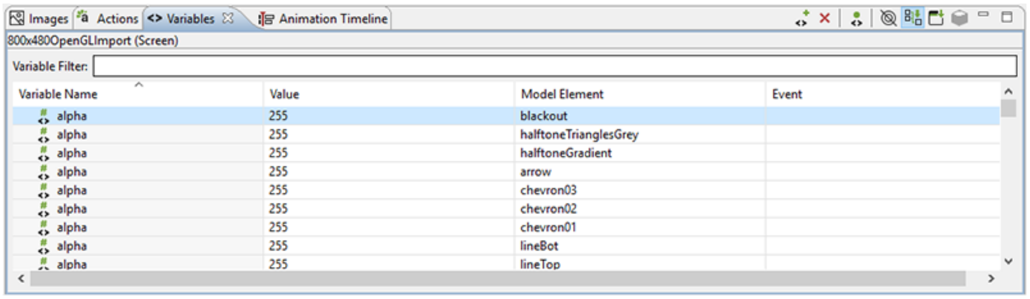
When a component is selected, its description, along with a graphical preview of the component, are shown if available. To create a new model object based on that component, select one of the components in the list and drag and drop it into your application.

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 22, *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.



800x480OpenGLImport (Screen)

Variable Filter:

Variable Name	Value	Model Element	Event
alpha	255	blackout	
alpha	255	halftoneTrianglesGrey	
alpha	255	halftoneGradient	
alpha	255	arrow	
alpha	255	chevron03	
alpha	255	chevron02	
alpha	255	chevron01	
alpha	255	lineBot	
alpha	255	lineTop	

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.

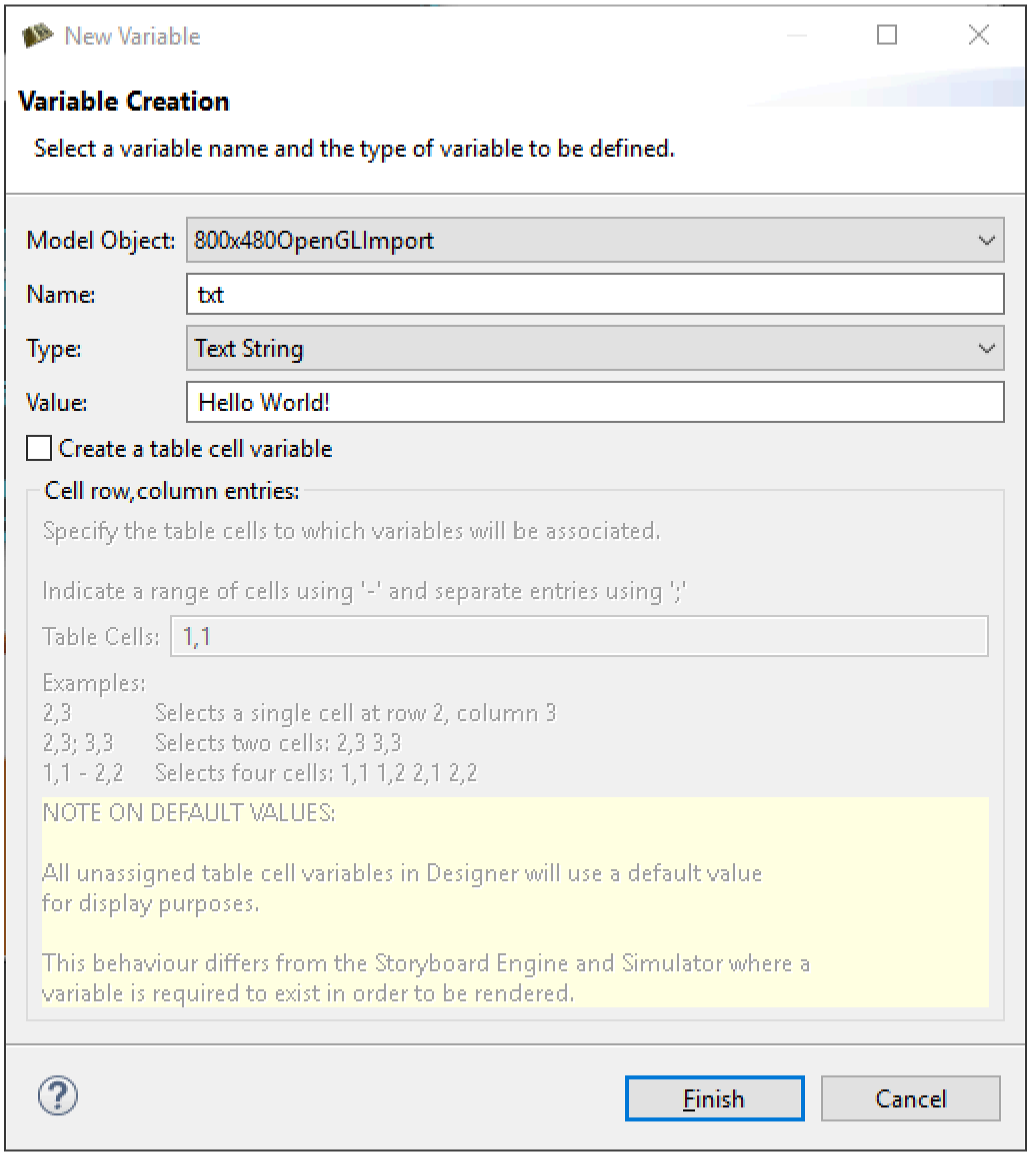
The content of the variable list can be sorted by selecting the appropriate variable table title and the variable values can be edited inline in the variable list by double-clicking on the appropriate field that you wish to change.

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

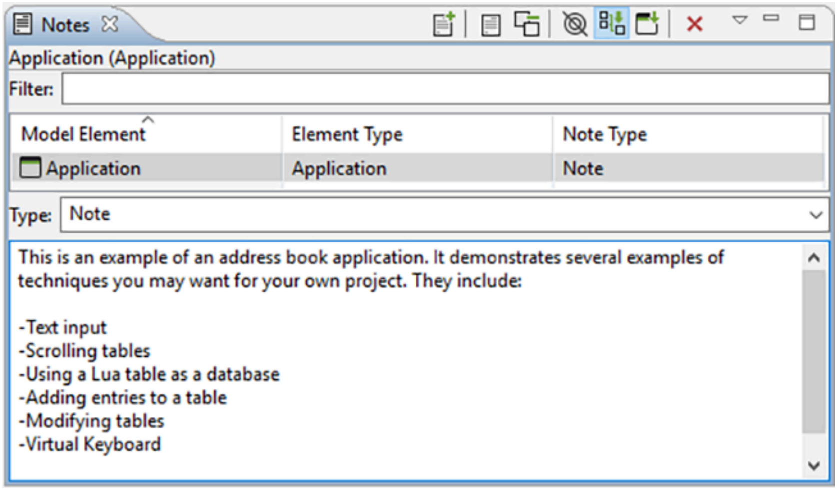
It is possible to associate a user-defined event to be generated when a variable's value is changed. These events are designed to facilitate the synchronization of user interface elements that may not be directly associated with the variables whose data is changing. A typical scenario would be to monitor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



Model Element	Element Type	Note Type
Application	Application	Note

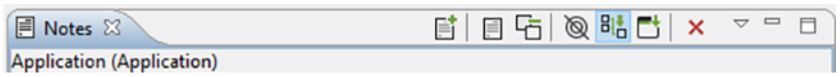
This is an example of an address book application. It demonstrates several examples of techniques you may want for your own project. They include:

-Text input
-Scrolling tables
-Using a Lua table as a database
-Adding entries to a table
-Modifying tables
-Virtual Keyboard

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.

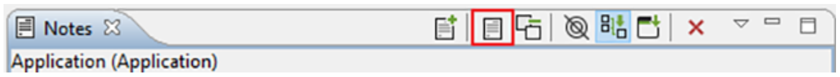


You can search for notes by typing in the filter text box. Sort the contents of the table by clicking on the header of one of the columns (e.g., clicking Model Element will sort the notes alphabetically by the name of the model object the note is attached to).

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).

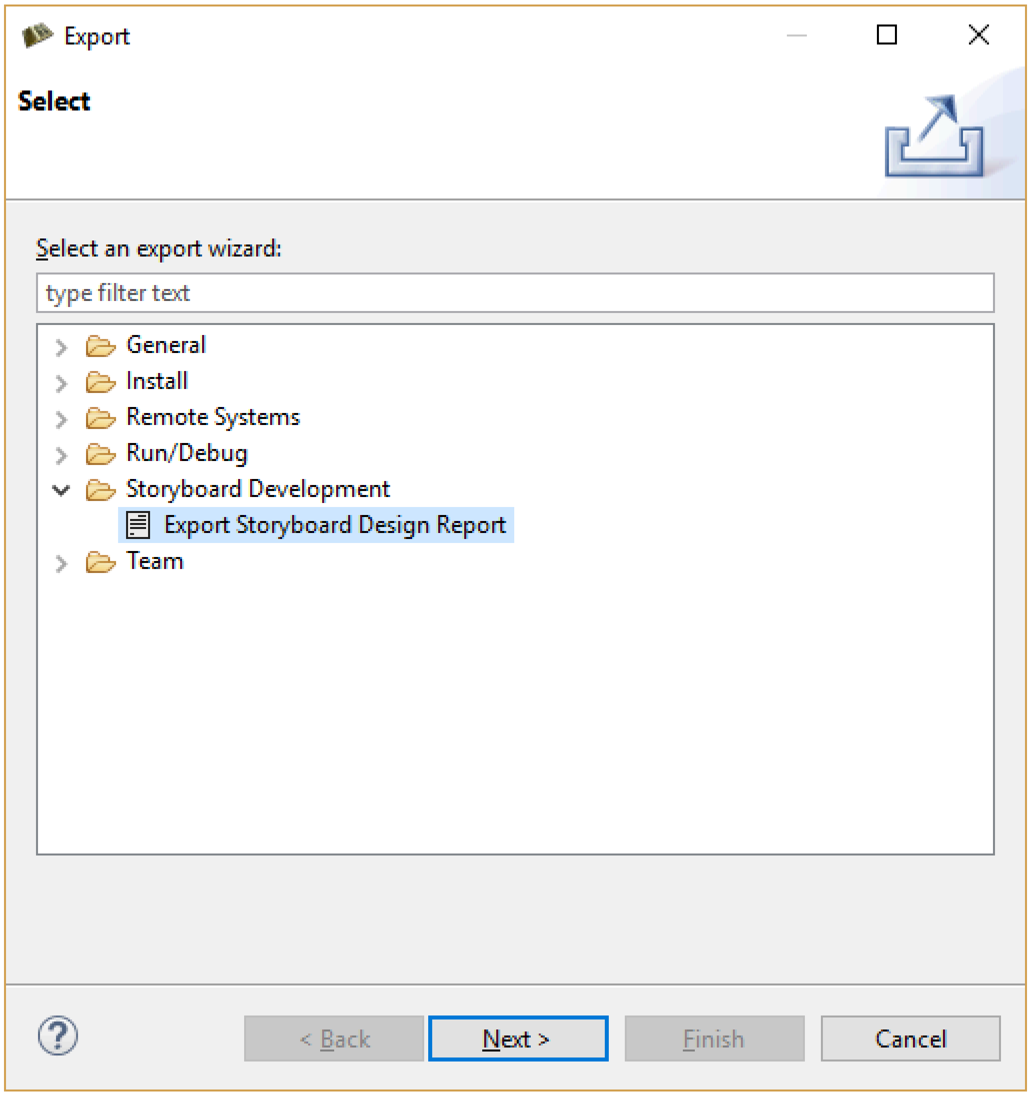


Storyboard Designer Utilities

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.

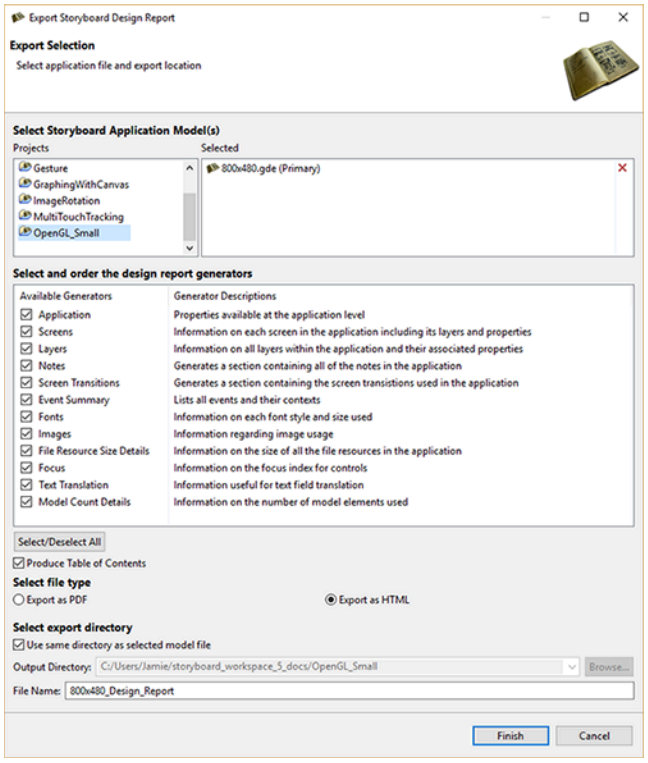
Design Notes

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.



Export Selection

Select application file and export location

Select Storyboard Application Model(s)

Projects	Selected
Gesture	800x480.gde (Primary)
GraphingWithCanvas	
ImageRotation	
MultiTouchTracking	
OpenGL_Small	

Select and order the design report generators

Available Generators	Generator Descriptions
✓ Application	Properties available at the application level
✓ Screens	Information on each screen in the application including its layers and properties
✓ Layers	Information on all layers within the application and their associated properties
✓ Notes	Generates a section containing all of the notes in the application
✓ Screen Transitions	Generates a section containing the screen transitions used in the application
✓ Event Summary	Lists all events and their contexts
✓ Fonts	Information on each font style and size used
✓ Images	Information regarding image usage
✓ File Resource Size Details	Information on the size of all the file resources in the application
✓ Focus	Information on the focus index for controls
✓ Text Translation	Information useful for text field translation
✓ Model Count Details	Information on the number of model elements used

✓ Produce Table of Contents

Select file type

Export as PDF [✓ Export as HTML]

Select export directory

✓ Use same directory as selected model file

Output Directory: C:/Users/Jamie/storyboard_workspace_5_docs/OpenGL_Small

File Name: 800x480_Design_Report

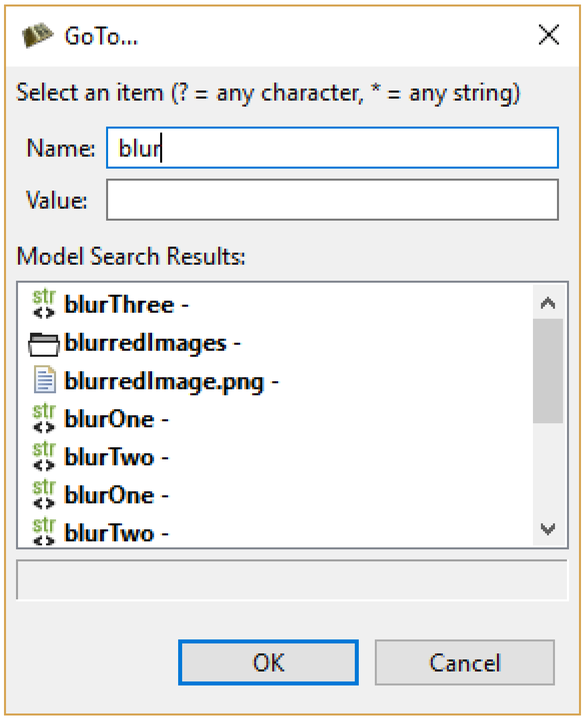
It is also possible to generate headless Design Note exports from the Storyboard Design files that can be used from a command line or scripting environment.

```
PATH_TO_INSTALL/Storyboard_Designer/storyboard/Storyboard -application
com.crank.gdt.designreport.designreport 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.



Select an item (? = any character, * = any string)

Name: blur

Value:

Model Search Results:

blurThree -
blurredImages -
blurredImage.png -
blurOne -
blurTwo -
blurOne -
blurTwo -

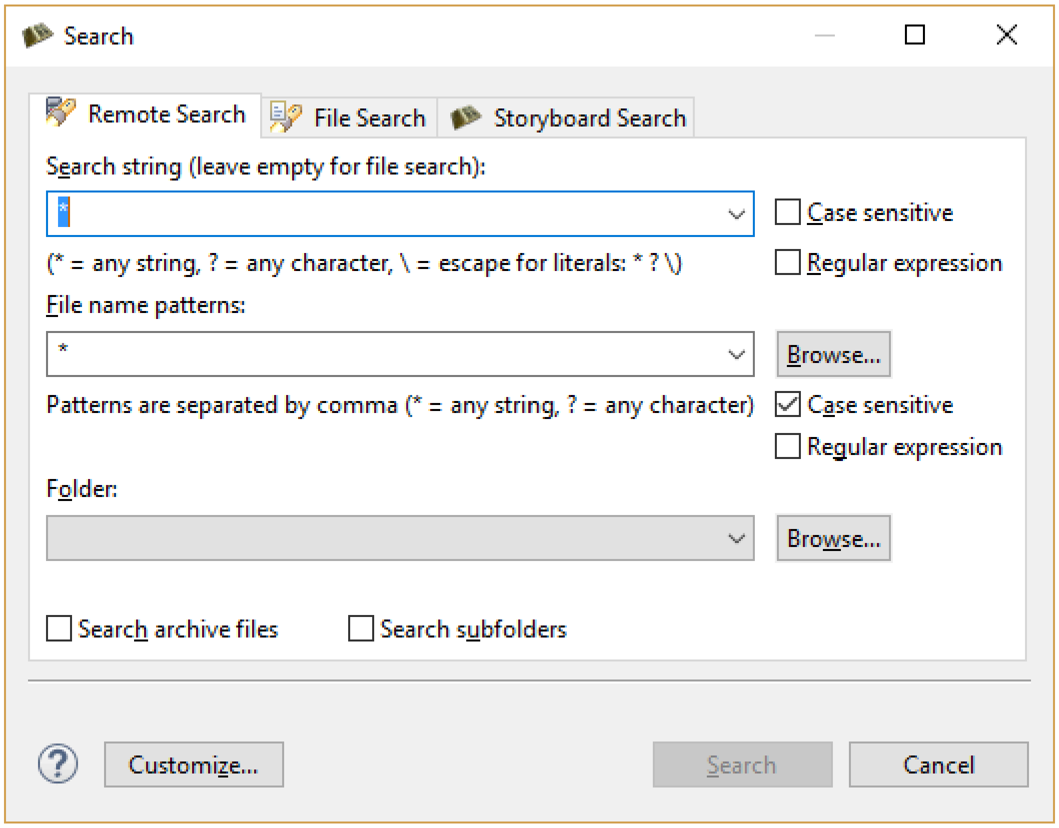
The Name and Value entries allow you to narrow the search criteria for the object you wish to find. The text entered here filters the results in the list to only display search results matching what you have typed:

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.



Search string (leave empty for file search):

*

(* = any string, ? = any character, \ = escape for literals: * ? \)

[ ] Case sensitive
[ ] Regular expression

File name patterns:

*

Patterns are separated by comma (* = any string, ? = any character)

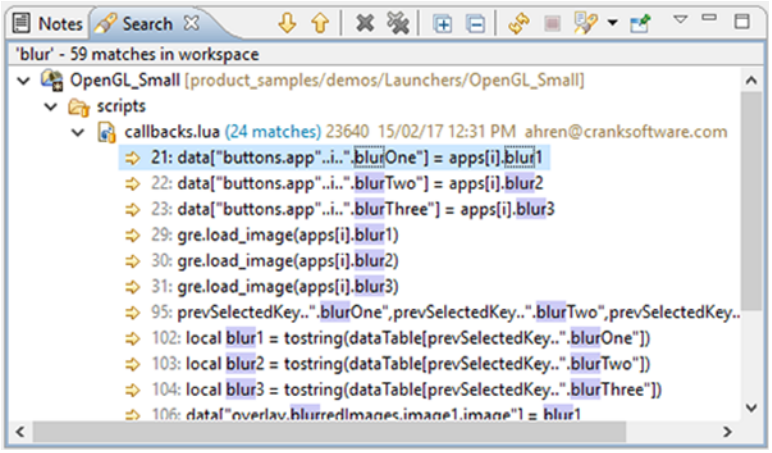
[✓] Case sensitive
[ ] Regular expression

Folder:

[ ] Search archive files
[ ] Search subfolders

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:



```
'blur' - 59 matches in workspace
OpenGL_Small [product_samples/demos/Launchers/OpenGL_Small]
  scripts
    callbacks.lua (24 matches) 23640  15/02/17 12:31 PM  ahren@cranksoftware.com
      21: data["buttons.app"..i..".blurOne"] = apps[i].blur1
      22: data["buttons.app"..i..".blurTwo"] = apps[i].blur2
      23: data["buttons.app"..i..".blurThree"] = apps[i].blur3
      29: gre.load_image(apps[i].blur1)
      30: gre.load_image(apps[i].blur2)
      31: gre.load_image(apps[i].blur3)
      95: prevSelectedKey..".blurOne",prevSelectedKey..".blurTwo",prevSelectedKey..
      102: local blur1 = tostring(dataTable[prevSelectedKey..".blurOne"])
      103: local blur2 = tostring(dataTable[prevSelectedKey..".blurTwo"])
      104: local blur3 = tostring(dataTable[prevSelectedKey..".blurThree"])
      106: data["overlay.blurredImages.image1.image"] = blur1
```

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

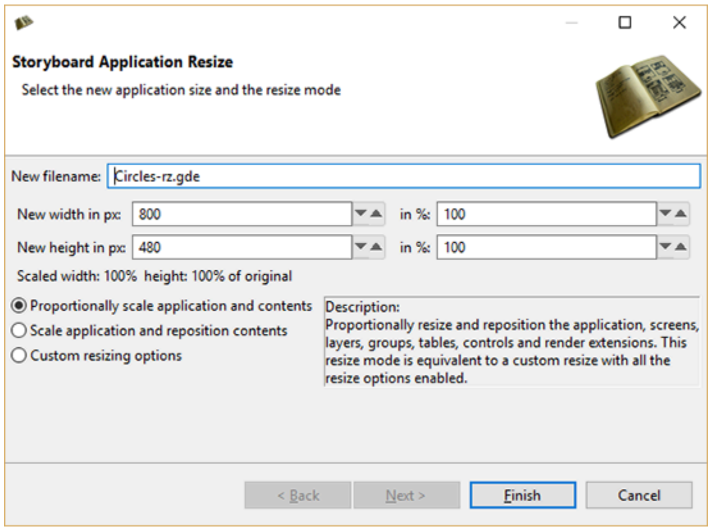
Resize Storyboard Application

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


Storyboard Application Resize

Select the new application size and the resize mode

New filename: Circles-rz.gde

New width in px: 800 in %: 100

New height in px: 480 in %: 100

Scaled width: 100% height: 100% of original

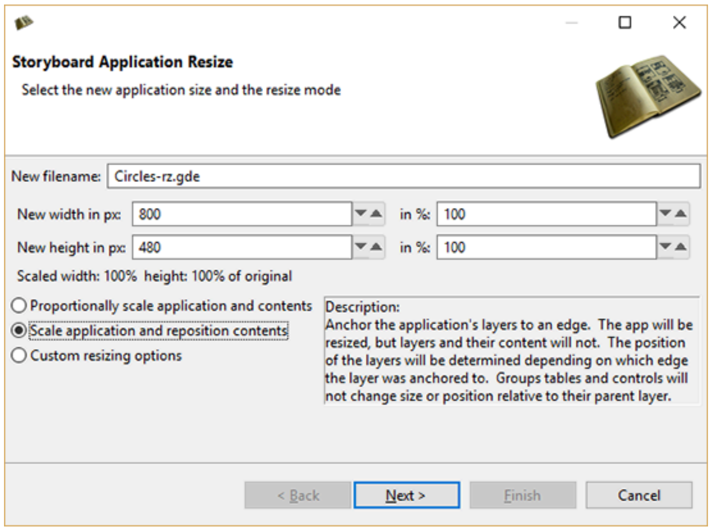
[✓ Proportionally scale application and contents]
Scale application and reposition contents
Custom resizing options

Description:
Proportionally resize and reposition the application, screens, layers, groups, tables, controls and render extensions. This resize mode is equivalent to a custom resize with all the resize options enabled.

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: Circles-rz.gde

New width in px: 800 in %: 100

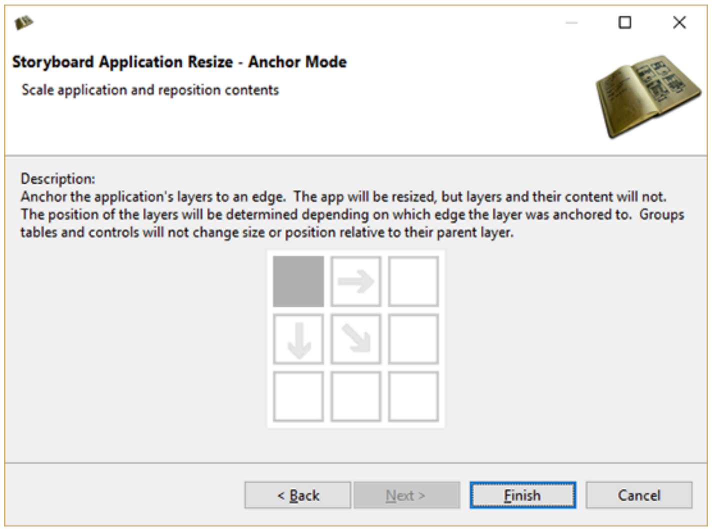
New height in px: 480 in %: 100

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.

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.

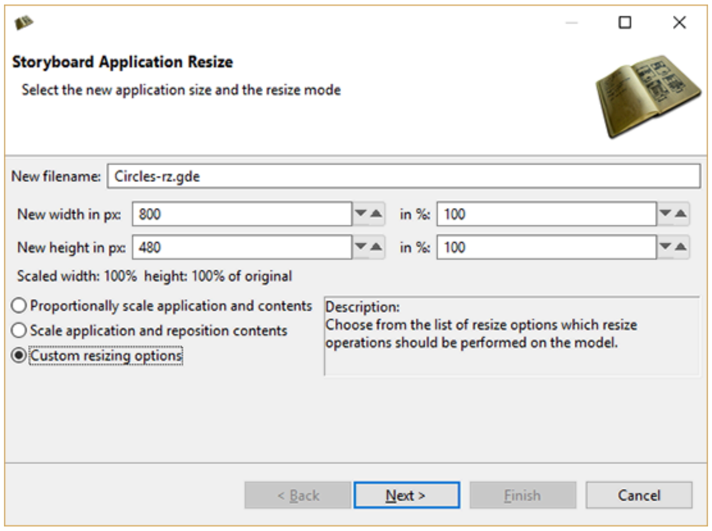

Storyboard Application Resize - Anchor Mode

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.

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: Circles-rz.gde

New width in px: 800 in %: 100

New height in px: 480 in %: 100

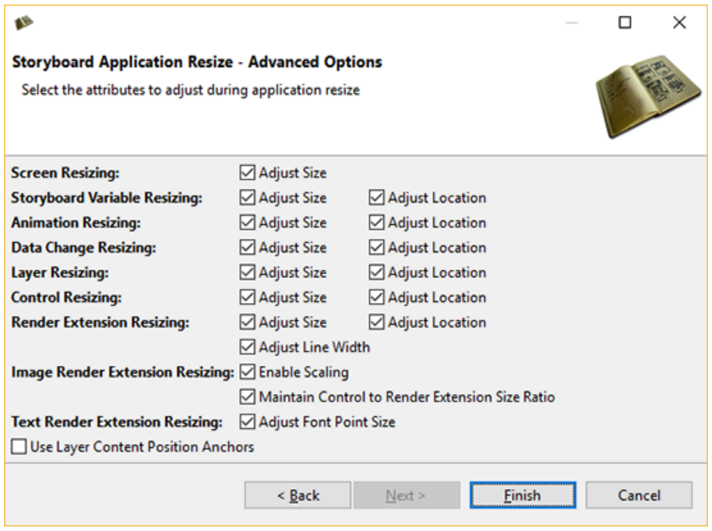
Scaled width: 100% height: 100% of original

○ 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.



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

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

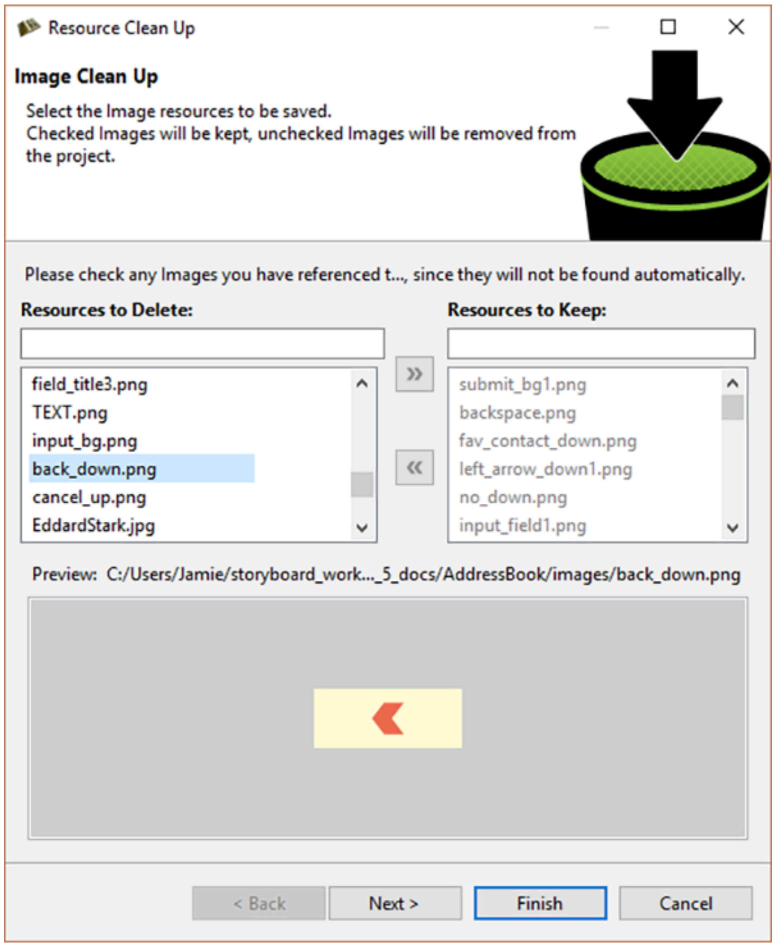
Adjust Font Point Size: Scales the text font size by the value of the smallest scale factor, be it that of height or that of width.

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.

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.


Resource Clean Up
Image Clean Up
Select the Image resources to be saved.
Checked Images will be kept, unchecked Images will be removed from the project.
Please check any Images you have referenced t..., since they will not be found automatically.
Resources to Delete:
field_title3.png
TEXT.png
input_bg.png
back_down.png
cancel_up.png
EddardStark.jpg
Resources to Keep:
submit_bg1.png
backspace.png
fav_contact_down.png
left_arrow_down1.png
no_down.png
input_field1.png
Preview: C:/Users/Jamie/storyboard_work..._5_docs/AddressBook/images/back_down.png
< Back
Next >
Finish
Cancel

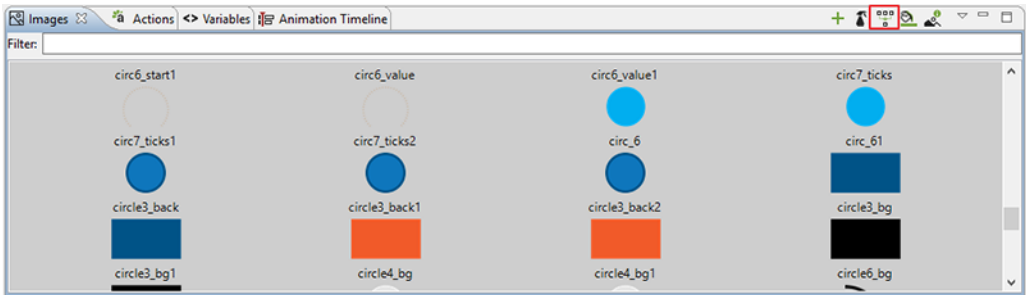
The Resource Clean Up wizard is used to remove resources that are present in the workspace but are no longer referenced by the project. In the wizard, the list on the left side contains the resources (fonts, images) and is used to select the resources that should be maintained/kept in the project. All 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.

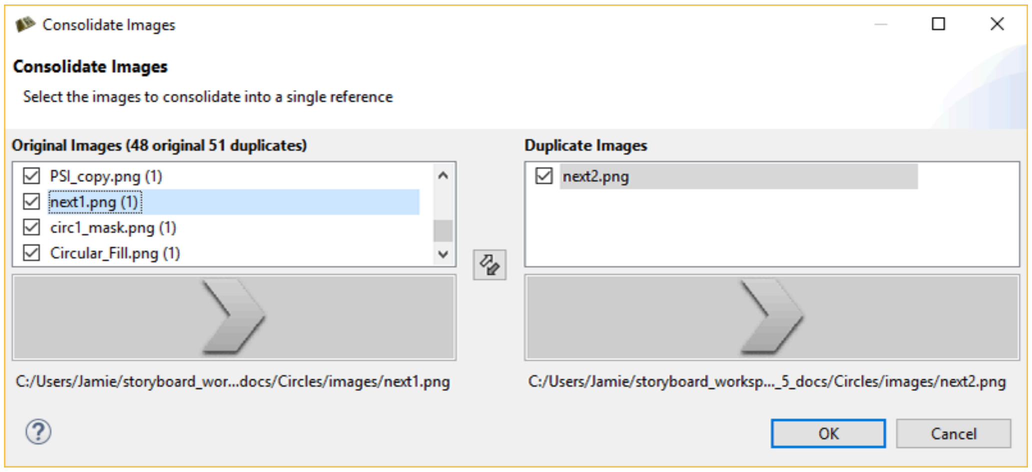
Once all of the resources to remove have been identified, selecting OK permanently removes those files from the file system.

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.



Consolidate Images

Consolidate Images

Select the images to consolidate into a single reference

Original Images (48 original 51 duplicates)

- ☑ PSI_copy.png (1)
- ☑ next1.png (1)
- ☑ circ1_mask.png (1)
- ☑ Circular_Fill.png (1)

Duplicate Images

- ☑ next2.png

C:/Users/Jamie/storyboard_wor...docs/Circles/images/next1.png

C:/Users/Jamie/storyboard_worksp..._5_docs/Circles/images/next2.png

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.

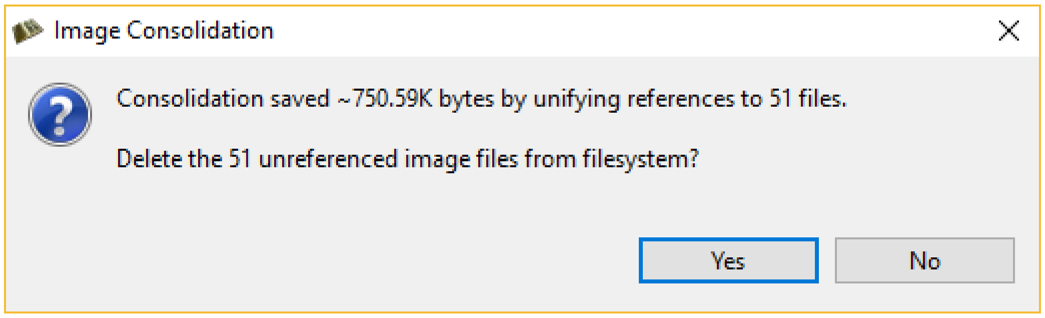


Image Consolidation

Consolidation saved ~750.59K bytes by unifying references to 51 files.

Delete the 51 unreferenced image files from filesystem?

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.

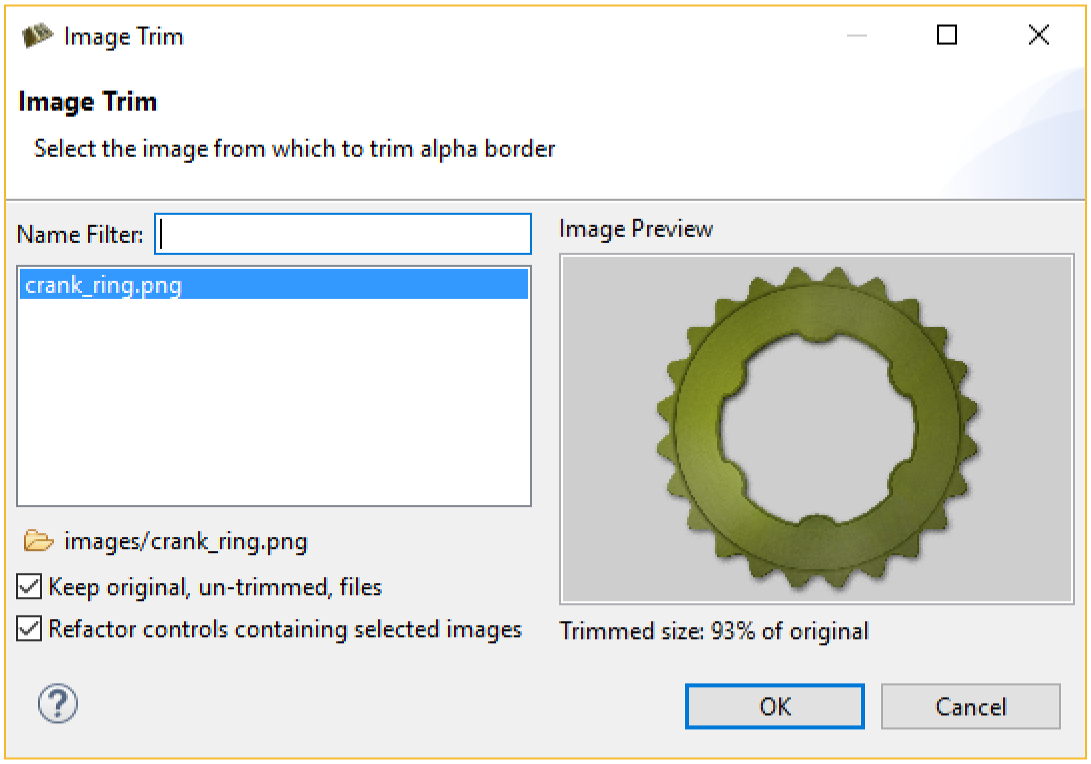


Image Trim
Image Trim
Select the image from which to trim alpha border
Name Filter:
crank_ring.png
Image Preview
images/crank_ring.png
Keep original, un-trimmed, files
Refactor controls containing selected images
Trimmed size: 93% of original
OK
Cancel

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.

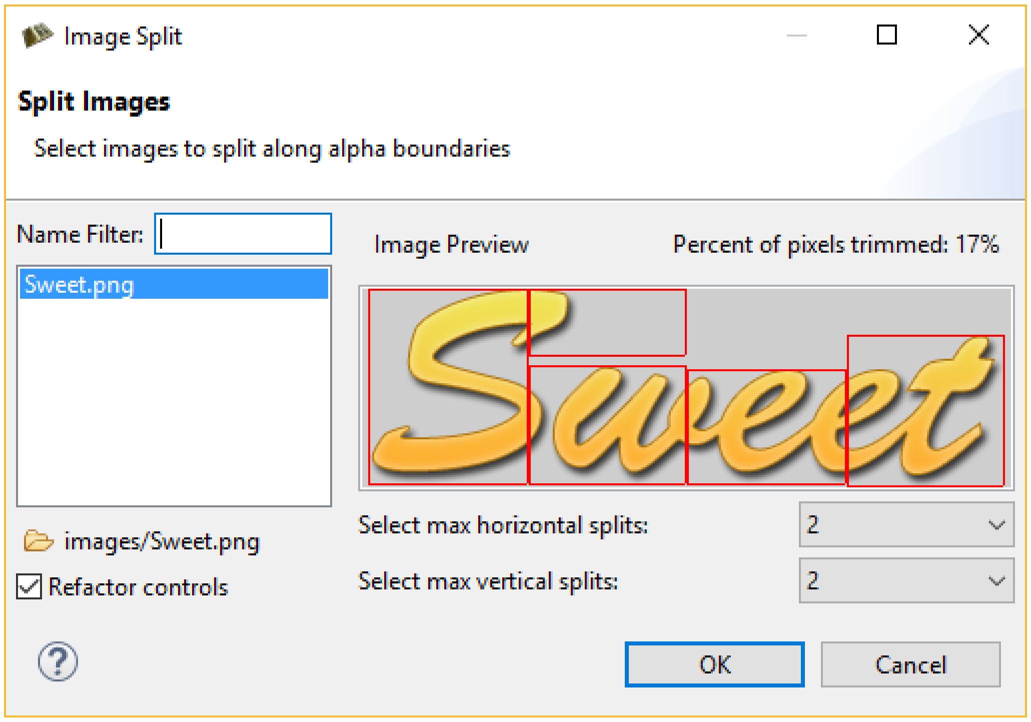


Image Split
Split Images
Select images to split along alpha boundaries
Name Filter:
Sweet.png
Image Preview
Percent of pixels trimmed: 17%
images/Sweet.png
Refactor controls
Select max horizontal splits: 2
Select max vertical splits: 2
OK
Cancel

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.

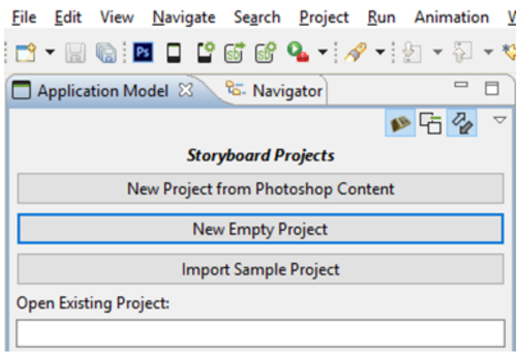
This can be used when you have one or more controls with static image content that will not change during runtime. It can be more efficient to flatten those render extensions into a single static image. The controls must be in the same layer for the wizard to work correctly.

Chapter 5. Creating A Storyboard Project

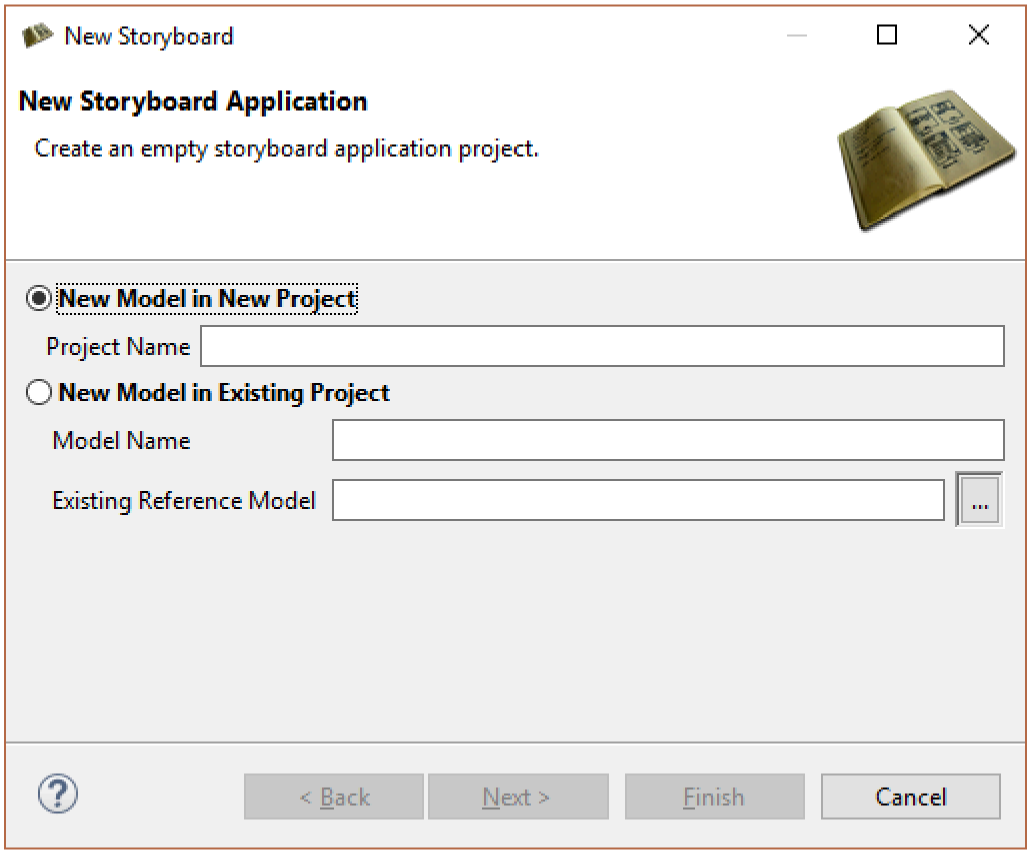
You can create a new Storyboard project in various ways: as a new empty application, as a Photoshop™ import, as an import of a sample, or from an existing Storyboard Designer project or Storyboard Engine *.gapp file.

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.


New Storyboard

New Storyboard Application

Create an empty storyboard application project.

(● New Model in New Project)

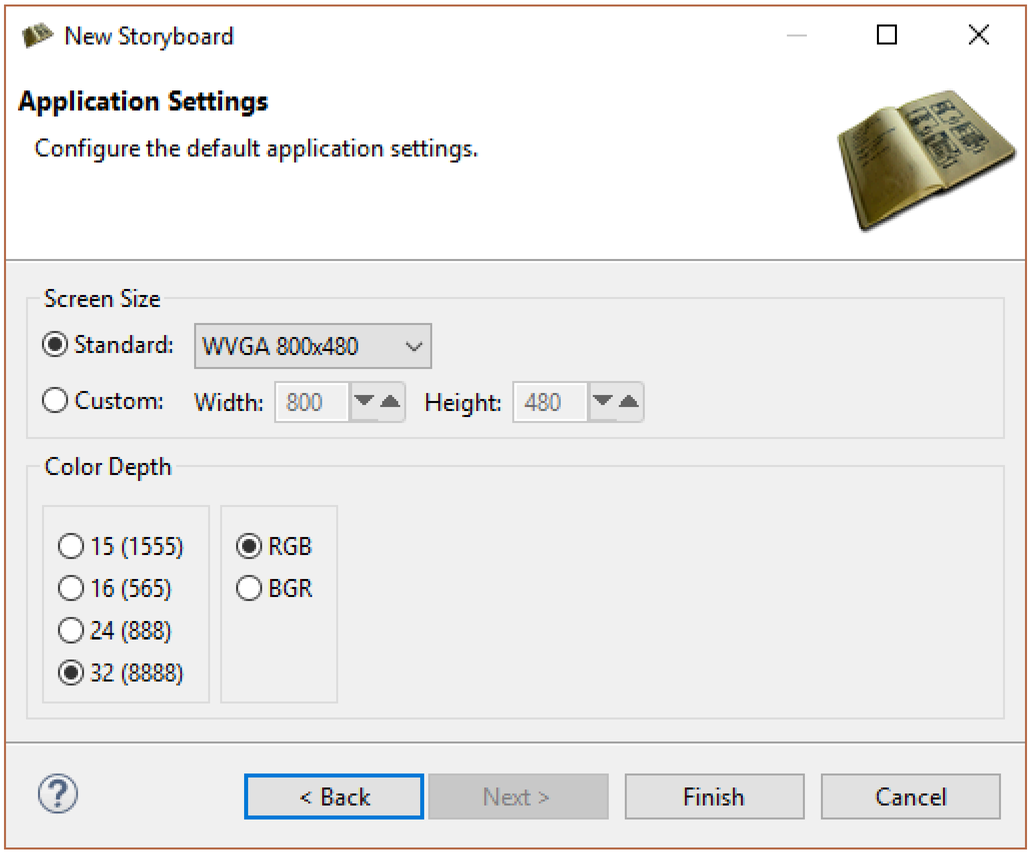
Project Name

○ New Model in Existing Project

Model Name

Existing Reference Model

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.


New Storyboard

Application Settings

Configure the default application settings.

Screen Size

[● Standard:] WVGA 800x480

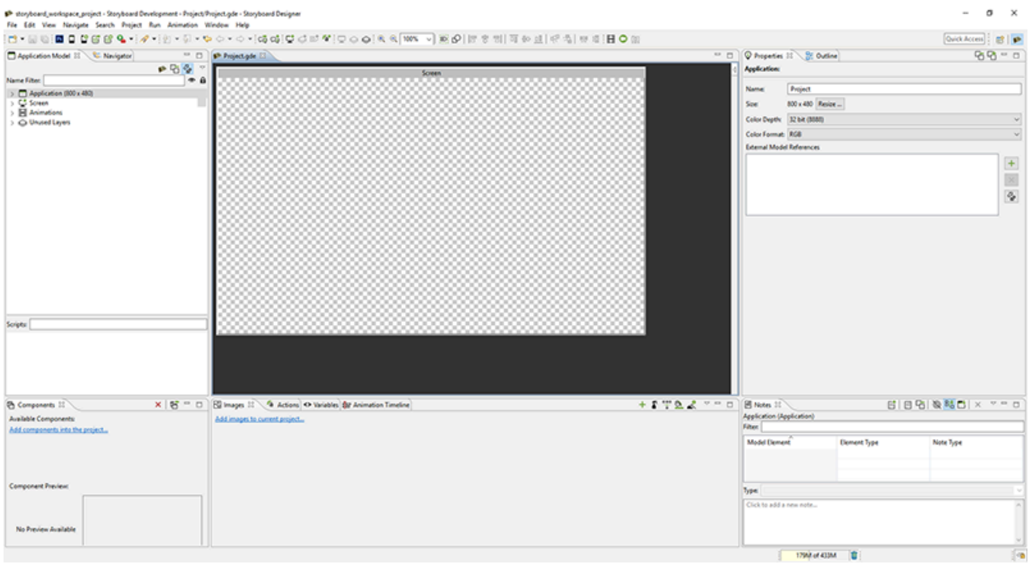
○ Custom: Width: 800 Height: 480

Color Depth

○ 15 (1555)
○ 16 (565)
○ 24 (888)
[● 32 (8888)]

[● RGB]
○ BGR

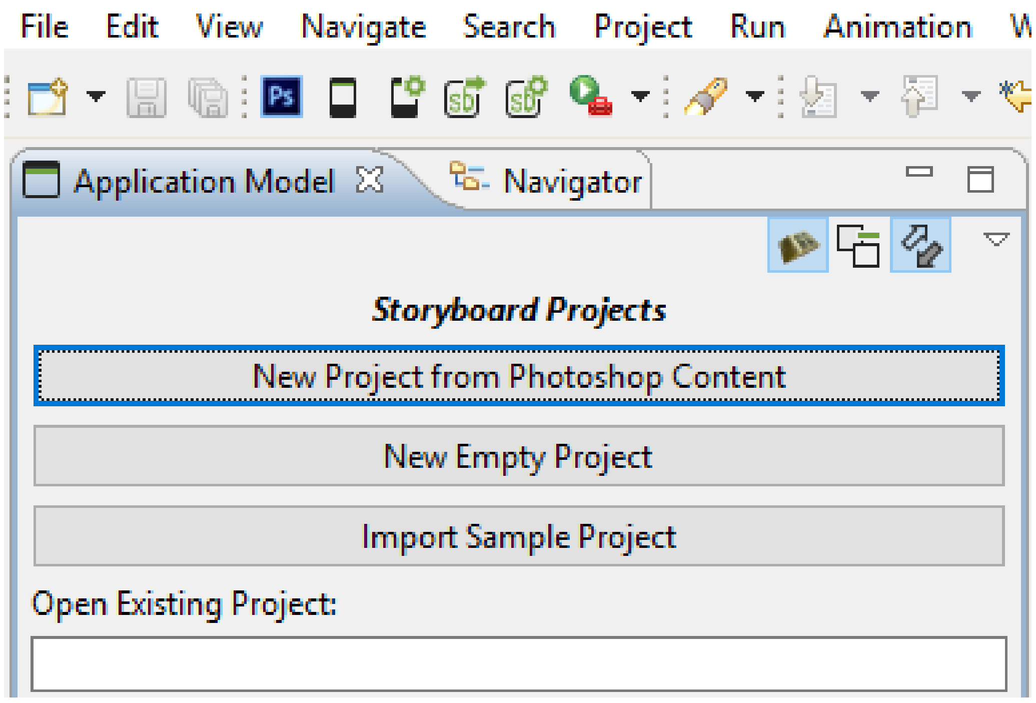
Select Finish to close the wizard, create the new project, and automatically open the Storyboard editor on the project:



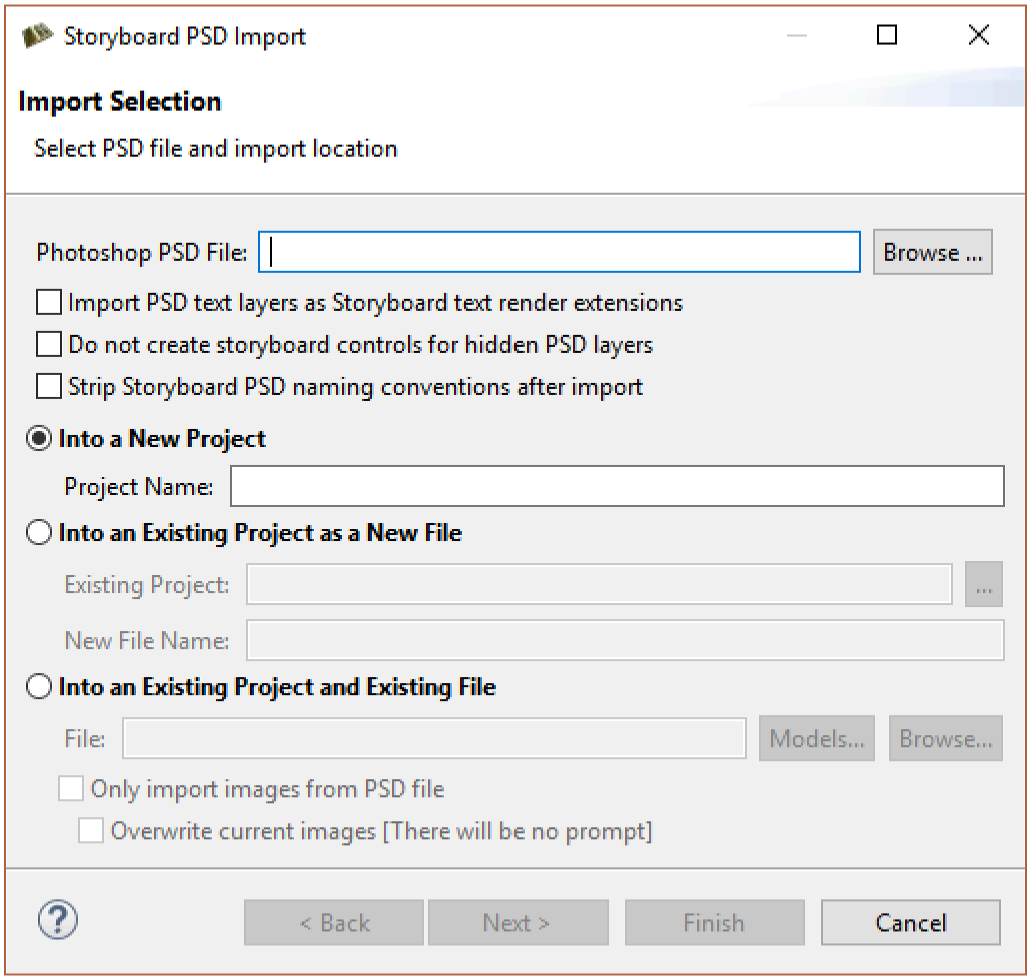
New Project from a Photoshop PSD File

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.


Storyboard PSD Import

Import Selection

Select PSD file and import location

Photoshop PSD File: [ ] Browse ...

[ ] Import PSD text layers as Storyboard text render extensions

[ ] Do not create storyboard controls for hidden PSD layers

[ ] Strip Storyboard PSD naming conventions after import

[● Into a New Project]

Project Name: [ ]

○ Into an Existing Project as a New File

Existing Project: [ ] ...

New File Name: [ ]

○ Into an Existing Project and Existing File

File: [ ] Models... Browse...

[ ] Only import images from PSD file

[ ] Overwrite current images [There will be no prompt]

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 displayed in the Layers view.

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

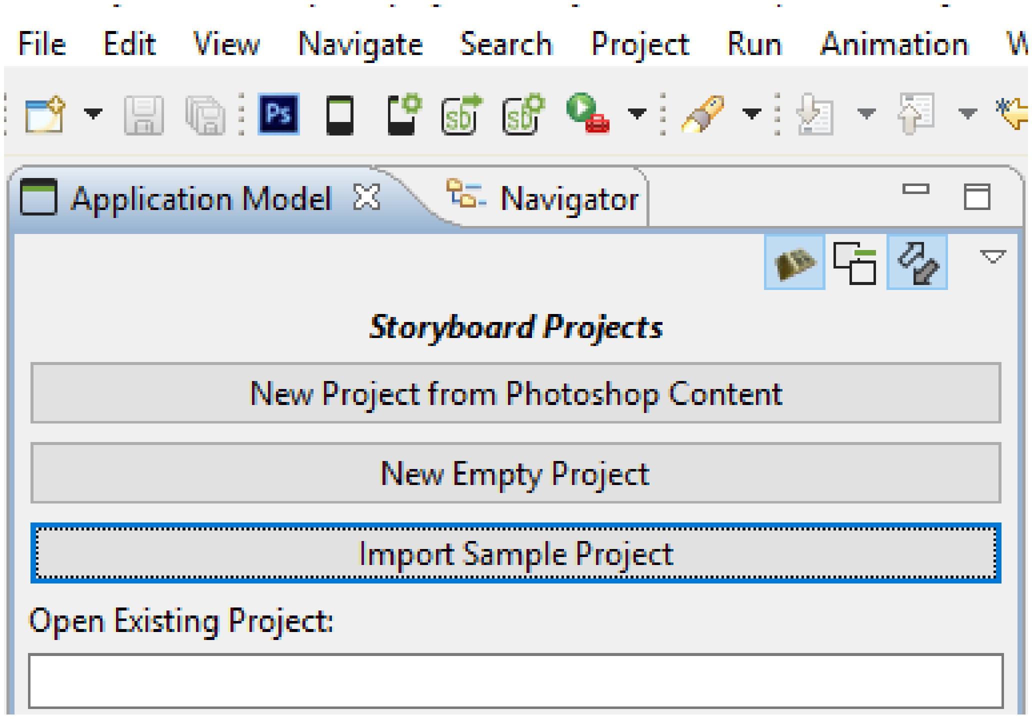
The Storyboard Suite installation contains a number of samples that you can use as reference for different UI development scenarios. These sample projects are generally not full-featured programs but are snippets designed to demonstrate how to use features of Storyboard including render extensions, actions,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

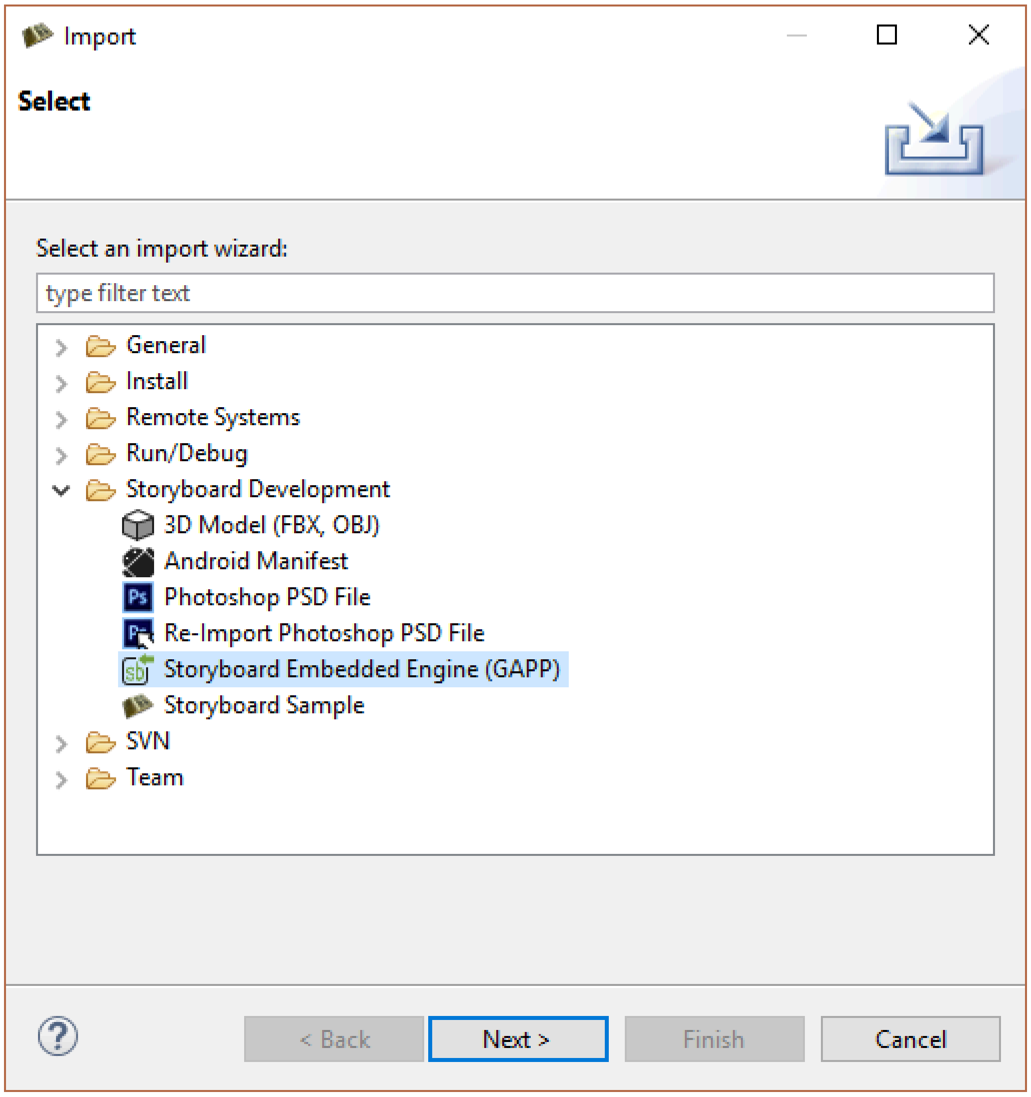
These options bring up a dialog allowing you to select one or more projects to import. The keyword filter allows you to search for projects that contain functionality that you may be interested in by searching both the project name as well as the project descriptions.



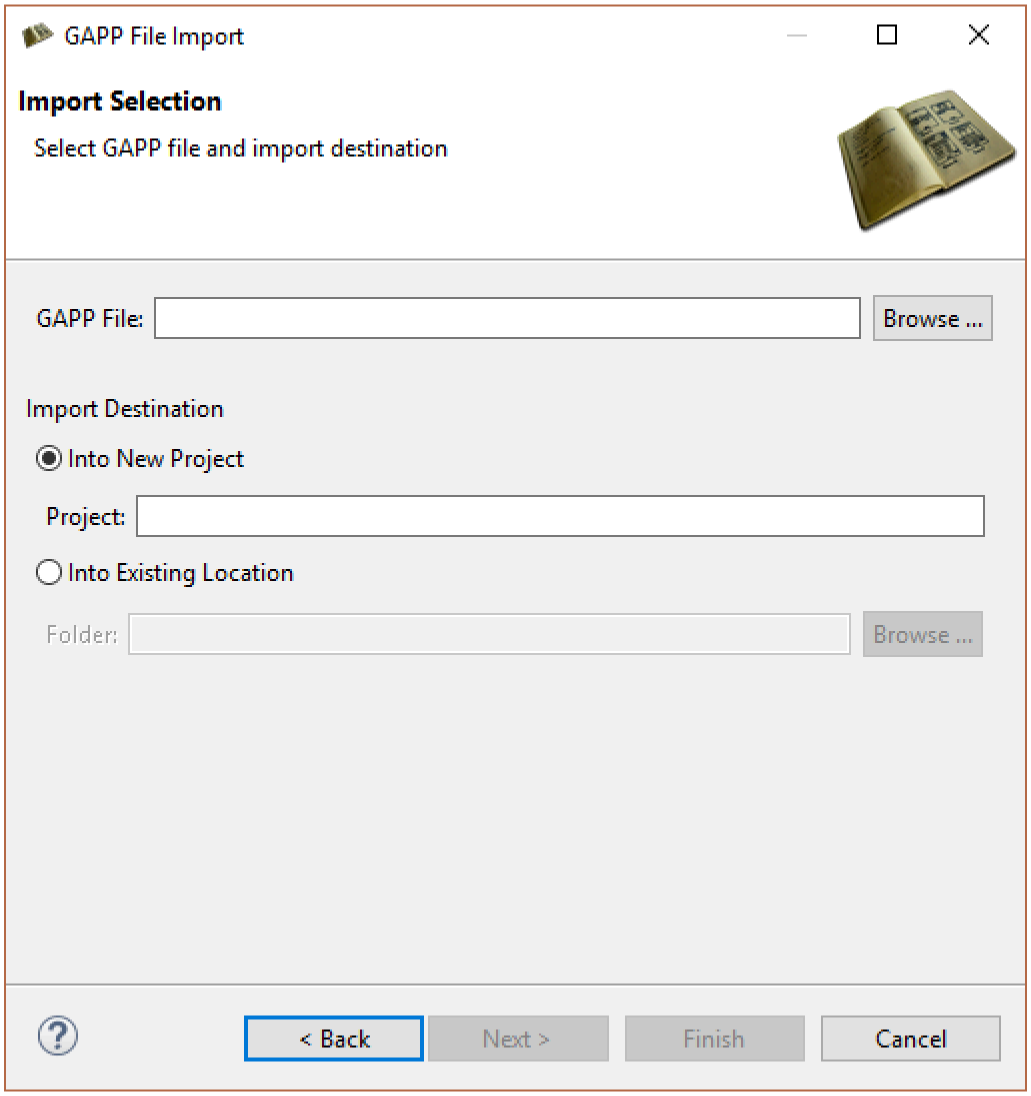
Once selected, a project will be created and the sample will automatically be opened into the Storyboard Designer editor.

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).



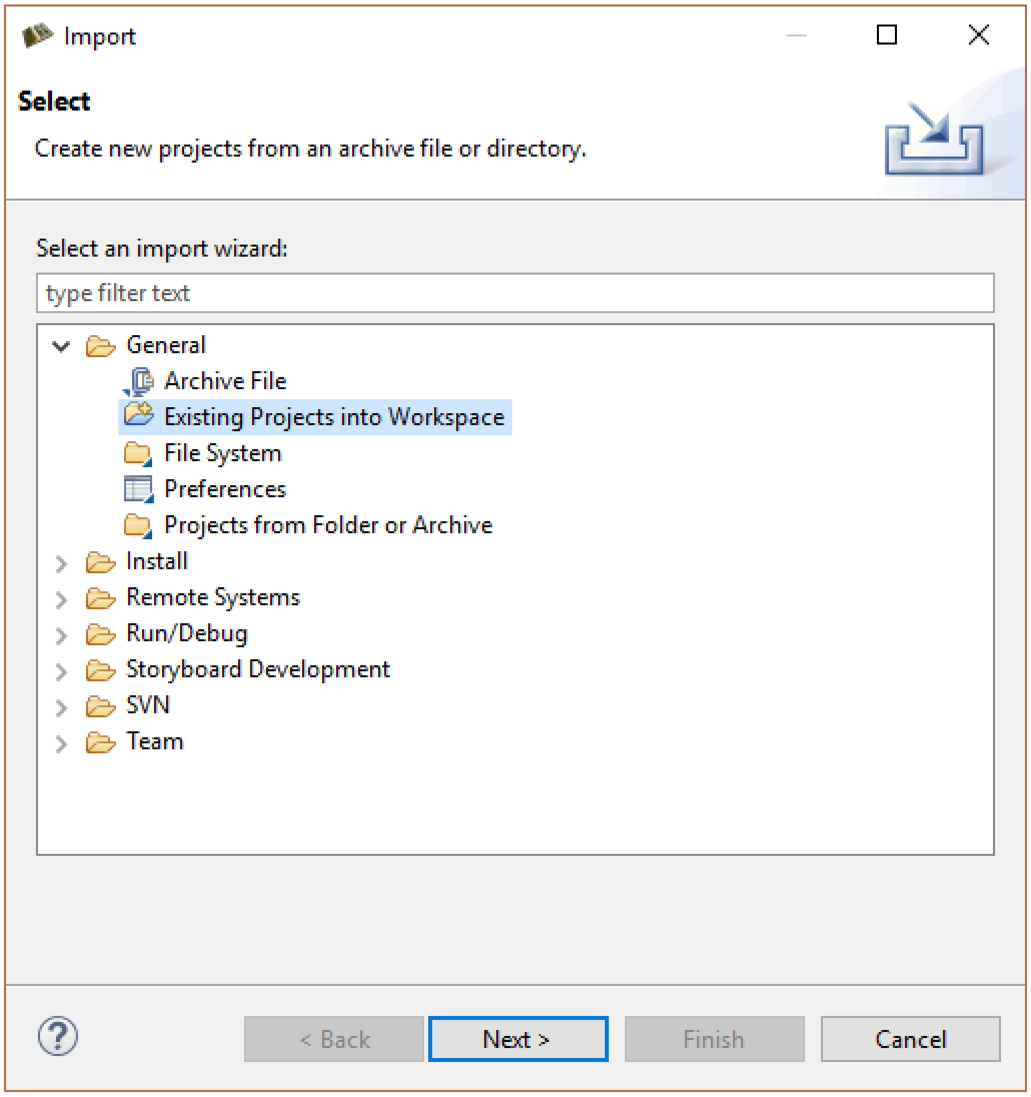
Select Next, and then select the embedded engine file to be used for import.



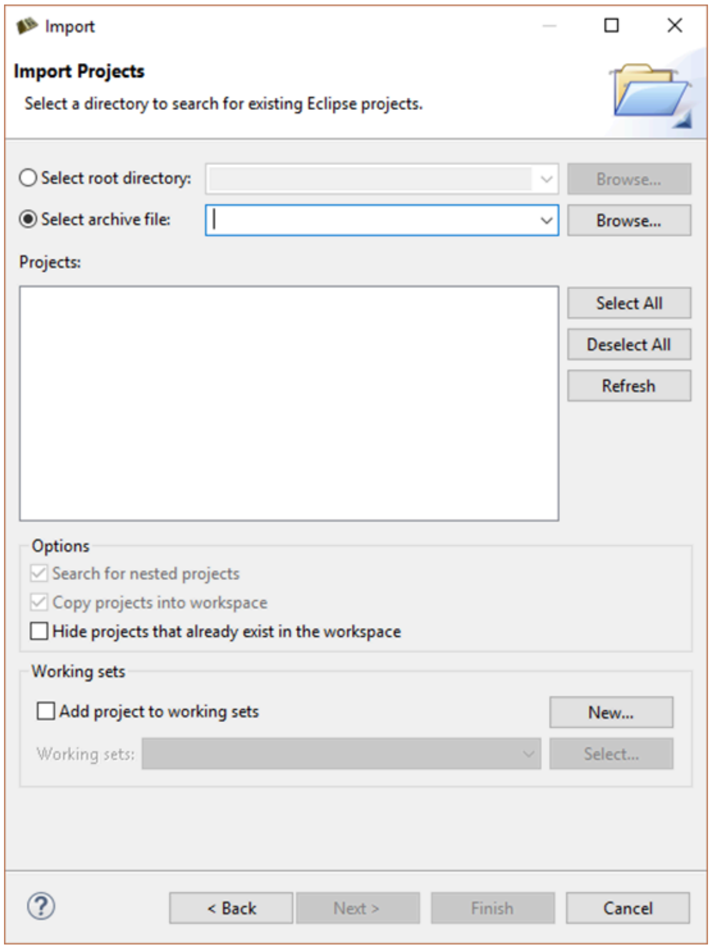
Existing Project Import

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.

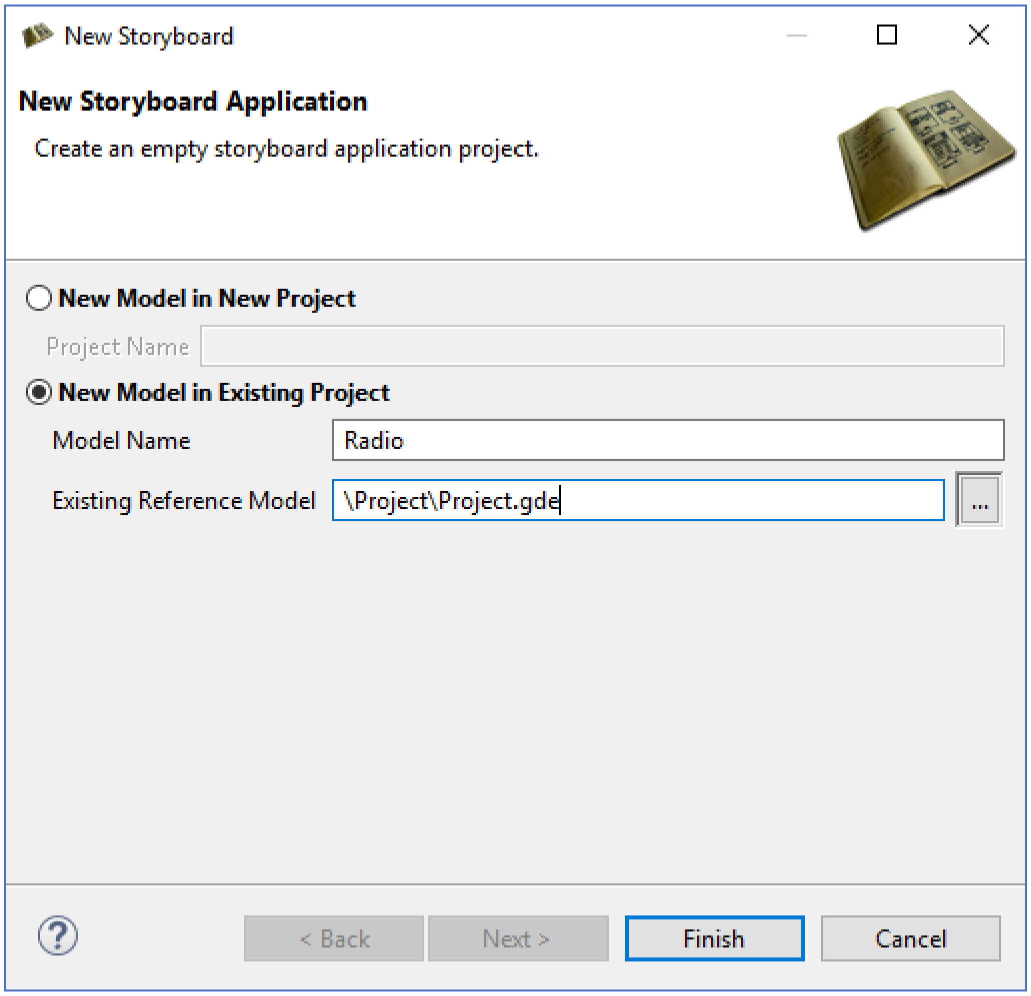
Once another application is required, for example to represent a distinctly themed area of the unified application, then you will want to create another Designer application model file within the base project.

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.



Create an empty storyboard application project.

( ) New Model in New Project

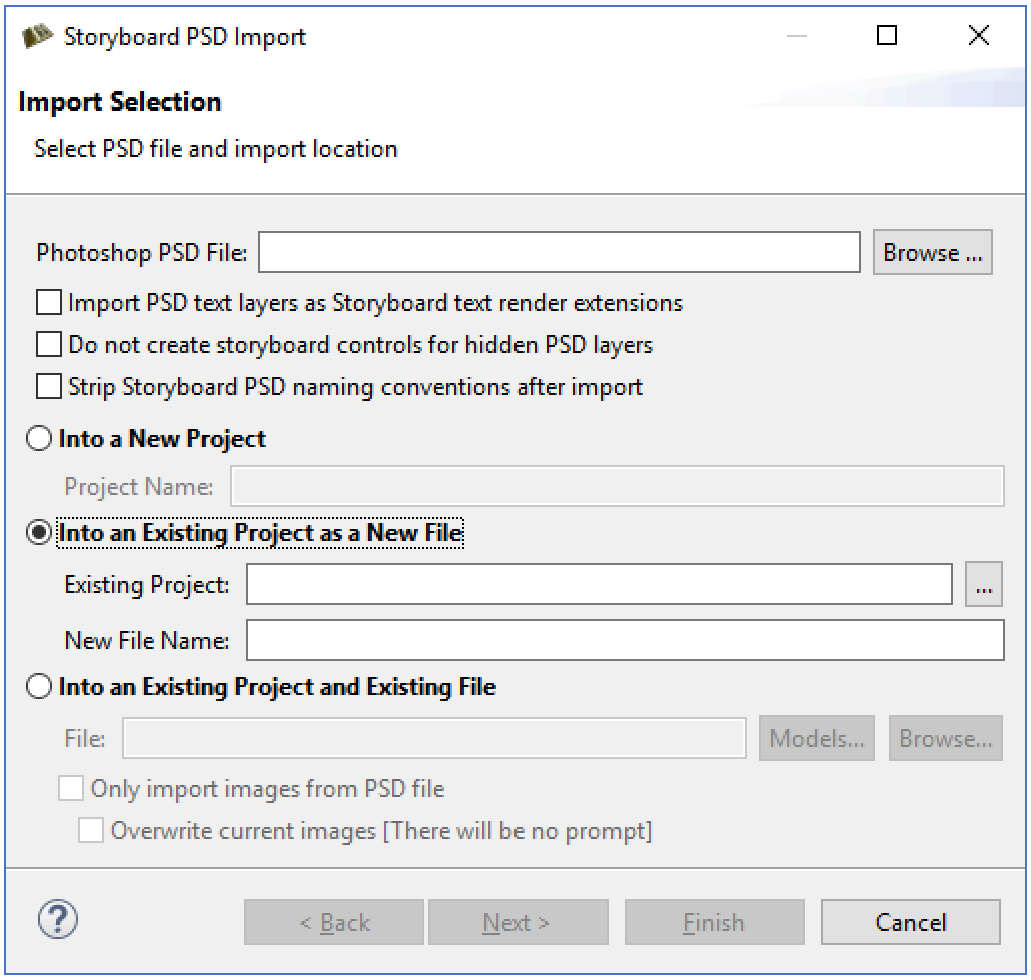
Project Name

(●) New Model in Existing Project

Model Name: Radio

Existing Reference Model: \Project\Project.gde

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



Storyboard PSD Import

Import Selection

Select PSD file and import location

Photoshop PSD File: Browse ...

☐ Import PSD text layers as Storyboard text render extensions

☐ Do not create storyboard controls for hidden PSD layers

☐ Strip Storyboard PSD naming conventions after import

○ Into a New Project

Project Name:

● Into an Existing Project as a New File

Existing Project: ...

New File Name:

○ Into an Existing Project and Existing File

File: Models... Browse...

☐ Only import images from PSD file

☐ Overwrite current images [There will be no prompt]

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 21, *Multi-File Application Development*.

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. Begin 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

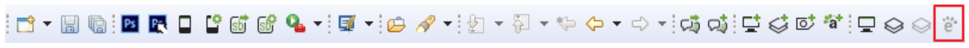
Event Editor

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

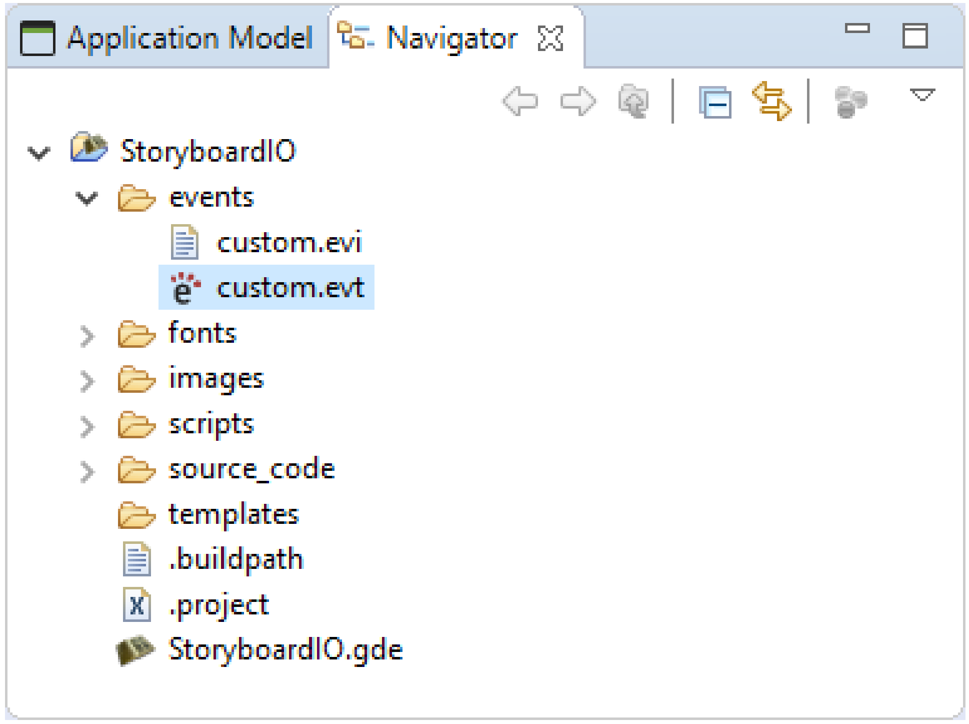
Opening the Event Editor

The different ways to open the Event Editor are:

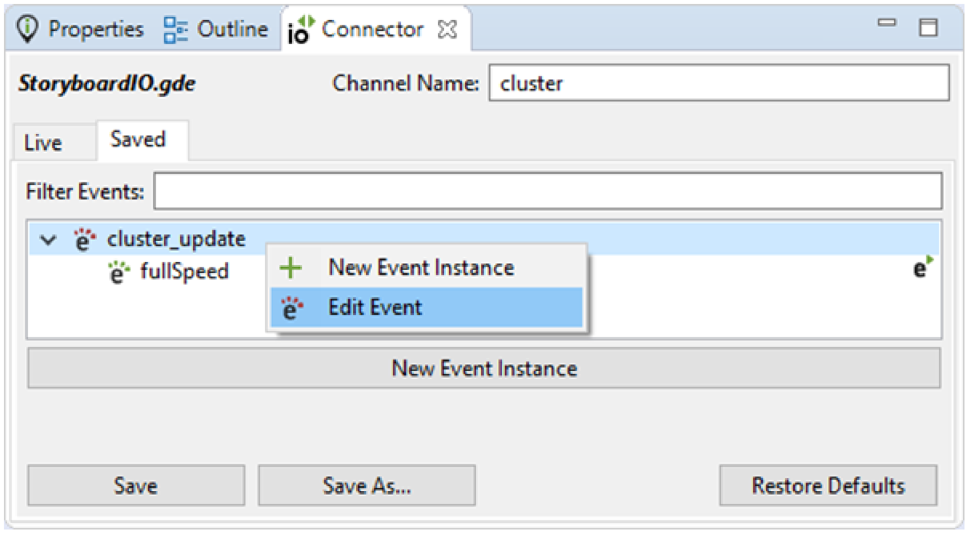
- From the main toolbar, with a Storyboard Editor open, click the Open Event Editor button.



- 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.



StoryboardIO.gde

Channel Name: cluster

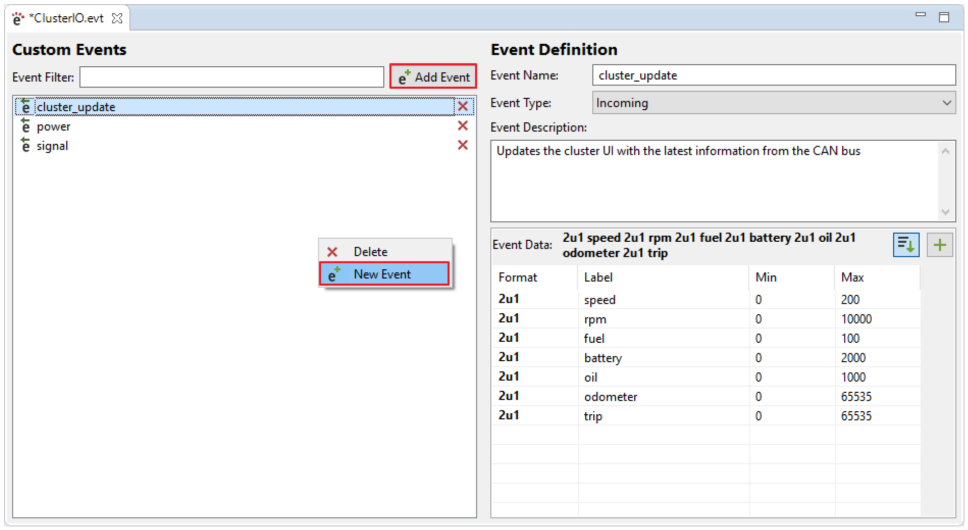
Filter Events:

- cluster_update
  - fullSpeed

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



Event Filter:

- cluster_update
- power
- signal

Event Name: cluster_update

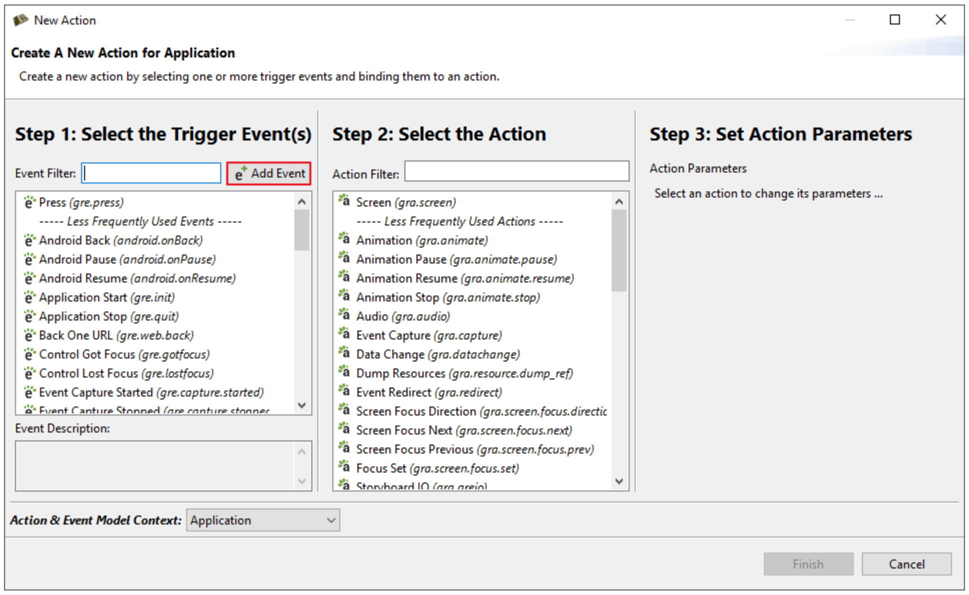
Event Type: Incoming

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	2000
2u1	oil	0	1000
2u1	odometer	0	65535
2u1	trip	0	65535

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



New Action

Create A New Action for Application

Create a new action by selecting one or more trigger events and binding them to an action.

Step 1: Select the Trigger Event(s)

Event Filter: Add Event

- Press *(gre.press)*
- ----- *Less Frequently Used Events* -----
- Android Back *(android.onBack)*
- Android Pause *(android.onPause)*
- Android Resume *(android.onResume)*
- Application Start *(gre.init)*
- Application Stop *(gre.quit)*
- Back One URL *(gre.web.back)*
- Control Got Focus *(gre.gotfocus)*
- Control Lost Focus *(gre.lostfocus)*
- Event Capture Started *(gre.capture.started)*
- Event Capture Stopped *(gre.capture.stopped*

Event Description:

Step 2: Select the Action

Action Filter:

- Screen *(gra.screen)*
- ----- *Less Frequently Used Actions* -----
- Animation *(gra.animate)*
- Animation Pause *(gra.animate.pause)*
- Animation Resume *(gra.animate.resume)*
- Animation Stop *(gra.animate.stop)*
- Audio *(gra.audio)*
- Event Capture *(gra.capture)*
- Data Change *(gra.datachange)*
- Dump Resources *(gra.resource.dump_ref)*
- Event Redirect *(gra.redirect)*
- Screen Focus Direction *(gra.screen.focus.directic*
- Screen Focus Next *(gra.screen.focus.next)*
- Screen Focus Previous *(gra.screen.focus.prev)*
- Focus Set *(gra.screen.focus.set)*
- Storyboard IO *(gra.greio)*

Step 3: Set Action Parameters

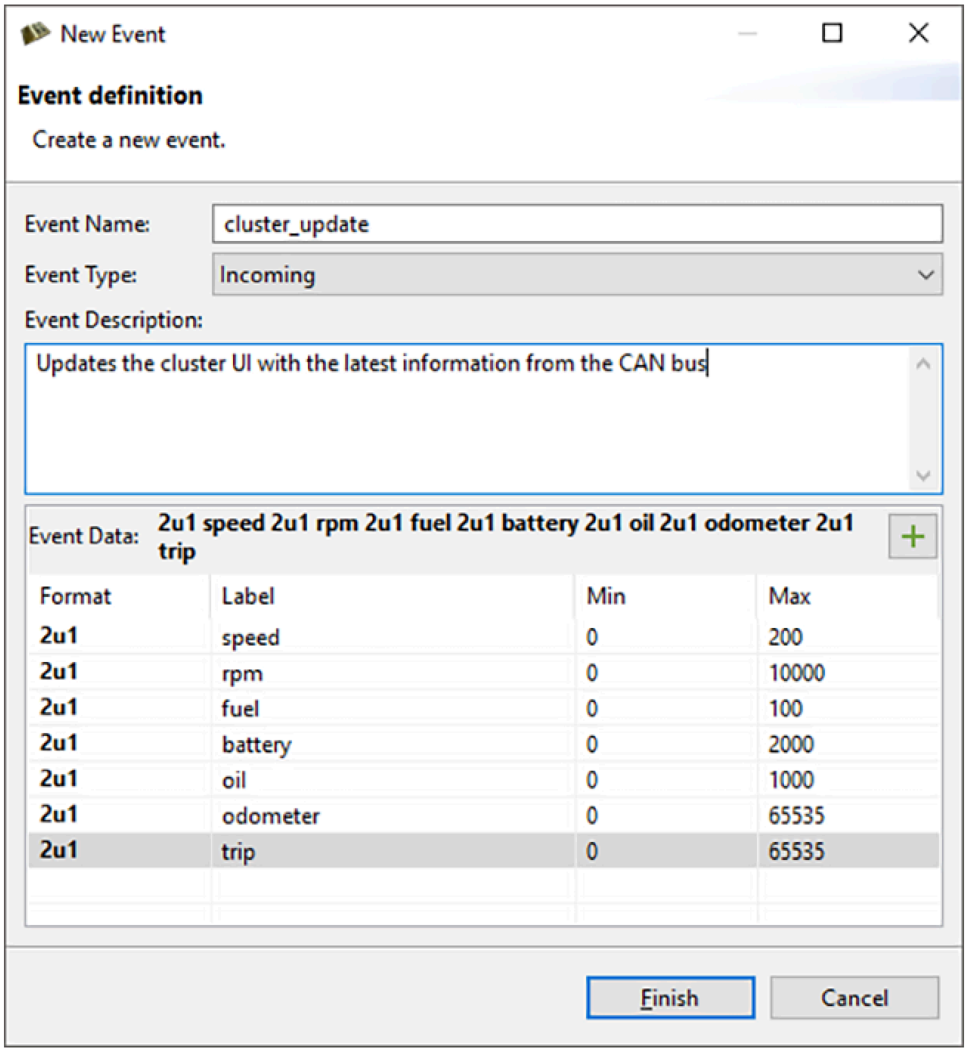
Action Parameters

Select an action to change its parameters ...

Action & Event Model Context: Application

Finish Cancel

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



New Event

Event definition

Create a new event.

Event Name: cluster_update

Event Type: Incoming

Event Description:

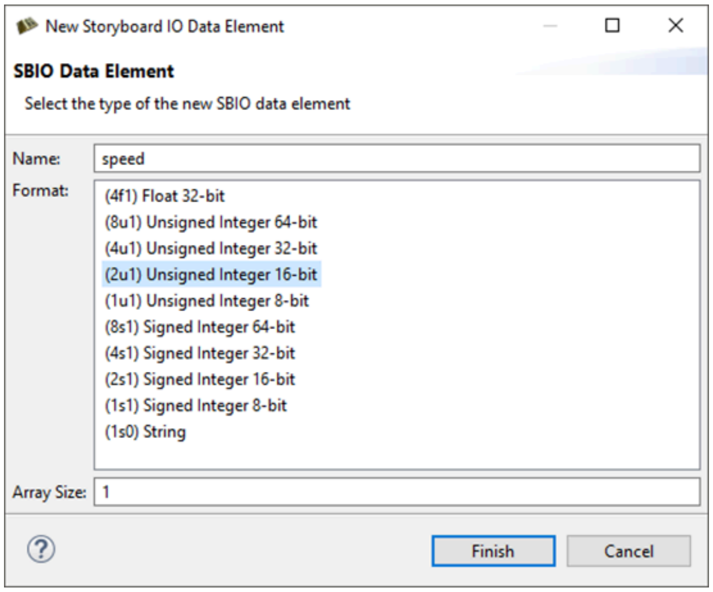
Updates the cluster UI with the latest information from the CAN bus

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	2000
2u1	oil	0	1000
2u1	odometer	0	65535
2u1	trip	0	65535

Finish Cancel

- **Name** - Must not be blank and must only contain alphanumerical 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.



SBIO Data Element

Select the type of the new SBIO data element

Name: speed

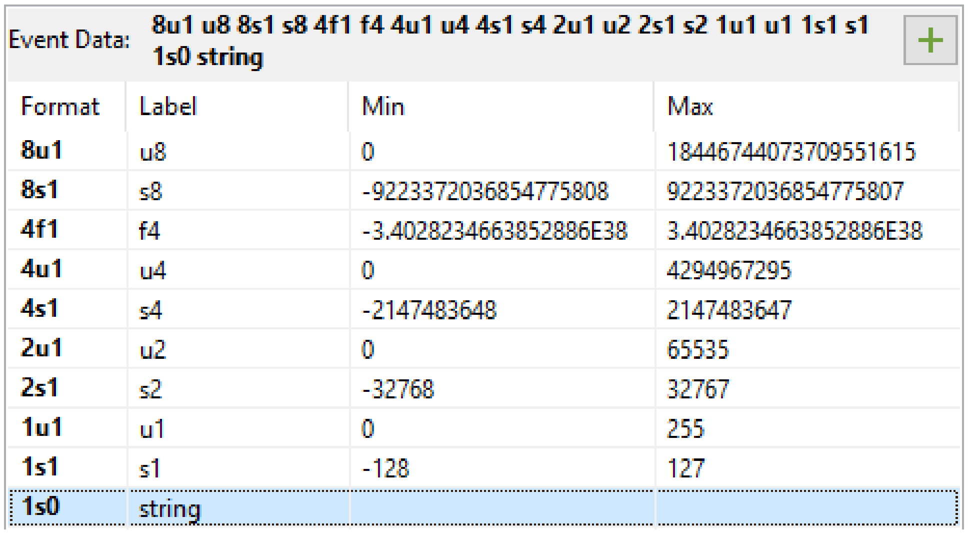
Format:
- (4f1) Float 32-bit
- (8u1) Unsigned Integer 64-bit
- (4u1) Unsigned Integer 32-bit
- (2u1) Unsigned Integer 16-bit
- (1u1) Unsigned Integer 8-bit
- (8s1) Signed Integer 64-bit
- (4s1) Signed Integer 32-bit
- (2s1) Signed Integer 16-bit
- (1s1) Signed Integer 8-bit
- (1s0) String

Array Size: 1

- **Name** - Must not be blank, must only contain alphanumerical 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

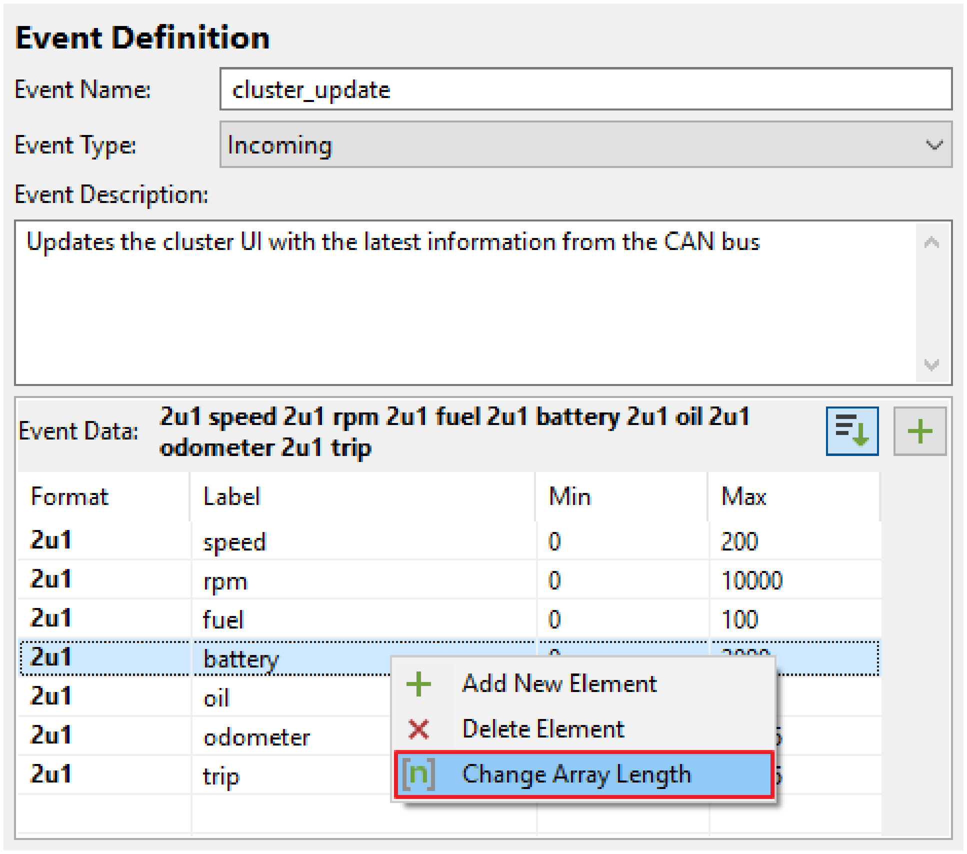
These are the different data element types and their default Min/Max attributes.


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: cluster_update

Event Type: Incoming

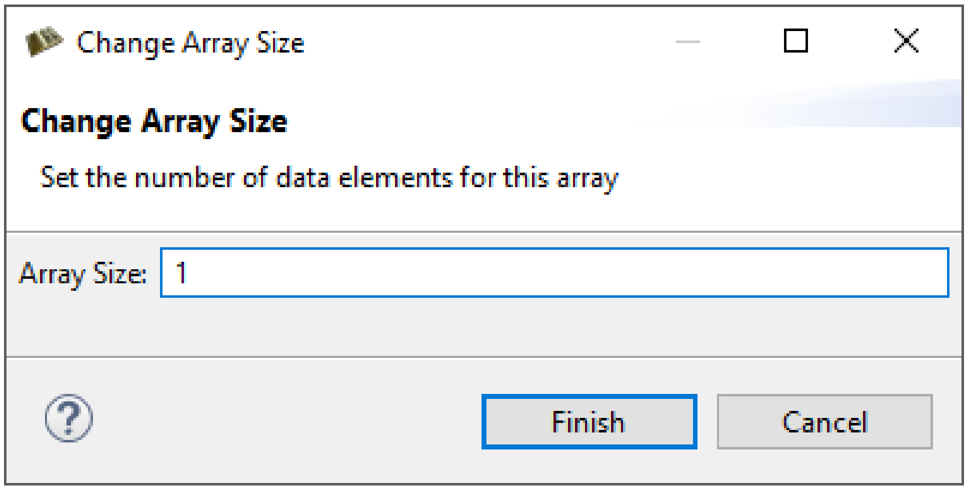
Event Description:

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		
2u1	oil		
2u1	odometer		
2u1	trip		

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

Change Array Size

Set the number of data elements for this array

Array Size: 1

Reordering Event Data Elements

The event format can be changed by reording 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 amonst themseleves, regardless of their type(i.e. 4u1, 4s1 and 4f1). To enable/disable the enforcement of sorting rules, toggle the button.

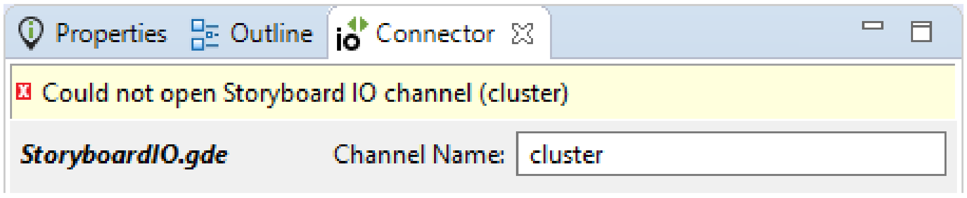
Editing Min and Max Attributes

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 attribtues 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.

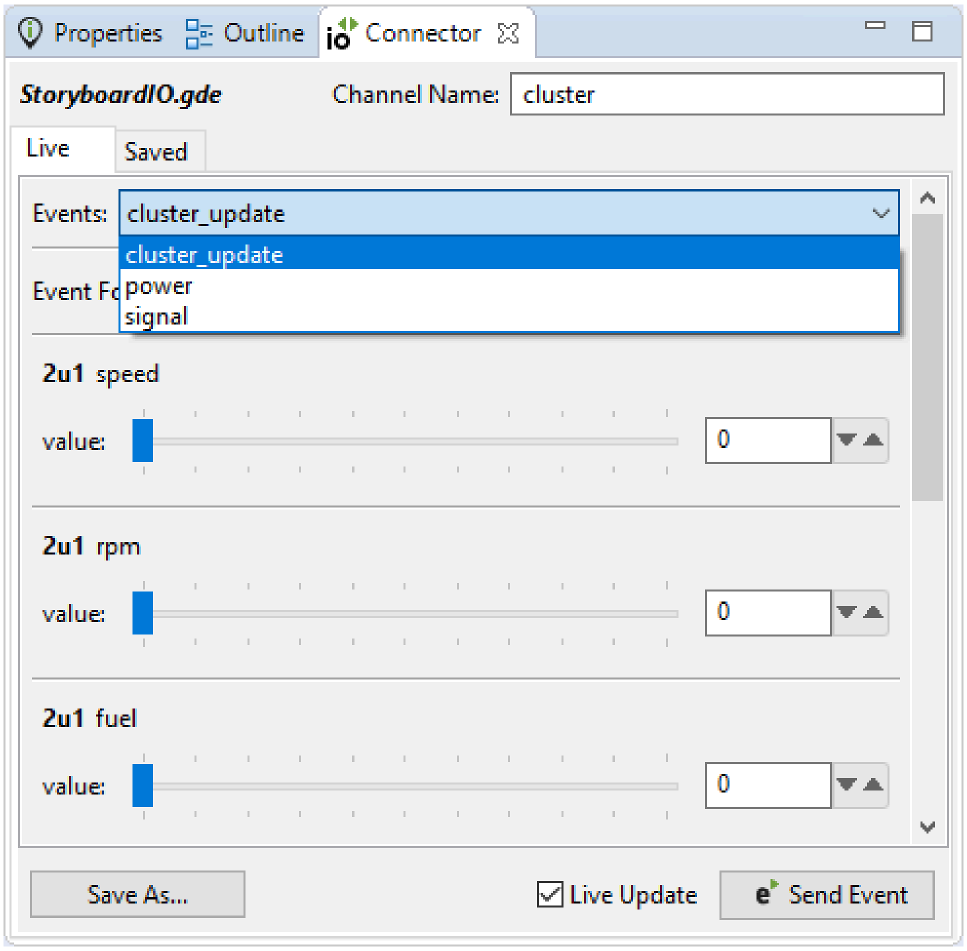


Could not open Storyboard IO channel (cluster)

StoryboardIO.gde Channel Name: cluster

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.



StoryboardIO.gde Channel Name: cluster

Events: cluster_update

cluster_update
power
signal

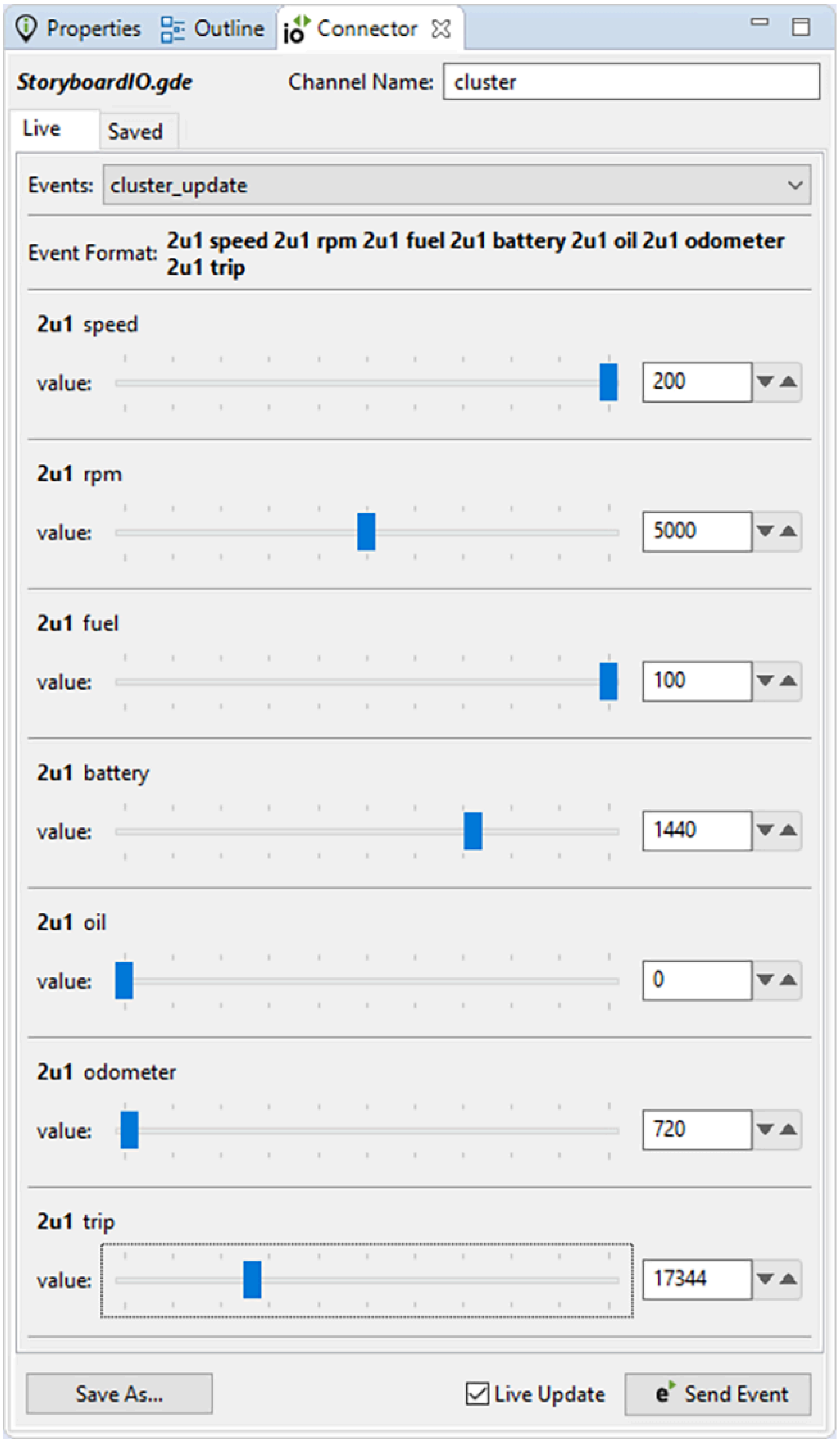
Event Fo

2u1 speed
value: 0

2u1 rpm
value: 0

2u1 fuel
value: 0

☑ Live Update



StoryboardIO.gde

Channel Name: cluster

Events: cluster_update

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

2u1 speed
value: 200

2u1 rpm
value: 5000

2u1 fuel
value: 100

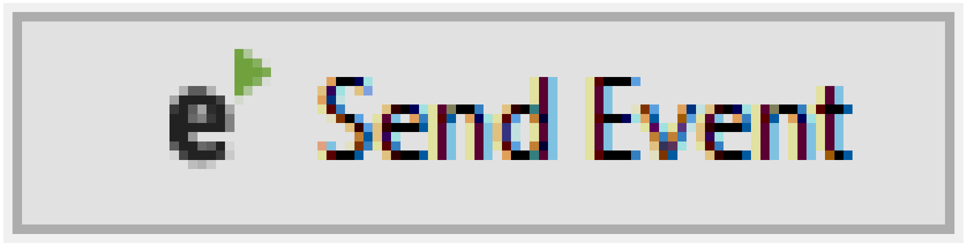
2u1 battery
value: 1440

2u1 oil
value: 0

2u1 odometer
value: 720

2u1 trip
value: 17344

☑ Live Update

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 

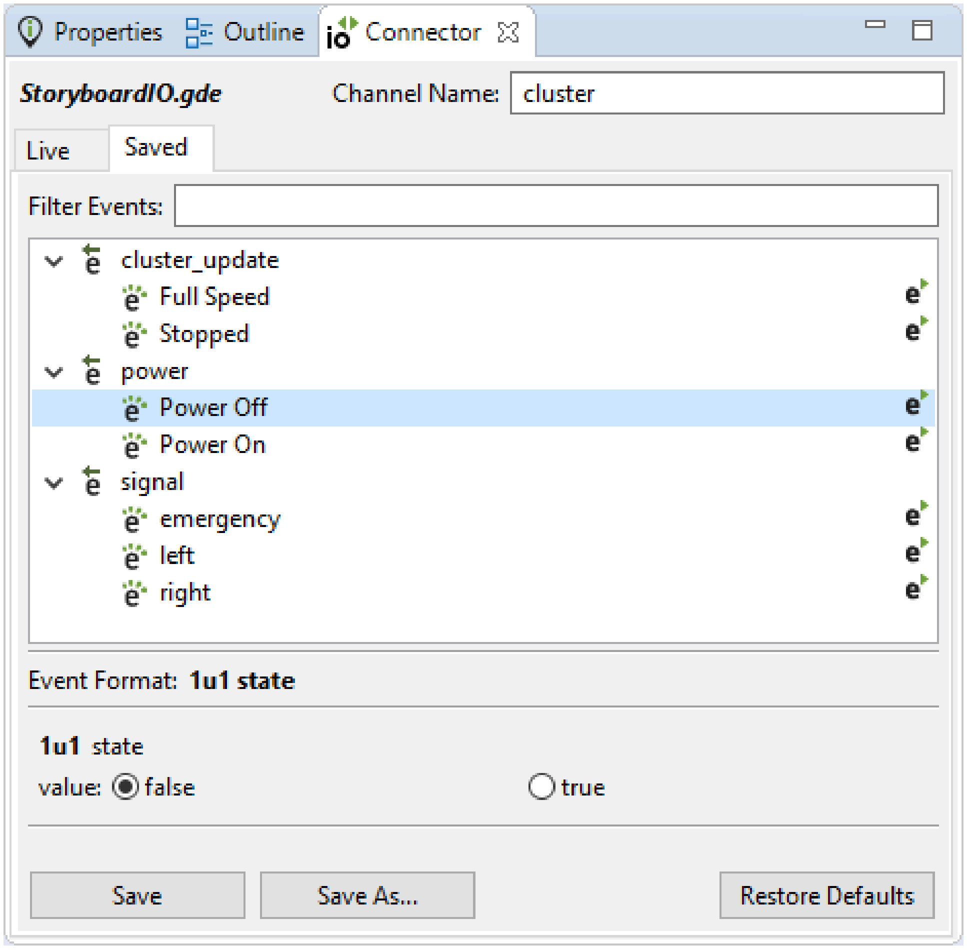
button.

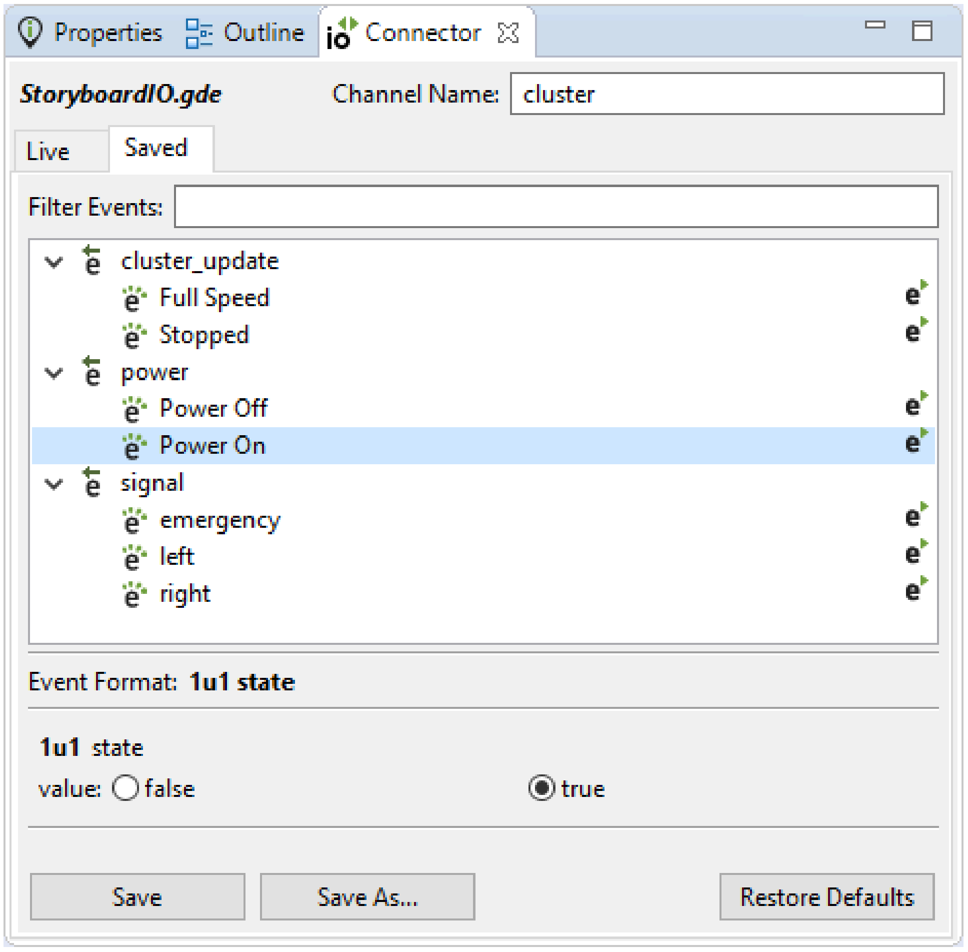
The Save As... button is used to save the current event configuration for reuse in Saved Mode, this is known as an **Event Instance**, these are the primary focus of the Saved Mode tab.

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.





StoryboardIO.gde Channel Name: cluster

Filter Events:

- cluster_update
  - Full Speed
  - Stopped
- power
  - Power Off
  - Power On
- signal
  - emergency
  - left
  - right

Event Format: **1u1 state**

1u1 state

value: ◯ false ● true

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.

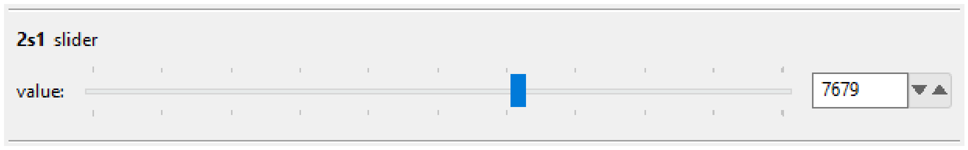
The Restore Defaults button is used to return the currently selected event instance back to it's saved state.

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

Data Element Editors

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

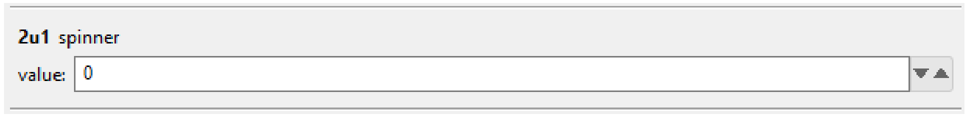
- Number Slider - Contains a slider and a number spinner. Not available on 8u/8s/4u element types.



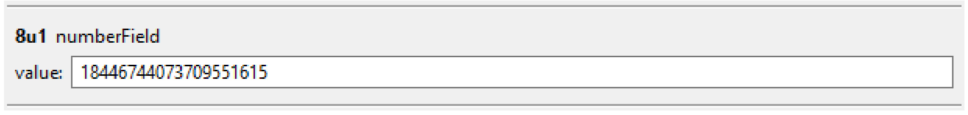
2s1 slider

value: 7679

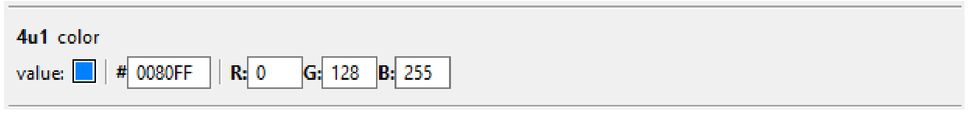
- Number Spinner - A number spinner, can perform mathematical calculations on, increment and decrement the current value. Not available on 8u/8s/4u element types.


2u1 spinner
value: 0

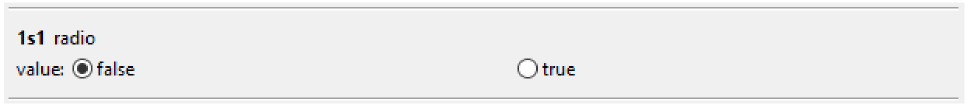
- Number Field - A text field for inputing numbers. Only available on 8u/8s/4u element types.


8u1 numberField
value: 18446744073709551615

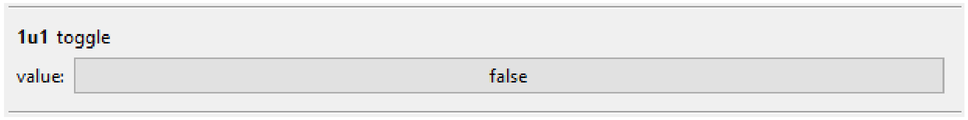
- Color Picker - Open Color Dialog or input RGB color values. Only available on 4u type elements.


4u1 color
value: # 0080FF **R:** 0 **G:** 128 **B:** 255

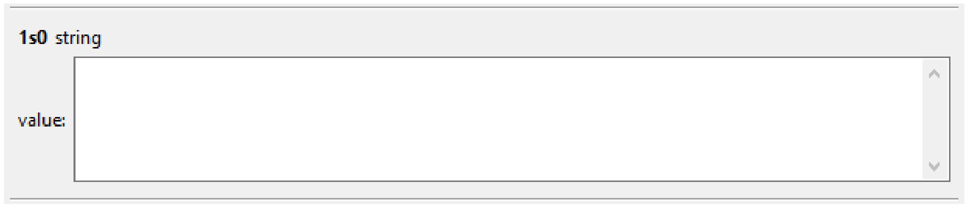
- Boolean Radio - True and False buttons.


1s1 radio
value: ● false ○ true

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


1u1 toggle
value: false

- 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.


1s0 string
value:

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:

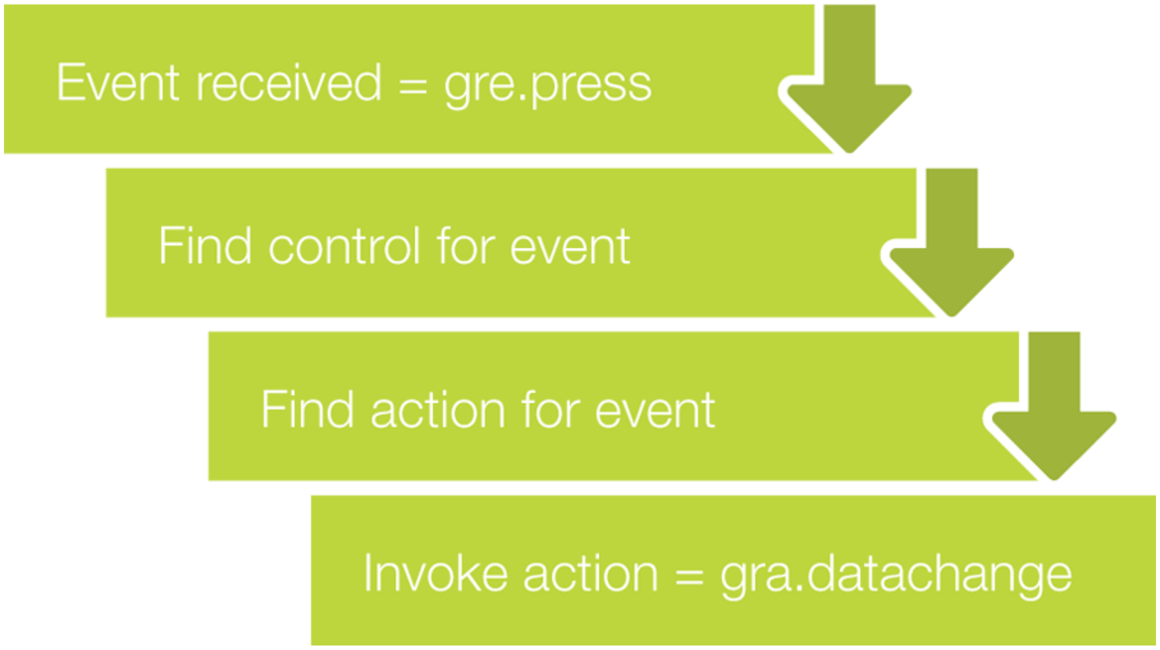
- Standard Input Events are generic input definitions and are pre-defined within Designer and as such are not user defined events. These events represent expected user interactions such mouse or touchscreen press, release and motion as well as keyboard input. There are other standard input events defined 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 `iodefs.h` and are also listed in the appendix Appendix D, *Standard Event Definitions*.

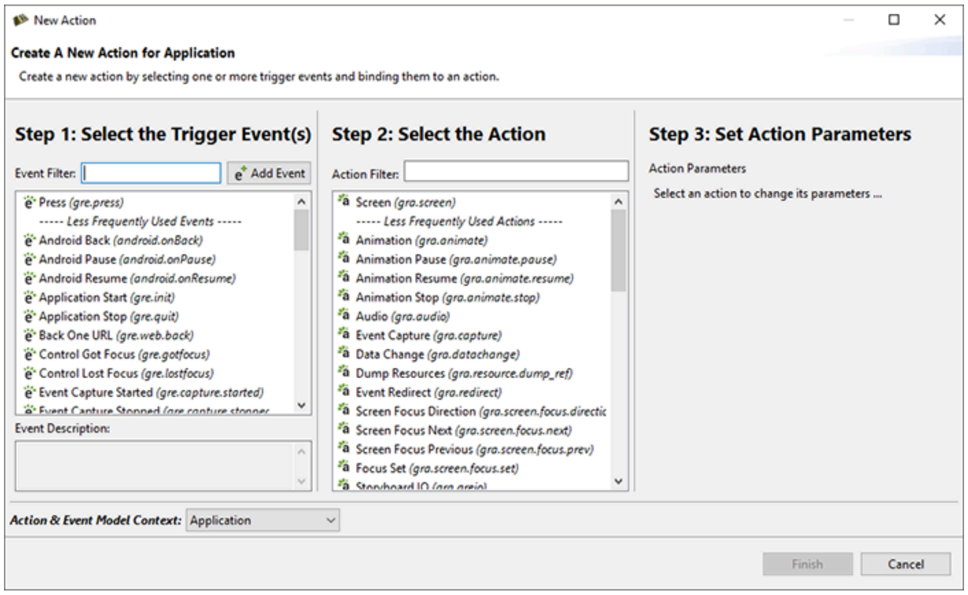
- UI Events are events that are defined to be used just within the context of the user interface. UI events are considered user events as they are custom to your application and you define them for your specific application needs. In general these events are generated from within the application in order 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.



New Action

Create A New Action for Application

Create a new action by selecting one or more trigger events and binding them to an action.

Step 1: Select the Trigger Event(s)

Event Filter: [Add Event]

- Press *(gre.press)*
- ----- *Less Frequently Used Events* -----
- Android Back *(android.onBack)*
- Android Pause *(android.onPause)*
- Android Resume *(android.onResume)*
- Application Start *(gre.init)*
- Application Stop *(gre.quit)*
- Back One URL *(gre.web.back)*
- Control Got Focus *(gre.gotfocus)*
- Control Lost Focus *(gre.lostfocus)*
- Event Capture Started *(gre.capture.started)*
- Event Capture Stopped *(gre.capture.stoppe*

Event Description:

Step 2: Select the Action

Action Filter:

- Screen *(gra.screen)*
- ----- *Less Frequently Used Actions* -----
- Animation *(gra.animate)*
- Animation Pause *(gra.animate.pause)*
- Animation Resume *(gra.animate.resume)*
- Animation Stop *(gra.animate.stop)*
- Audio *(gra.audio)*
- Event Capture *(gra.capture)*
- Data Change *(gra.datachange)*
- Dump Resources *(gra.resource.dump_ref)*
- Event Redirect *(gra.redirect)*
- Screen Focus Direction *(gra.screen.focus.directic*
- Screen Focus Next *(gra.screen.focus.next)*
- Screen Focus Previous *(gra.screen.focus.prev)*
- Focus Set *(gra.screen.focus.set)*
- Storyboard IO *(gra.greio)*

Step 3: Set Action Parameters

Action Parameters

Select an action to change its parameters ...

Action & Event Model Context: Application

[Finish] [Cancel]

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.

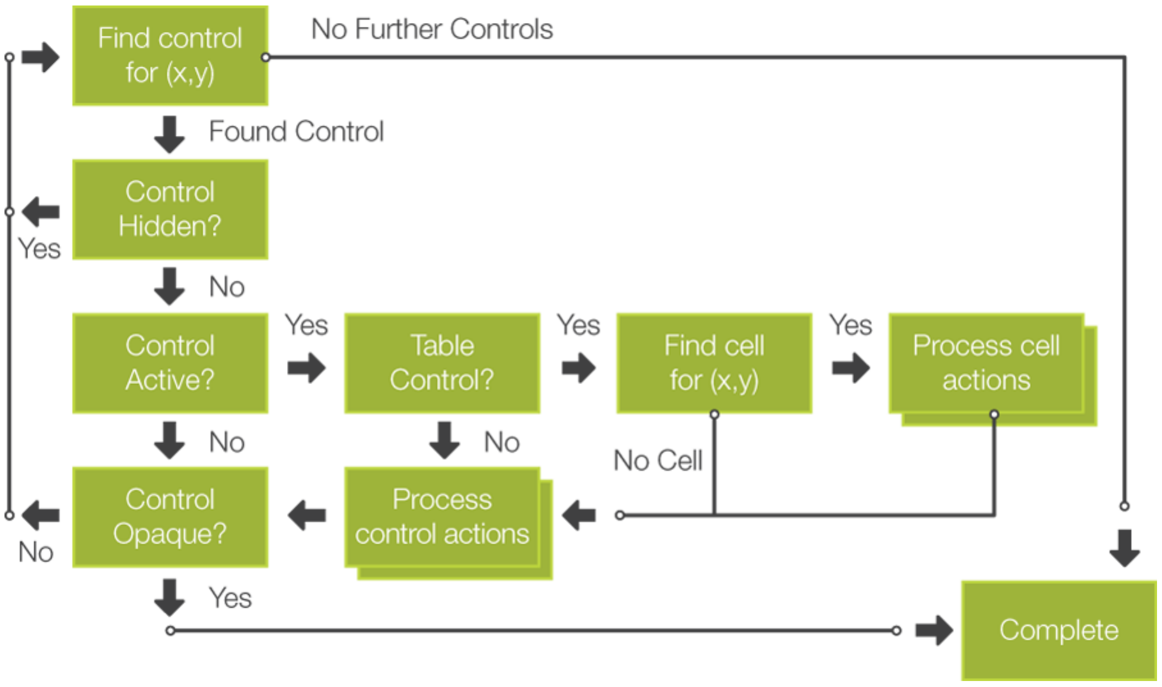
The controls that are matched by pointer events also have the potential to become the newly focused controls if they have enabled a focus index property (off by default). Focus controls are used for non-targeted events that do not have a specific display location.

- 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.

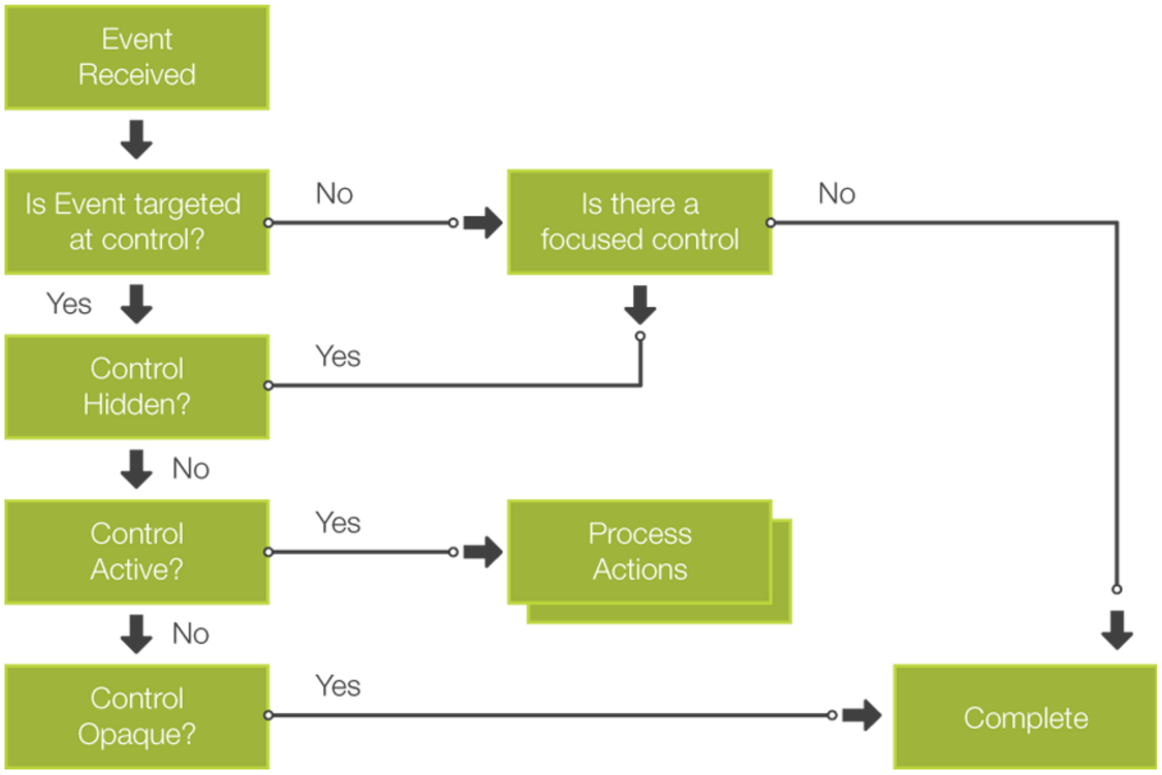
- Targeted events. These are events that have a specific model target provided within the event payload. In these situations, the control list will be created based on what the target indicates. If the target is a control then it will be used to start processing.

If there is a list of controls then the event processing will to 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:

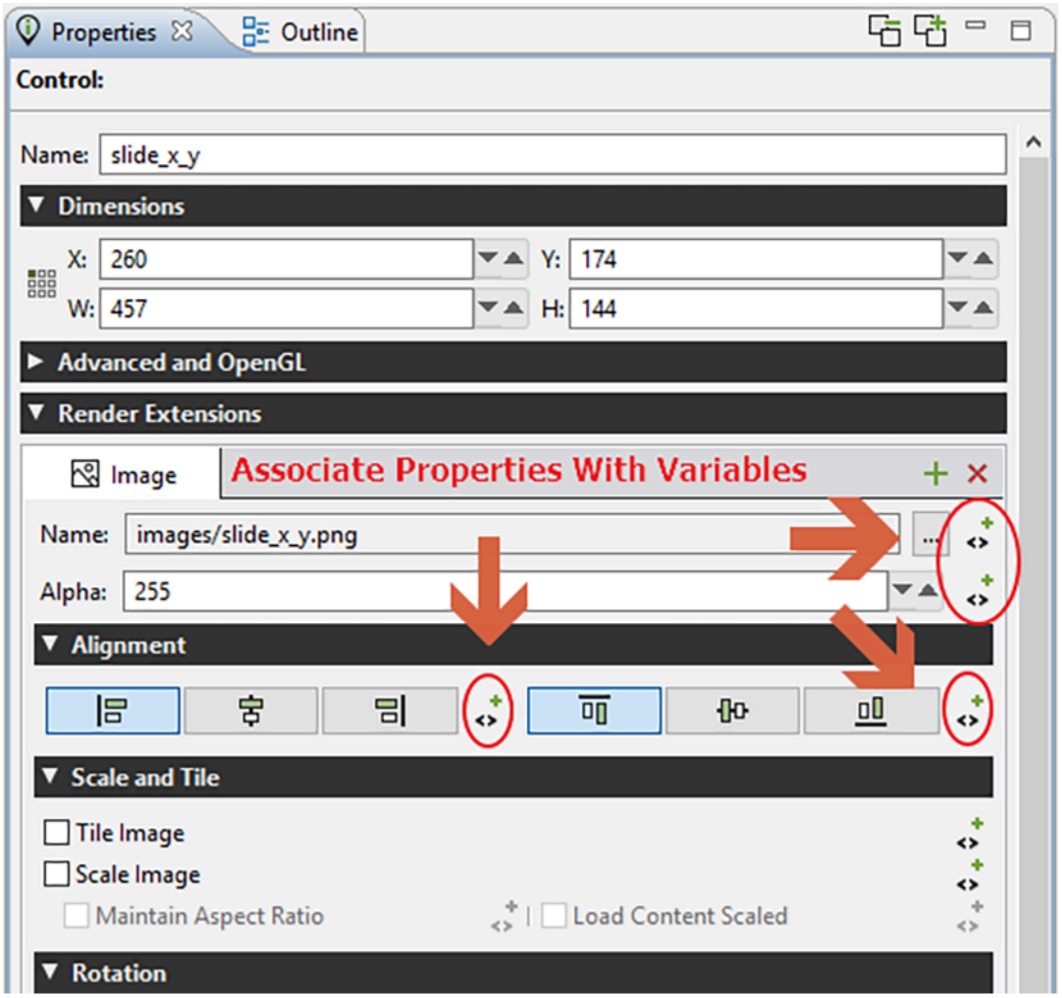


The following diagram illustrates the flow for an event which does not have positional information:


Event Received
Is Event targeted at control?
No
Is there a focused control
No
Yes
Control Hidden?
Yes
No
Control Active?
Yes
Process Actions
No
Control Opaque?
Yes
Complete

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.



Name: slide_x_y

X: 260
Y: 174
W: 457
H: 144

Name: images/slide_x_y.png
Alpha: 255

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 <> 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.

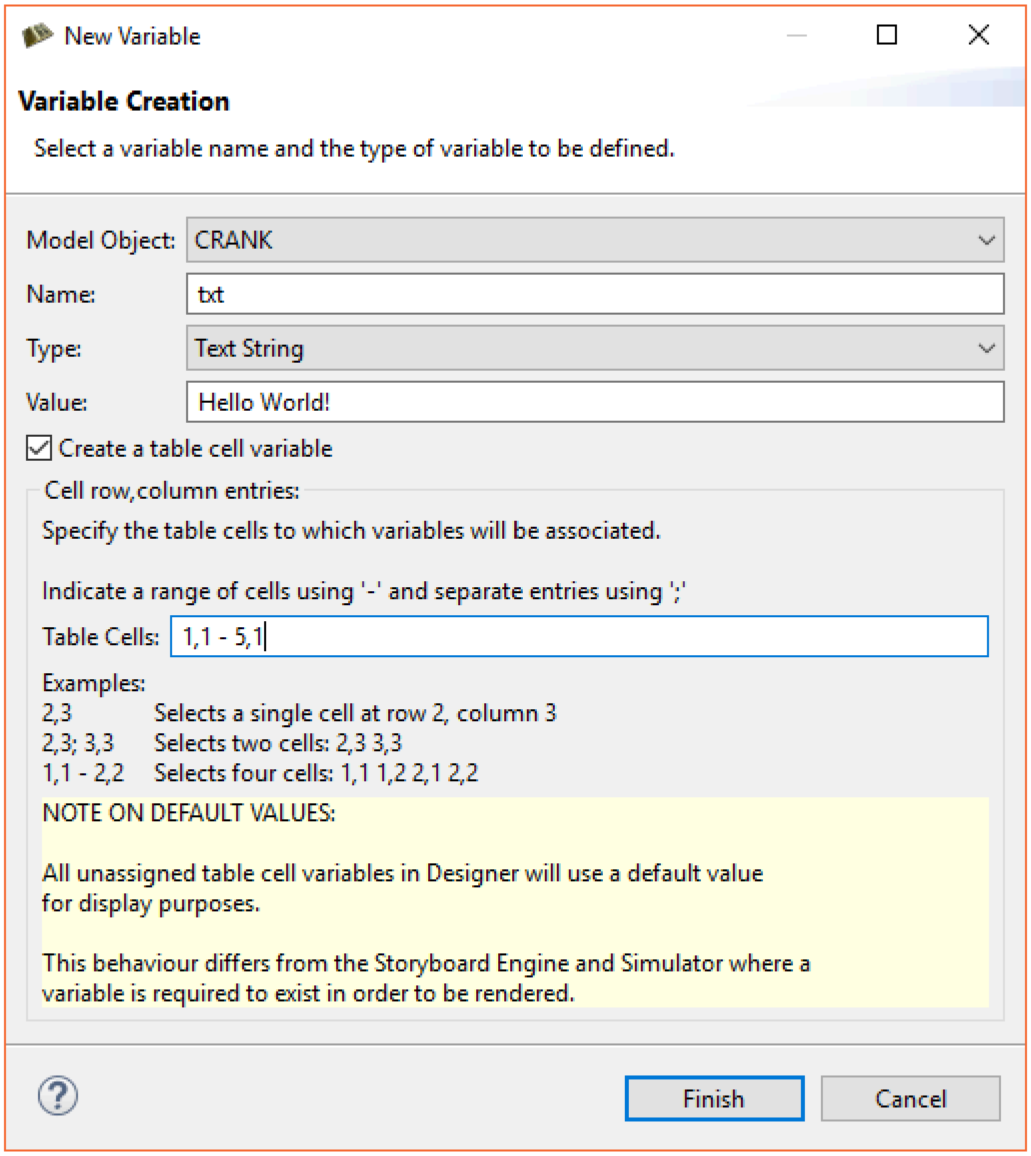
To change the variable binding to a new or different variable, then just repeat the process of clicking on the variable binding <> indicator to re-open the Variable Selection dialog. To remove a variable binding on a property you can right click on the variable binding <> indicator and then select Unbind. 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.



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.

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:

gra.screen	Transition to a new screen immediately
gra.screen.fade	Fade the new screen into the current screen over time
gra.screen.path	Slide the new screen in and the old screen out over time, from one of the following directions • Left • Right • Top • Bottom
gra.screen.scale	Grow the new screen over the current screen

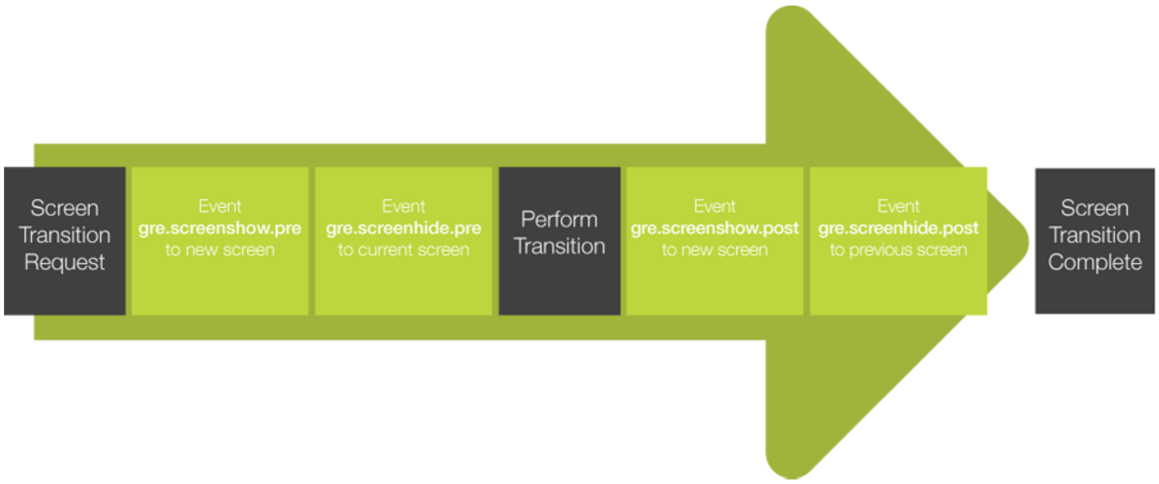
All transition actions, which are time-based, take similar arguments that control the duration of the transition, the rate at which the transitions will occur, the orientation of the transition, and the number of frames that should be used. Using these arguments, the designer can control the user experience (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:

gre.screenshow.pre	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 gra.datachange action or Lua before their transition content is updated.
gre.screenhide.pre	This event is generated for the previous screen being hidden. The event will be generated before the transition starts.
gre.screenshow.post	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:


Screen Transition Request
Event gre.screenshow.pre to new screen
Event gre.screenhide.pre to current screen
Perform Transition
Event gre.screenshow.post to new screen
Event gre.screenhide.post to previous screen
Screen Transition Complete

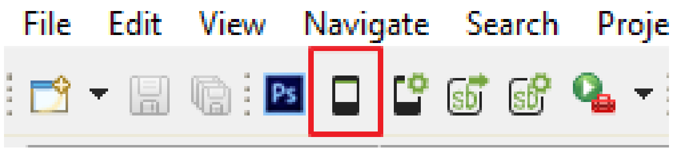
The transitions are written such that if graphics hardware layer support is are available, then these layers, assuming they are available for use, will be leveraged to lower the processing overhead for the system during the transition period. Experience has demonstrated that it is possible to achieve 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.

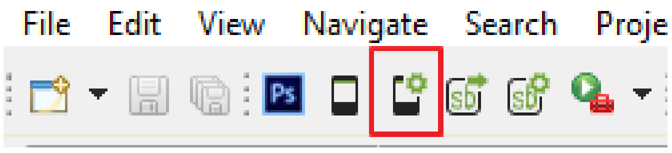
- From the main menu Run > Simulate Storyboard Project
- From the main toolbar using the Simulate Storyboard Project

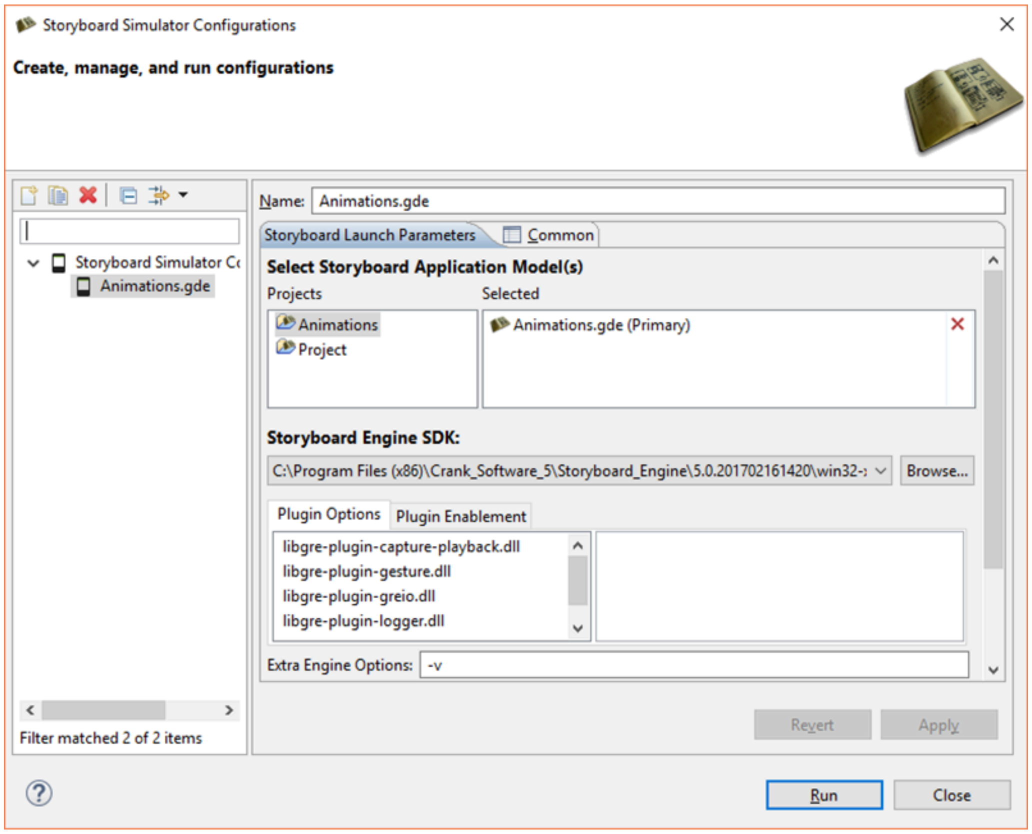


- By right clicking on an application model file (gde) and selecting Storyboard Simulator
- By right clicking within the Storyboard editor and selecting Simulate...
- By using the key command ALT/CMD+L within the Storyboard editor

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.




Storyboard Simulator Configurations

Create, manage, and run configurations

Name: Animations.gde

Storyboard Launch Parameters | Common

Select Storyboard Application Model(s)

Projects: Animations Project

Selected: Animations.gde (Primary)

Storyboard Engine SDK: C:\Program Files (x86)\Crank_Software_5\Storyboard_Engine\5.0.201702161420\win32-

Plugin Options | Plugin Enablement

libgre-plugin-capture-playback.dll
libgre-plugin-gesture.dll
libgre-plugin-greio.dll
libgre-plugin-logger.dll

Extra Engine Options: -v

Filter matched 2 of 2 items

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 the section called “Plugins 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 24, *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 21, *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 though 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
context_target	The current context (app, screen, layer, or control) that the event was targeted at
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[mapargs.context_target])
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.

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

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:

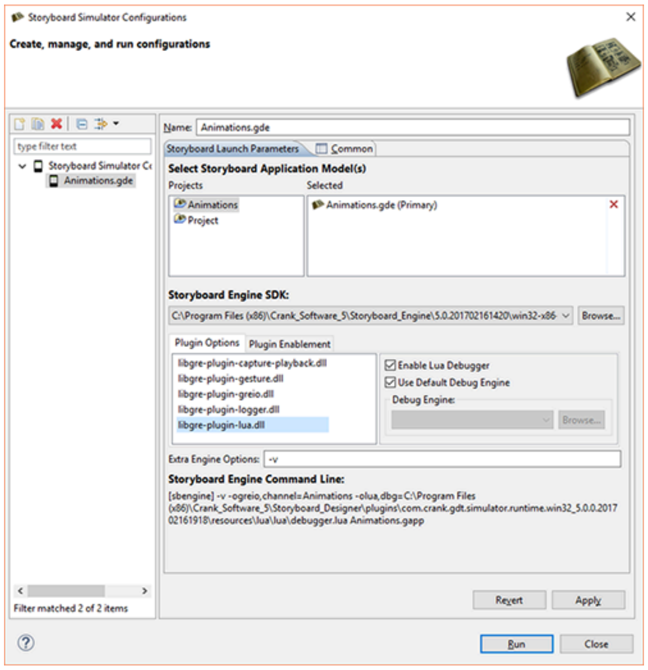
1. Create a Storyboard application launch configuration. This topic is discussed in more detail in the chapter Chapter 11, *Simulating your Application*.

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.



Storyboard Simulator Configurations

Create, manage, and run configurations

Name: Animations.gde

Storyboard Launch Parameters | Common

Select Storyboard Application Model(s)

Projects: Animations, Project

Selected: Animations.gde (Primary)

Storyboard Engine SDK:

C:\Program Files (x86)\Crank_Software_5\Storyboard_Engine\5.0.201702161420\win32-x86

Plugin Options | Plugin Enablement

libgre-plugin-capture-playback.dll
libgre-plugin-gesture.dll
libgre-plugin-greio.dll
libgre-plugin-logger.dll
libgre-plugin-lua.dll

☑ Enable Lua Debugger
☑ Use Default Debug Engine
Debug Engine:

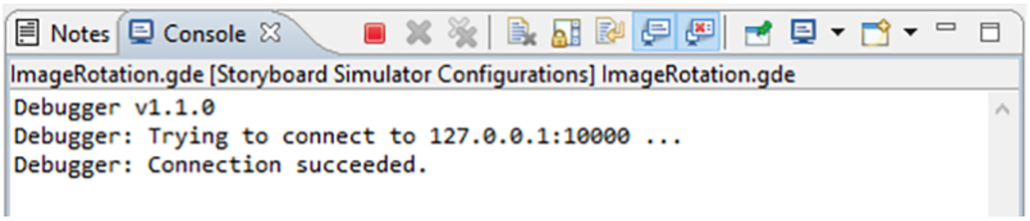
Extra Engine Options: -v

Storyboard Engine Command Line:

```
[sbengine] -v -ogreio,channel=Animations -olua,dbg=C:\Program Files
(x86)\Crank_Software_5\Storyboard_Designer\plugins\com.crank.gdt.simulator.runtime.win32_5.0.0.2017
02161918\resources\lua\lua\debugger.lua Animations.gapp
```

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

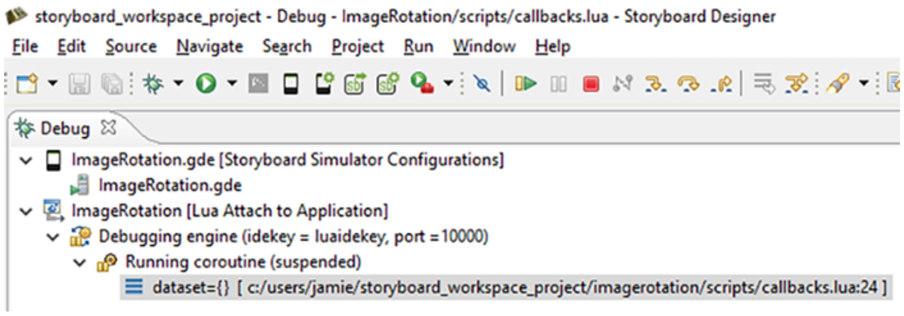
This will launch the Storyboard application and at the same time initialize the debug client within the Storyboard application and the server running within the Designer environment. The two will connect automatically over a network socket and your application should immediately begin running. To confirm that the connection has taken place and the Lua debugger is running, check the output console for the following:



ImageRotation.gde [Storyboard Simulator Configurations] ImageRotation.gde

```
Debugger v1.1.0
Debugger: Trying to connect to 127.0.0.1:10000 ...
Debugger: Connection succeeded.
```

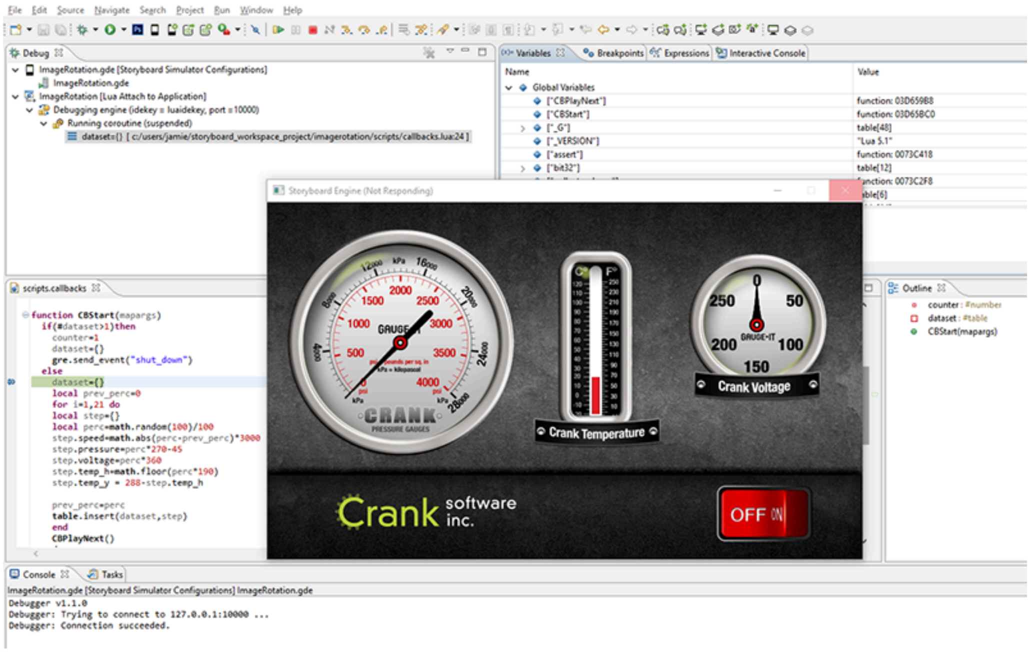
If you change to the Debugger perspective, the Debug view should look like this:


storyboard_workspace_project - Debug - ImageRotation/scripts/callbacks.lua - Storyboard Designer
File Edit Source Navigate Search Project Run Window Help
Debug

```
ImageRotation.gde [Storyboard Simulator Configurations]
    ImageRotation.gde
ImageRotation [Lua Attach to Application]
    Debugging engine (idekey = luaidekey, port = 10000)
        Running coroutine (suspended)
            dataset={} [ c:/users/jamie/storyboard_workspace_project/imagerotation/scripts/callbacks.lua:24 ]
```

You are now able to start debugging your Lua code

The Lua debugger, when a breakpoint is activated will automatically switch to the Debug Perspective. This perspective provides an alternative layout of views that are specifically related to debugging activities.



```
ImageRotation.gde [Storyboard Simulator Configurations]
    ImageRotation.gde
ImageRotation [Lua Attach to Application]
    Debugging engine (idekey = luaidekey, port = 10000)
        Running coroutine (suspended)
            dataset={} [ c:/users/jamie/storyboard_workspace_project/imagerotation/scripts/callbacks.lua:24 ]
```

Name	Value
Global Variables	
["CBPlayNext"]	function: 03D659B8
["CBStart"]	function: 03D65BC0
["_G"]	table[48]
["_VERSION"]	"Lua 5.1"
["assert"]	function: 0073C418
["bit32"]	table[12]
	function: 0073C2F8
	table[6]

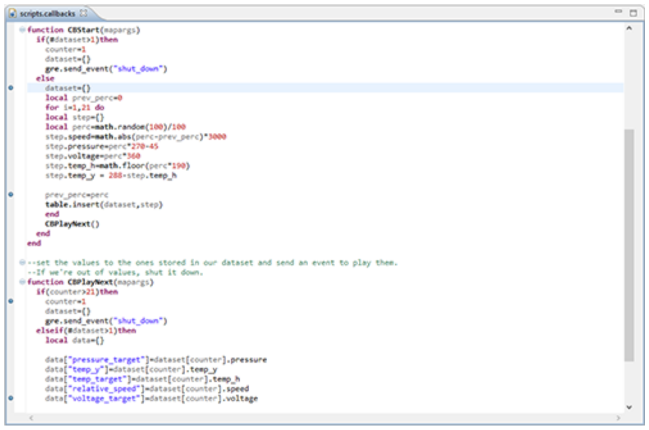
```
function CBStart(mapargs)
  if(#dataset>1)then
    counter=1
    dataset={}
    gre.send_event("shut_down")
  else
    dataset={}
    local prev_perc=0
    for i=1,21 do
    local step={}
    local perc=math.random(100)/100
    step.speed=math.abs(perc-prev_perc)*3000
    step.pressure=perc*270-45
    step.voltage=perc*360
    step.temp_h=math.floor(perc*190)
    step.temp_y = 288-step.temp_h

    prev_perc=perc
    table.insert(dataset,step)
    end
    CBPlayNext()
```

```
counter : #number
dataset : #table
CBStart(mapargs)
```

```
ImageRotation.gde [Storyboard Simulator Configurations] ImageRotation.gde
Debugger v1.1.0
Debugger: Trying to connect to 127.0.0.1:10000 ...
Debugger: Connection succeeded.
```

Variables	Variables are displayed in a hierarchical manner in the Variables view. Global variables are displayed in a special table and listed as `globals` while function parameters and local variables are displayed as top level elements. Strings and numeric values are displayed directly, while tables can be navigated by double clicking their nodes and driving down.
Breakpoints	Breakpoints can be placed directly in the editor for Lua script files. Breakpoints can be toggled on/off by double clicking in the margins of the Lua script file where the execution should be stopped. While it is possible to place breakpoints on all lines of a Lua script file, not all lines are breakable due to the manner in which the Lua script is 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 di-

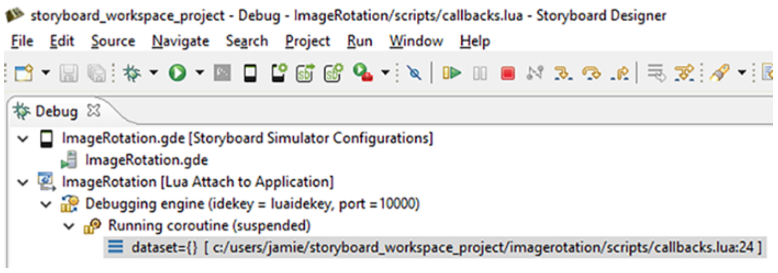
rectly to the script source file from withing the Breakpoints view by right clicking and selecting Go To File.



```
function CBStart(mapargs)
  if(#dataset>1)then
    counter=1
    dataset={}
    gre.send_event("shut_down")
  else
    dataset={}
    local prev_perc=0
    for i=1,21 do
    local step={}
    local perc=math.random(100)/100
    step.speed=math.abs(perc-prev_perc)*3000
    step.pressure=perc*270-45
    step.voltage=perc*360
    step.temp_h=math.floor(perc*190)
    step.temp_y = 288-step.temp_h

    prev_perc=perc
    table.insert(dataset,step)
    end
    CBPlayNext()
  end
end

--set the values to the ones stored in our dataset and send an event to play them.
--If we're out of values, shut it down.
function CBPlayNext(mapargs)
  if(counter>21)then
    counter=1
    dataset={}
    gre.send_event("shut_down")
  elseif(#dataset>1)then
    local data={}

    data["pressure_target"]=dataset[counter].pressure
    data["temp_y"]=dataset[counter].temp_y
    data["temp_target"]=dataset[counter].temp_h
    data["relative_speed"]=dataset[counter].speed
    data["voltage_target"]=dataset[counter].voltage
```

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.


storyboard_workspace_project - Debug - ImageRotation/scripts/callbacks.lua - Storyboard Designer
File
Edit
Source
Navigate
Search
Project
Run
Window
Help
Debug

```
ImageRotation.gde [Storyboard Simulator Configurations]
    ImageRotation.gde
ImageRotation [Lua Attach to Application]
    Debugging engine (idekey = luaidekey, port = 10000)
        Running coroutine (suspended)
            dataset={} [ c:/users/jamie/storyboard_workspace_project/imagerotation/scripts/callbacks.lua:24 ]
```

Lua Executables

Storyboard Suite 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 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 Selectiondialog 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.

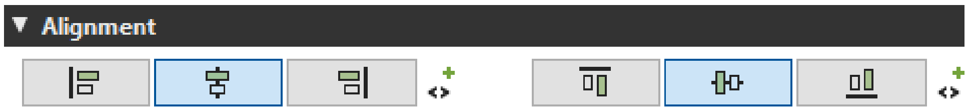
Image Rotation

Rotate At Center

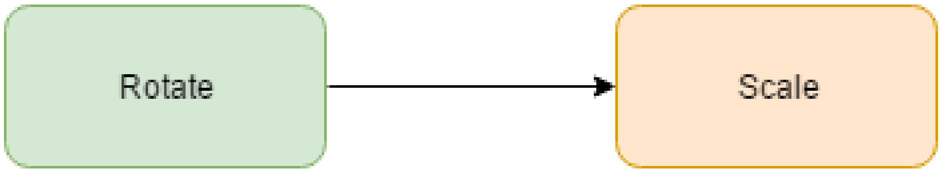
Storyboard image render extensions have an option to facilitate the rotation of an image at the center of the image. For images that are not scaled, this option behaves exactly as one might expect it would and as the rotation property is changed the image rotates accordingly.

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:

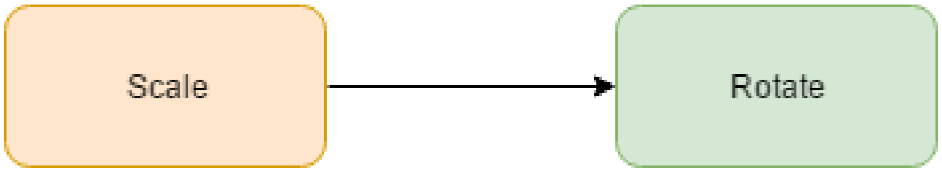


The rotate at center option uses a specific transformation path that will scale the image to fit the original size of the render extension after the rotation has taken place, this means that as it is rotated, it will be scaled to fit into its original bounds.

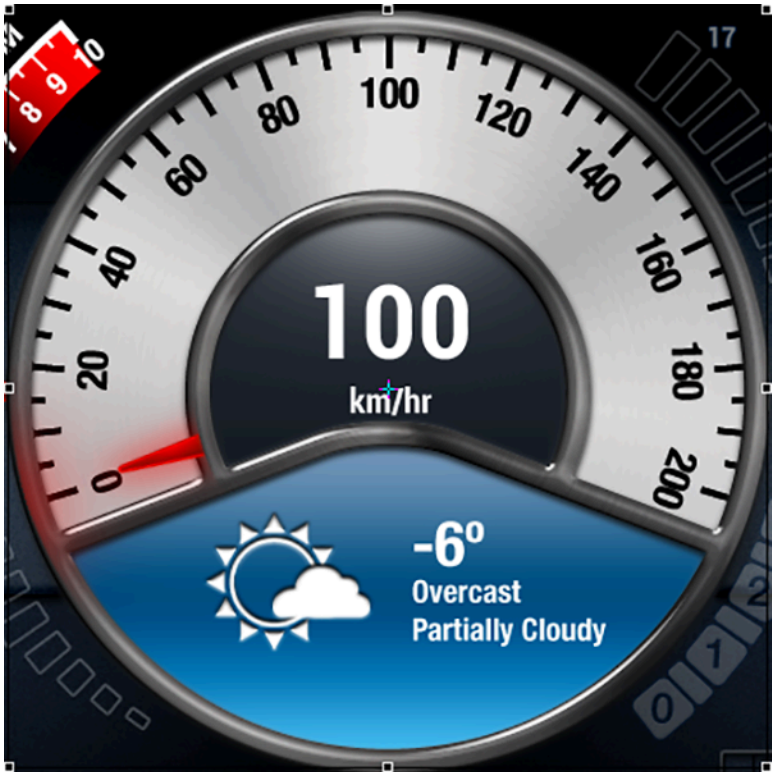


Rotate At Custom Point

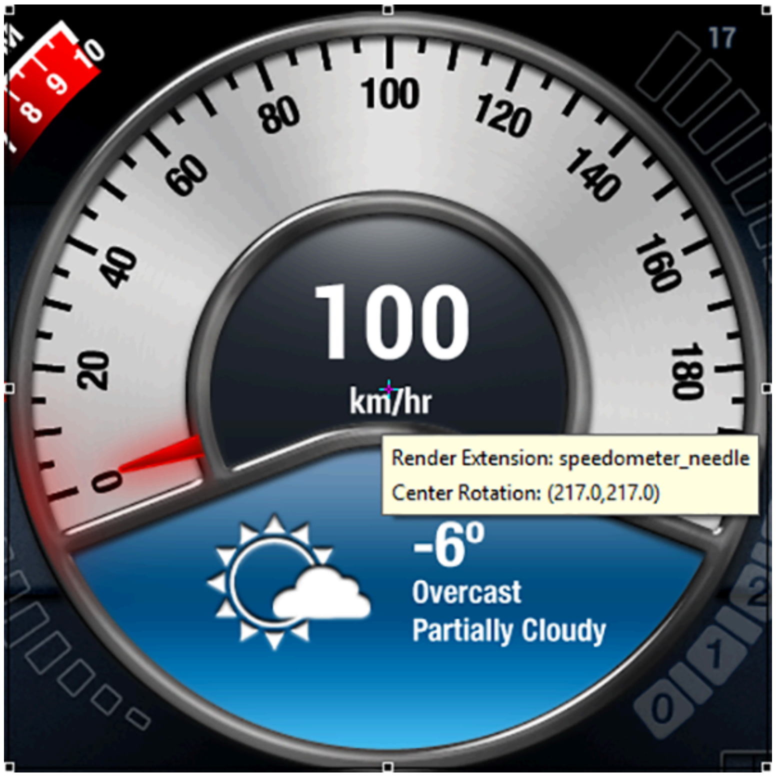
The option to use custom rotation points works well with scaled images. The image will be scaled to fit its render extension bounds before being rotated and the image rotation will be performed without modifying the size of the image on screen.



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.


100
km/hr
Render Extension: speedometer_needle
Center Rotation: (217.0,217.0)
-6º
Overcast
Partially Cloudy

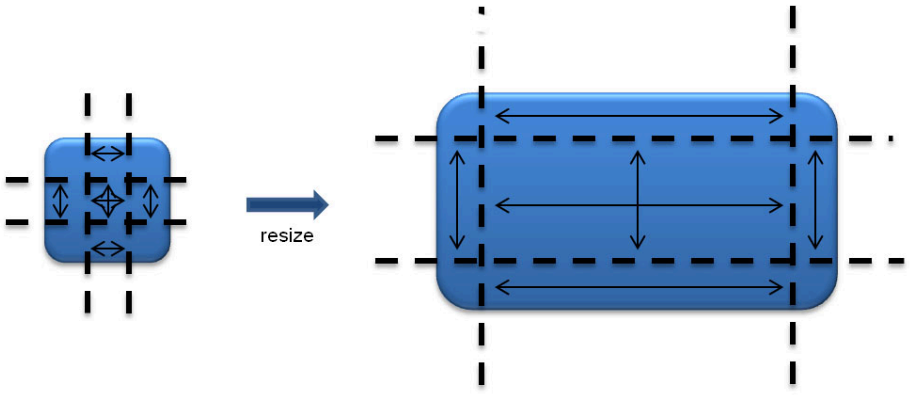
When images are rotated dynamically at runtime, there is rarely a one-to-one correspondence between the original source pixels and the destination pixels so some level of interpolation, or mapping, is going to occur. There are many different algorithms for this interpolation each of which trades visual 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 may 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.

Chapter 14. 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.

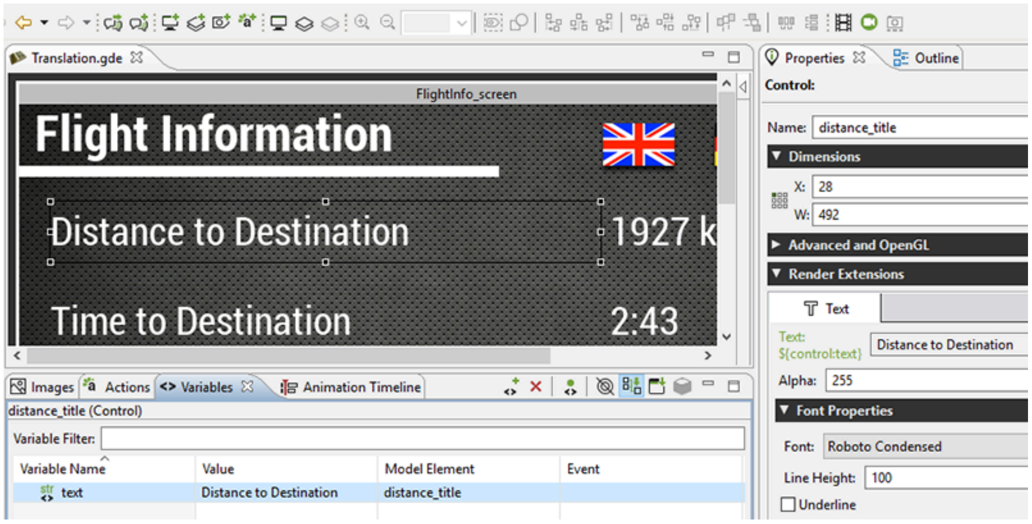
By using the font files directly Storyboard offers developers total control over what resources are used on the embedded system. For example a system that uses a specific font for only numeric elements can be edited to remove all other glyphs. Similarly several different fonts can be merged together 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.

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.

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.



Control:

Name: distance_title

Dimensions

X: 28

W: 492

Advanced and OpenGL

Render Extensions

Text

Text: ${control:text} Distance to Destination

Alpha: 255

Font Properties

Font: Roboto Condensed

Line Height: 100

Underline

distance_title (Control)

Variable Filter:

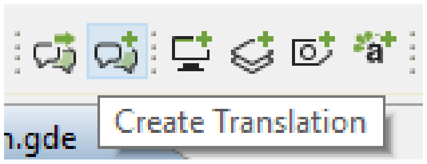
Variable Name	Value	Model Element	Event
text	Distance to Destination	distance_title	

Changes in string content is automatically reflected, making the translation activity significantly less labour intensive. To apply the translated content to the application, simply update a number of data variables with the appropriate UTF-8 encoded text string.

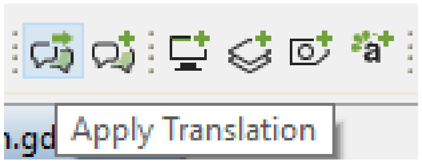
Two examples of translation are provided in the Storyboard samples. To further understand the internationalization process, be sure to check out *translation* and *thermostat* Storyboard samples available via File > Import > Storyboard Sample.

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 data change to a dynamic variable. In all situations the UI will automatically refresh to show the new content.

Storyboard provides two editor functions that allow translations to be quickly prototyped in the context of the design environment.



The Create Translation toolbar item scans through the application, identifies all of the bound text variables that are used, and extracts them to a comma separated (csv) file that contains the Storyboard variable key and the translated text string. By default, this file is saved in the `translations` directory of the project. This file can serve as the basis for performing a dynamic load of translated content or can be used by designers to ensure that the UI is appropriate for different language configurations.



The Apply Translation toolbar item scans the `translations` directory of a project for suitable translation files and presents a dialog that allows the user to immediately change all of the variable definitions in a project to the values declared in the translation file. This feature allows developers a quick way to preview their content in different language configurations.

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)

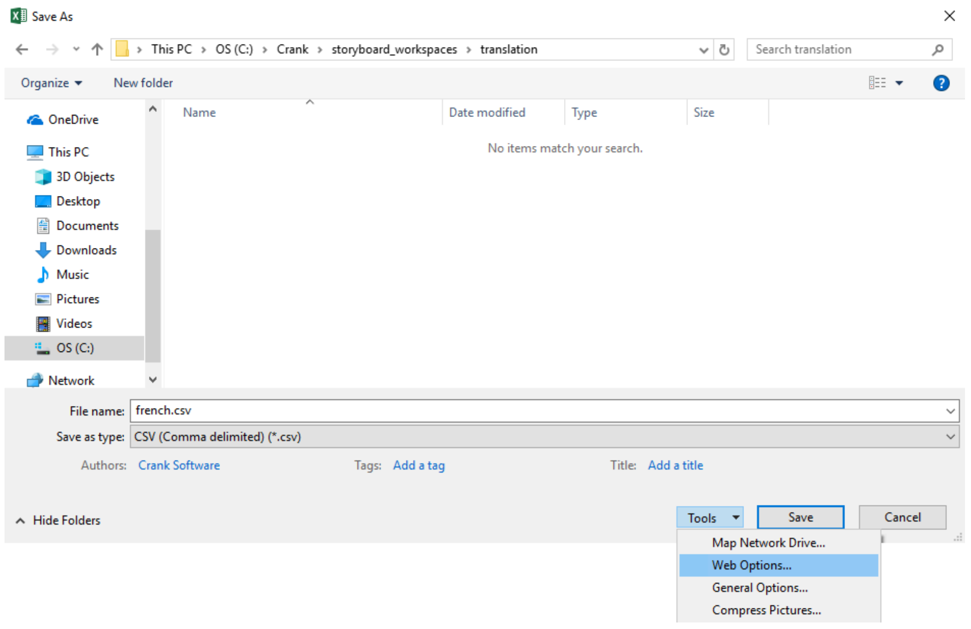
Microsoft Excel is a popular spreadsheet software solution that is widely adopted in many industries.

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

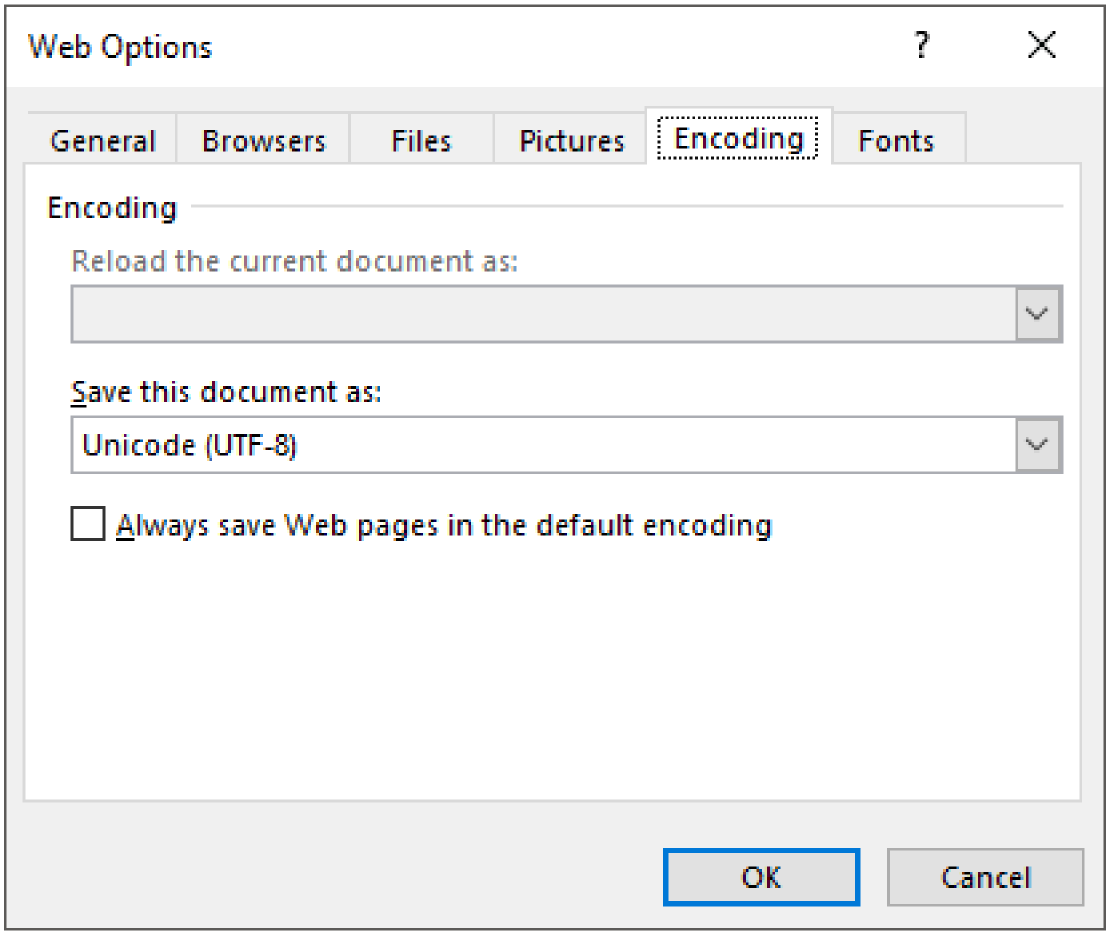
3. Choose 'CSV (Comma delimited) (*.csv)' in 'Save as type:'

4. At the bottom of the dialog, select tools > Web Options....



File name: french.csv
Save as type: CSV (Comma delimited) (*.csv)
Authors: Crank Software Tags: Add a tag Title: Add a title

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



Reload the current document as:

Save this document as: Unicode (UTF-8)

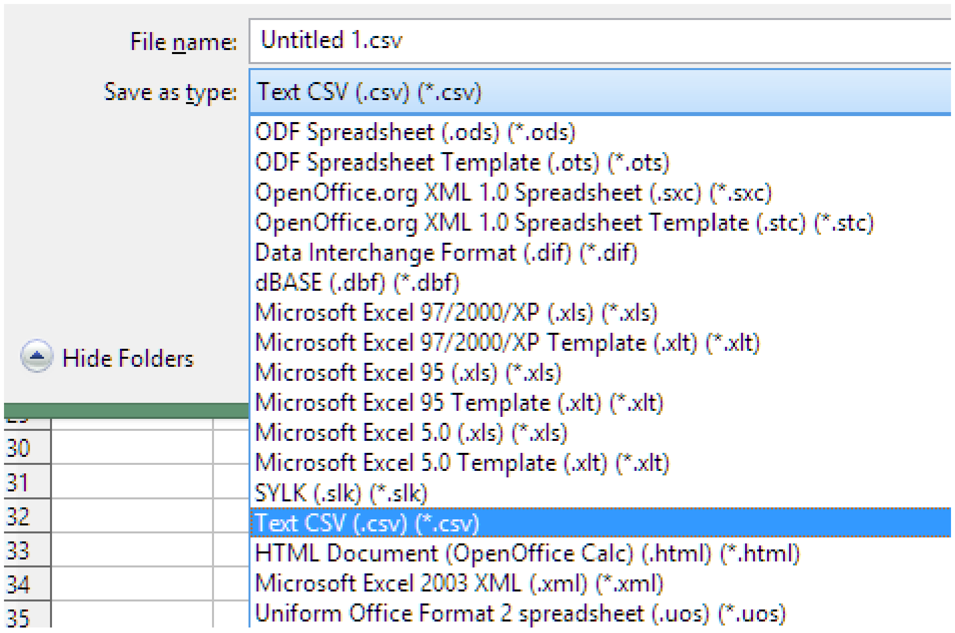
☐ Always save Web pages in the default encoding

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)

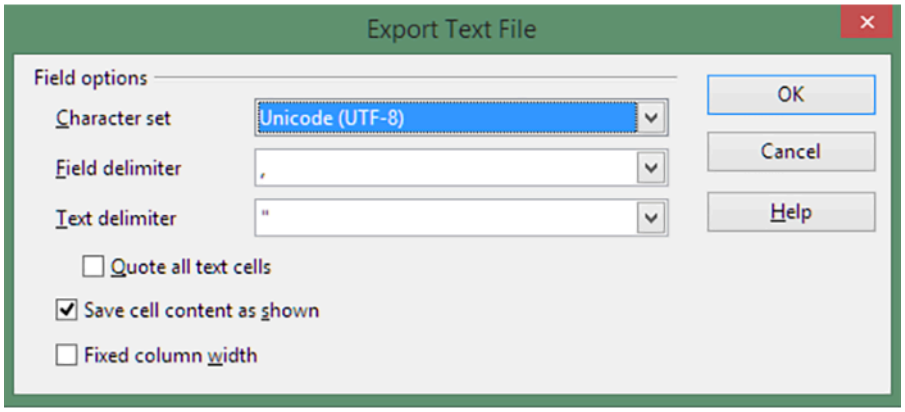


File name: Untitled 1.csv

Save as type: Text CSV (.csv) (*.csv)

- ODF Spreadsheet (.ods) (*.ods)
- ODF Spreadsheet Template (.ots) (*.ots)
- OpenOffice.org XML 1.0 Spreadsheet (.sxc) (*.sxc)
- OpenOffice.org XML 1.0 Spreadsheet Template (.stc) (*.stc)
- Data Interchange Format (.dif) (*.dif)
- dBASE (.dbf) (*.dbf)
- Microsoft Excel 97/2000/XP (.xls) (*.xls)
- Microsoft Excel 97/2000/XP Template (.xlt) (*.xlt)
- Microsoft Excel 95 (.xls) (*.xls)
- Microsoft Excel 95 Template (.xlt) (*.xlt)
- Microsoft Excel 5.0 (.xls) (*.xls)
- Microsoft Excel 5.0 Template (.xlt) (*.xlt)
- SYLK (.slk) (*.slk)
- Text CSV (.csv) (*.csv)
- HTML Document (OpenOffice Calc) (.html) (*.html)
- Microsoft Excel 2003 XML (.xml) (*.xml)
- Uniform Office Format 2 spreadsheet (.uos) (*.uos)

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).



Field options

Character set: Unicode (UTF-8)

Field delimiter: ,

Text delimiter: "

☐ Quote all text cells

☑ Save cell content as shown

☐ Fixed column width

Chapter 15. Working with Rich Text

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

Table 15.1. Supported Rich Text Tags

NAME	DESCRIPTION	TAG	OPTIONS
Paragraph	The <p> tag specifies a paragraph.	<p> </p>	style - Set of attributes that define how to style the text.
Span	The <span> tag is used for styling text.	<span> </span>	style - Set of attributes that define how to style the text.
Bold	The <b> tag specifies bold text.	<b> </b>	None
Italic	The <i> tag specifies italic text.	<i> </i>	None
Underline	The <u> tag specifies underlined text.	<u> </u>	None
Break	The <br> tag specifies a line break.	<br> or </br>	None
Non-Breaking Text	The <nobr> tag specifies text that can't break.	<nobr> </nobr>	None
Font File	This is used to specify a font face for a font family (local file only, truetype fonts only).	<style> @font-face { font-family: roboto-bold; src: url('file:fonts/roboto_bold.ttf') } </style>	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". 2. font-weight - Defines the boldness of the font. Only support value is "bold", otherwise it's treated as "normal"
Ordered Lists	The <ol> tag is used to define an ordered list.	<ol> </ol>	style - Set of attributes that define how to style the list.
Unordered Lists	The <ul> tag is used to define an unordered list.	<ul> </ul>	style - Set of attributes that define how to style the list.
List Items	The <li> tag is used to specify a list item.	<li> </li>	None

Table 15.2. Supported Style Attributes

NAME	DESCRIPTION	ATTRIBUTE NAME	PARAMETERS
Text Color	The color attribute specifies the color of the text.	<p style="color:red;">This is a paragraph.</p> or <p style="color:#FF0000;">This is a paragraph.</p>	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	<p style="background-color:red;">This is a paragraph.</p> or <p style="background-color:#FF0000;">This is a paragraph.</p>	Color can be a color name (ie. "red", "blue") or a hex value (ie. "#FF0000")
Text Alignment	The text-align attribute specifies the alignment of the text block.	<p style="text-align:left">Left Aligned </p>	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>	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>
```

Chapter 16. 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

`gre.gesture.up`	1
`gre.gesture.down`	5
`gre.gesture.right`	3
`gre.gesture.left`	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. These events are all of the form gre.mtXaction, where X is the number of touch points present (between 2 and 5), and action is the name of the event, which will be one of, move, pinch or rotate.

All multi-touch gesture events have the same format of a gre pointer event, with a few extra data fields.

`gre.mtXmove`

This event has an x_move and y_move data field, which will be the difference in x and y of the midpoint of all present touch touches between the current and last event sent from the touchscreen.

`gre.mtXpinch`

This event has a 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 newspacing/oldspacing, so a value of 1.1 indicates a growth of 10% and a value of 0.9 indicates a shrink of 10%

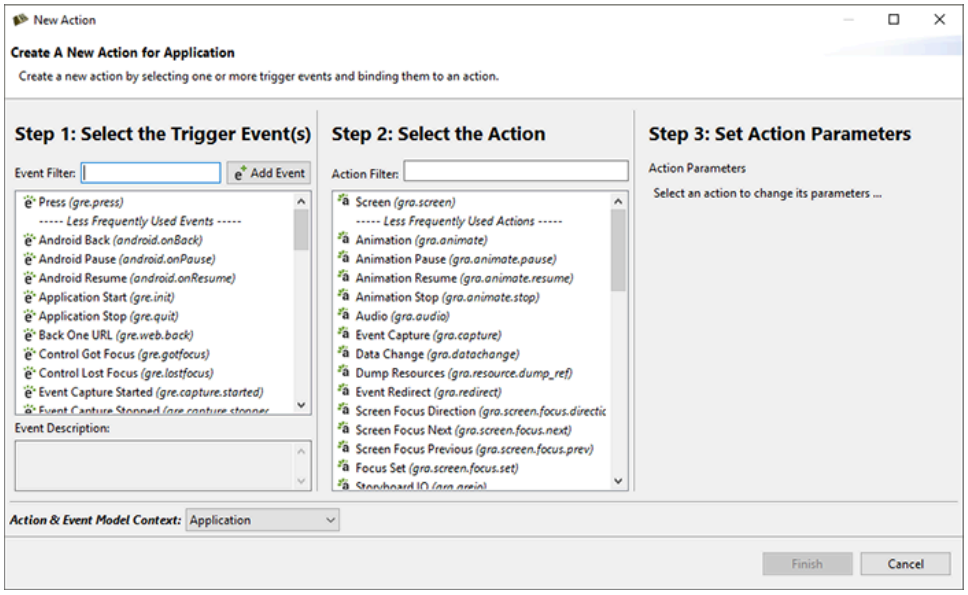
`gre.mtXrotate`

This event has a 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.

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.


New Action

Create A New Action for Application

Create a new action by selecting one or more trigger events and binding them to an action.

Step 1: Select the Trigger Event(s)

Event Filter: Add Event

- Press *(gre.press)*
- ----- *Less Frequently Used Events* -----
- Android Back *(android.onBack)*
- Android Pause *(android.onPause)*
- Android Resume *(android.onResume)*
- Application Start *(gre.init)*
- Application Stop *(gre.quit)*
- Back One URL *(gre.web.back)*
- Control Got Focus *(gre.gotfocus)*
- Control Lost Focus *(gre.lostfocus)*
- Event Capture Started *(gre.capture.started)*
- Event Capture Stopped *(gre.capture.stopped*

Event Description:

Step 2: Select the Action

Action Filter:

- Screen *(gra.screen)*
- ----- *Less Frequently Used Actions* -----
- Animation *(gra.animate)*
- Animation Pause *(gra.animate.pause)*
- Animation Resume *(gra.animate.resume)*
- Animation Stop *(gra.animate.stop)*
- Audio *(gra.audio)*
- Event Capture *(gra.capture)*
- Data Change *(gra.datachange)*
- Dump Resources *(gra.resource.dump_ref)*
- Event Redirect *(gra.redirect)*
- Screen Focus Direction *(gra.screen.focus.directic*
- Screen Focus Next *(gra.screen.focus.next)*
- Screen Focus Previous *(gra.screen.focus.prev)*
- Focus Set *(gra.screen.focus.set)*
- Storyboard IO *(gra.greio)*

Step 3: Set Action Parameters

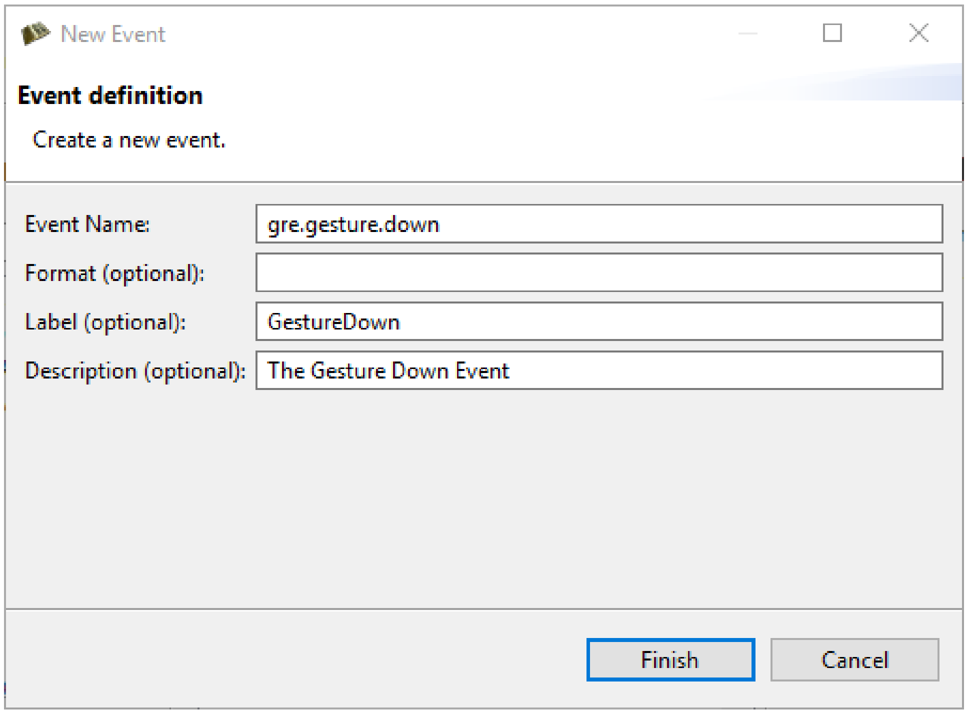
Action Parameters

Select an action to change its parameters ...

Action & Event Model Context: Application

Finish Cancel

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`

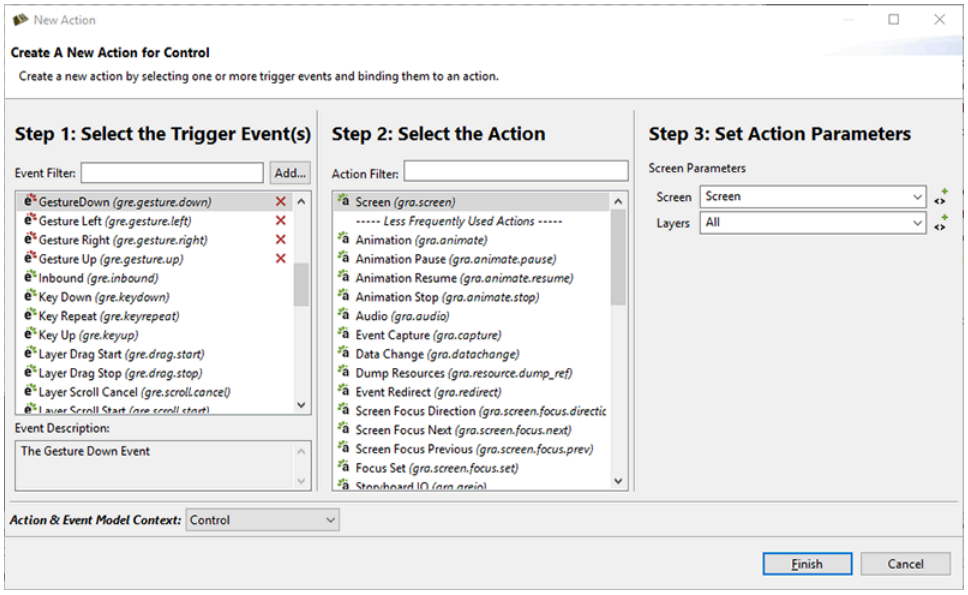


Event definition

Create a new event.

Event Name:	gre.gesture.down
Format (optional):	
Label (optional):	GestureDown
Description (optional):	The Gesture Down Event

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



Create A New Action for Control

Create a new action by selecting one or more trigger events and binding them to an action.

Step 1: Select the Trigger Event(s)

Event Filter:

- GestureDown *(gre.gesture.down)*
- Gesture Left *(gre.gesture.left)*
- Gesture Right *(gre.gesture.right)*
- Gesture Up *(gre.gesture.up)*
- Inbound *(gre.inbound)*
- Key Down *(gre.keydown)*
- Key Repeat *(gre.keyrepeat)*
- Key Up *(gre.keyup)*
- Layer Drag Start *(gre.drag.start)*
- Layer Drag Stop *(gre.drag.stop)*
- Layer Scroll Cancel *(gre.scroll.cancel)*
- Layer Scroll Start *(gre.scroll.start)*

Event Description:

The Gesture Down Event

Step 2: Select the Action

Action Filter:

- Screen *(gra.screen)*
- *----- Less Frequently Used Actions -----*
- Animation *(gra.animate)*
- Animation Pause *(gra.animate.pause)*
- Animation Resume *(gra.animate.resume)*
- Animation Stop *(gra.animate.stop)*
- Audio *(gra.audio)*
- Event Capture *(gra.capture)*
- Data Change *(gra.datachange)*
- Dump Resources *(gra.resource.dump_ref)*
- Event Redirect *(gra.redirect)*
- Screen Focus Direction *(gra.screen.focus.directic*
- Screen Focus Next *(gra.screen.focus.next)*
- Screen Focus Previous *(gra.screen.focus.prev)*
- Focus Set *(gra.screen.focus.set)*
- Storyboard IO *(gra.greio)*

Step 3: Set Action Parameters

Screen Parameters

Screen	Screen
Layers	All

Action & Event Model Context: Control

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

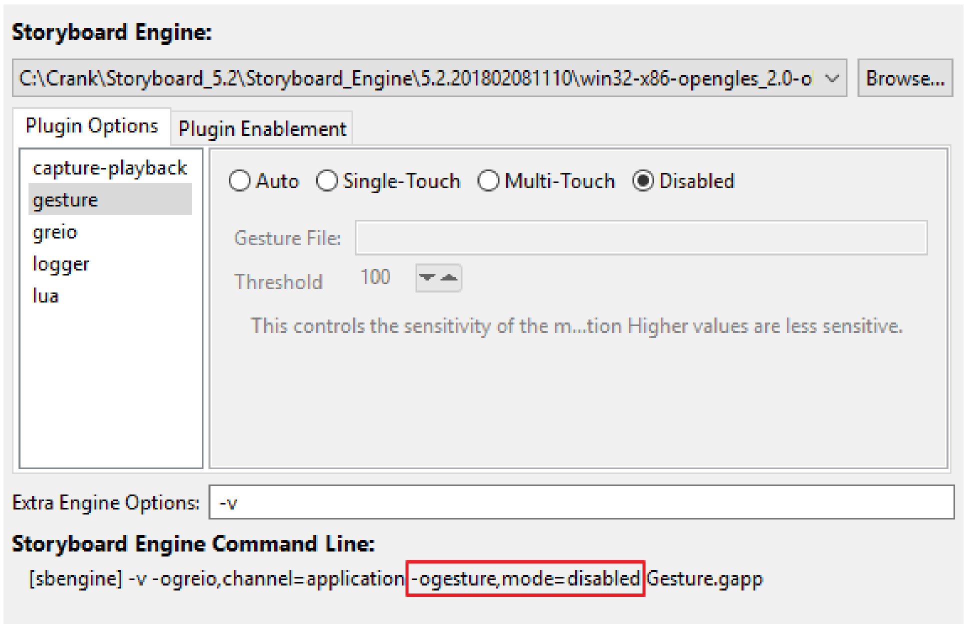
The gesture plugin can be disabled or forced to single or multi-touch mode by passing one of the folowing options to Storyboard Engine. See the section on Plugin Options for details about command line arguments.

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.


Storyboard Engine:

C:\Crank\Storyboard_5.2\Storyboard_Engine\5.2.201802081110\win32-x86-opengles_2.0-o

( ) Auto ( ) Single-Touch ( ) Multi-Touch (●) Disabled

Gesture File:

Threshold 100

This controls the sensitivity of the m...tion Higher values are less sensitive.

Extra Engine Options: -v

Storyboard Engine Command Line:

```
[sbengine] -v -ogreio,channel=application -ogesture,mode=disabled Gesture.gapp
```

Chapter 17. 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 it's 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:

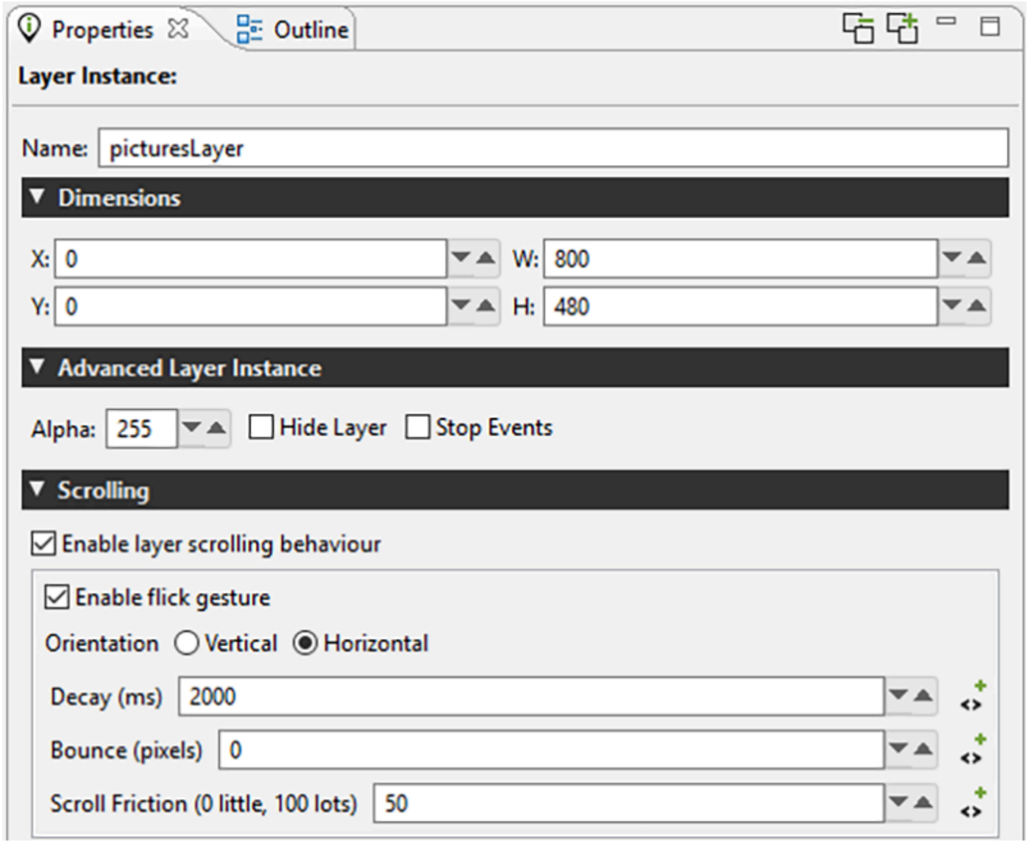
Cell based scrolling using Actions and cell re-positioning
Pixel based scrolling using table `xoffset` and `yoffset`
It is not suggested to mix and match between the two techniques.

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 18, *Working with Scrolling Content*.

Chapter 18. 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 17, *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.



Layer Instance:

Name: picturesLayer

X: 0
W: 800
Y: 0
H: 480

Alpha: 255 ☐ Hide Layer ☐ Stop Events

☑ Enable layer scrolling behaviour

☑ Enable flick gesture

Orientation ○ Vertical ● Horizontal

Decay (ms) 2000

Bounce (pixels) 0

Scroll Friction (0 little, 100 lots) 50

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
Decay	Maximum time in milliseconds the scrolling layer will scroll after a flick

Bounce	Number of pixels for the scrolling layer to bounce when reaching the edge of the contents within the layer
Scroll Friction	A number from 0 to 100 to determine the friction level of the scrolling layer. With higher friction it becomes more resistant to moving when swiped and more inclined to slowing down when released

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`

```
    positionPercent = (-1 * objectYOffset) / (contentTotalHeight - objectHeight)
```

Here, we'll assumed 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 * tablInfo.yoffset) /
    (totalHeight - tableInfo.height)

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

Chapter 19. 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.

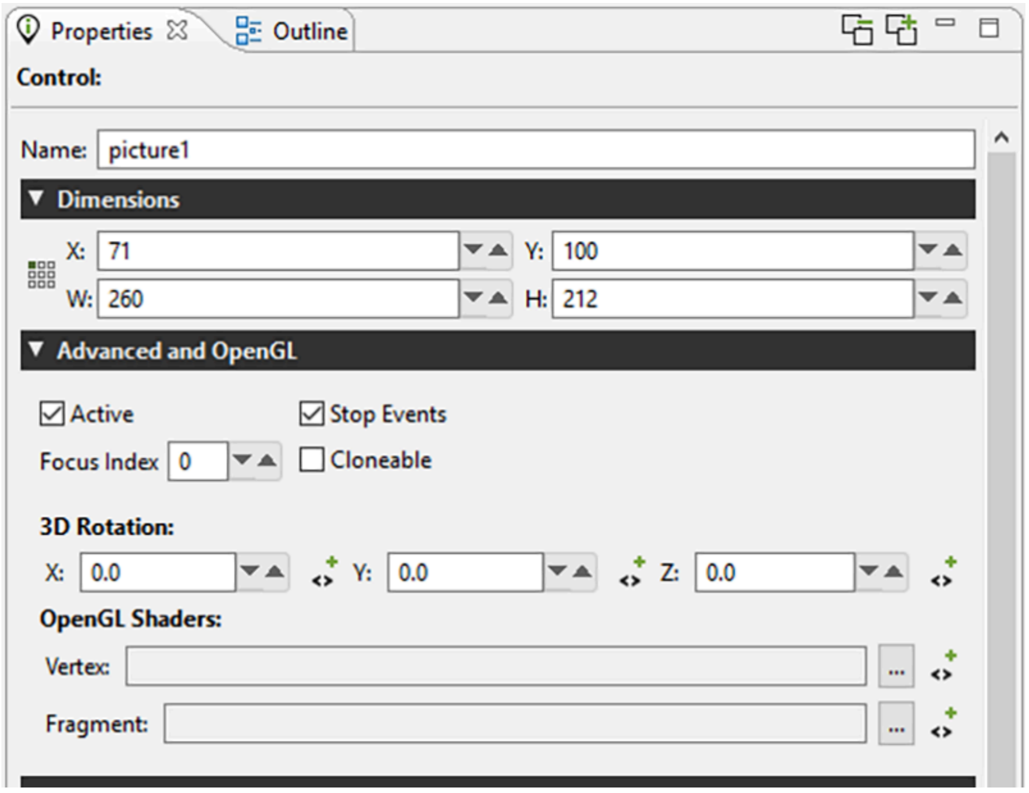
- Use an unlit shading model. Prebake lighting/ambient occlusion first to still have a nice lighting on the model without the runtime cost
- Turn culling on. This will have the impact of drawing less content and will speed up overall rendering operations.
- Use smaller texture maps and consolidate texture 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.



Control:

Name: picture1

Dimensions

X: 71 Y: 100
W: 260 H: 212

Advanced and OpenGL

☑ Active ☑ Stop Events
Focus Index 0 ☐ Cloneable

3D Rotation:
X: 0.0 Y: 0.0 Z: 0.0

OpenGL Shaders:
Vertex:
Fragment:

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 the 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:

`grd_a`	This variable is resolved at the Application level
`grd_l`	This variable is resolved at the Layer level (layer where the control is)
`grd_g`	This variable is resolved at the Group level (group where the control is)
`grd_c`	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:

`r`	float
`g`	float
`b`	float
`a`	float

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 OpenGLES 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/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TextureTool/TextureTool.html]

Chapter 20. Working with Audio and Video

Storyboard Suite 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 Suite 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 re-open 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 application 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.

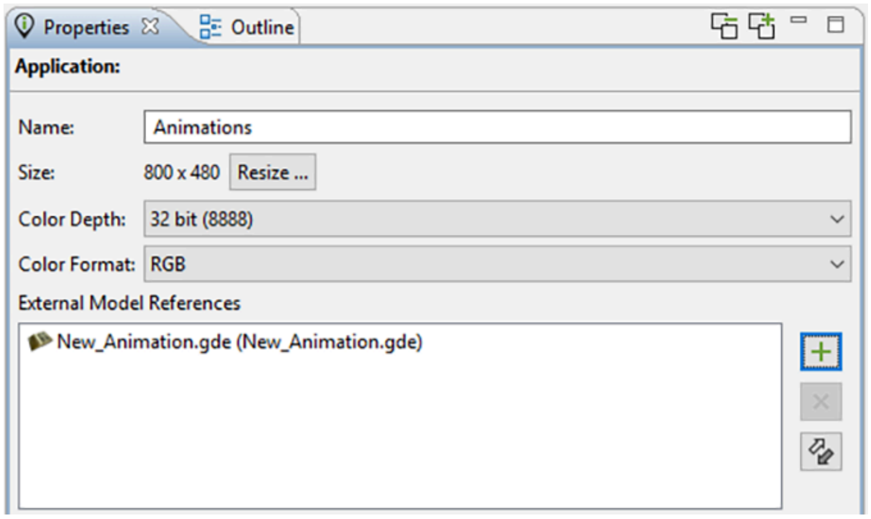
Chapter 21. Multi-File Application Development

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



Application:

Name: Animations

Size: 800 x 480 Resize ...

Color Depth: 32 bit (8888)

Color Format: RGB

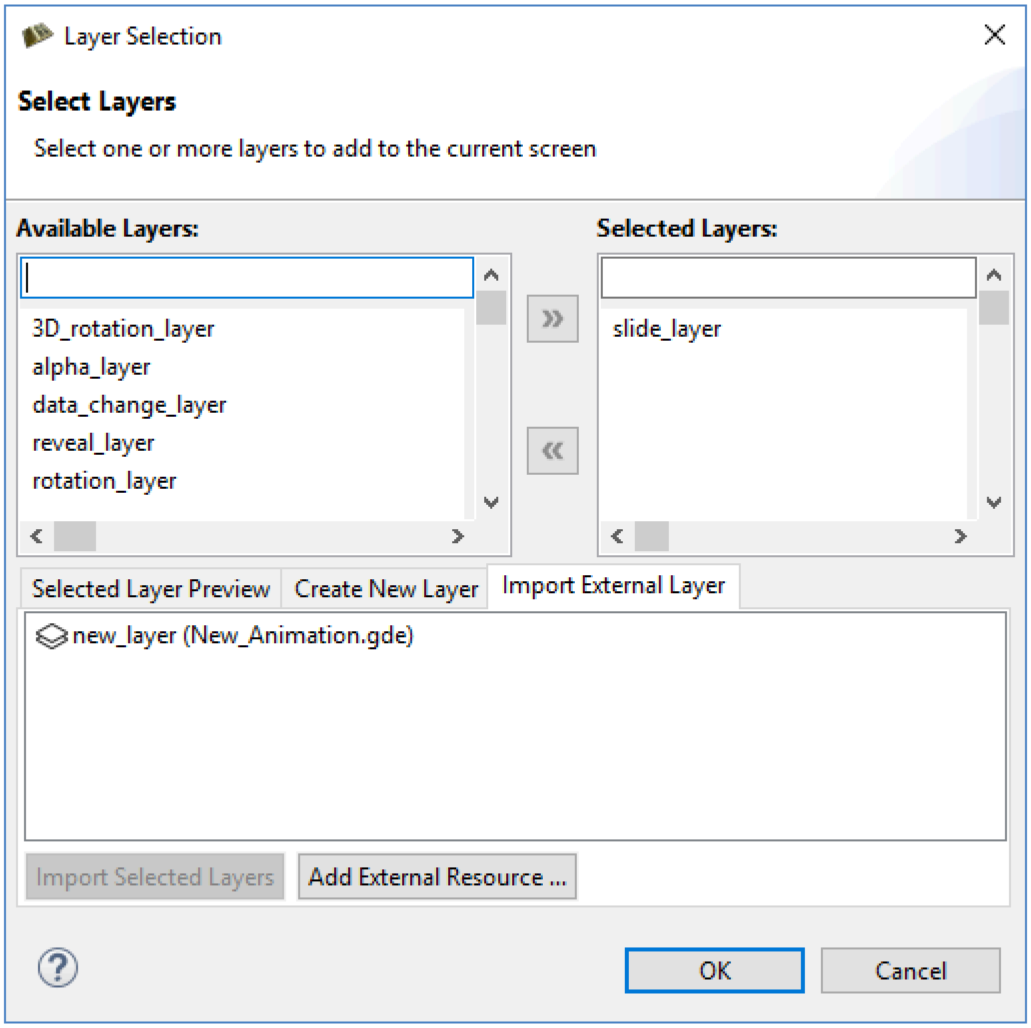
External Model References

New_Animation.gde (New_Animation.gde)

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

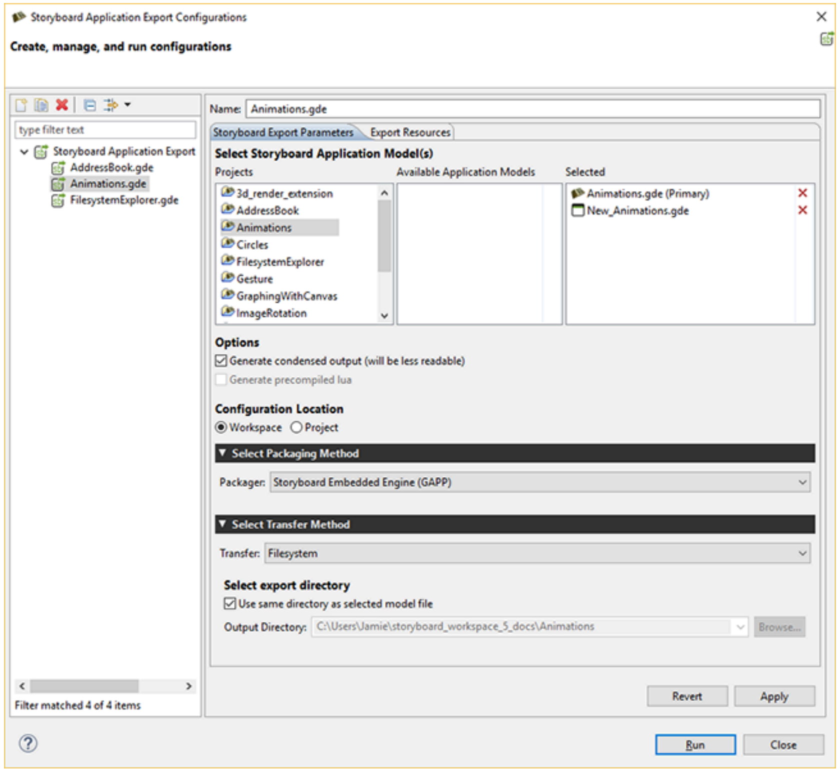


Once a layer is added as an external layer, then a copy of the original layer from the external application is snapshotted and incorporated directly into the source application. At this point, the external layer can be used with any source application screens just like any other layers.

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.


Storyboard Application Export Configurations

Create, manage, and run configurations

type filter text

Storyboard Application Export
- AddressBook.gde
- Animations.gde
- FilesystemExplorer.gde

Filter matched 4 of 4 items

Name: Animations.gde

Select Storyboard Application Model(s)

Projects: 3d_render_extension, AddressBook, Animations, Circles, FilesystemExplorer, Gesture, GraphingWithCanvas, ImageRotation

Selected: Animations.gde (Primary), New_Animations.gde

Options
- [✓] Generate condensed output (will be less readable)
- [ ] Generate precompiled lua

Configuration Location: [✓ Workspace] Project

Select Packaging Method

Packager: Storyboard Embedded Engine (GAPP)

Select Transfer Method

Transfer: Filesystem

Select export directory
- [✓] Use same directory as selected model file

Output Directory: C:\Users\Jamie\storyboard_workspace_5_docs\Animations

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 22. 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 is it 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 23. 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

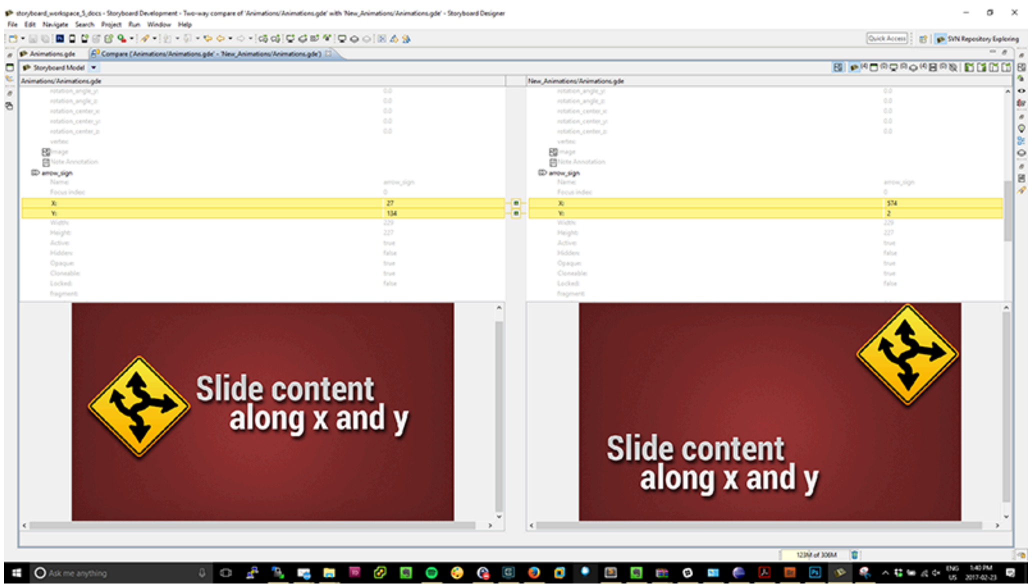
Comparing and Merging Model Files

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.

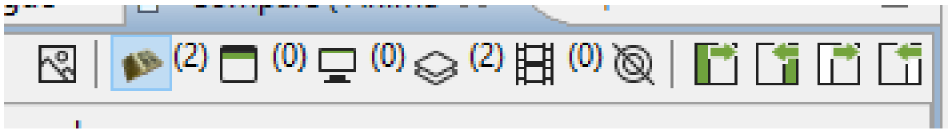


Triggering a comparison provides a hierarchical breakdown of the models' objects, with two sides representing the two files. Any differences between the two models will be highlighted in yellow. The two types of differences are property changes and additions/deletions.

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/redo 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}
```

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 is 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.

Comparing and Merging Projects

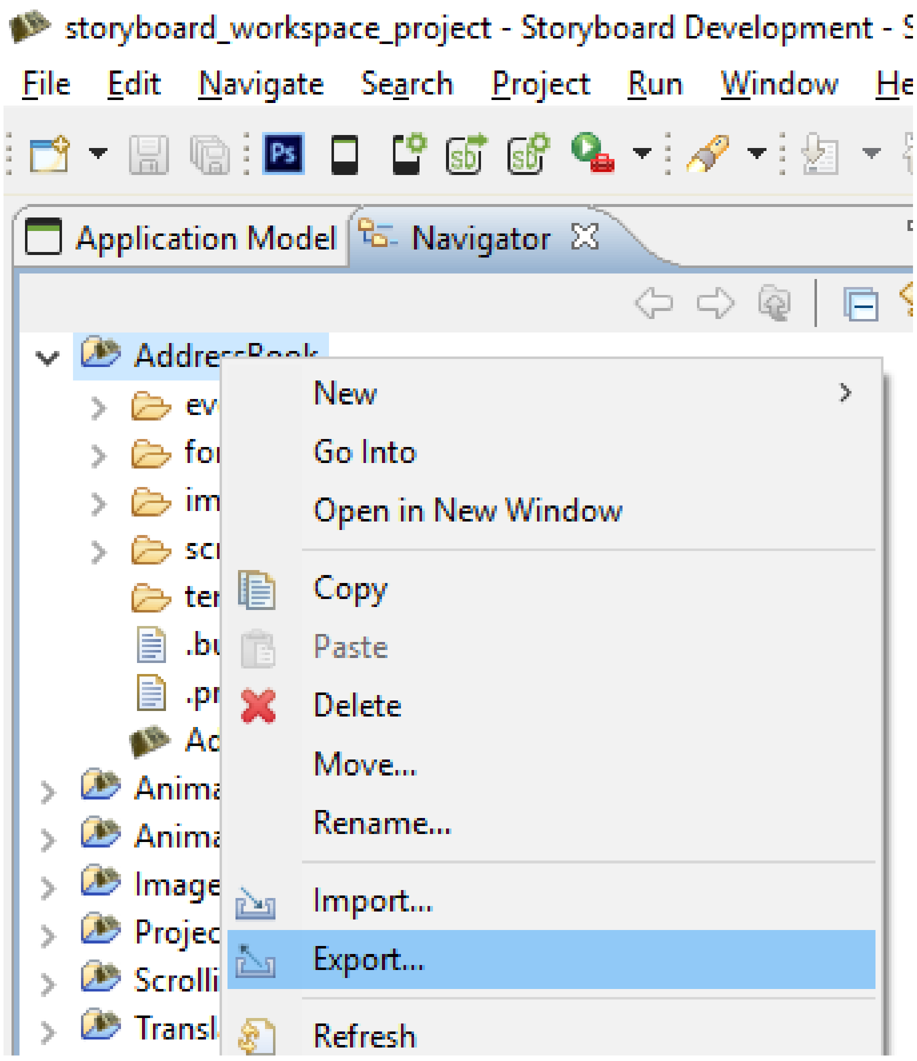
Entire projects and directory structures can be compared as easily as comparing single model files. This will allow developers to understand which resources such as images, fonts and script files have changed in addition to the changes to the model logic. The same as a model file comparison, this can 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.

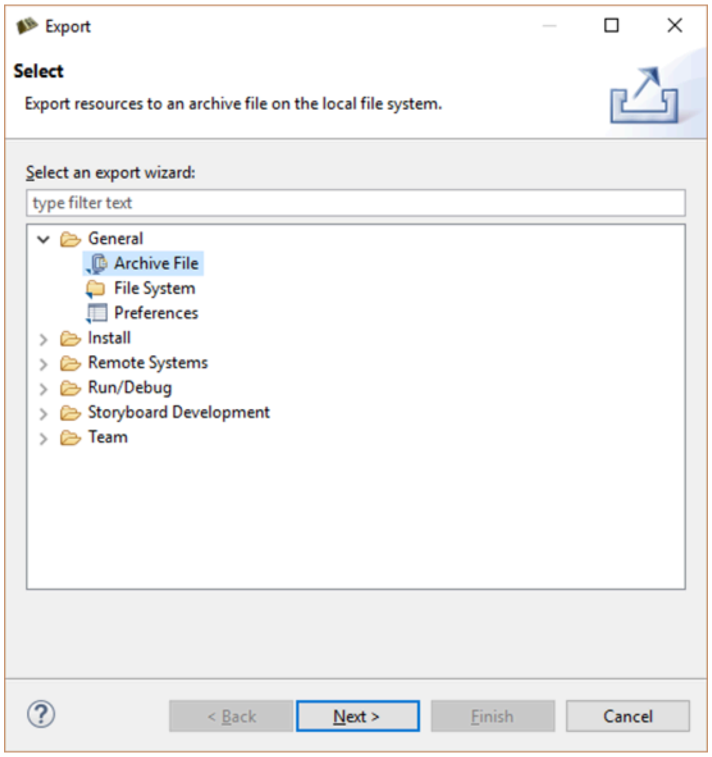
Exporting Storyboard Projects for Sharing

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.

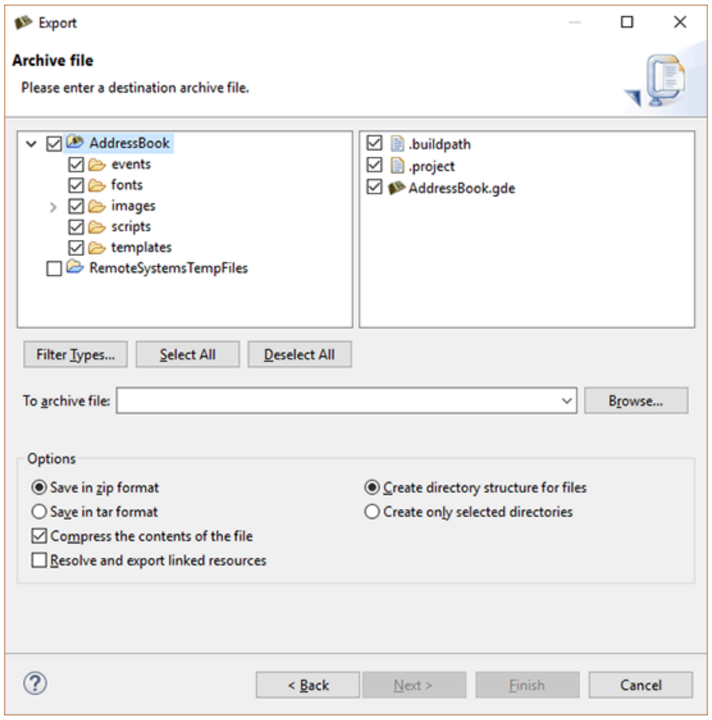
When your Storyboard Project is complete and you are ready to export, right click on the project folder and select Export.



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.



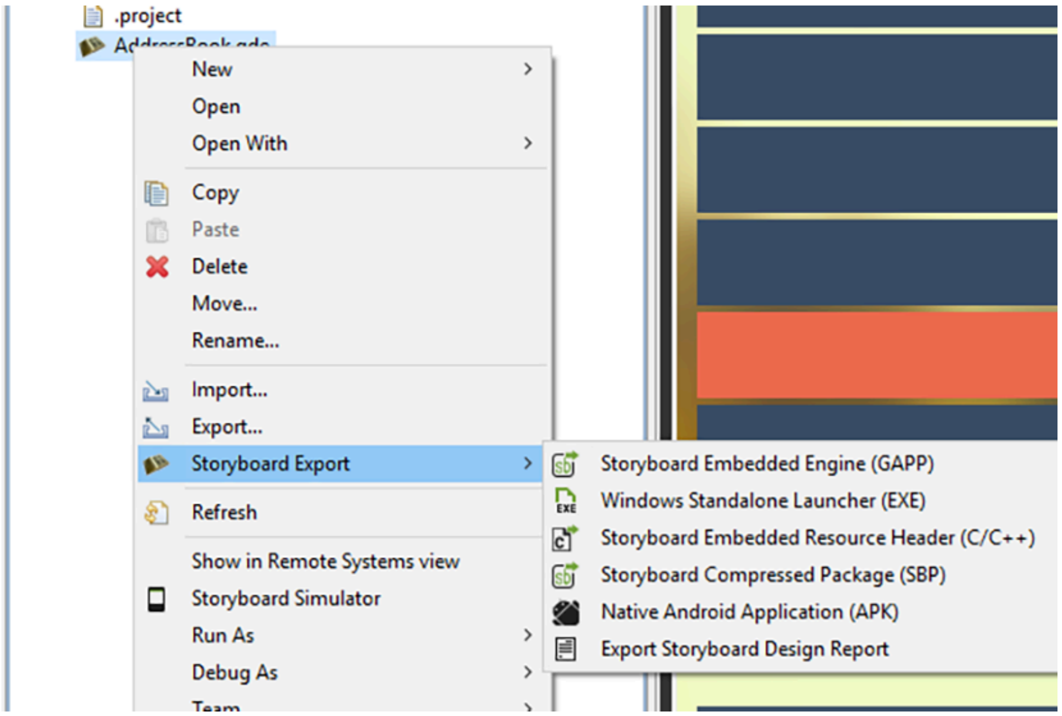
Chapter 24. Exporting and Running on your Embedded Target

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.

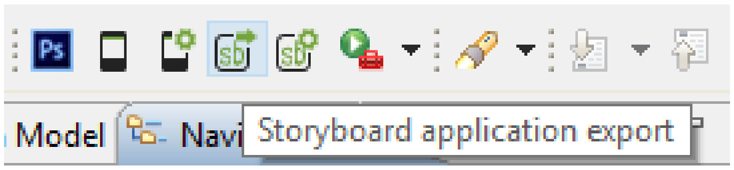
In the past deployment exports were done through a few different export wizards, but as of Storyboard Suite 5.0 these have been consolidated into a single export configuration system. This export system allows the user to create, save, and share their application export parameters. These configurations can be customized through the Storyboard Application Export Configuration dialog.

Export Workflow

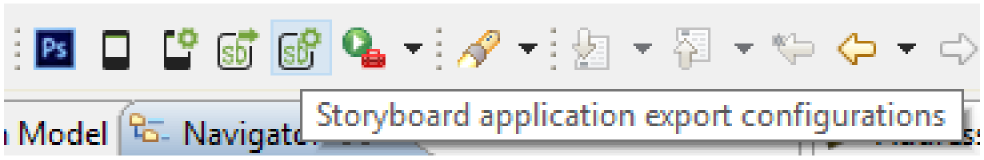
To export a Storyboard Deployment Bundle, right-click the Storyboard application file (*.gde) and select Storyboard Export, and choose the deployment type.


.project
AddressBook.gde
New
Open
Open With
Copy
Paste
Delete
Move...
Rename...
Import...
Export...
Storyboard Export
Refresh
Show in Remote Systems view
Storyboard Simulator
Run As
Debug As
Team
Storyboard Embedded Engine (GAPP)
Windows Standalone Launcher (EXE)
Storyboard Embedded Resource Header (C/C++)
Storyboard Compressed Package (SBP)
Native Android Application (APK)
Export Storyboard Design Report

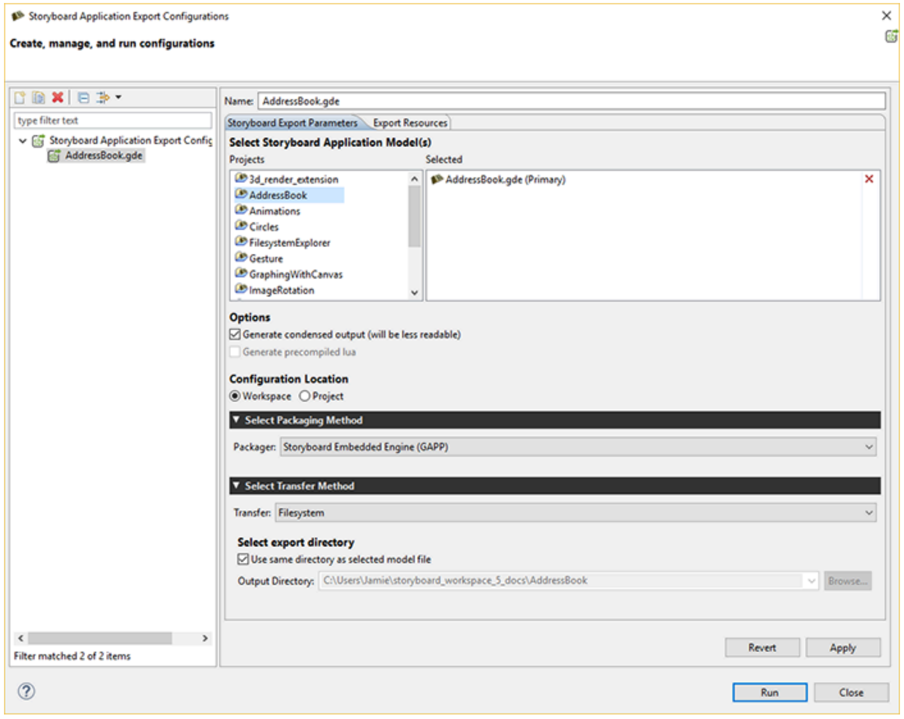
An export can also be launched by clicking on the Storyboard Application Export button in the toolbar. This button will launch the first export configuration found for the current application, or create one if none exist.



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.


Storyboard Application Export Configurations

Create, manage, and run configurations

type filter text

Storyboard Application Export Config
AddressBook.gde

Filter matched 2 of 2 items

Name: AddressBook.gde

Storyboard Export Parameters | Export Resources

Select Storyboard Application Model(s)

Projects	Selected
3d_render_extension	AddressBook.gde (Primary)
AddressBook	
Animations	
Circles	
FilesystemExplorer	
Gesture	
GraphingWithCanvas	
ImageRotation	

Options

[✓] Generate condensed output (will be less readable)

[ ] Generate precompiled lua

Configuration Location

[✓ Workspace] Project

Select Packaging Method

Packager: Storyboard Embedded Engine (GAPP)

Select Transfer Method

Transfer: Filesystem

Select export directory

[✓] Use same directory as selected model file

Output Directory: C:\Users\Jamie\storyboard_workspace_5_docs\AddressBook

Browse... Revert Apply Run Close

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.myapp).

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 or IP address), 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 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 full 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).

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=<PathToGDEFile[,PathToAdditionalGDEFile,...]>
    configuration=<ConfigurationName>
```

Mac:

```
..../Storyboard.app/Contents/MacOS/Storyboard -application
    com.crank.gdt.ui.configexport
    model=<PathToGDEFile[,PathToAdditionalGDEFile,...]>
    configuration=<ConfigurationName>
```

Linux:

```
..../Storyboard -application com.crank.gdt.ui.configexport
    model=<PathToGDEFile[,PathToAdditionalGDEFile,...]>
    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=<PathToGDEFile[,PathToAdditionalGDEFile,...]>
    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.

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.

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 24.1. Options

OPTION	DESCRIPTION
-i	Displays which version of sbengine and related libs are being used

OPTION	DESCRIPTION
-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_CONSOLEDEVICE=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 muti-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 25. 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 at the top right of the page. Here you will need the device's name and Device ID. To get these connect the device to your computer and find it in iTunes. With the device selected click on the device's serial number, and it will switch to the Device's Identifier. Then click Edit > Copy Identifier (UDID). Now return to your Provisioning Portal, paste the identifier and enter the name of the device.

Application IDs

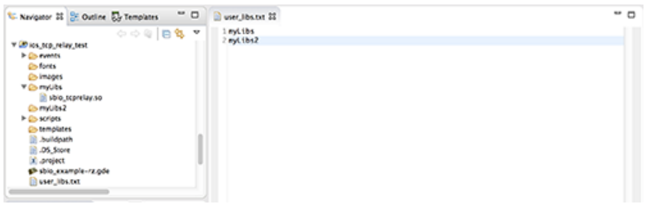
Log into your iOS Development account and click on the iOS Provisioning Portal link at the right hand side of the page. In your Provisioning Portal, click App IDs in the left tab bar. For this section we recommend you set up a generic App ID and have it accept all of you applications. However if you wish to enable other iOS like Push Notifications or In-App Purchase, you will need to make an ID for that individual app. To create a generic App ID, click the Add New App ID Button. Now enter a description of the app the Id will match with, e.g., "Application Development ID". Now enter the Bundle identifier. If this is a generic App ID simply type "*". If this is for a specific app, enter the app's identifier. For example, "com.cranksoftware.storyboardApp". Now click submit to go back to the previous page, find the App ID, and configure all of the options that this specific application needs.

Provisioning Profile

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 Provisioning in the left tab bar. Click the New Profile button. Create a name for this profile. Select the certificates that will be used by this profile. Select the App ID that will be used by this profile. Select the Devices that will be used by this profile. Once this is completed, download the YourProvisionProfileName.mobileprovision file and save it to your computer. When you are exporting your Storyboard application, you will need to tell Storyboard where this file is.

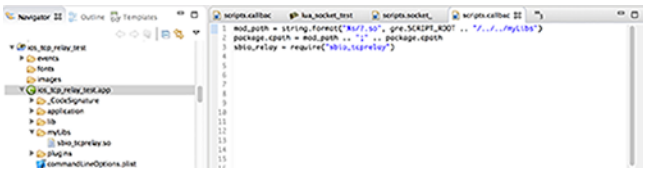
Adding Extra Libraries for iOS

Users can also add additional libraries that aren't included in the runtime to an exported iOS application. Similar to Android, Storyboard needs to know about them beforehand so that we can pass them through the code signing process. To do this, create a text file called user_libs.txt that contains any folder names as strings that lead to added libraries that we need to check for signing. The folders should be placed at the project root. This text file should also be placed in the root of the project. A valid setup looks like this:



```
myLibs
myLibs2
```

Now that your libraries have been signed and are available for use, if you need to access them through Lua you can do so using the helper variable gre.SCRIPT_ROOT. You can look at the generated .app file to determine the script root relative path to your library folders, which will be located at the root of the .app folder. That path needs to then be appended to the package.cpath variable in Lua using a semicolon. In the above example, it would look like this:

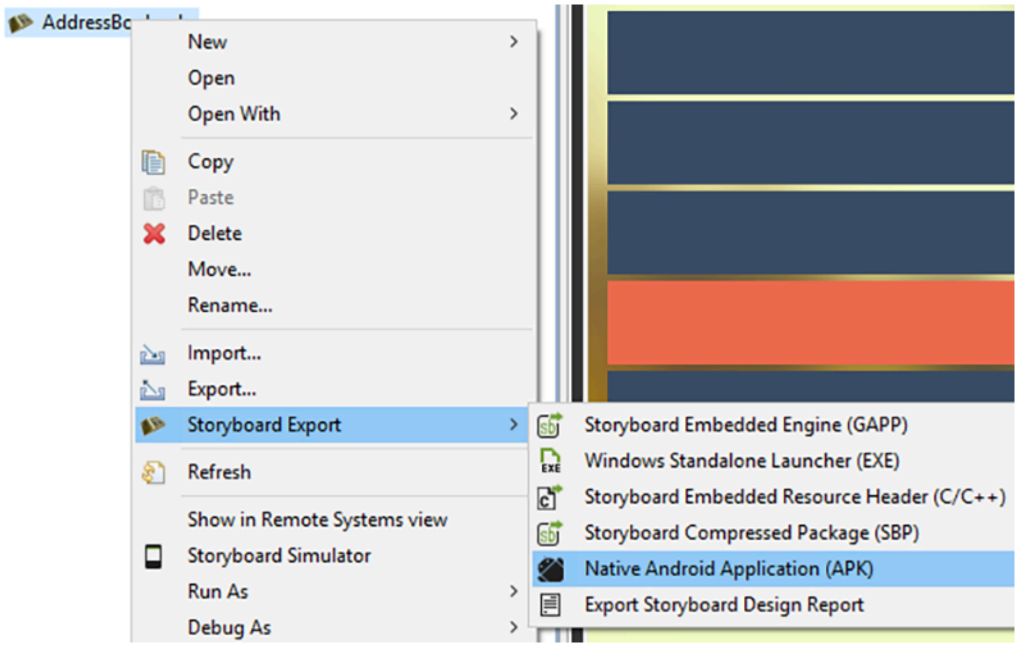


```
mod_path = string.format("%s/?.so", gre.SCRIPT_ROOT .. "/../../myLibs")
package.cpath = mod_path .. ";" .. package.cpath
sbio_relay = require("sbio_tcprelay")
```

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*.

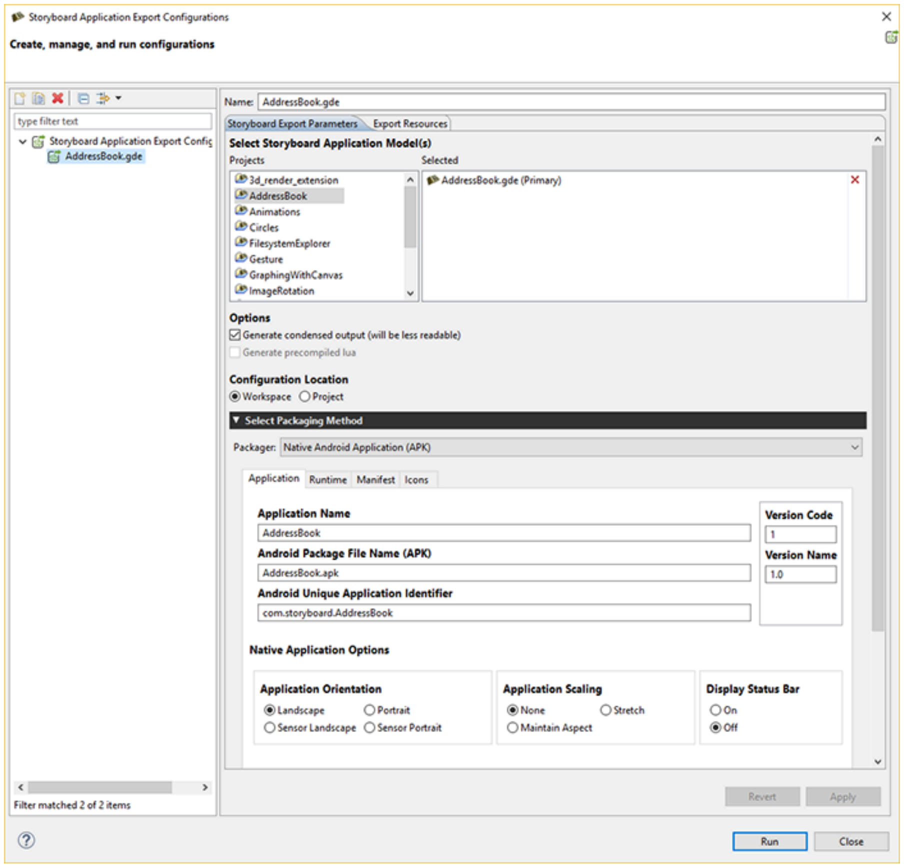
To create an Android package, right-click the project .gde File > Storyboard Export > Export as Native Android Application.


AddressBo
New
Open
Open With
Copy
Paste
Delete
Move...
Rename...
Import...
Export...
Storyboard Export
Refresh
Show in Remote Systems view
Storyboard Simulator
Run As
Debug As
Storyboard Embedded Engine (GAPP)
Windows Standalone Launcher (EXE)
Storyboard Embedded Resource Header (C/C++)
Storyboard Compressed Package (SBP)
Native Android Application (APK)
Export Storyboard Design Report

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.



Storyboard Application Export Configurations

Create, manage, and run configurations

type filter text

Storyboard Application Export Config
AddressBook.gde

Filter matched 2 of 2 items

Name: AddressBook.gde

Storyboard Export Parameters | Export Resources

Select Storyboard Application Model(s)

Projects	Selected
3d_render_extension	AddressBook.gde (Primary)
AddressBook	
Animations	
Circles	
FilesystemExplorer	
Gesture	
GraphingWithCanvas	
ImageRotation	

Options
- [✓] Generate condensed output (will be less readable)
- [ ] Generate precompiled lua

Configuration Location
[✓ Workspace] Project

Select Packaging Method

Packager: Native Android Application (APK)

Application | Runtime | Manifest | Icons

Application Name: AddressBook
Android Package File Name (APK): AddressBook.apk
Android Unique Application Identifier: com.storyboard.AddressBook
Version Code: 1
Version Name: 1.0

Native Application Options

Application Orientation: [✓ Landscape] Portrait Sensor Landscape Sensor Portrait
Application Scaling: [✓ None] Stretch Maintain Aspect
Display Status Bar: On [✓ Off]

Revert | Apply

Run | Close

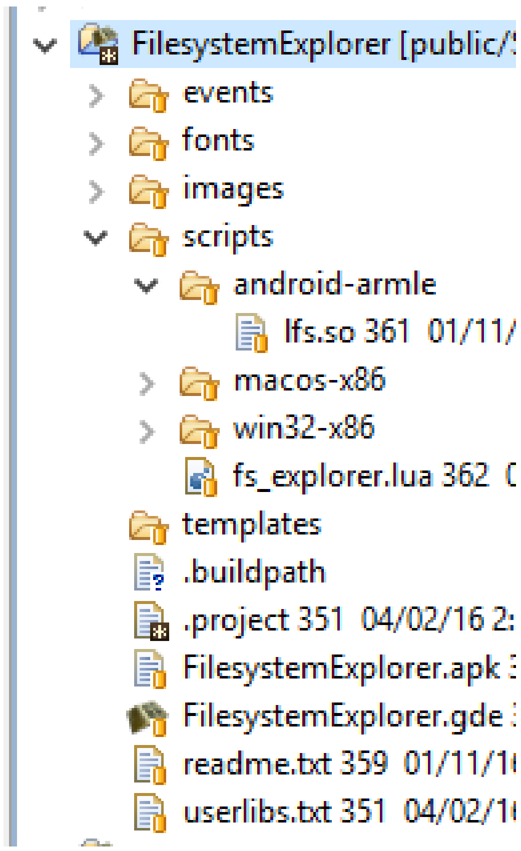
Select the Runtime tab to set Storyboard runtime options (see Storyboard Engine Options for list of available runtime options). Select the Manifest tab for advanced options to modify the default Permissions, Manifest file or Keystore settings (optional). Select the Icons tab for selecting the icons for 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.

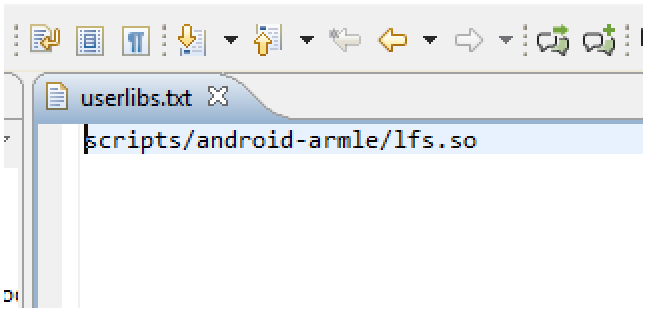
Adding Extra Libraries for Android

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.

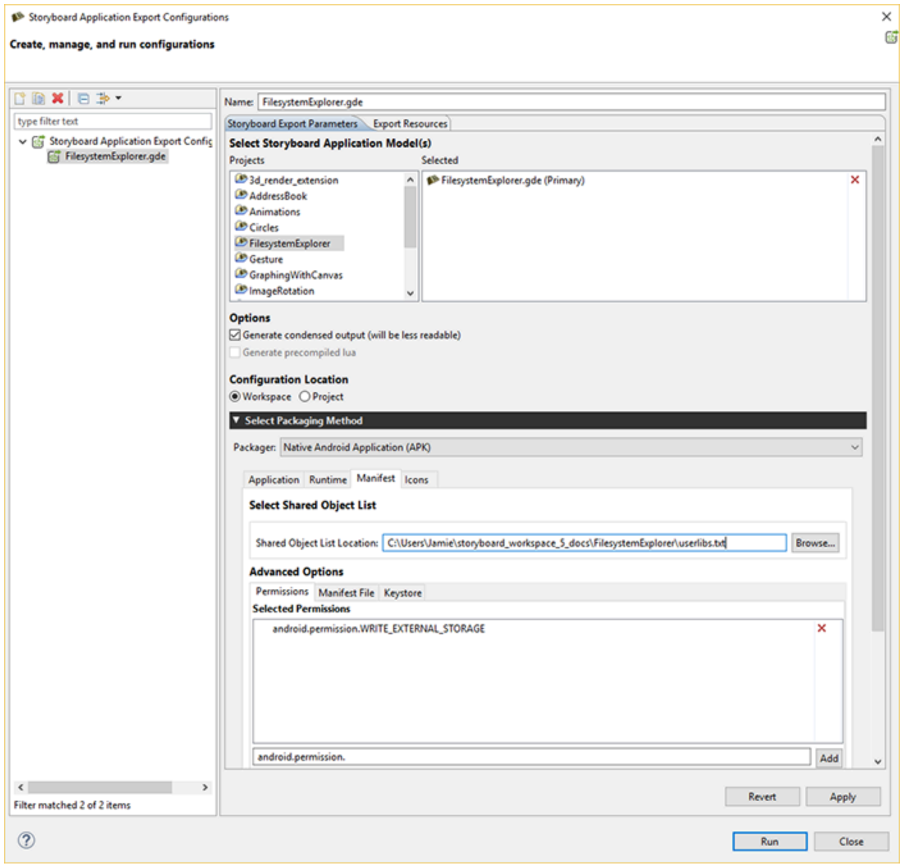


Create a text file, which we'll call user_libs.txt, with the following contents:



```
scripts/android-armle/lfs.so
```

When exporting the app make sure to include the path to this file and hit finish.


Storyboard Application Export Configurations

Create, manage, and run configurations

Name: FilesystemExplorer.gde

Storyboard Export Parameters | Export Resources

Select Storyboard Application Model(s)

Projects: 3d_render_extension, AddressBook, Animations, Circles, FilesystemExplorer, Gesture, GraphingWithCanvas, ImageRotation

Selected: FilesystemExplorer.gde (Primary)

Options

[✓] Generate condensed output (will be less readable)

[ ] Generate precompiled lua

Configuration Location

[✓ Workspace] Project

Select Packaging Method

Packager: Native Android Application (APK)

Application | Runtime | Manifest | Icons

Select Shared Object List

Shared Object List Location: C:\Users\Jamie\storyboard_workspace_5_docs\FilesystemExplorer\userlibs.txt

Advanced Options

Permissions | Manifest File | Keystore

Selected Permissions

android.permission.WRITE_EXTERNAL_STORAGE

android.permission.

Filter matched 2 of 2 items

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.CALL_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 create 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 26. 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 and inter-process 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 can be specified in the command option for sbengine. The Storyboard application can transmit events on one or more channels. 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.

The Storyboard IO plugin provides a single communication channel which clients can used 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 on top of native message passing and communication APIs, dependent on the operating system:

Linux	SysV message queue
MacOS	SysV message queue
QNX	QNX POSIX message queue (`mqueue` 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 `mqueue` 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.

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 allocated 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;
}

/*
 * 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",
             "1s0 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

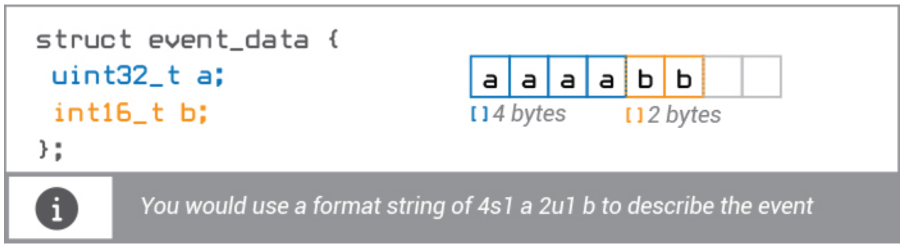
Any data associated with an event as its payload must be a linearly contiguous block of memory. In order for clients receiving the event to decode the event data memory block back into structured content a key must be provided. The format string that is provided as part of the event is this decoding 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

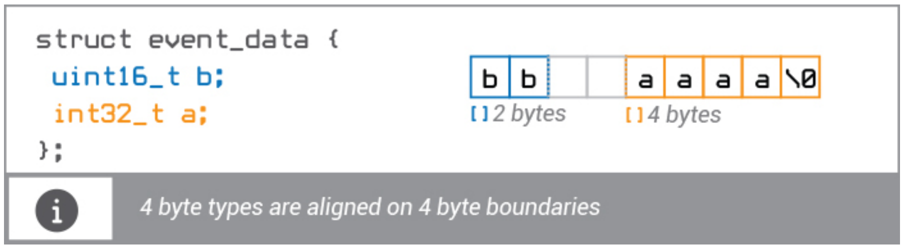
So, if you were transmitting the following C/C++ structure you would presume that the bytewise memory layout would be:


```
struct event_data {
 uint32_t a;
 int16_t b;
};
```
a
a
a
a
b
b
[ ] 4 bytes
[ ] 2 bytes
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;
};
```

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.

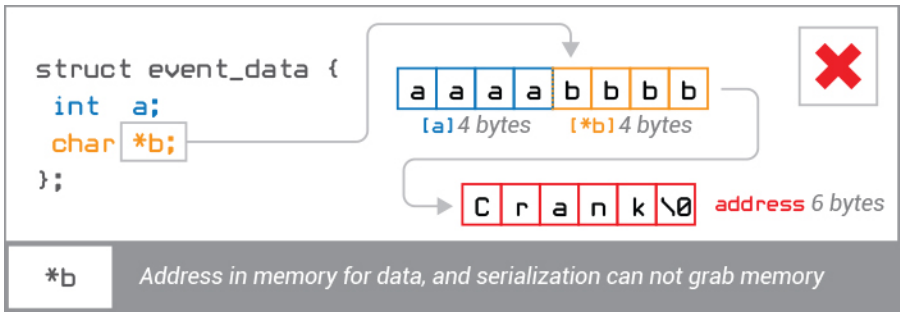
Here the 32 bit/4 byte member of the structure `a` comes after `b` but there are two additional bytes of padding inserted to ensure `a` starts on a 4 byte memory boundary. Since the format string must describe the linear memory layout for clients, we would have to change the format string to accomodate 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:

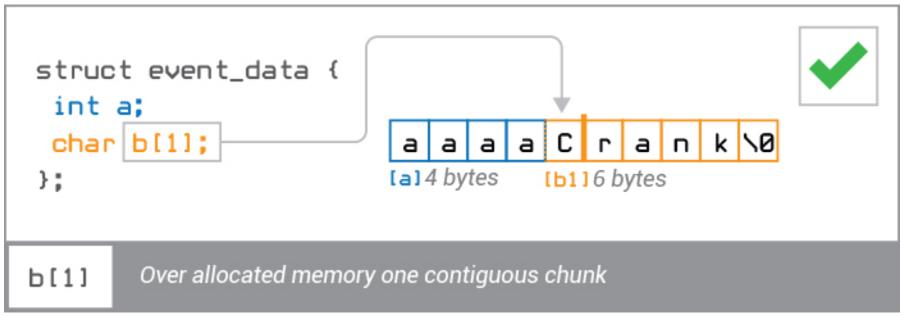
```
char * event_data = "Crank";
```

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++ idom 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",
  "1s0",
  ptr, strlen(ptr)+1);
```

```
x = 1;
ret = gre_io_add_mdata(&md_buffer,
  "Test.Number",
  "4u1",
  &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 the 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.

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_unserialze()` 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 backend)

Table 26.1. 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 issue. You can use iogen and iorcv to verify permissions.

Table 26.2. 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

Possible Issue	Resolution
	to trigger the event was executed. If you see `WARN [0.135]:Can't open greio channel` 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 26.3. 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 26.4. 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. `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 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 Suite 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.vebosity' 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 `iorvc` 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 27. Optimizing Your Storyboard Application

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 screen manager listens for data changes and checks the state of controls to determine when the display needs to be refreshed. If the data for controls is changing rapidly this may cause thrashing of the display and possible flicker if not using double buffering. When changing data values, moving controls, or generating events which would cause the display to be updated, 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`

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 of the performance events.

For more information on the performance monitoring plugin and its capabilities, 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 internal variables that can be used at runtime to display performance information. The following Storyboard variables, can be used to extract information from the runtime:

grd_fps (string, 1s0)	The frame rate of display updates averaged over the last 5 seconds of display. This value is only generated if the `-oscreen_mgr,fps` option is passed along to the sbengine binary. 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.
grd.animation.name (string, 1s0)	The name of the last completed animation.
grd.animation.frames (number, 4s0)	The number of frames rendered for the last animation run.

grd.animation.duration (number, 4s1)	The duration in milliseconds (ms) of the last animation run.

This sample demonstrates how you can use a Lua script to extract and print these values to the display.

```
function show_metrics(mapargs)
    local fps_key = "grd_fps"
    local name_key = "grd.animation.name"
    local frame_key = "grd.animation.frames"
    local duration_key = "grd.animation.duration"

    local msg
    local data = gre.get_data(fps_key, name_key, frame_key, duration_key)

    -- FPS generated every 5s, assuming: -oscreen_mgr,fps
    if(data[fps_key]) then
        msg = string.format("Screen FPS: %d", data[fps_key])
        print(msg)
    end

    -- Animation data only available after animation complete
    if(data[name_key]) then
        msg = string.format("Animation %s took %d ms @ %d fps", data[name_key],
        data[duration_key], data[frame_key])
        print(msg)
    end
end
```

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 you application don't scale the image, this is a performance hit at image render time. It's far better to use you 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 (3)

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.
- Set sbengine's `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.

Chapter 28. Extending Storyboard Functionality

Storyboard Suite, 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 Suite 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:

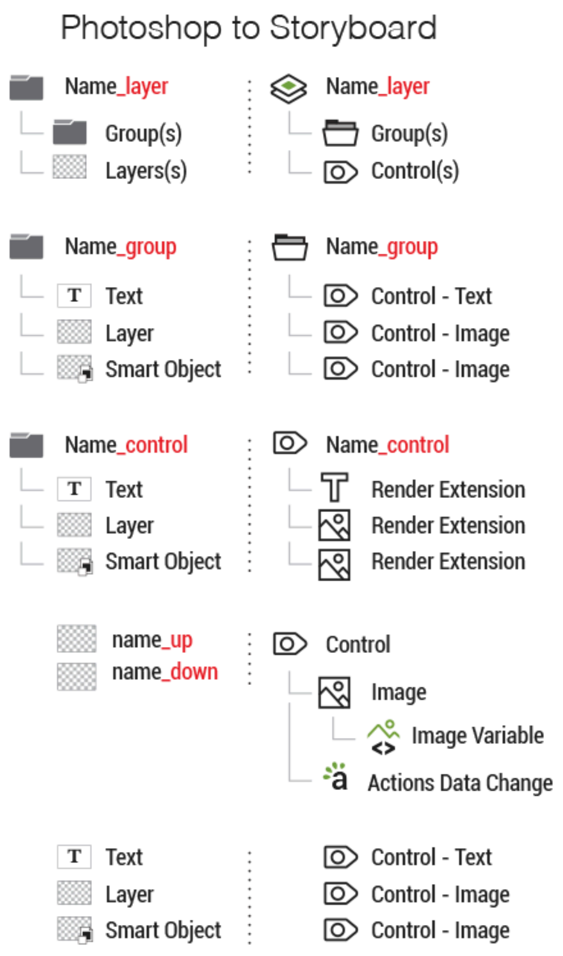
boolean	A boolean true/false value. An optional attribute, 'default' can be used to specify the default value.
color	A color value. An optional attribute, 'default' can be used to specify the default value as a hexadecimal value (0xRRGGBB).
file	A file string. An optional 'extensions' argument list contains 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 29. 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.



PSD File Requirements

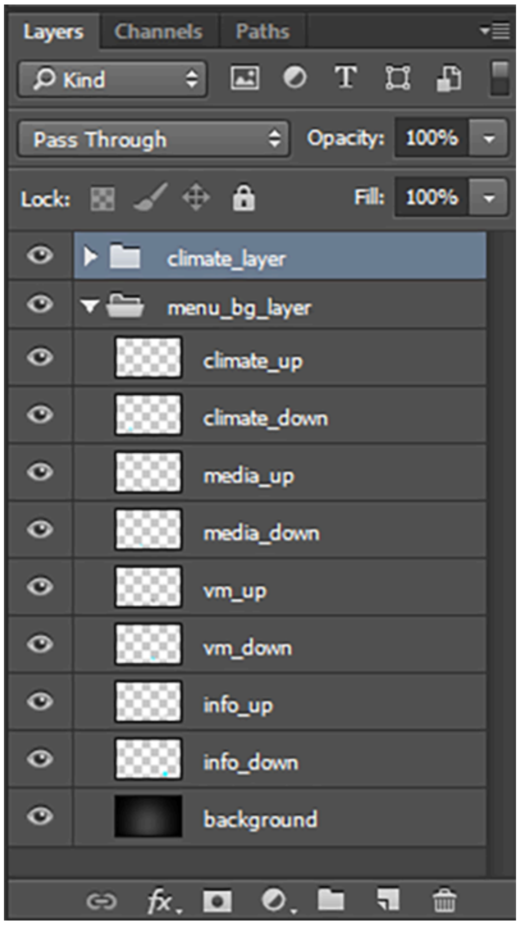
In order to import files from Photoshop™ to Storyboard, documents must be in a RGB color mode as an 8-bit file. Other formats are not supported.

Application Model Hierarchy

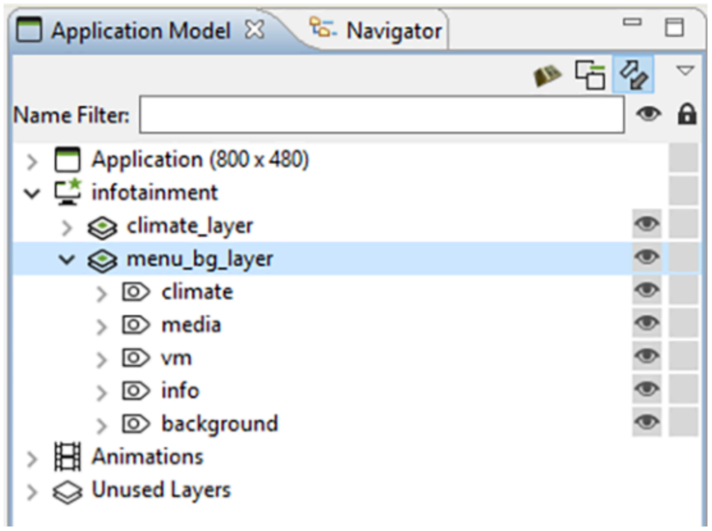
Understanding the Storyboard Application Model Hierarchy will help users decide how to name and organize content in Photoshop to better control how a Storyboard application comes together when using the PSD import feature.

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.



climate_layer
menu_bg_layer
climate_up
climate_down
media_up
media_down
vm_up
vm_down
info_up
info_down
background



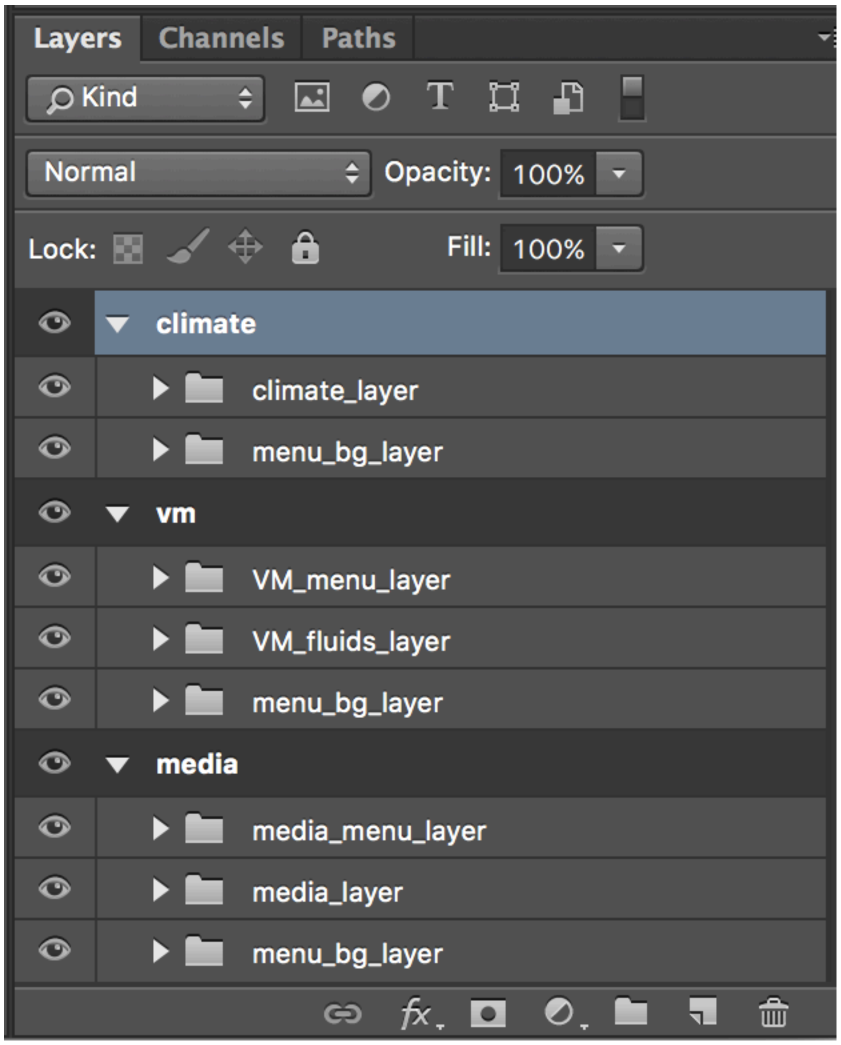
Application (800 x 480)
infotainment
climate_layer
menu_bg_layer
climate
media
vm
info
background
Animations
Unused Layers

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-orginize 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.

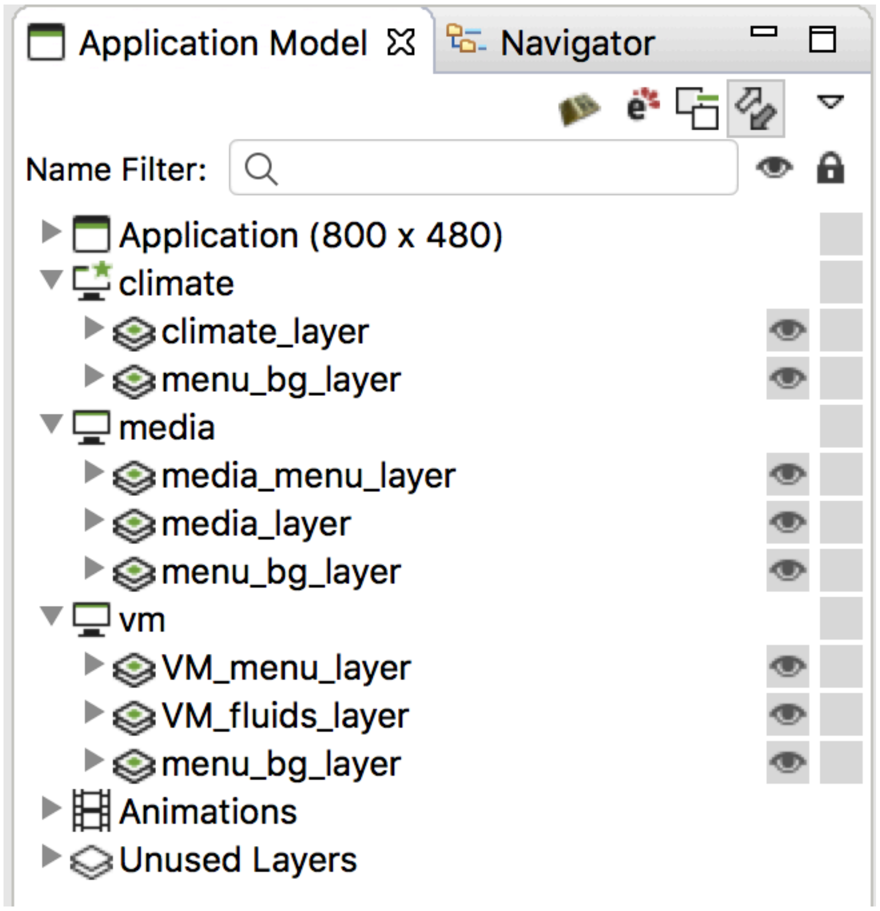
Below is how a new project is laid out after a PSD file with artboards has been imported into Storyboard.



climate
- climate_layer
- menu_bg_layer

vm
- VM_menu_layer
- VM_fluids_layer
- menu_bg_layer

media
- media_menu_layer
- media_layer
- menu_bg_layer



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

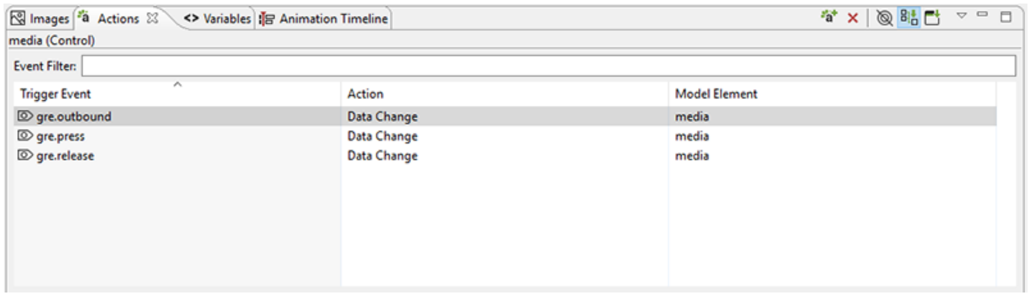
Photoshop group folders named with _group at the end will import to Storyboard as a group. Any content within these groups will import as controls within that group.

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.



media (Control)

Event Filter:

Trigger Event	Action	Model Element
gre.outbound	Data Change	media
gre.press	Data Change	media
gre.release	Data Change	media

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 30. Re-Importing Photoshop and Updating Content

Re-Importing Photoshop Content

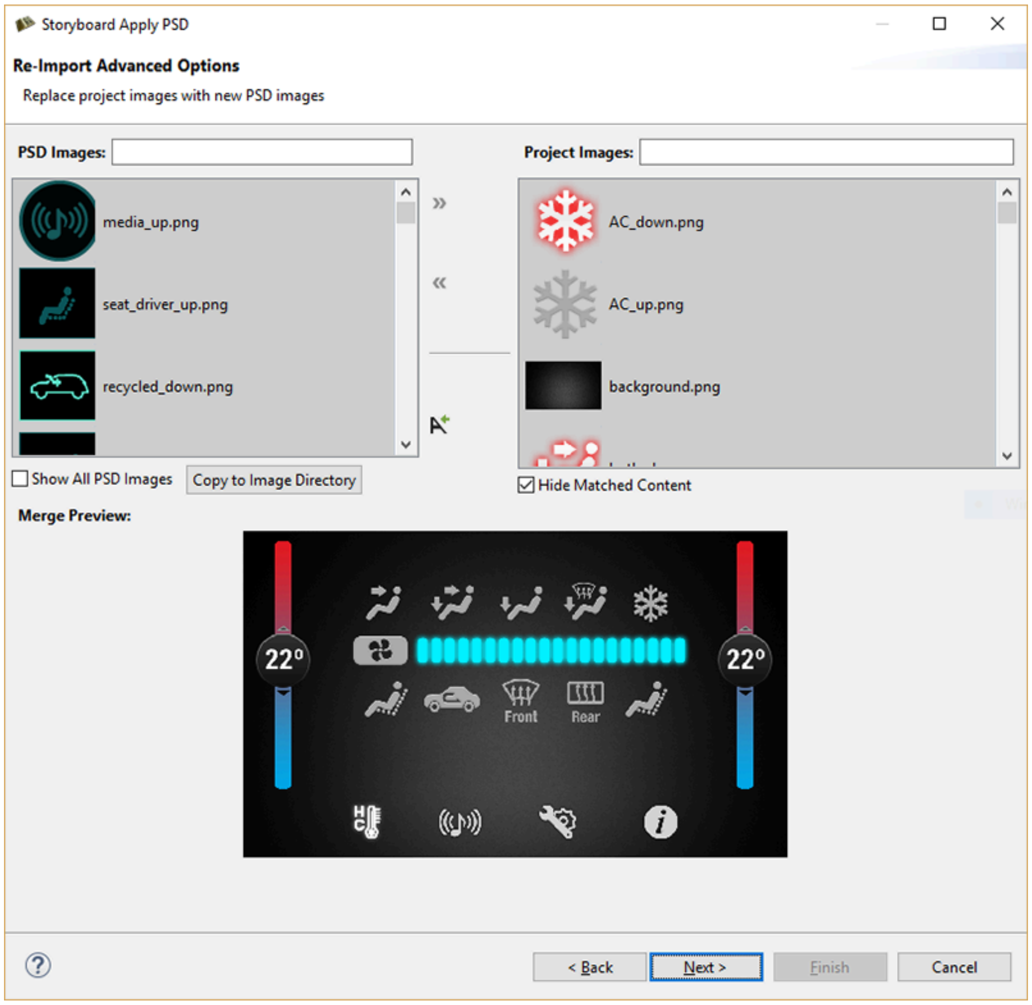
The Photoshop™ Re-Import feature enables the user to re-import a new or revised PSD file to replace existing images in a project's image directory. The re-import wizard is initiated from the main menu » File » Import

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.


Storyboard Apply PSD

Re-Import Advanced Options

Replace project images with new PSD images

PSD Images:

- media_up.png
- seat_driver_up.png
- recycled_down.png

☐ Show All PSD Images

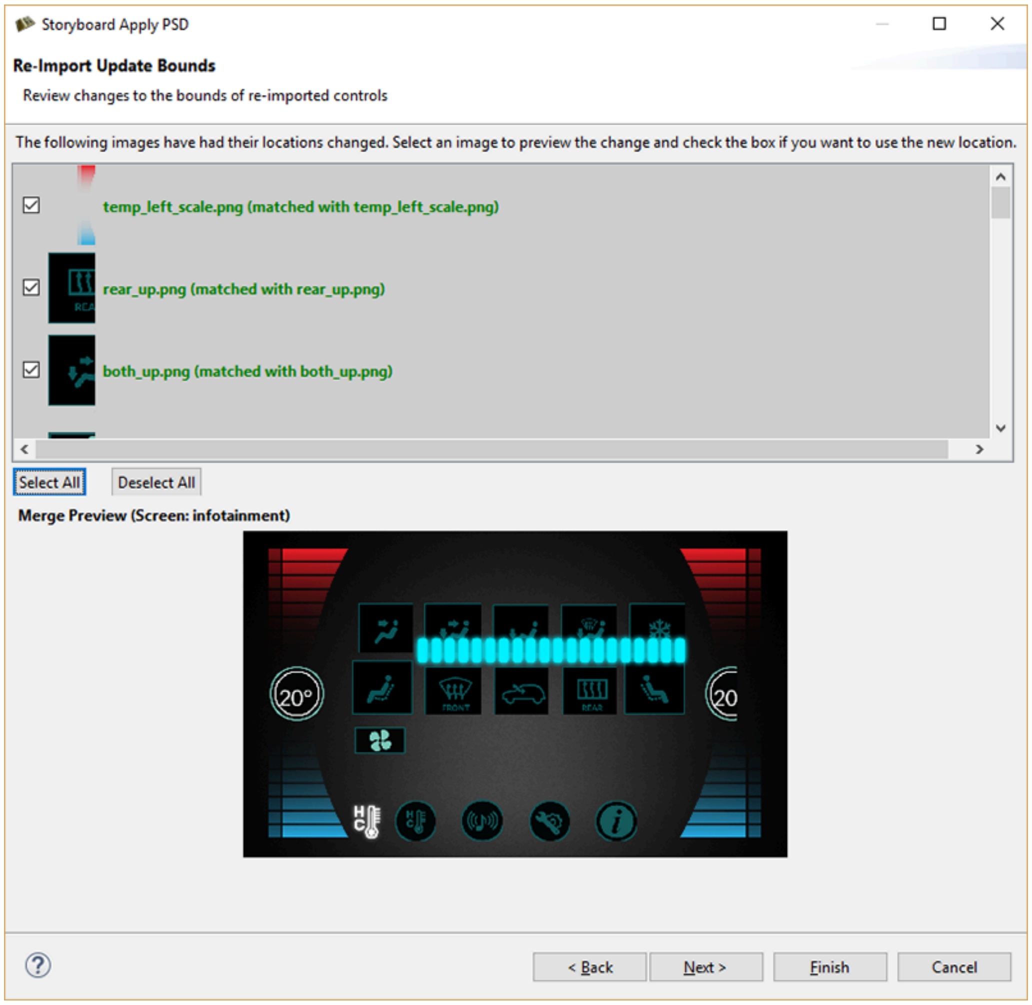
Project Images:

- AC_down.png
- AC_up.png
- background.png

☑ Hide Matched Content

Merge Preview:

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'.



Re-Import Update Bounds

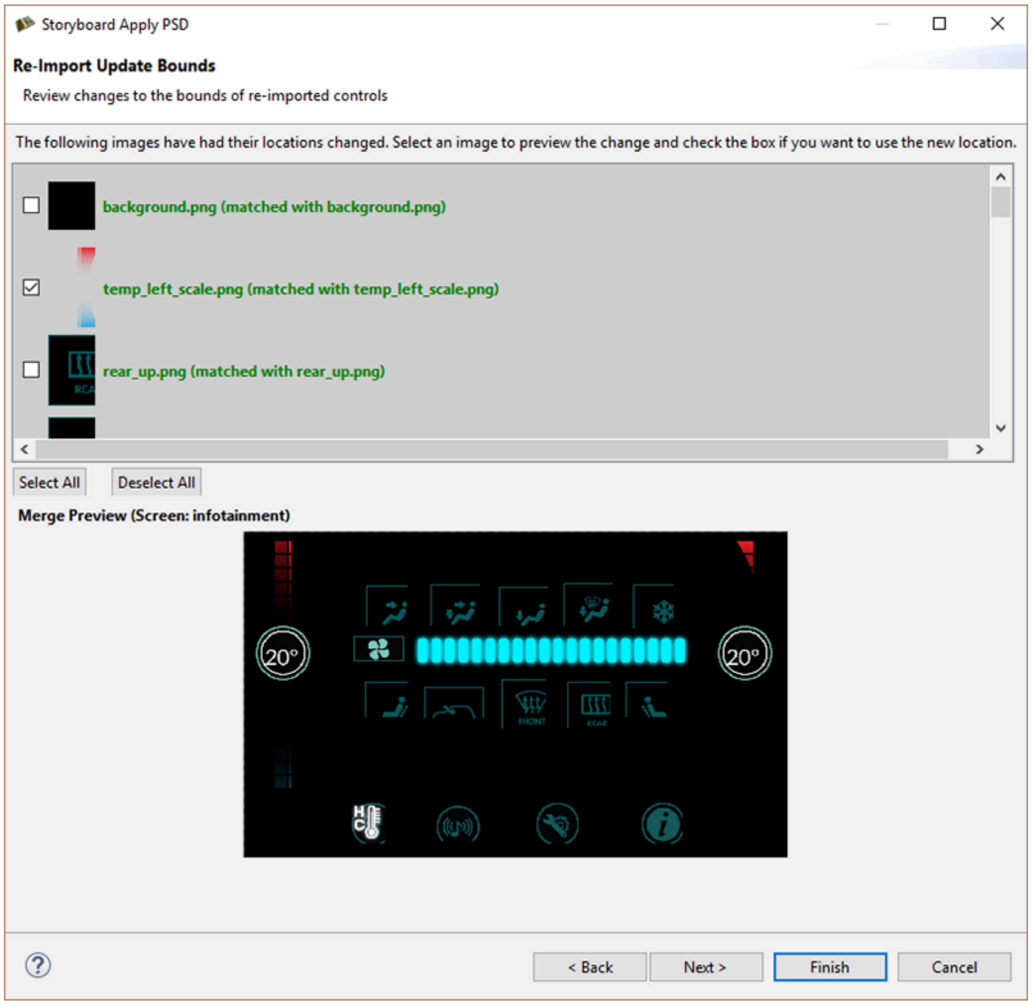
Review changes to the bounds of re-imported controls

The following images have had their locations changed. Select an image to preview the change and check the box if you want to use the new location.

- [✓] temp_left_scale.png (matched with temp_left_scale.png)
- [✓] rear_up.png (matched with rear_up.png)
- [✓] both_up.png (matched with both_up.png)

Merge Preview (Screen: infotainment)

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.


Storyboard Apply PSD

Re-Import Update Bounds

Review changes to the bounds of re-imported controls

The following images have had their locations changed. Select an image to preview the change and check the box if you want to use the new location.

- [ ] background.png (matched with background.png)
- [x] temp_left_scale.png (matched with temp_left_scale.png)
- [ ] rear_up.png (matched with rear_up.png)

Merge Preview (Screen: infotainment)

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_PLUGINS`environment 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.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

    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 mentioned in section 6.1.12 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

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

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, active, zindex, xoffset, yoffset

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

alpha, hidden, active, x, y, zindex, width, height, xoffset, yoffset, effect

Note

Any change to the width and height values affect all layers.

The “effect” attribute requires the libgre-plugin-effects-3d.so plugin. This plugin is only available for OpenGL ES 2.0.

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.move_control

```
gre.move_control(
    control_name,
    dx,
    dy,
    x,
    y
)
```

Move a controls to a new position. The control_name is the name of the control or a variable. Setting dx and 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:

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)

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:

layer_name	The model full path of the layer 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)

gre.set_focus

```
gre.set_focus(
    fqn
)
```

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:

fqn	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:
local success, error = gre.send_event("my_event", “io_channel_name”)
if(success == false) then
  print(error)
  --handle error
end

```
-- Send to the event directed at a particular control target:
gre.send_event("my_event", "my_layer.my_control")
```

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 (layer_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 x_y_string = gre.poly_string(x_points, y_points)
print("X Y String: " .. x_y_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
)
```

Calculate the area in pixels which the given string will occupy on the screen. Optionally calculate how many characters can fit into a predefined screen area.

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 (optional, by default there is no clip)

Returns:

A table containing the following entries:
"num_bytes" number of bytes that will fit in the clip
"width" string width in pixels as clipped by clip width
"height" string height in pixels
"line_height" height in pixels of the specified font

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")
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 lightuserdata 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:

animation_id	The animation to trigger
data	A table containing the tags and values for the extra arguments to set
id	The animation id used in the case of multiple animations with the same name
context	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:

animation_id	The animation to stop
data	A table containing the tags and values for the extra arguments to set
id	The animation id used in the case of multiple animations with the same name
context	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:

animation_id	The running animation to pause
data	A table containing the tags and values for the extra arguments to set
id	The animation id used in the case of multiple animations with the same name
context	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:

animation_id	The running animation to pause
data	A table containing the tags and values for the extra arguments to set
id	The animation id used in the case of multiple animations with the same name
context	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

--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 tweend 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 CBAnimation(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:

x The x position to simulate the touch event at
y The y position to simulate the touch event at
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 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
```

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.

Returns:

Nothing

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 as a table of key value pairs with two current keys defined. The key 'process_used' indicates the memory used by the sbengine process. The key 'heap_used' indicates only the heap (malloc) memory used by the sbengine process. Not all rendering engine platforms support all metrics, in which case the value will be set to 0 indicating no information.

Parameters:

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:

```
-- Generate an error message
gre.log(1, "This is an error message")

-- Generate an diagnostic information message
gre.log(4, "This is an info  message")

-- Generate an unfiltered message
gre.log(-1, "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.torgb(
 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.torgb(grey)
```

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 rext you are targeting. This will target the first rich text rext f
table - a table containing optional parameters

Optional parameters:
(these will override any parameters found in the targeted rich text rext)

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)

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,y2` 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,y2` 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

    cavans: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:0x00000)

```
-- 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 as an external Lua module and is located in the `lib/gredom.so` file of the Storyboard Engine target distributions.

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.get_application

```
gre.get_application()
```

Get an object handle for the application root

Returns:
A DOMOBJECT object that represents the application.

gredom.get_object

```
gre.get_object(fqn)
```

Get an object handle for the

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.

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 grecom.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.

Lua DOM Samples

```
require("gredom")

-- 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

        -- 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
            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_XRDONLY:	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_XRDONLY`.

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_addr,
    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_addr 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:

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_addr,
    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_addr	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 gre_io_open()
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 Manager Options

Table C.1. Action Manager Options

OPTION	DESCRIPTION
None	None

Table C.2. Data Manager Options

OPTION	DESCRIPTION
None	None

Table C.3. IO Manager Options

OPTION	DESCRIPTION
-oio_mgr,queue_size=4096	This will limit the event queue size to a maximum of 4K. 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.

Table C.4. Model Manager Options

OPTION	DESCRIPTION
-omodel_mgr,fps=25	Limits the frame rate for all animations to a maximum fps specified (25 in this example).
-omodel_mgr,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 statistics, future values are reserved.
-omodel_mgr,plugin_path=/temp	Sets the plugin path to the specified directory (/temp in this example). This setting overrides the SB_PLUGINS environment variable setting.
-omodel_mgr,scroll_drag_px	The number of pixels that need to be moved before a scrollable container considers this a scroll operation (default 10).
-omodel_mgr,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).

OPTION	DESCRIPTION
-omodel_mgr,scroll_bounce_rate	The animation rate that should be used on a scrollable container for bounce back animations (default easein).

Table C.5. Render Manager Options: Windows, win32, swrender, OpenGL ES 2.0, x86

OPTION	DESCRIPTION
-orender_mgr,x=[xpos]	This will position the application at the defined x-position
-orender_mgr,y=[ypos]	This will position the application at the defined y-position
-orender_mgr,sysdpi	This will enable the window upscaling performed by Windows according to the system DPI setting

Table C.6. Render Manager Options: Linux, sdl, x86

OPTION	DESCRIPTION
-orender_mgr,quality=[0|1|2]	Quality of image rotation. The default, 0 is fastest but lowest quality rendering. A setting of 2 is highest quality but slowest performing.

Table C.7. Render Manager Options: Linux, fbdev, x86, armle

OPTION	DESCRIPTION
-orender_mgr,dblbuffer	Enable double buffering (fullscreen redraws)
-orender_mgr,fb=[device path]	The path to the framebuffer device to use (default=/dev/fb0)
-orender_mgr,fullscreen	Run in fullscreen mode
-orender_mgr,rotated=[90|180|270]	Rotate the application by the defined angle.
-orender_mgr,quality=[0|1|2]	Quality of image rotation. The default, 0 is fastest but lowest quality rendering. A setting of 2 is highest quality but slowest performing.

Table C.8. Render Manager Options: Linux, directfb, x86, armle

OPTION	DESCRIPTION
-orender_mgr,dumpconfig	Dump the layer and graphics configuration information at startup
-orender_mgr,double	Enable double buffering (fullscreen redraws)
-orender_mgr,layer=[index]	Set the directfb layer index that content will render to (default=0)
-orender_mgr,quality=[0|1|2]	Quality of image rotation. The default, 0 is fastest but lowest quality rendering. A setting of 2 is highest quality but slowest performing.

Table C.9. Render Manager Options: Linux, Windows CE, Windows Compact 7, Mac OSX, Neutrino 6.5, OpenGL ES 2.0, armle (Beagleboard)

OPTION	DESCRIPTION
-orender_mgr,quality=[0|1|2]	Set the rendering quality including shade model and texture filter, and image rotation quality. A value of 0=nearest/flat, 1=linear/flat, 2=nearest/smooth (default=linear/flat). For image rotation, lower numbers mean faster algorithms, but better quality
-orender_mgr,fullscreen	Run in fullscreen mode
-orender_mgr,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.
-orender_mgr,vbo	Enables the use of vertex buffer objects
-orender_mgr,scale=[aspect]	Scale the application to the physical display size. If aspect is passed the application will retain the proper aspect ratio when scaled.
-orender_mgr,backbuffer	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.
-orender_mgr,npot	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.
-orender_mgr,fontsize=[size]	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.
-orender_mgr,window_w=[w]	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'
-orender_mgr,window_h=[h]	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'

OPTION	DESCRIPTION
-orender_mgr,linejoin=[0|1]	Set line join style for path drawing, drawing joins can have a performance impact on frame rate. 0=none, 1=round (default=1)
-orender_mgr,clipmode=[stencil|scissor]	Set the clipping mode to use, may have performance impacts. Each implementation defaults to the best performance.
-orender_mgr,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.
-orender_mgr,display=[index]	Connect to the given display index, this option is only available for the QNX Screen OpenGL ES 2.0 render manager or the Linux i.MX6 OpenGL ES 2.0 render manager where the value is the selected framebuffer index.
-orender_mgr,fb=[x]	This option pertains specifically to iMX6 hardware platforms. Starting at 0, x defines the framebuffer number to render to.
-orender_mgr,x=[xpos]	When using the QNX Screen engine this will position the application at the defined x-position
-orender_mgr,y=[ypos]	When using the QNX Screen engine this will position the application at the defined y-position.
-orender_mgr,rotated=[90|180|270]	Rotate the application by the defined angle.
-orender_mgr,zorder=[z]	When using the QNX Screen engine this will position the application window at the defined Z index.
-orender_mgr,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.
-orender_mgr,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.
-orender_mgr,font_block_size=[number]	This specifies the memory size of the shared blocks that are used within the font glpyh 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.
-orender_mgr,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)

OPTION	DESCRIPTION
-orender_mgr,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.

Table C.10. Render Manager Options: QNX Neutrino 6.5, Linux, Fujitsu Jade, armle

OPTION	DESCRIPTION
-orender_mgr,mainlayer=[number]	The main layer to use for rendering, defaults to layer 0
-orender_mgr,display=[number]	The display to connect to, defaults to display 0
-orender_mgr,conf_file=[path]	A path to a LCD configuration file which includes the display settings

Table C.11. Render Manager Options: WinCE 6.0, Windows Compact 7, win32, armle

OPTION	DESCRIPTION
-orender_mgr,quality=[0|1|2]	Quality of image rotation. 0 is fastest but lowest quality. 2 highest and slowest
-orender_mgr,fullscreen	Removes the window border and fills the display outside of the application area
-orender_mgr,dumpcaps	Print the device capabilities and acceleration flags
-orender_mgr,nohwcursor	Disables the HW cursor

Table C.12. Resource Manager Options

OPTION	DESCRIPTION
-oresource_mgr,image=[number]	This will set the maximium number of bytes in the image cache. After the cache is full, the behaviour for allocations is controlled via the `image_oom` 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.
-oresource_mgr,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 `fail` then when an allocation can't be fulfilled the request will fail. If this is set to the value `lru` then when an allocation can't be fulfilled the request will attempt to remove other entries from the cache until

OPTION	DESCRIPTION
	it can either fit the request or it fails. The default behaviour is `lru`.
-oresource_mgr,font=[number]	Similar to the `image` 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 `font_oom` 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 glpyh cache of 4096 bytes or `1M` to indicate 1048576 bytes of memory.
-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.

Table C.13. Screen Manager Options

OPTION	DESCRIPTION
-oscreen_mgr,swcursor	Enables the rendering of a software cursor
-oscreen_mgr,redraw_complete	Generate a completed event for every screen update
-oscreen_mgr,dl=1	Disable the use of a particular hardware layer
-oscreen_mgr,fps	Display the frames per second of the display updates in the Storyboard Engine console output and also set the `grd_fps` variable with the calculated value.
-oscreen_mgr,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.

Plugins and 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 on option to a plugin use the -o option to sbengine in this format -o[PLUGIN_NAME],[PLUGIN OPTION]

Table C.14. 3D model rendering: libgre-plugin-model3d.so

OPTION	DESCRIPTION
-omodel3d,novbo	Disable the use of vertex buffer objects, by default Vertex buffer objects are used for rendering.

Table C.15. Capture/Playback: libgre-plugin-capture-playback.so

OPTION	DESCRIPTION
-ocapture_playback,mode=[capture|playback]	Specifies the behavior of the plugin for either play-back or capture. If capture is specified then input events (gre.press, gre.release, gre.motion, gre.key-press etc) will be logged and stored in an ouptut file. If playback is specified then the contents of an input file are read and the input events are injected back into the application.
-ocapture_playback,file=filename	If "capture" is specified as the mode it indicates the contents of the file specified will be overwritten with the new event stream. If "playback" is specified then the contents of the file specified will be used as an event stream source.
-ocapture_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.
-ocapture_playback,no_defaults	Capture option to specify that there are no default events set to be captured. By default we capture the following events: 1 gre.press 2 gre.release 3 gre.keyup 4 gre.keydown 5 gre.keyup 6 gre.keyrepeat 7 gre.motion 8 gre.mtevent
-ocapture_playback,verbosity	Indicates that the plugin should log events that it is either capturing or playing back to the standard out-put
-ocapture_playback,loop=count	Playback option that indicates how many times the playback should iterate through its content. The de-fault is to playback the content once, otherwise if a `count` is specified the plugin will replay the full content count times.
-ocapture_playback,quit_playback	Playback option that indicates if a `gre.quit` event should be automatically generated after playback is complete. The default is to not generate a quit event.

Table C.16. Gesture: libgre-plugin-gesture.so

OPTION	DESCRIPTION
-ogesture,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.
-ogesture,mode=[disabled|single|multi]	By default, sbengine will search your application for use of any gesture events and gestures will be enbled as required. This behaviour can be overridden with any of the following mode options: disabled will not generate any gesture events. single allows for generation of single touch gesture events. multi allows for generation of both single and multi-touch gesture events.
-ogesture,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.

Table C.17. Linux Input Support: libgre-plugin-dev-input.so

OPTION	DESCRIPTION
-odev-input,mouse=/dev/input/event0	The name of the mouse device, for example `/dev/input/event0`
-odev-input,kbd=/dev/input/event1	The name of the keyboard device, for example `/dev/input/event1`
	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.

Table C.18. Lua Scripting: libgre-plugin-lua.so

OPTION	DESCRIPTION
-olua,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 `gre.set_event` call. In this case the data manager will be released to process the notifications, the send_event 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 `gre.set_data` calls. If 2 is specified then the data manager is held from emitting change notifications until the end of script execution even if a `gre.send_event` call is made.

OPTION	DESCRIPTION
-olua,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 `collectgarbage` 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.

Table C.19. Linux Multi-Touch Protocol: libgre-plugin-mtdev.so

OPTION	DESCRIPTION
-omtdev,device=[path to touch device]	Plugin for Linux Multi-Touch Protocol to be used with kernels supporting multi-touch events. This plugin is only available for the imx6 runtime currently.
-omtdev,rotation=[0|90|180|270]	Clockwise rotation of the touch input coordinates. This option is not often required. By default, mtdev 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.
-omtdev,calibrate	Puts mtdev into raw mode to be used in conjunction with the mtcalib application (See Crank Public SVN) to acquire the bounds parameters for screen calibration. Use the -omtdev,bounds option to pass the parameters to sbengine.
-omtdev,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.
-omtdev,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
-omtdev,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

Table C.20. Linux Touchscreen Support: libgre-plugin-tslib.so

OPTION	DESCRIPTION
-otslib,pressure=1	Set the pressure value with corresponds to a press, the default is any value greater than 0 is a press.
-otslib,motion=5	The number of consecutive motion events to compress, can be useful on a device which delivers a

OPTION	DESCRIPTION
	high rate of motion events, default is to not compress.
-otslib,calibrate	Put tslib into raw mode which is used for calibration.
	If you do not have the following `tslib` variables setup the plugin will not load or function properly. TSLIB_CONSOLEDEVICE TSLIB_TSDEVICE TSLIB_CALIBFILE TSLIB_CONFFILE

Table C.21. Logger: libgre-plugin-logger.so

OPTION	DESCRIPTION
-ologger,output=[filename]	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. `+` At the beginning of the file name will append to the log file, otherwise the file will be overridden on each invocation of sbengine. `%D` in the filename will be replaced by a date stamp with YYYY-MM-DD format. `%T` in the filename will be replaced by a 24h time stamp with HHMMSS format. Both `%D` and `%T` may be used on the same filename. A valid command would be: `-ologger,output=/logs/log-%D-%T.txt`, provided the /logs/ directory existed prior to runtime.
-ologger,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.
-ologger,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.
-ologger,slogger=[opcode] (QNX ONLY)	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 _SLOG_SETCODE(major, minor) macro that's defined in sys/slog.h.

OPTION	DESCRIPTION
	This option is only available for systems running QNX.
-ologger,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.
-ologger,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.
-ologger,perf	This option enables the logging of performance data to the standard output (or file if `perf_file` is used). If a value of 0 is specified to the perf option then performance logging is enabled, but the capture of data is not immediately started and can be toggled using the `gra.perf_state` action. If the value is set to 1 or is not specified, then performance data will be immediately captured. For example `sbengine -ologger,perf=1` will enable performance logging with the immediate capture of performance metrics.
-ologger,perf_file=[filename]	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 sbengine.
-ologger,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.
-ologger,perf_event_prefix=[prefix]	This option specifies the prefix to use in the perf event name. This defaults to perf.
-ologger,perf_filter=[keyword]	This option enables filtering of perf types based on the keyword provided. The sb_Event 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 delimited by a ':'. `sbengine -ologger,perf,perf_filter=sb_RenderExtension` Each keyword can also be negated by the '^' symbol. Therefor we could ignore all render extension types by passing in the following command: `sbengine -ologger,perf,per-`

OPTION	**DESCRIPTION**
	`f_filter=^sb_RenderExtension` Filtering applies to the perf option.
-ologger,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.
-ologger,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 defualt, 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.
-ologger,metrics_event_prefix=[prefix]	This option specifies the prefix to use in the metrics event names. This defaults to metrics.
-ologger,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: `sbengine -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.
-ologger,retailmsg	This option is only available on Windows Compact Embedded platforms. The logger will output using RETAILMSG instead of sending output to stdio.
	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

Table C.22. QNX input support: libgre-plugin-gfi-input.so

OPTION	DESCRIPTION
-ogfi-input,mouse=/dev/devi/mouse0	The name of the mouse device, for example `/dev/devi/mouse0`
-ogfi-input,rotated=[90|270]	If specified, this indicates that the input co-ordinates should be rotated by 90 or 270 degrees
	By default the input system used the gfi interface based on the devi drivers. The devi driver must be run with the `-P` option. If you pass the `mouse` option then the mouse/touchscreen is used in raw mode.

Table C.23. Storyboard IO: libgre-plugin-greio.so

OPTION	DESCRIPTION
-ogreio,channel=name	The value specifies the name that the applications Storyboard IO channel will use. This name can then be used by `gre_io_open` or `iogen` clients to send events to the application.
-ogreio,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.

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/touchscreen has entered a control (if dragging a pointer or finger). This event is generated once the coordinates enter a control boundary. If mouse motion events are disabled in the render manager 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/touchscreen has left a control (if dragging a pointer or finger). This event is generated once the coordinates leave a control boundary. If mouse motion events are disabled in the render manager 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:

No data payload

`gre.table.drag_stop`

This event is generated when a user stops dragging a scrolling table.

Data:

No data payload

`gre.table.scroll_start`

This event is generated when the scroll animation begins.

Data:

No data payload

`gre.table.scroll_stop`

This event is generated when the scroll animation completes.

Data:

No data payload

`gre.table.scroll_cancel`

This event is generated when the scroll animation is interrupted.

Data:

No data payload

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.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

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 MW_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 finishes 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 recognize
```

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	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 *down* a negative value moves the table *up*
delta_col	The number of columns to move. A positive value moves the table *right* a negative value moves the table *left*
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 `direction` is assigned `set`
col	The column to navigate to. This is only used if `direction` is assigned `set`

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.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 gra.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.

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 target of the event to generate (optional)
	format	The format of the event data (optional)
	data	The data payload for the event

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 fading the old screen into the new one.

Plugin:	`libgre-plugin-screen-path.so`	
Options:	screen	The screen to transition to
	rate	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 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	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	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.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	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 delta

gra.screen.glflip

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	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.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	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 delta

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	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 delta

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	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 delta

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.timer

Start, stop and control a timer.

Plugin: `libgre-plugin-timer.so`

Options:	name	The name to use to identify this timer (required)
	rtime	The time delay in milliseconds *relative* 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,re-
peat=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,re-
peat=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 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.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:

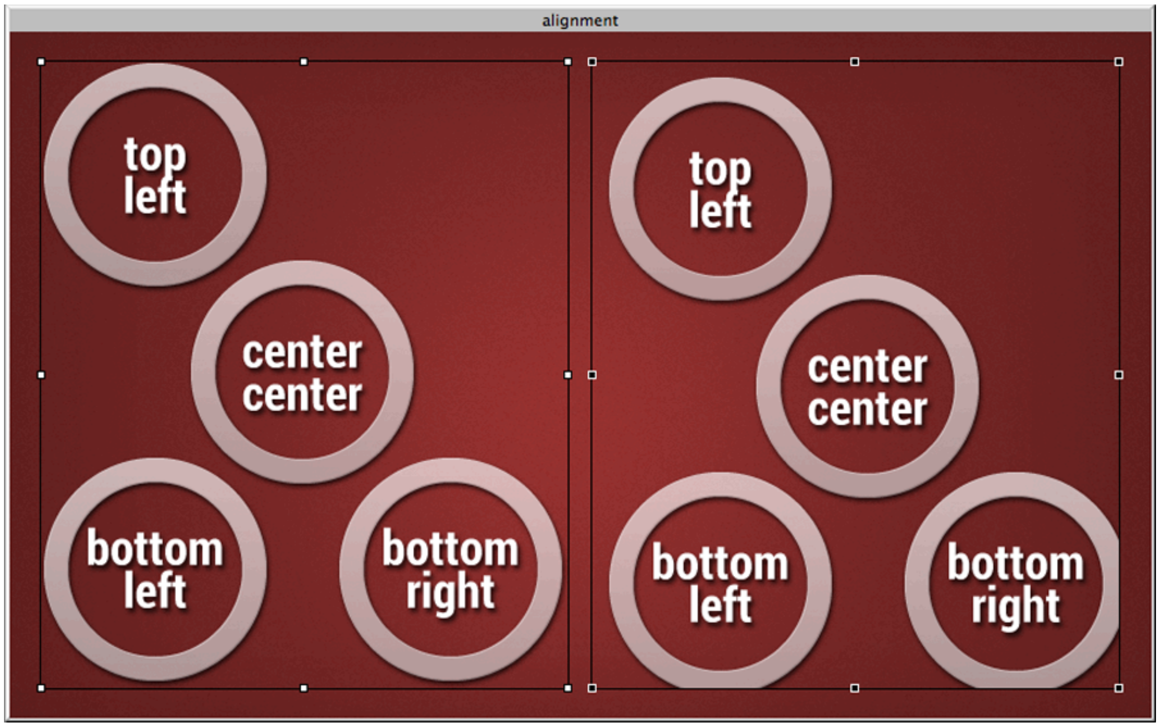
0	top (default)
1	top
2	center
3	bottom

Horizontal Alignment	The horizontal alignment within the control (number, optional, default: 0). This alignment value is a numeric value corresponding to the following:

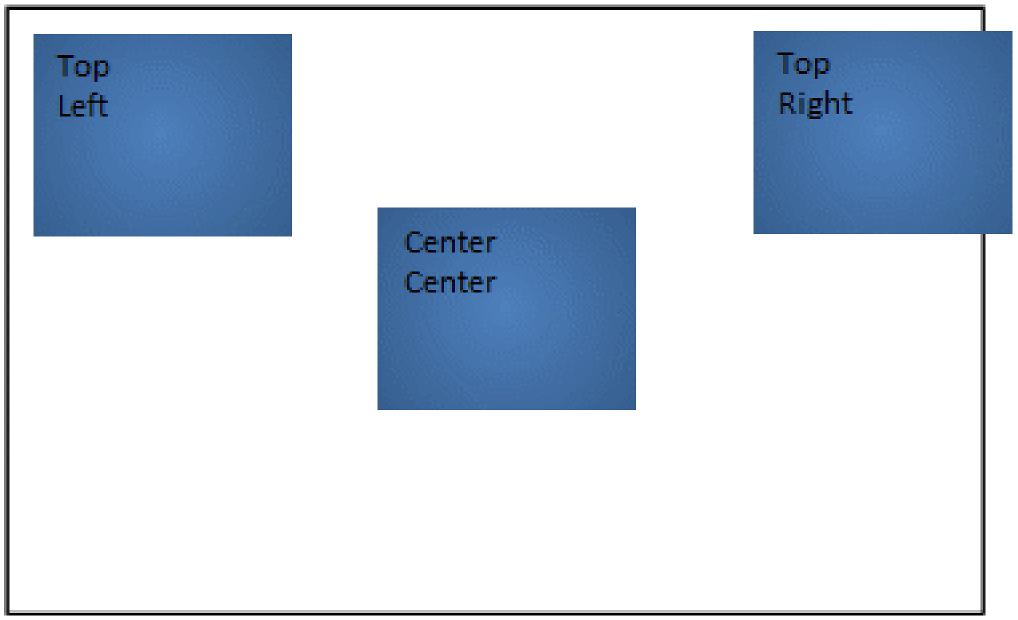
0	left (default)
1	left
2	center
3	right

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

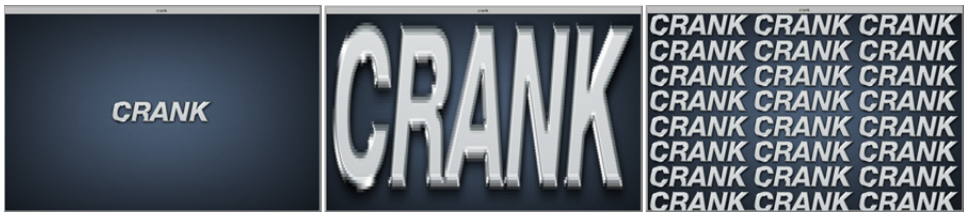
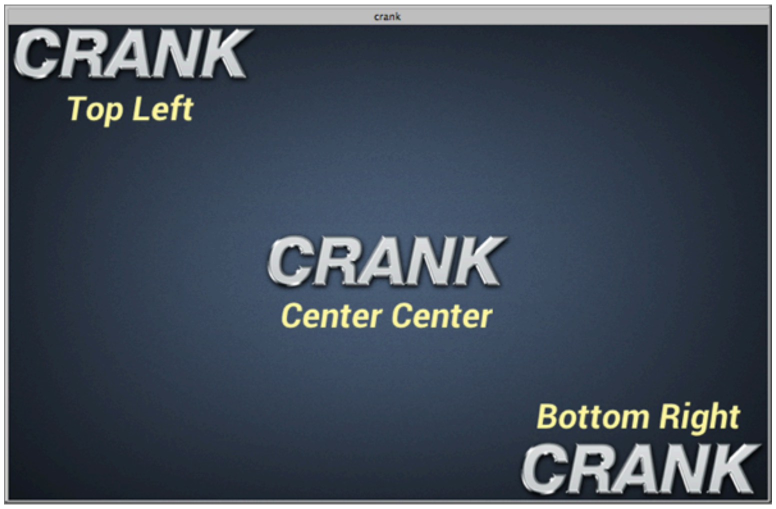


Image Alignment Sample



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 ("wrap"), 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.

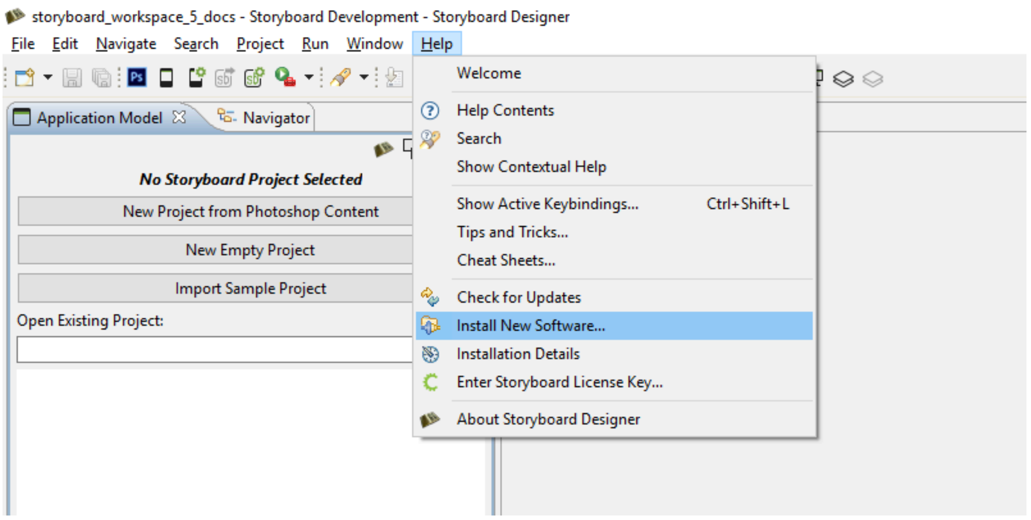
Part I. Storyboard Tutorials

Table of Contents

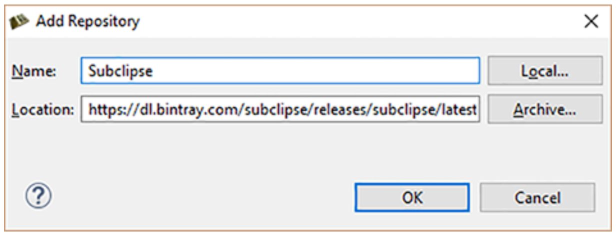
Chapter 31. Importing Sample Projects from Crank's Public SVN

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.

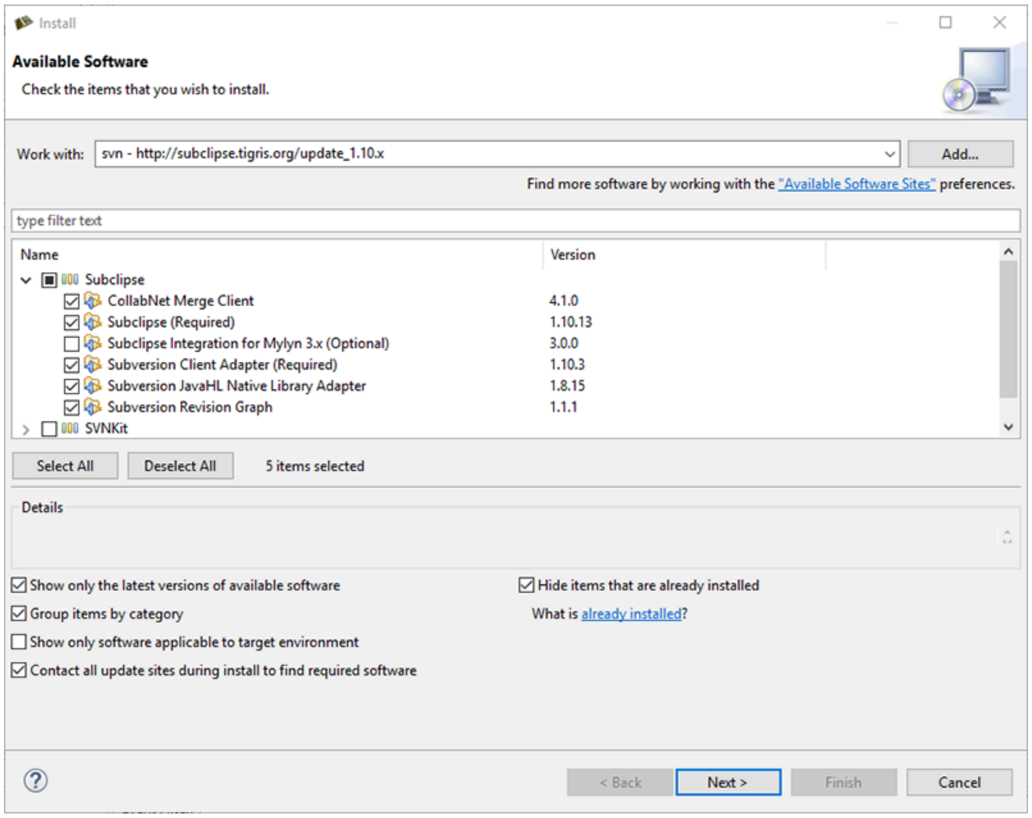

storyboard_workspace_5_docs - Storyboard Development - Storyboard Designer
File Edit Navigate Search Project Run Window Help
Application Model
Navigator
No Storyboard Project Selected
New Project from Photoshop Content
New Empty Project
Import Sample Project
Open Existing Project:
Welcome
Help Contents
Search
Show Contextual Help
Show Active Keybindings... Ctrl+Shift+L
Tips and Tricks...
Cheat Sheets...
Check for Updates
Install New Software...
Installation Details
Enter Storyboard License Key...
About Storyboard Designer

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.



Name: Subclipse
Location: https://dl.bintray.com/subclipse/releases/subclipse/latest

After you click OK you will be presented with a list of software to install. There is no need for Mylyn so you can uncheck it. Click Next.


Install

Available Software

Check the items that you wish to install.

Work with: svn - http://subclipse.tigris.org/update_1.10.x

Find more software by working with the "Available Software Sites" preferences.

type filter text

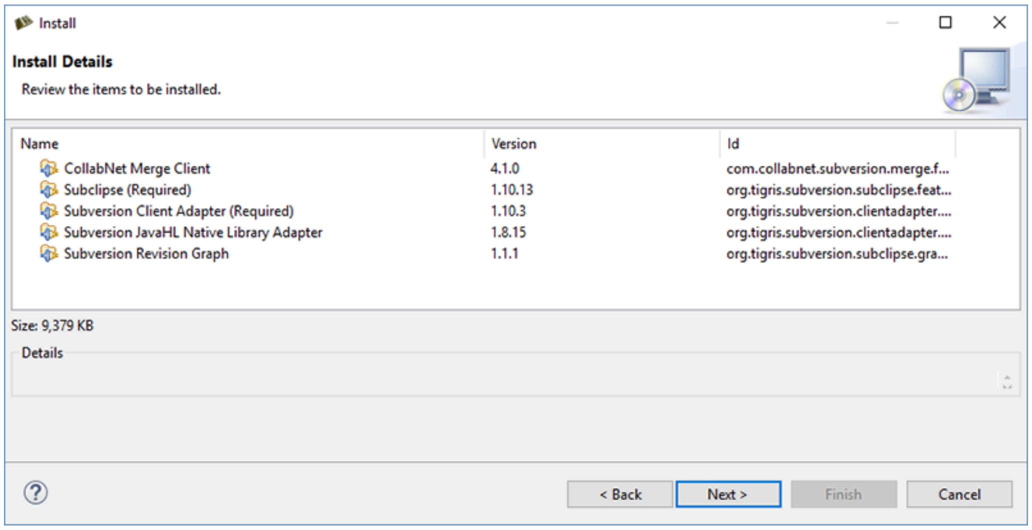
Name	Version
Subclipse	
☑ CollabNet Merge Client	4.1.0
☑ Subclipse (Required)	1.10.13
☐ Subclipse Integration for Mylyn 3.x (Optional)	3.0.0
☑ Subversion Client Adapter (Required)	1.10.3
☑ Subversion JavaHL Native Library Adapter	1.8.15
☑ Subversion Revision Graph	1.1.1
☐ SVNKit	

5 items selected

Details

☑ Show only the latest versions of available software
☑ Group items by category
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software
☑ Hide items that are already installed
What is already installed?

Details of the software to be installed. Click Next.


Install

Install Details

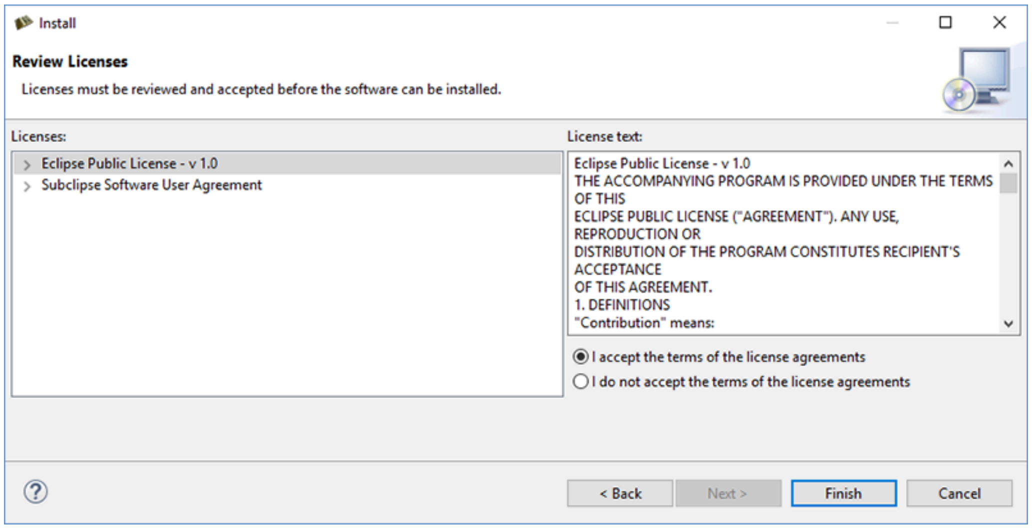
Review the items to be installed.

Name	Version	Id
CollabNet Merge Client	4.1.0	com.collabnet.subversion.merge.f...
Subclipse (Required)	1.10.13	org.tigris.subversion.subclipse.feat...
Subversion Client Adapter (Required)	1.10.3	org.tigris.subversion.clientadapter....
Subversion JavaHL Native Library Adapter	1.8.15	org.tigris.subversion.clientadapter....
Subversion Revision Graph	1.1.1	org.tigris.subversion.subclipse.gra...

Size: 9,379 KB

Details

Accept license agreements. Click Finish.



Install

Review Licenses

Licenses must be reviewed and accepted before the software can be installed.

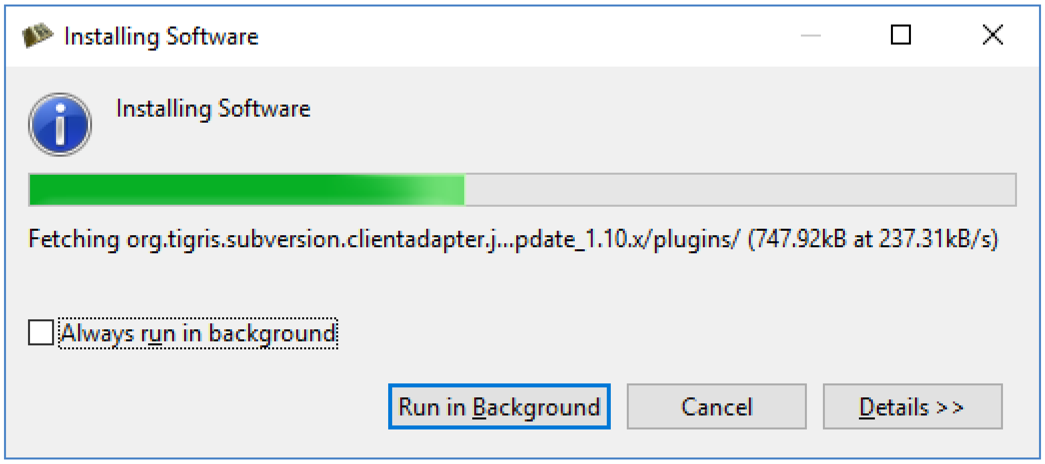
Licenses:
- Eclipse Public License - v 1.0
- Subclipse Software User Agreement

License text:

Eclipse Public License - v 1.0
THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS
OF THIS
ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE,
REPRODUCTION OR
DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S
ACCEPTANCE
OF THIS AGREEMENT.
1. DEFINITIONS
"Contribution" means:

[✓ I accept the terms of the license agreements]
I do not accept the terms of the license agreements

Software installation progress.



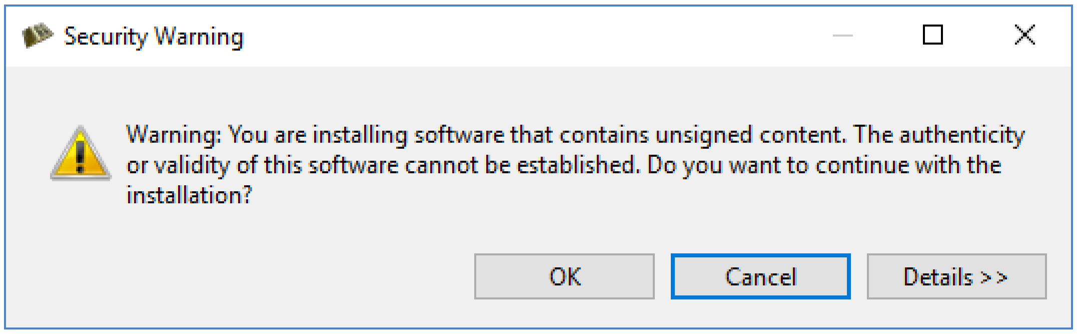
Installing Software

Installing Software

Fetching org.tigris.subversion.clientadapter.j...pdate_1.10.x/plugins/ (747.92kB at 237.31kB/s)

Always run in background

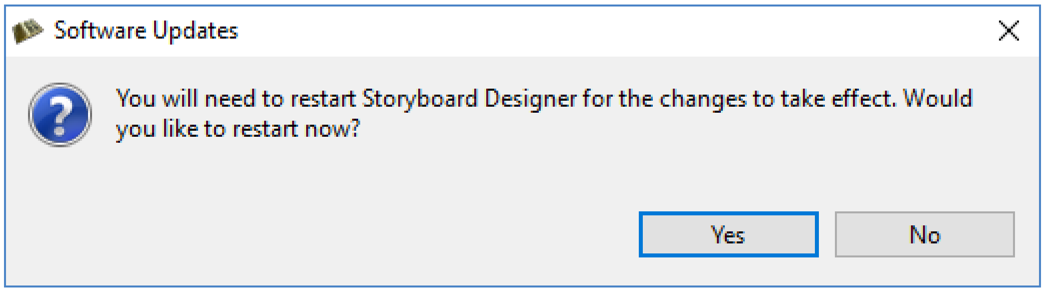
Generic warning that the software being installed is not signed. Click OK.



Security Warning

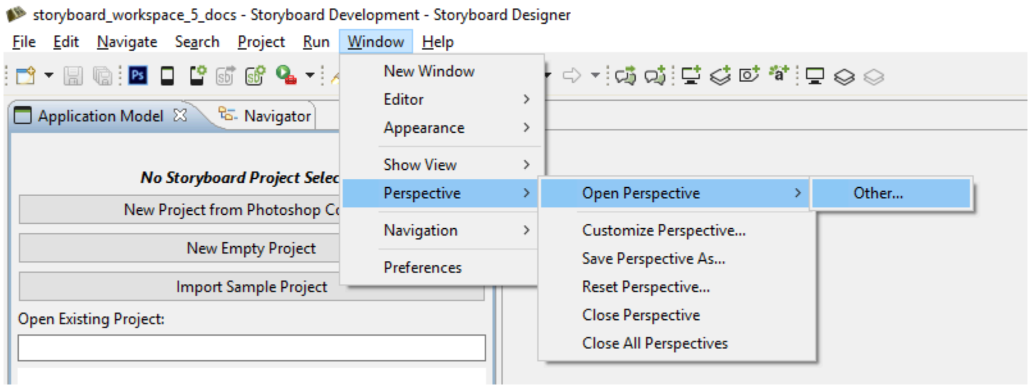
Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

After the Subclipse SVN client is installed you will need to restart Storyboard Designer. Click Restart Now.

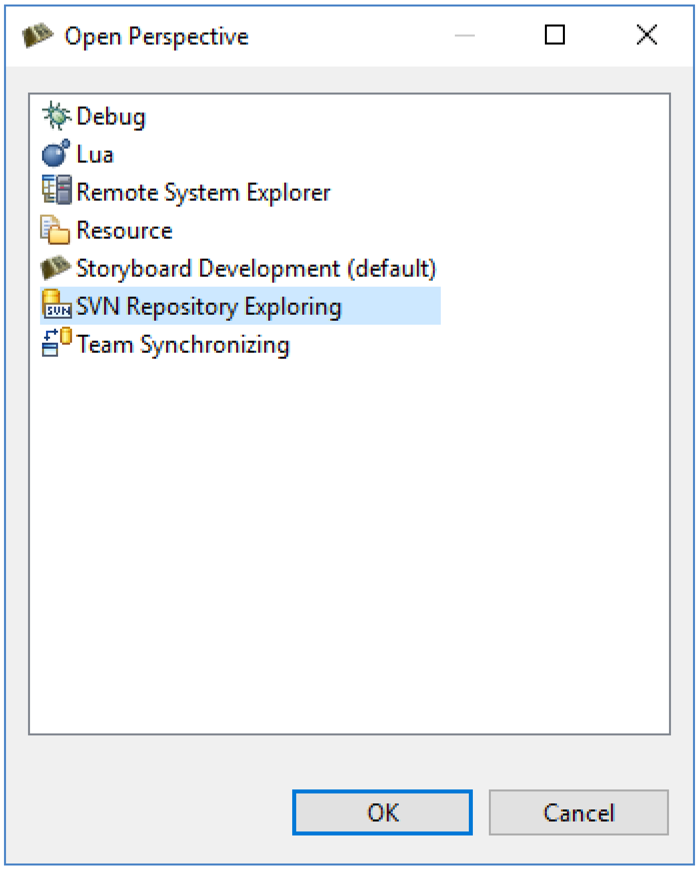

Software Updates

You will need to restart Storyboard Designer for the changes to take effect. Would you like to restart now?

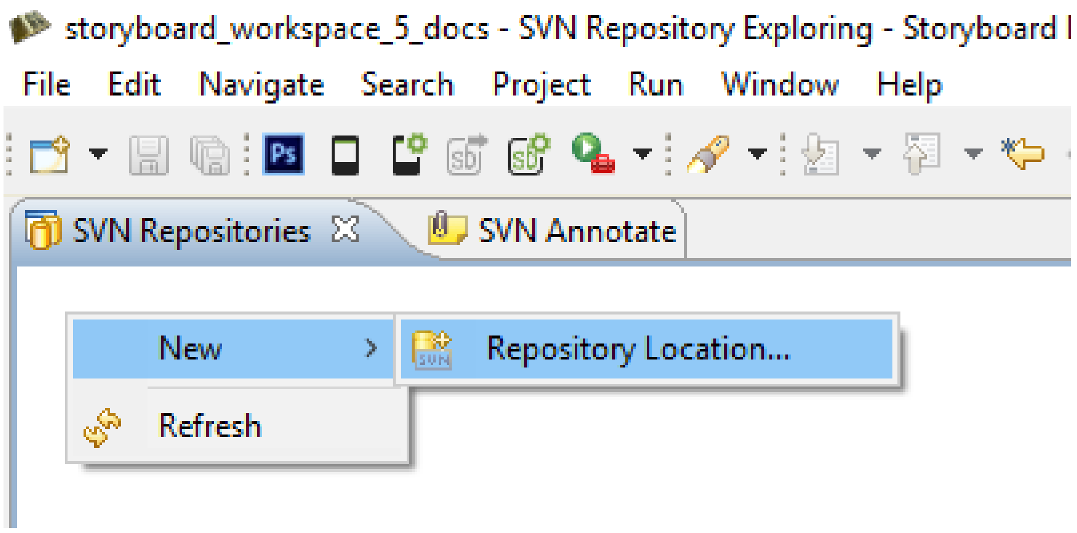
Now that Subclipse is installed we need to go to that perspective to add a repository.



Select SVN Repository Exploring. Click OK.

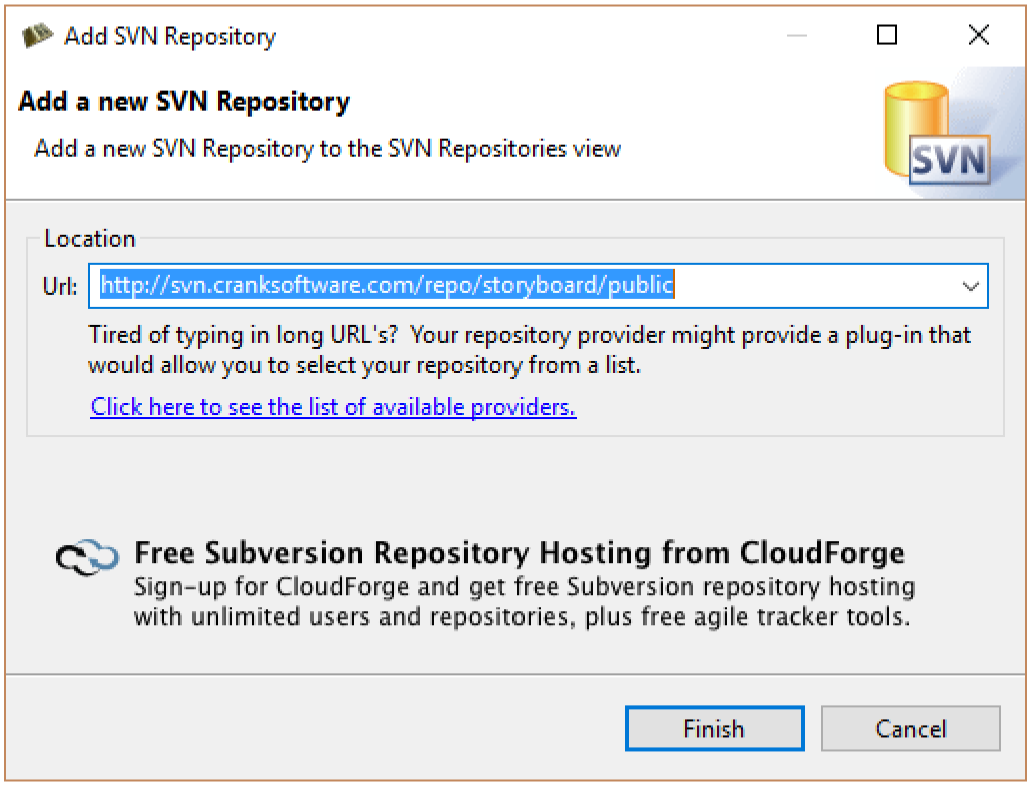

Open Perspective

To add a repository we simply right click in the SVN perspective window and select New > Repository Location.


storyboard_workspace_5_docs - SVN Repository Exploring - Storyboard I
File Edit Navigate Search Project Run Window Help
SVN Repositories
SVN Annotate
New
Repository Location...
Refresh

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**



Add a new SVN Repository

Add a new SVN Repository to the SVN Repositories view

Location

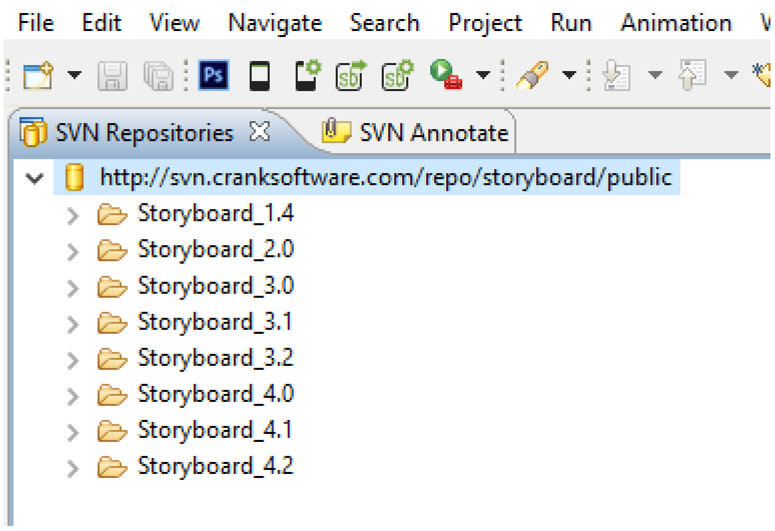
Url: http://svn.cranksoftware.com/repo/storyboard/public

Tired of typing in long URL's? Your repository provider might provide a plug-in that would allow you to select your repository from a list.

Click here to see the list of available providers.

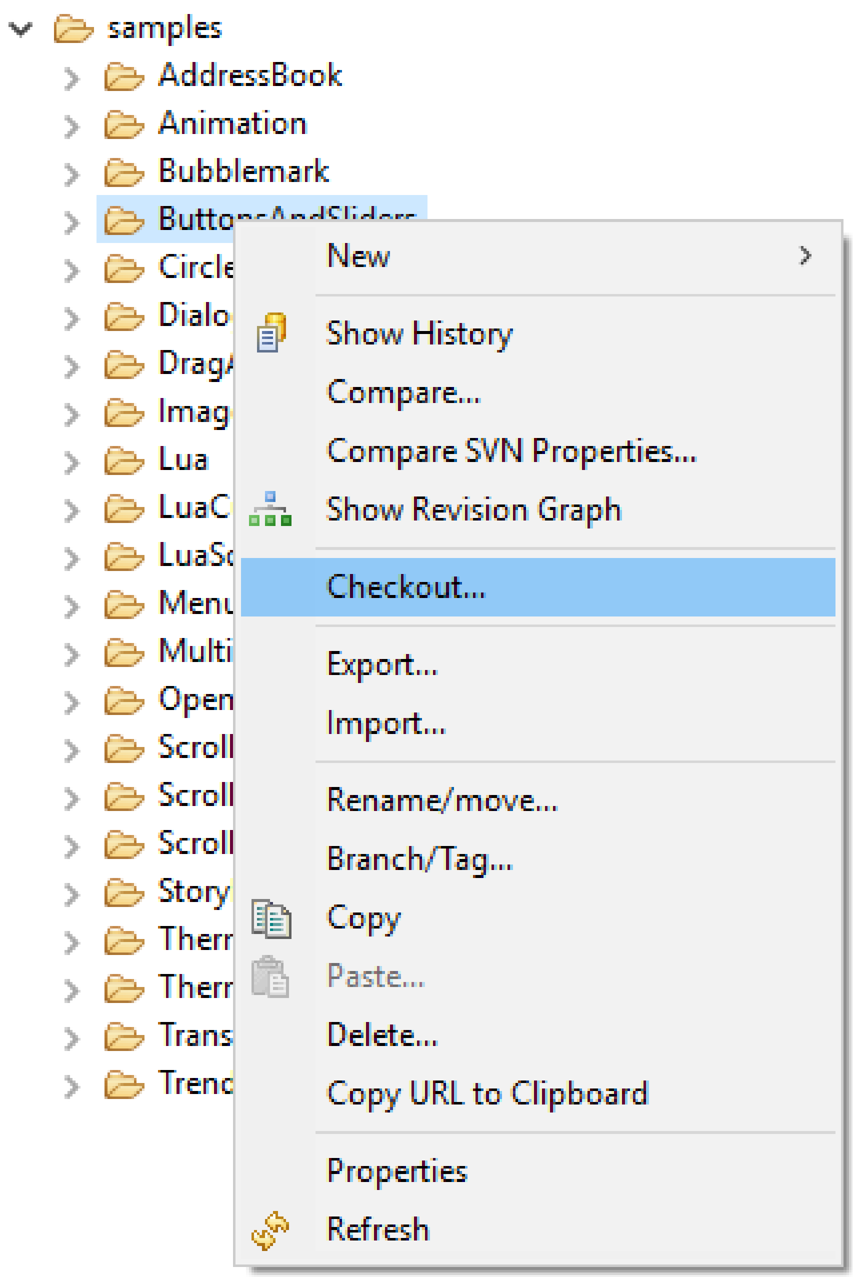
Free Subversion Repository Hosting from CloudForge
Sign-up for CloudForge and get free Subversion repository hosting with unlimited users and repositories, plus free agile tracker tools.

You are now connected to the Crank Public Repository. By expanding the directories you can see the different demos available for checkout.



http://svn.cranksoftware.com/repo/storyboard/public
Storyboard_1.4
Storyboard_2.0
Storyboard_3.0
Storyboard_3.1
Storyboard_3.2
Storyboard_4.0
Storyboard_4.1
Storyboard_4.2

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.


samples
AddressBook
Animation
Bubblemark
ButtonsAndSliders
Circle
Dialo
Drag
Imag
Lua
LuaC
LuaSc
Menu
Multi
Open
Scroll
Scroll
Scroll
Story
Therr
Therr
Trans
Trenc
New
Show History
Compare...
Compare SVN Properties...
Show Revision Graph
Checkout...
Export...
Import...
Rename/move...
Branch/Tag...
Copy
Paste...
Delete...
Copy URL to Clipboard
Properties
Refresh

Chapter 32. Working with Multiple Application Design Files

Storyboard Suite'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.

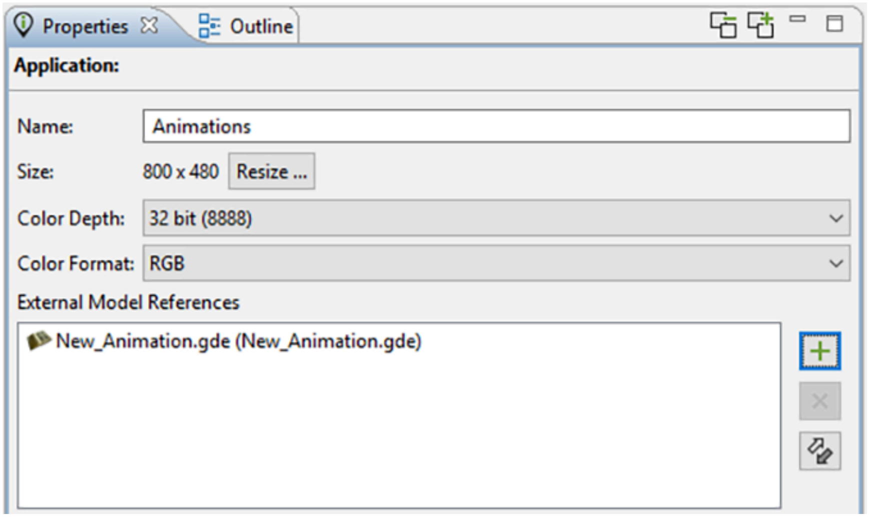
Creating a Project

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.



Application:

Name: Animations

Size: 800 x 480 Resize ...

Color Depth: 32 bit (8888)

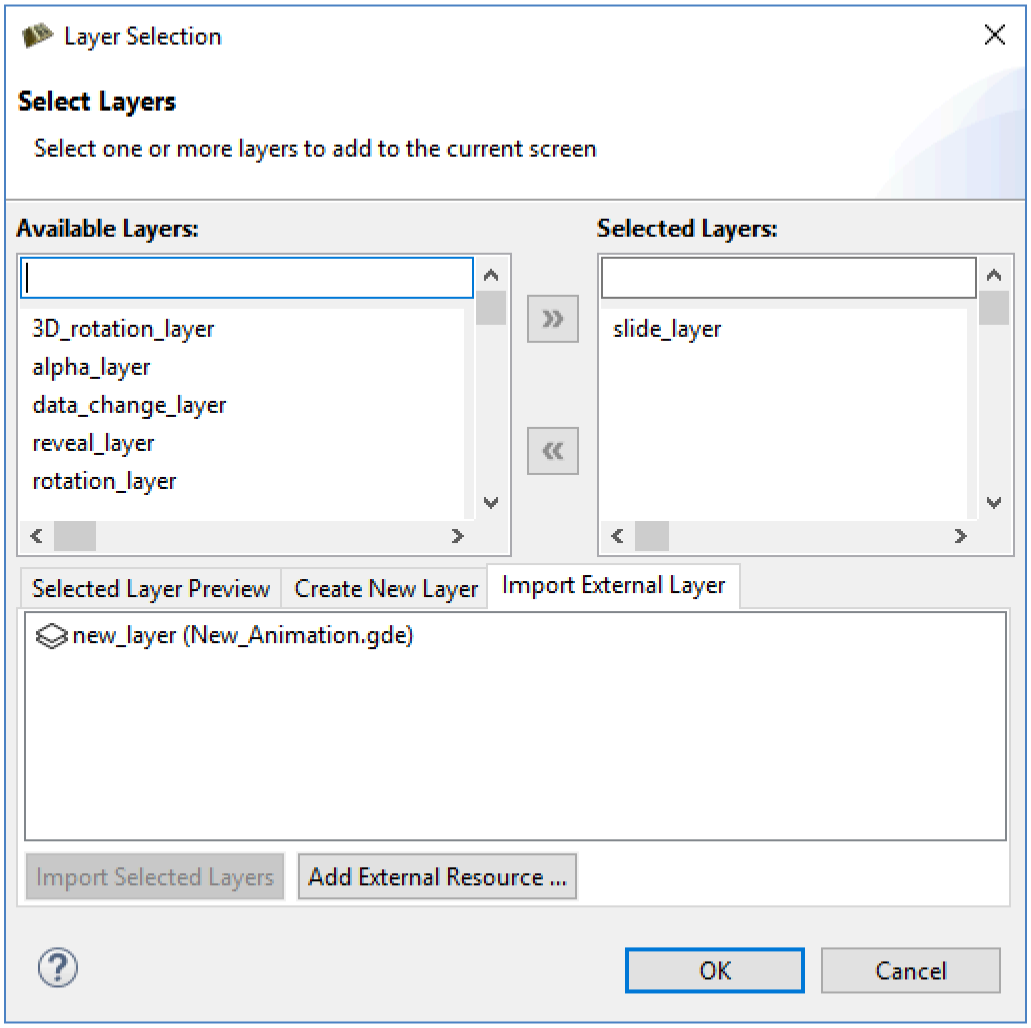
Color Format: RGB

External Model References

New_Animation.gde (New_Animation.gde)

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 Suite will recognize the external content and incorporate it to function like any other layer.


Layer Selection

Select Layers

Select one or more layers to add to the current screen

Available Layers:

- 3D_rotation_layer
- alpha_layer
- data_change_layer
- reveal_layer
- rotation_layer

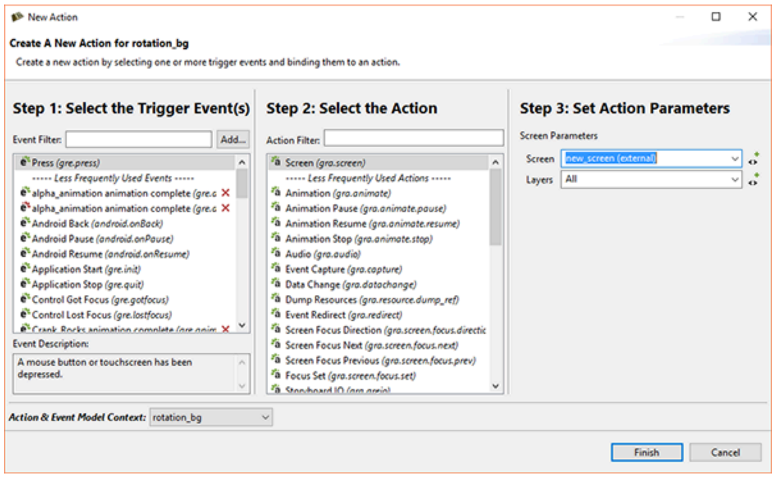
Selected Layers:

- slide_layer

Selected Layer Preview | Create New Layer | Import External Layer

new_layer (New_Animation.gde)

To create a transition from a source application screen to an external application screen, add actions to an application that perform a screen change.


New Action

Create A New Action for rotation_bg

Create a new action by selecting one or more trigger events and binding them to an action.

Step 1: Select the Trigger Event(s)

Event Filter:

- Press (gre.press)
- ----- Less Frequently Used Events -----
- alpha_animation animation complete (gre.a
- alpha_animation animation complete (gre.a
- Android Back (android.onBack)
- Android Pause (android.onPause)
- Android Resume (android.onResume)
- Application Start (gre.init)
- Application Stop (gre.quit)
- Control Got Focus (gre.gotfocus)
- Control Lost Focus (gre.lostfocus)
- Crank Rocks animation complete (gre.anim

Event Description:

A mouse button or touchscreen has been depressed.

Step 2: Select the Action

Action Filter:

- Screen (gra.screen)
- ----- Less Frequently Used Actions -----
- Animation (gra.animate)
- Animation Pause (gra.animate.pause)
- Animation Resume (gra.animate.resume)
- Animation Stop (gra.animate.stop)
- Audio (gra.audio)
- Event Capture (gra.capture)
- Data Change (gra.datachange)
- Dump Resources (gra.resource.dump_ref)
- Event Redirect (gra.redirect)
- Screen Focus Direction (gra.screen.focus.directic
- Screen Focus Next (gra.screen.focus.next)
- Screen Focus Previous (gra.screen.focus.prev)
- Focus Set (gra.screen.focus.set)
- Storyboard IO (gra.greio)

Step 3: Set Action Parameters

Screen Parameters

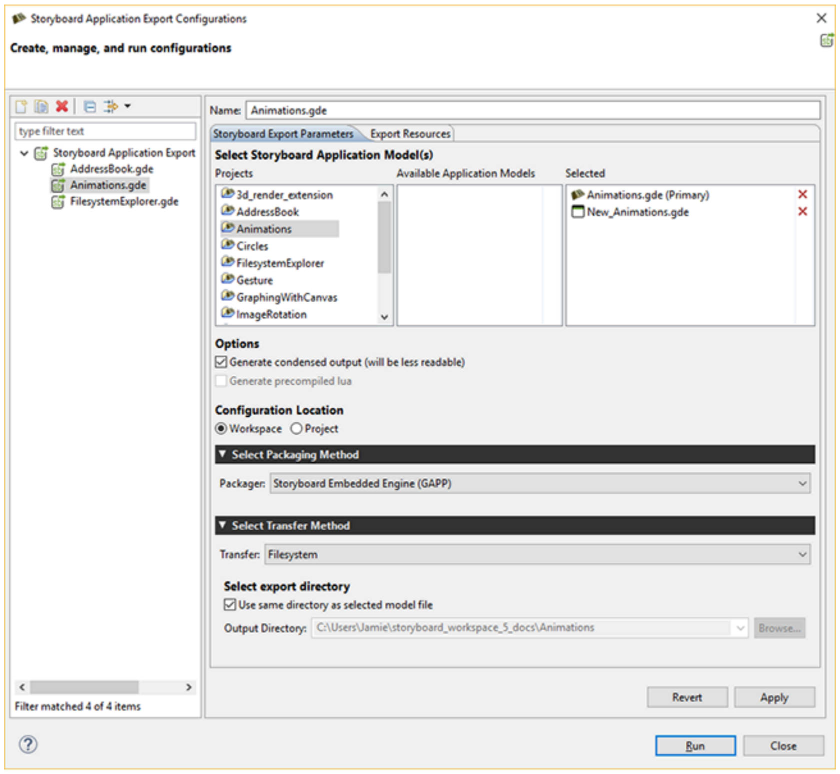
Screen: new_screen (external)

Layers: All

Action & Event Model Context: rotation_bg

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.



Storyboard Application Export Configurations

Create, manage, and run configurations

type filter text

Storyboard Application Export
- AddressBook.gde
- Animations.gde
- FilesystemExplorer.gde

Filter matched 4 of 4 items

Name: Animations.gde

Storyboard Export Parameters | Export Resources

Select Storyboard Application Model(s)

Projects	Available Application Models	Selected
3d_render_extension		Animations.gde (Primary)
AddressBook		New_Animations.gde
Animations		
Circles		
FilesystemExplorer		
Gesture		
GraphingWithCanvas		
ImageRotation		

Options
- [✓] Generate condensed output (will be less readable)
- [ ] Generate precompiled lua

Configuration Location
[● Workspace] ○ Project

Select Packaging Method

Packager: Storyboard Embedded Engine (GAPP)

Select Transfer Method

Transfer: Filesystem

Select export directory
- [✓] Use same directory as selected model file

Output Directory: C:\Users\Jamie\storyboard_workspace_5_docs\Animations

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.

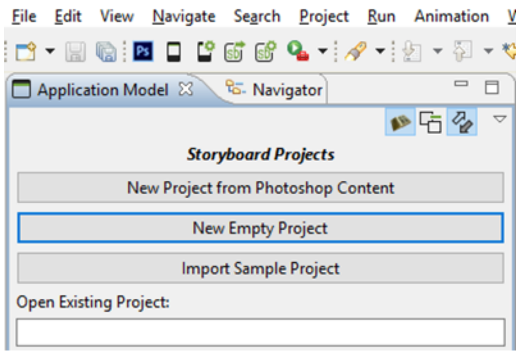
Screens: Screens from all applications are compared. Two or more screens with the same name prompt the user to resolve differences between the two.

Chapter 33. Creating a 3D Model Application

This quick tutorial will show how to use the 3D Model Control with a new project

New Project

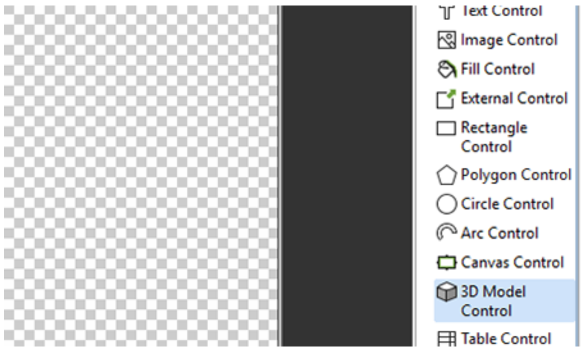
Create a new project by either File > New > Storyboard Application or selecting New Empty Project from the Application View.



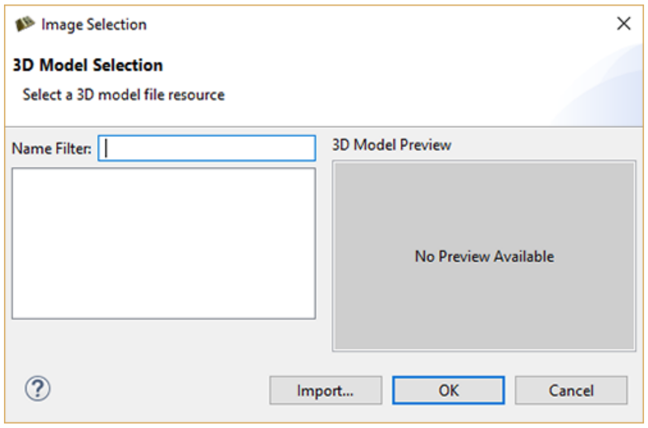
Give your Project a name and click Finish.

3D Model Control

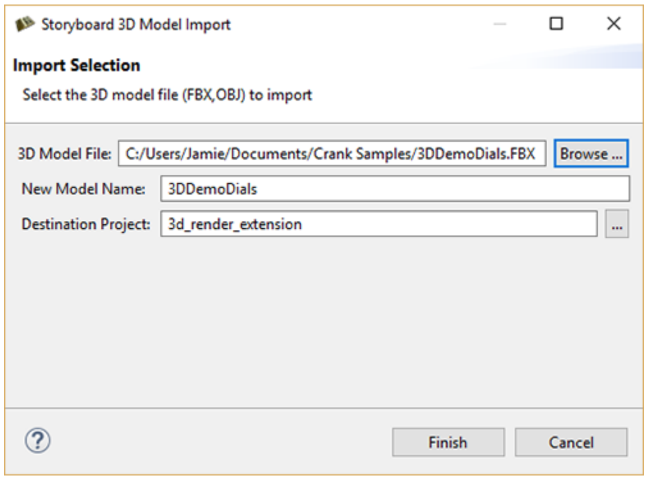
Once the empty project loads add a 3D Model Control to the screen.



The 3D Model Selection dialog will pop up enabling you can navigate to the 3D model you want to load.

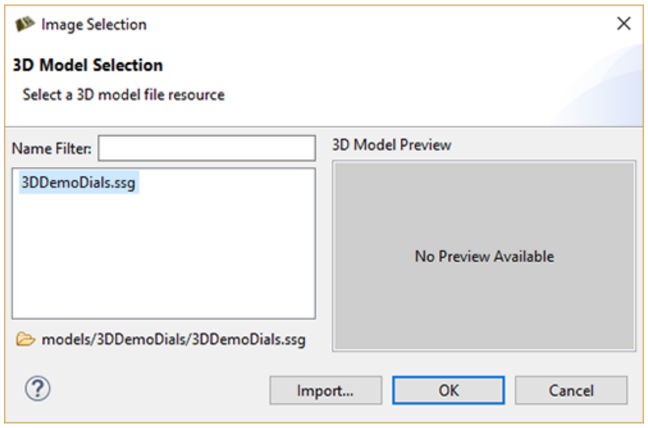


Click on the Import button to select your FBX or OBJ model.


Storyboard 3D Model Import
Import Selection
Select the 3D model file (FBX,OBJ) to import
Browse ...
Finish
Cancel

3D Model File: C:/Users/Jamie/Documents/Crank Samples/3DDemoDials.FBX
New Model Name: 3DDemoDials
Destination Project: 3d_render_extension

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.

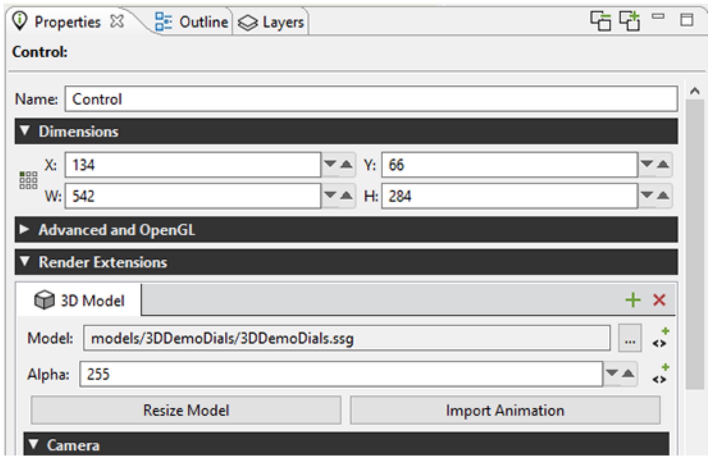


3DDemoDials.ssg

models/3DDemoDials/3DDemoDials.ssg

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.



Name: Control

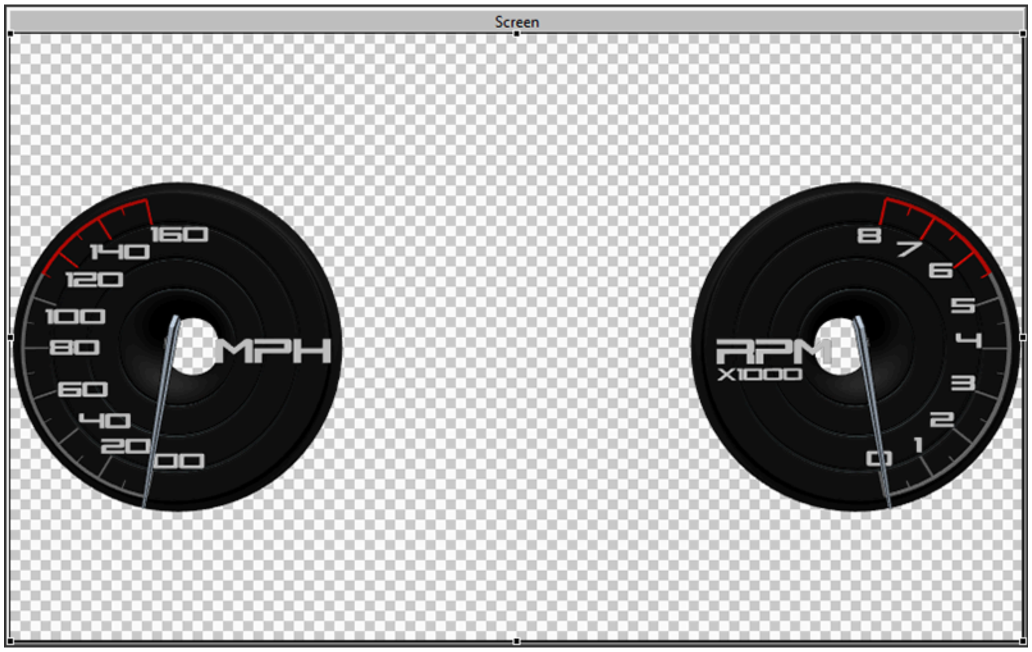
X: 134 Y: 66

W: 542 H: 284

Model: models/3DDemoDials/3DDemoDials.ssg

Alpha: 255

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 Suite Demo Images

Table of Contents

Chapter 34. NXP

Experience multi-market demo applications built with Storyboard Suite 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_imx6sxsabresd.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

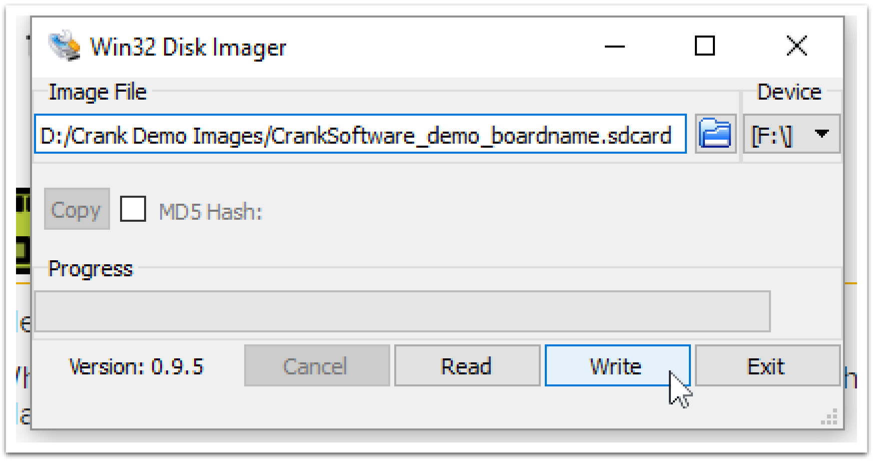


Image File: D:/Crank Demo Images/CrankSoftware_demo_boardname.sdcard

Device: [F:\]

Version: 0.9.5

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 35. STMicroelectronics

Experience multi-market demo applications built with Storyboard Suite 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 Suite Evaluation [http://www.cranksoftware.com/storyboard_suite_eval]
- STM32F746G-Discovery (“target platform”) or
- STM32F439l (“target platform”) or
- STM32F429l (“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 ST-M32F429
- demos/fbake
  - Sample white goods Storyboard application for the STM32F746G-Discovery or STM32F429

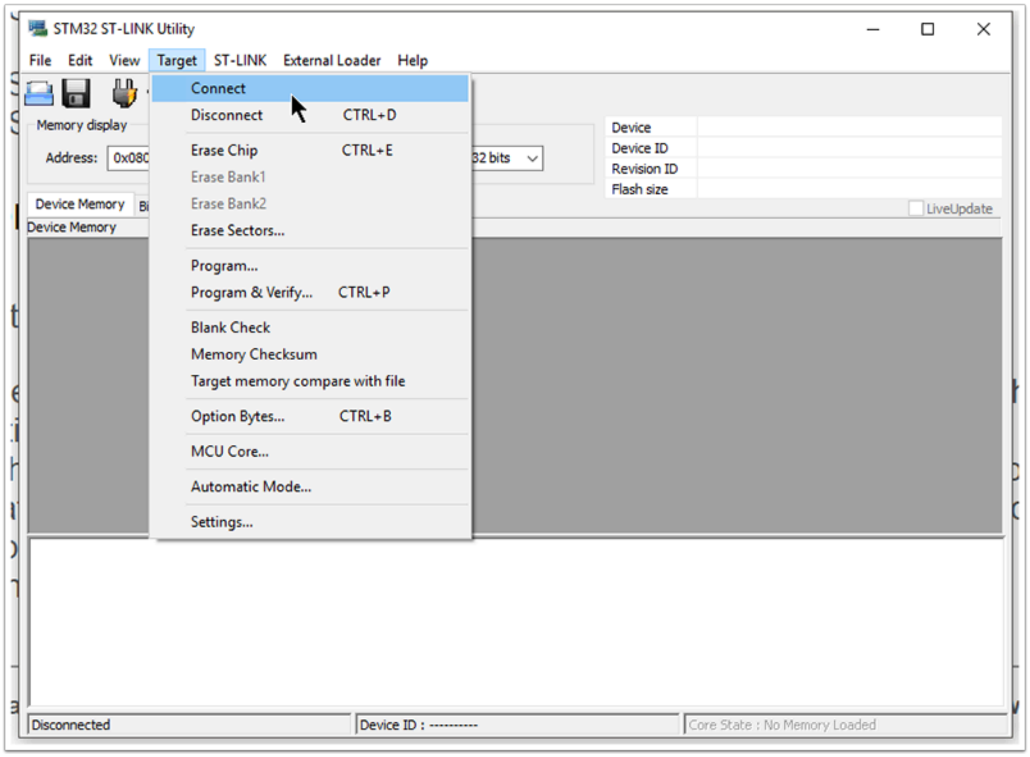
- demos/HomeAutomation-640
  - Sample home automation 640 x 480 Storyboard application for the STM32F439
- demos/HomeAutomation-480
  - Sample home automation 480 x 272 Storyboard application for the STM32F746GDiscovery or STM32F429
- demos/medical
  - Sample medical Storyboard application for the STM32F746G-Discovery or STM32F429
- demos/SB_App.txt
  - Storyboard application definition file

STM32 ST-LINK Utility

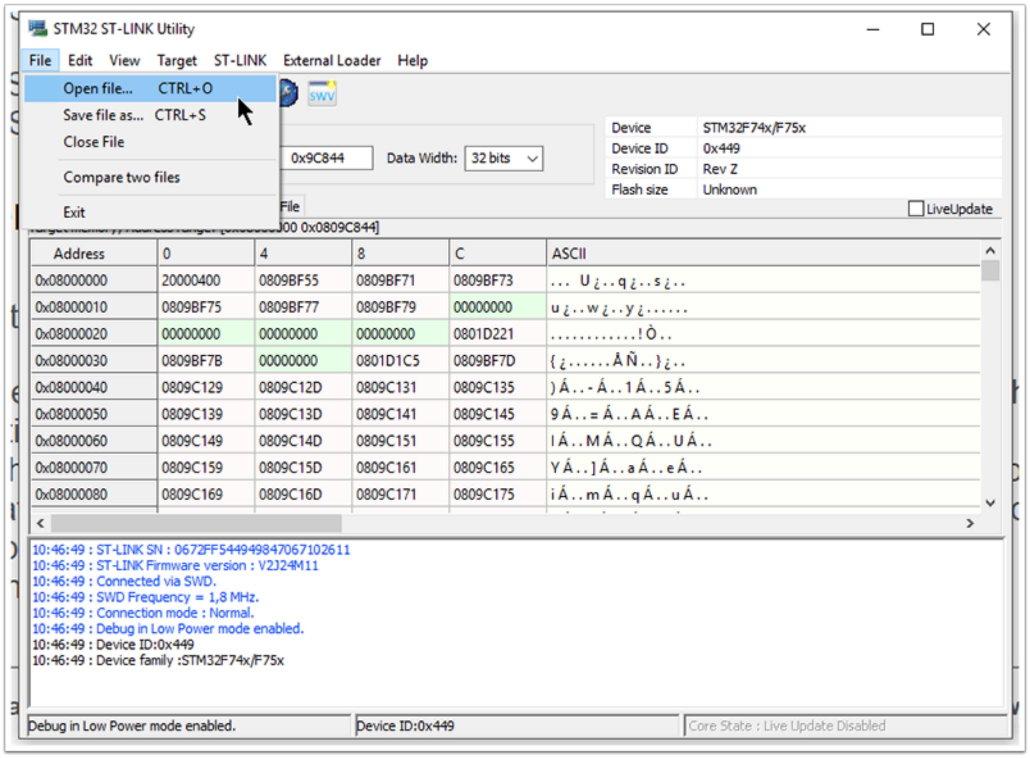
In order to use the STM32 demo images the following steps must be performed. These steps are specific to the STM32F746G_Discovery but use the correct .bin file for the STM32 target platform being used.

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 Suite Evaluation: STM32 Storyboard



2. Browse to the Storyboard_STM32F746G_Discovery.bin file



Data Width: 32 bits

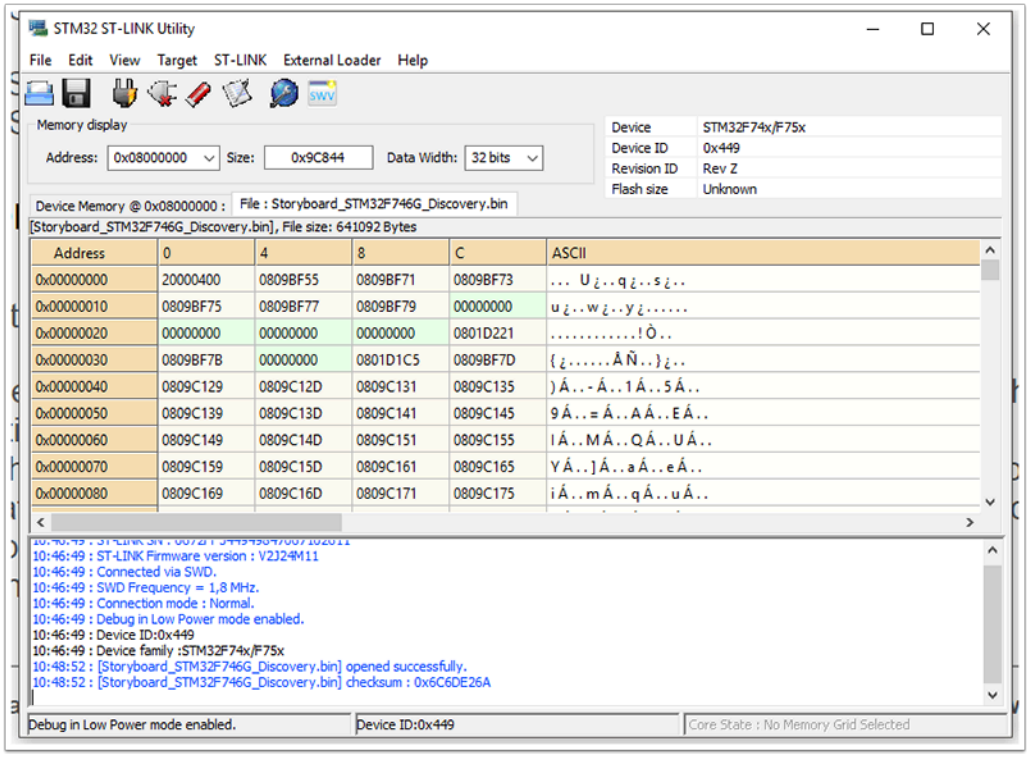
Device	STM32F74x/F75x
Device ID	0x449
Revision ID	Rev Z
Flash size	Unknown

Address	0	4	8	C	ASCII
0x08000000	20000400	0809BF55	0809BF71	0809BF73	... U¿..q¿..s¿..
0x08000010	0809BF75	0809BF77	0809BF79	00000000	u¿..w¿..y¿......
0x08000020	00000000	00000000	00000000	0801D221	............!Ò..
0x08000030	0809BF7B	00000000	0801D1C5	0809BF7D	{¿......ÅÑ..}¿..
0x08000040	0809C129	0809C12D	0809C131	0809C135	)Á..-Á..1Á..5Á..
0x08000050	0809C139	0809C13D	0809C141	0809C145	9Á..=Á..AÁ..EÁ..
0x08000060	0809C149	0809C14D	0809C151	0809C155	IÁ..MÁ..QÁ..UÁ..
0x08000070	0809C159	0809C15D	0809C161	0809C165	YÁ..]Á..aÁ..eÁ..
0x08000080	0809C169	0809C16D	0809C171	0809C175	iÁ..mÁ..qÁ..uÁ..

```
10:46:49 : ST-LINK SN : 0672FF544949847067102611
10:46:49 : ST-LINK Firmware version : V2J24M11
10:46:49 : Connected via SWD.
10:46:49 : SWD Frequency = 1,8 MHz.
10:46:49 : Connection mode : Normal.
10:46:49 : Debug in Low Power mode enabled.
10:46:49 : Device ID:0x449
10:46:49 : Device family :STM32F74x/F75x
```

Debug in Low Power mode enabled. | Device ID:0x449 | Core State : Live Update Disabled

3. The Storyboard_STM32F746G_Discovery.bin file is now loaded in the utility



Memory display

Address: 0x08000000 Size: 0x9C844 Data Width: 32 bits

Device	STM32F74x/F75x
Device ID	0x449
Revision ID	Rev Z
Flash size	Unknown

Device Memory @ 0x08000000 : | File : Storyboard_STM32F746G_Discovery.bin

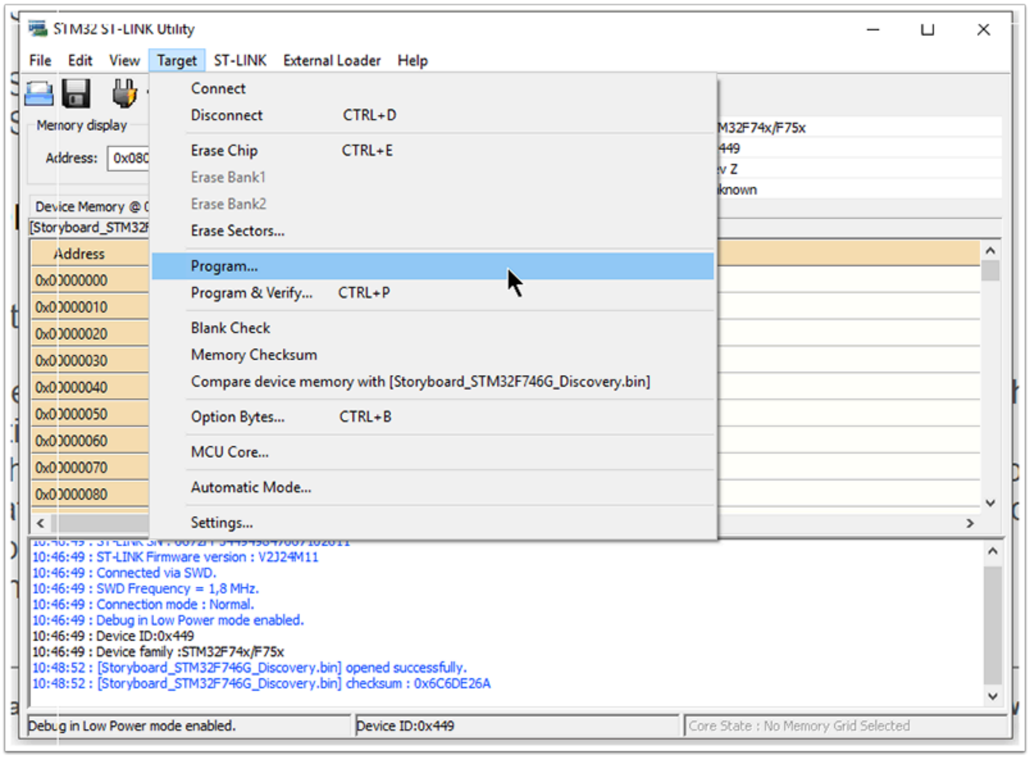
[Storyboard_STM32F746G_Discovery.bin], File size: 641092 Bytes

Address	0	4	8	C	ASCII
0x00000000	20000400	0809BF55	0809BF71	0809BF73	... U¿..q¿..s¿..
0x00000010	0809BF75	0809BF77	0809BF79	00000000	u¿..w¿..y¿......
0x00000020	00000000	00000000	00000000	0801D221	............!Ò..
0x00000030	0809BF7B	00000000	0801D1C5	0809BF7D	{¿......ÅÑ..}¿..
0x00000040	0809C129	0809C12D	0809C131	0809C135	)Á..-Á..1Á..5Á..
0x00000050	0809C139	0809C13D	0809C141	0809C145	9Á..=Á..AÁ..EÁ..
0x00000060	0809C149	0809C14D	0809C151	0809C155	IÁ..MÁ..QÁ..UÁ..
0x00000070	0809C159	0809C15D	0809C161	0809C165	YÁ..]Á..aÁ..eÁ..
0x00000080	0809C169	0809C16D	0809C171	0809C175	iÁ..mÁ..qÁ..uÁ..

```
10:46:49 : ST-LINK Firmware version : V2J24M11
10:46:49 : Connected via SWD.
10:46:49 : SWD Frequency = 1,8 MHz.
10:46:49 : Connection mode : Normal.
10:46:49 : Debug in Low Power mode enabled.
10:46:49 : Device ID:0x449
10:46:49 : Device family :STM32F74x/F75x
10:48:52 : [Storyboard_STM32F746G_Discovery.bin] opened successfully.
10:48:52 : [Storyboard_STM32F746G_Discovery.bin] checksum : 0x6C6DE26A
```

Debug in Low Power mode enabled. | Device ID:0x449 | Core State : No Memory Grid Selected

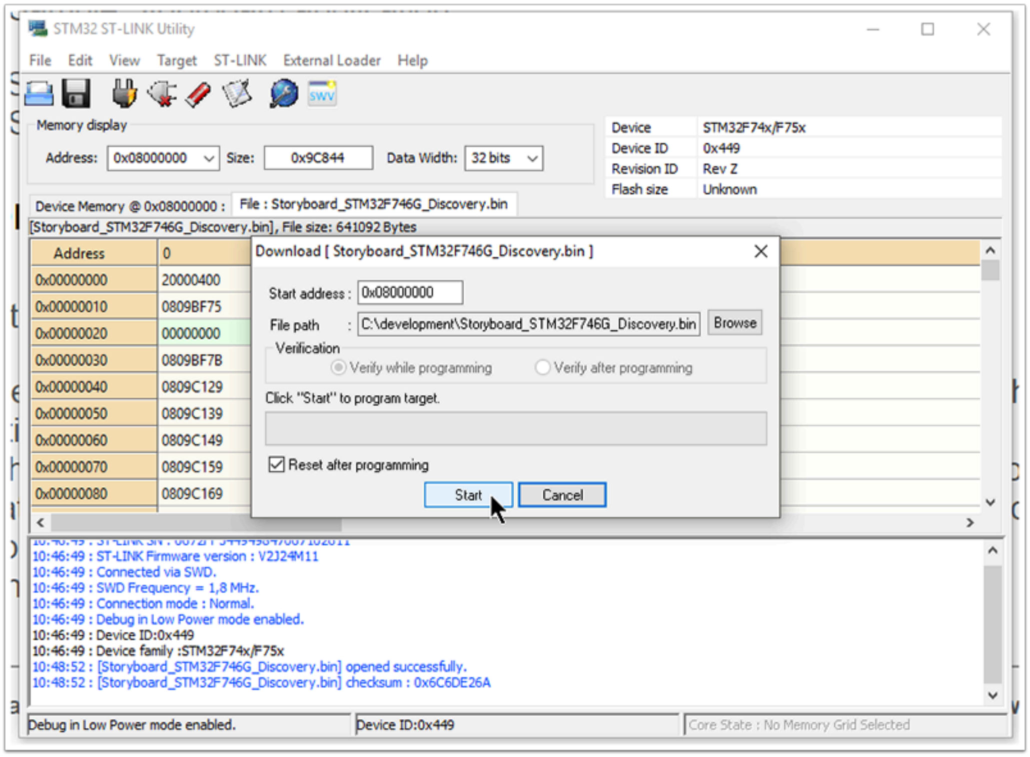
4. Program the Storyboard_STM32F746G_Discovery.bin file to the target platform



```
10:46:49 : ST-LINK Firmware version : V2J24M11
10:46:49 : Connected via SWD.
10:46:49 : SWD Frequency = 1,8 MHz.
10:46:49 : Connection mode : Normal.
10:46:49 : Debug in Low Power mode enabled.
10:46:49 : Device ID:0x449
10:46:49 : Device family :STM32F74x/F75x
10:48:52 : [Storyboard_STM32F746G_Discovery.bin] opened successfully.
10:48:52 : [Storyboard_STM32F746G_Discovery.bin] checksum : 0x6C6DE26A
```

Debug in Low Power mode enabled. | Device ID:0x449 | Core State : No Memory Grid Selected

5. Start the programming process


STM32 ST-LINK Utility
File Edit View Target ST-LINK External Loader Help
Memory display
Download [Storyboard_STM32F746G_Discovery.bin]
Browse
Verification
Start
Cancel

Address: 0x08000000 Size: 0x9C844 Data Width: 32 bits

Device	STM32F74x/F75x
Device ID	0x449
Revision ID	Rev Z
Flash size	Unknown

Device Memory @ 0x08000000 : File : Storyboard_STM32F746G_Discovery.bin

[Storyboard_STM32F746G_Discovery.bin], File size: 641092 Bytes

Address	0
0x00000000	20000400
0x00000010	0809BF75
0x00000020	00000000
0x00000030	0809BF7B
0x00000040	0809C129
0x00000050	0809C139
0x00000060	0809C149
0x00000070	0809C159
0x00000080	0809C169

Start address : 0x08000000
File path : C:\development\Storyboard_STM32F746G_Discovery.bin
( ) Verify while programming ( ) Verify after programming
Click "Start" to program target.
[✓] Reset after programming

```
10:46:49 : ST-LINK Firmware version : V2J24M11
10:46:49 : Connected via SWD.
10:46:49 : SWD Frequency = 1,8 MHz.
10:46:49 : Connection mode : Normal.
10:46:49 : Debug in Low Power mode enabled.
10:46:49 : Device ID:0x449
10:46:49 : Device family :STM32F74x/F75x
10:48:52 : [Storyboard_STM32F746G_Discovery.bin] opened successfully.
10:48:52 : [Storyboard_STM32F746G_Discovery.bin] checksum : 0x6C6DE26A
```

Debug in Low Power mode enabled. | Device ID:0x449 | Core State : No Memory Grid Selected

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 36. Atmel

Experience multi-market demo applications built with Storyboard Suite on your favorite Atmel 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/atmel] the demo image that corresponds with your hardware, which includes everything you need to get and 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

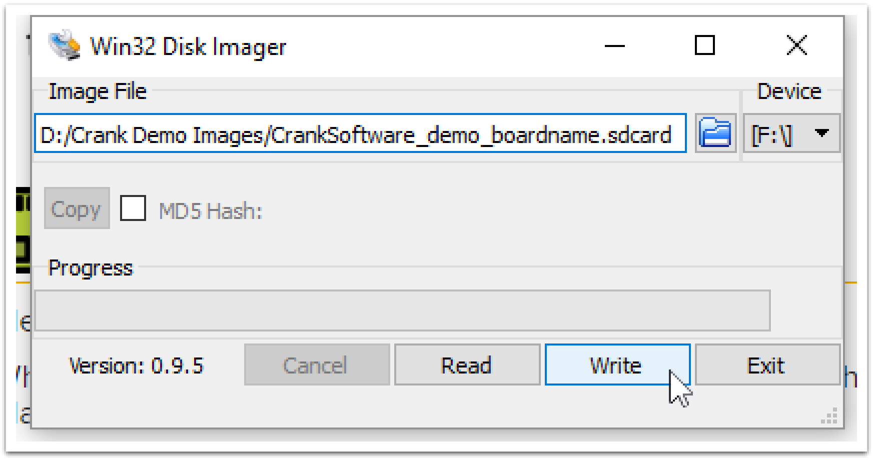


Image File: D:/Crank Demo Images/CrankSoftware_demo_boardname.sdcard

Device: [F:\]

Version: 0.9.5

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 37. Renesas

Experience multi-market demo applications built with Storyboard Suite 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 38. Linux

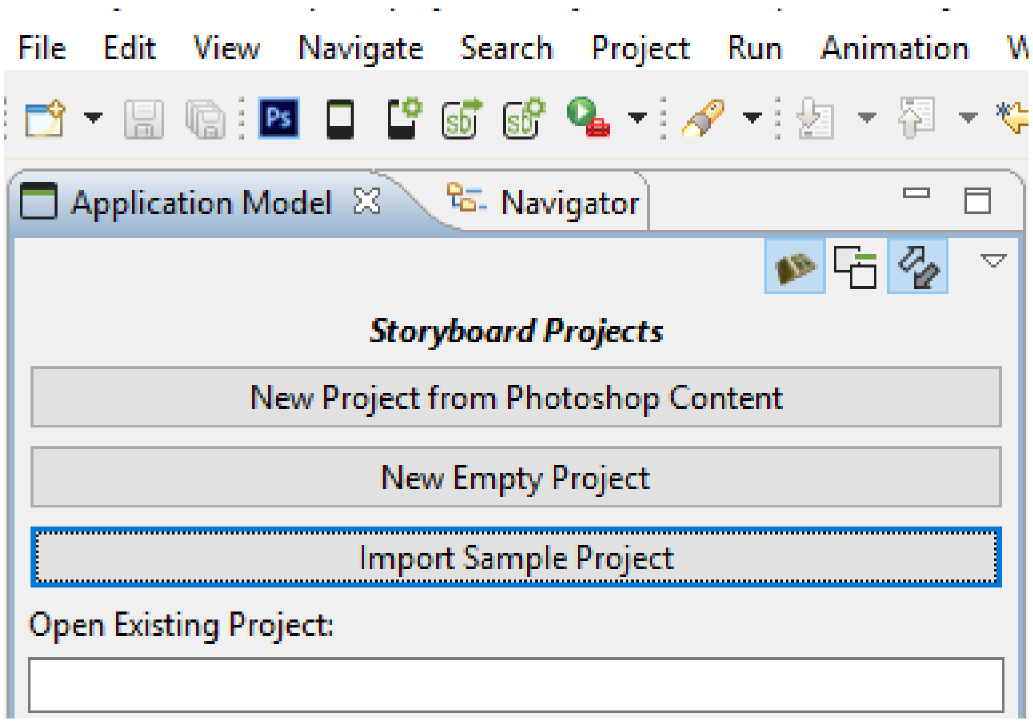
TI AM355 Starter Kit

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.

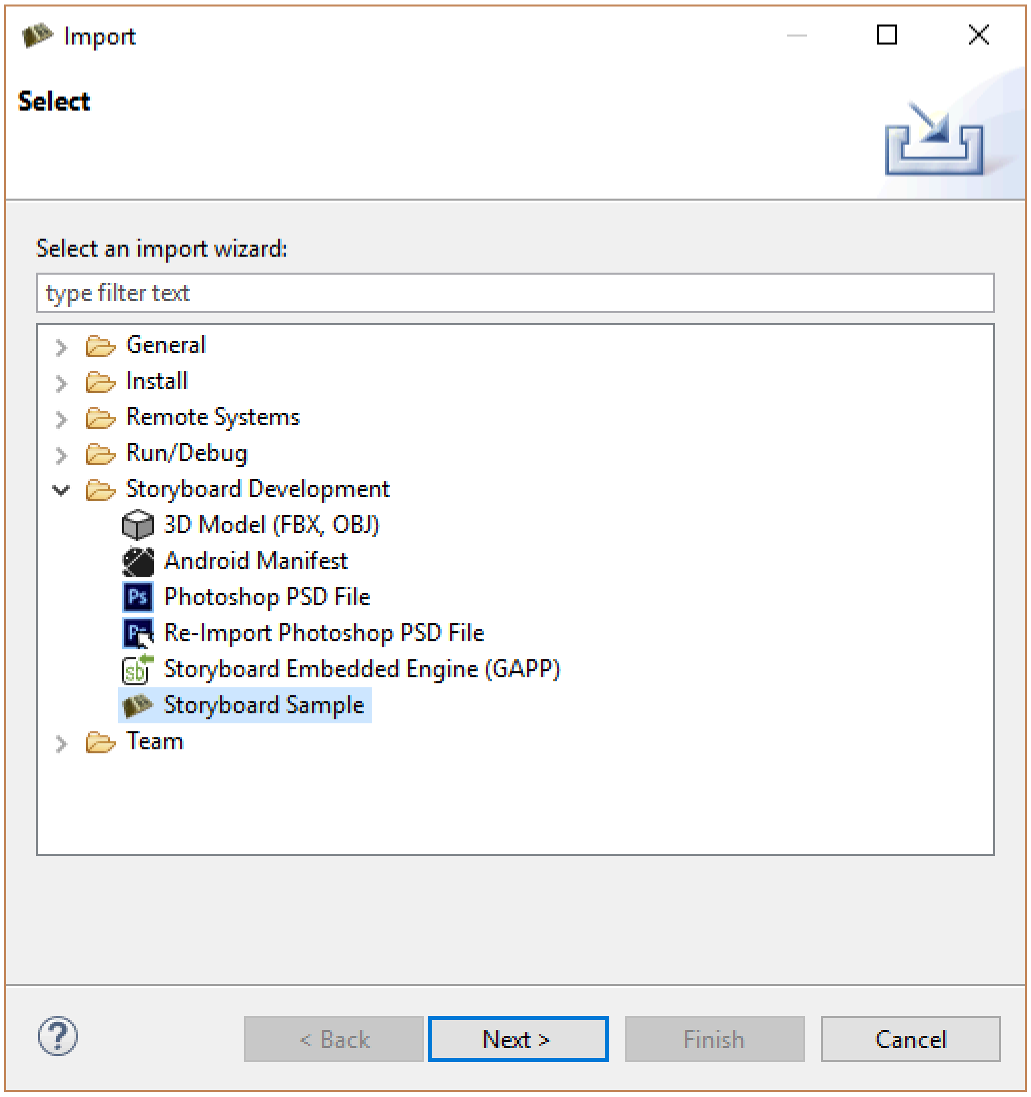
The following steps describe how to take a Storyboard sample and place it down on the TI AM335 board. It is assumed that the TI AM335 board has been setup correctly running Linux and that it is connected via a serial cable to either a laptop or desktop computer.

Step 1: Importing A Storyboard Sample

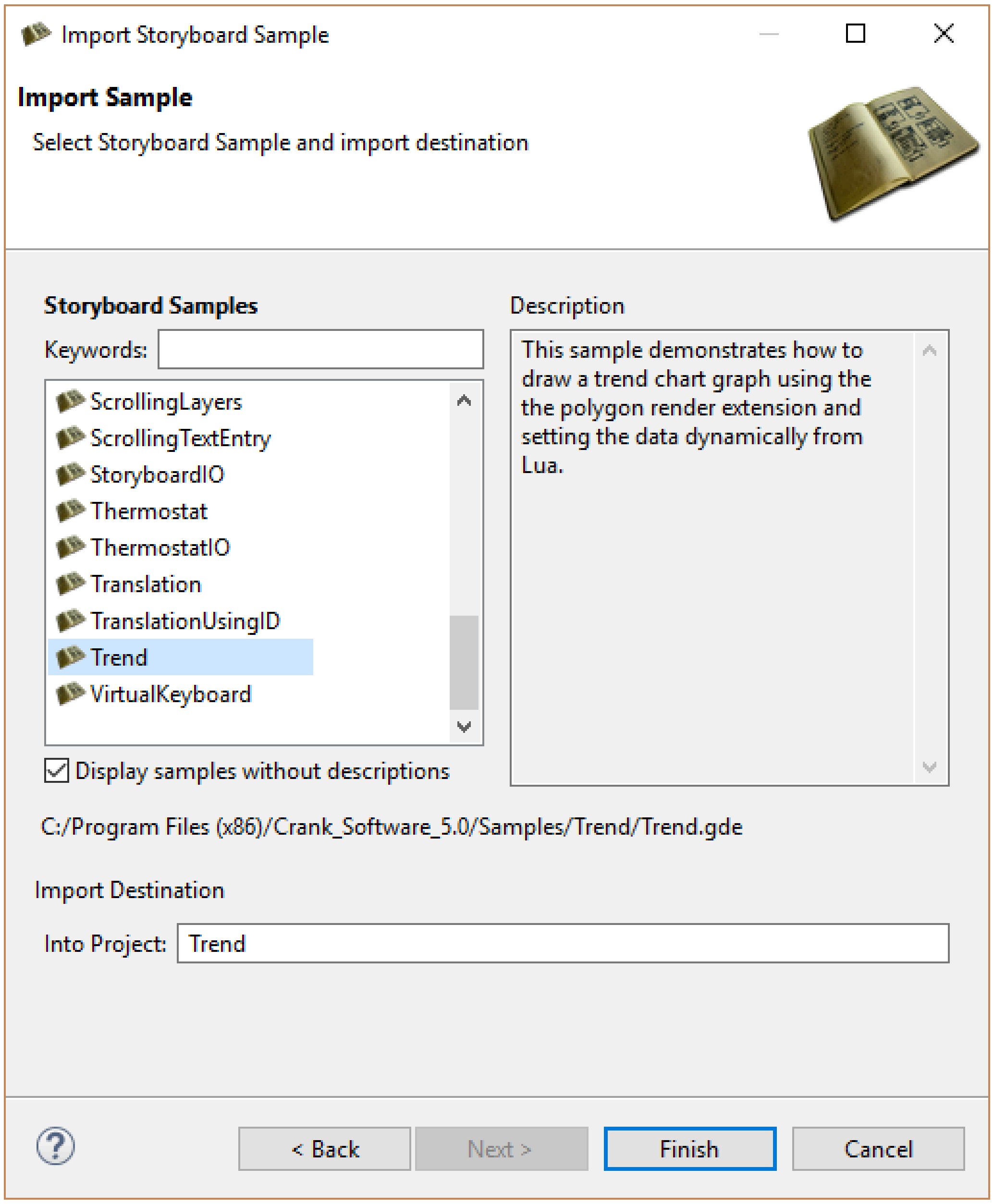
To import a Storyboard sample the user can right-click within the Navigator view and select Import or by selecting Import Sample Project from the Application Model view.



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.



Keywords:

☑ Display samples without descriptions

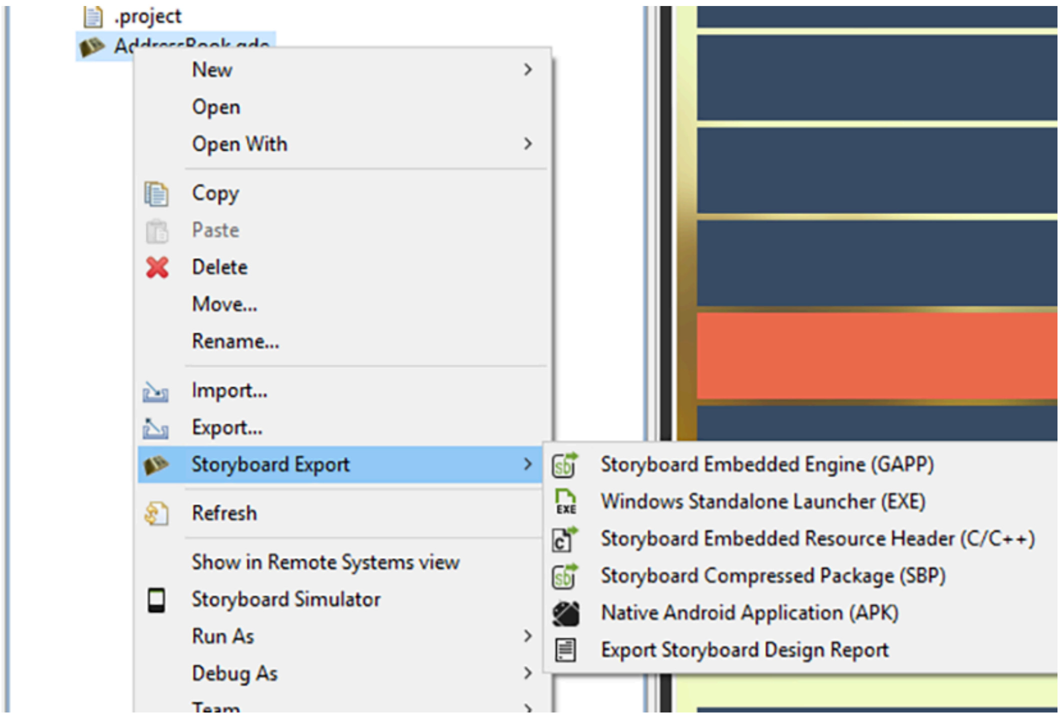
Description: This sample demonstrates how to draw a trend chart graph using the the polygon render extension and setting the data dynamically from Lua.

C:/Program Files (x86)/Crank_Software_5.0/Samples/Trend/Trend.gde

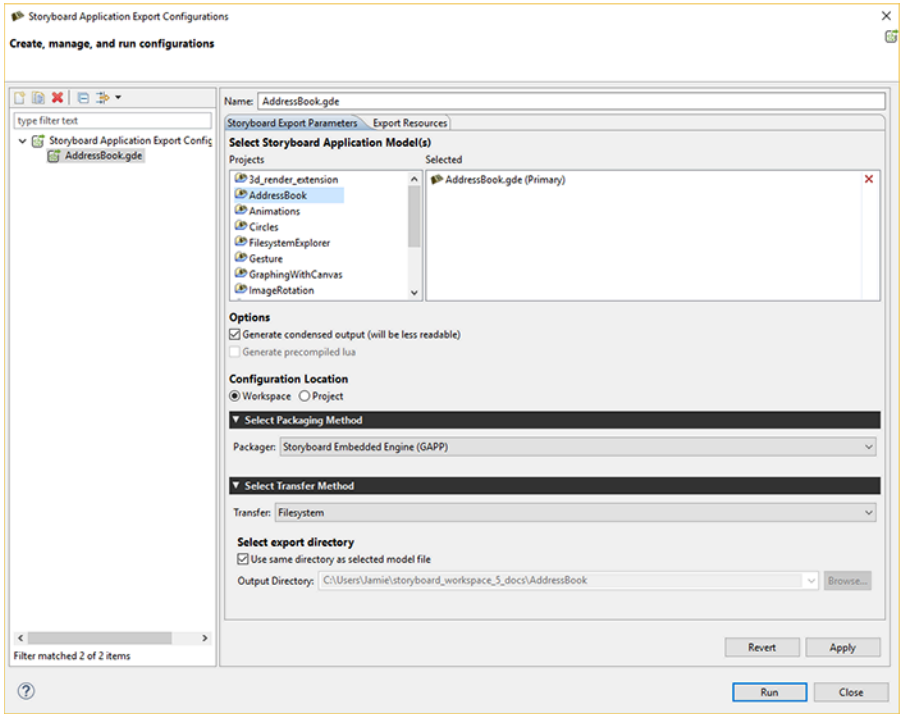
Into Project: Trend

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.


.project
AddressBook.gde
New
Open
Open With
Copy
Paste
Delete
Move...
Rename...
Import...
Export...
Storyboard Export
Refresh
Show in Remote Systems view
Storyboard Simulator
Run As
Debug As
Team
Storyboard Embedded Engine (GAPP)
Windows Standalone Launcher (EXE)
Storyboard Embedded Resource Header (C/C++)
Storyboard Compressed Package (SBP)
Native Android Application (APK)
Export Storyboard Design Report

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).


Storyboard Application Export Configurations

Create, manage, and run configurations

Name: AddressBook.gde

Select Storyboard Application Model(s)

Projects: 3d_render_extension, AddressBook, Animations, Circles, FilesystemExplorer, Gesture, GraphingWithCanvas, ImageRotation

Selected: AddressBook.gde (Primary)

Options
- [✓] Generate condensed output (will be less readable)
- [ ] Generate precompiled lua

Configuration Location: [✓ Workspace] Project

Select Packaging Method
Packager: Storyboard Embedded Engine (GAPP)

Select Transfer Method
Transfer: Filesystem

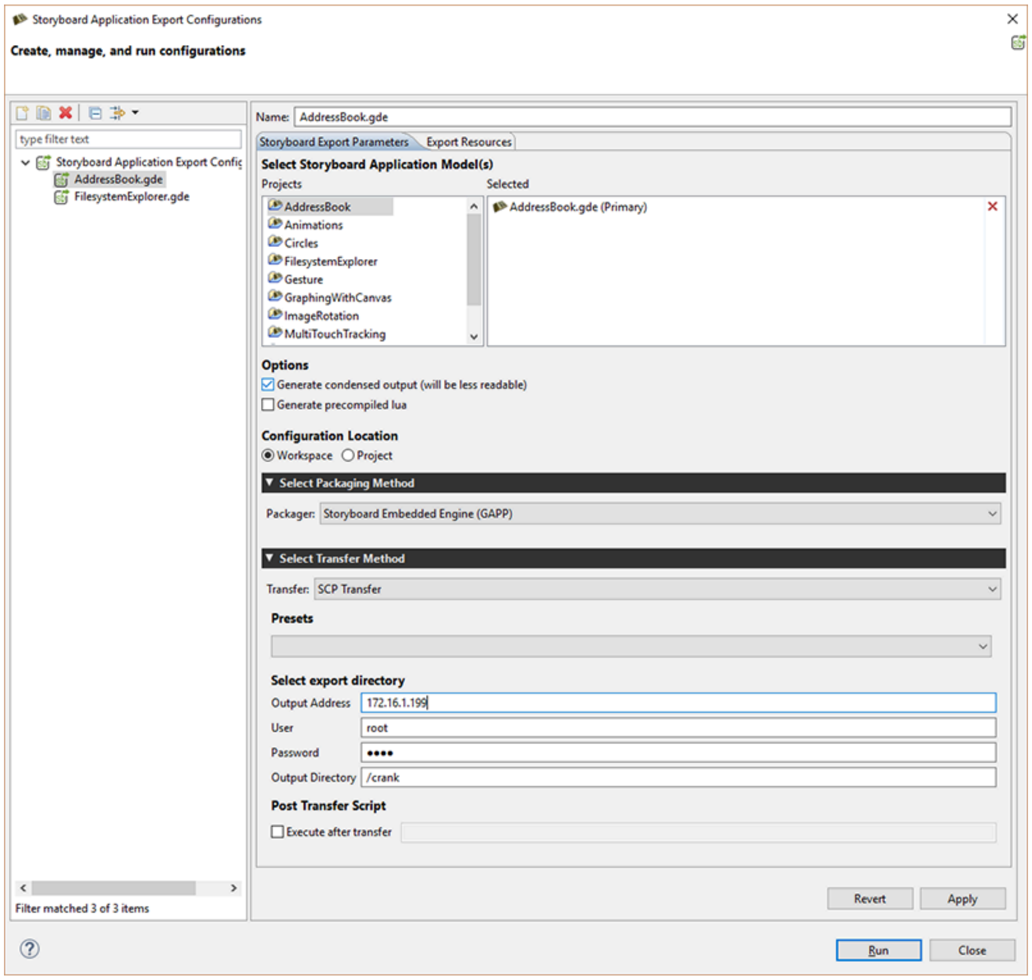
Select export directory
- [✓] Use same directory as selected model file

Output Directory: C:\Users\Jamie\storyboard_workspace_5_docs\AddressBook

Filter matched 2 of 2 items

Storyboard Application Export Config
type filter text
Storyboard Export Parameters
Export Resources
Browse...
Revert
Apply
Run
Close

Selecting SCP Transfer enables you to remotely connect to your target platform and transfer via SCP.



Storyboard Application Export Configurations

Create, manage, and run configurations

type filter text

- Storyboard Application Export Config
  - AddressBook.gde
  - FilesystemExplorer.gde

Filter matched 3 of 3 items

Name: AddressBook.gde

Storyboard Export Parameters | Export Resources

Select Storyboard Application Model(s)

Projects: AddressBook, Animations, Circles, FilesystemExplorer, Gesture, GraphingWithCanvas, ImageRotation, MultiTouchTracking

Selected: AddressBook.gde (Primary)

Options

- [✓] Generate condensed output (will be less readable)
- [ ] Generate precompiled lua

Configuration Location

[✓ Workspace] Project

Select Packaging Method

Packager: Storyboard Embedded Engine (GAPP)

Select Transfer Method

Transfer: SCP Transfer

Presets

Select export directory

Output Address 172.16.1.199
User root
Password ••••
Output Directory /crank

Post Transfer Script

- [ ] Execute after transfer

Step 3: Selecting The Storyboard Embedded Engine

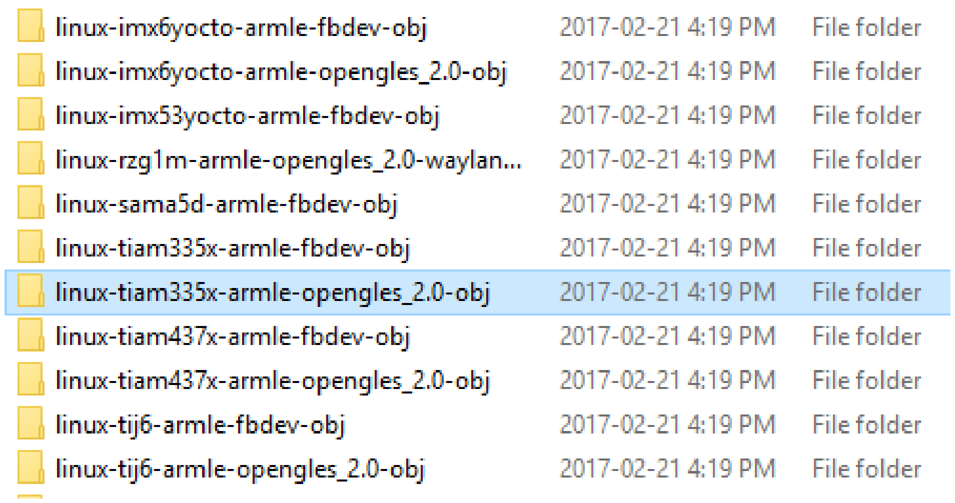
The Storyboard Embedded Engine is the optimized runtime component that resides on the target platform that interprets the data bundle to display the Storyboard application. The Storyboard Embedded Engine is categorized by operating system, system architecture, and rendering technology.

All the supported Storyboard Embedded Engines are shipped with Storyboard Suite 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/plug-
ins
```

```
export         LD_LIBRARY_PATH=/mnt/usb/linux-codesourcery-armle-open-
gles_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

Chapter 39. Release Notes 5.0

Introduction

Welcome to the release of Storyboard 5.0

Storyboard Designer

The Storyboard Suite 5.0 release includes new features, stability and performance enhancements. Storyboard 5.0 will automatically convert Storyboard 4.x and earlier workspaces and project files. Once converted to Storyboard 5.0 format, Storyboard projects are no longer compatible with Storyboard 4 and earlier projects.

Changes

Eclipse

Storyboard Designer is now based on the Eclipse Framework NEON 4.6

Lua Editor

Code completion for mapargs standard arguments in Lua editor.

x64

Storyboard Designer has been switched from a 32 bit application to 64 bit on all hosted platforms.

Standalone EXE export

A stand-alone launcher that includes a Storyboard runtime and a Storyboard application can now be exported from Storyboard Deisgner.

Single Model File

Storyboard Designer gives the option to export your Storyboard application as a single file for added security on the target. This file is a single ZIP compressed file that includes the model (gapp) file and all of its assets (images, fonts, scripts etc)

Components

Templates have been rebranded and are now called Components. New look, same great taste :)

SCP For Target Application Deployment

Export a Storyboard Application directly to a target from within Storyboard Designer.

Locking of Storyboard Designer Elements

Now have the ability to lock Storyboard Elements so they can not be moved or their properties changed.

Storyboard Canvas Plugin

The canvas plugin offers a programmatic way to perform a series of drawing operations without having to deal with all the setup of cloned dynamic render extensions.

Custom Animation Tweens

Users will be able to provide their own Lua function to implement specialized animation rates.

Newlines in Text Properties Panel

Support for CTRL+ENTER to automatically insert a newline has been added to the UI creator.

Known Issues

Linux x64 Installer

Users of 32 bit Linux will need to download 64 bit libraries to run the installer.

Require Mac OS X 10.8 or greater

Storyboard Designer requires Mac OS X 10.8 or greater due to the requirement by the Java 8 runtime from Oracle.

Resource Cleanup tool

Images that are not referenced directly, but are pulled in at runtime-based on a Lua script action, will not be identified as used by the Resource Cleanup wizard and may be removed accidentally.

Storyboard Engine

Storyboard Engine continues to move forward with the goal of providing our customers with a broad list of supported targets.

Changes

FFMPEG

ffmpeg is now the generic video player for Storyboard cross platform.

Multisampling

Multisampling is now turned off by default unless there is a 3D model contained in the Storyboard application.

Known Issues

OpenGL on Windows

A recent Windows update of the driver for Intel based graphics chips to version 20.19.15.4568 (2016-12-16) will prevent the Storyboard OpenGLES 2.0 runtime from running correctly. You can roll back the update or if you have a hybrid card, try using the discrete graphics processor instead of the integrated graphics processor.

Texture Memory

"ERROR (1):ERROR: 505 : create_image_texture@2791" relates to running out of texture memory. Since the texture is purged when the image resource gets released you can use the resource manager "image" pool size option, i.e sbengine -oresource_mgr,image=4096 your_app.gapp, to get around this issue. This option enables you to manually set the image pool size. (By default sbengine will use all available memory) When the image pool size is reached, older images are released to create room for newer images.

3D Layer Rotation

3D layer rotation does not respect alpha. When a 3D layer, that has transparency, is rotated it gets rendered with a white fill.

9-Patch

9-Patch images currently can't be rotated.

Linux Wayland Runtimes

The Linux Wayland runtimes do not support keyboards.

Flickering Graphics

Using the Yocto Jethro Linux kernel (3.14) with the boundary devices branch for the nitrogen6x you might encounter flickering graphics. If so, executing the following line resolves the issue ... echo 10 >/sys/devices/soc0/backlight_lvds0.17/backlight/backlight_lvds0.17/brightness

Chapter 40. Release Notes 5.1

Introduction

Welcome to the release of Storyboard 5.1

Storyboard Designer

The Storyboard Suite 5.1 release includes new features, stability and performance enhancements. Storyboard 5.1 will automatically convert Storyboard 4.x and earlier workspaces and project files. Once converted to Storyboard 5.1 format, Storyboard projects are no longer compatible with Storyboard 4 and earlier projects.

Changes

Eclipse

Storyboard Designer is now based on the Eclipse Framework NEON 4.6

x64

Storyboard Designer has been switched from a 32 bit application to 64 bit on all hosted platforms.

Cloning

When cloning an object, we now check to see if a control with the new name already exists.

Keyboard Shortcuts to Group and Ungroup

Added concept of grouping and un-grouping through a new action bound to Control/Command + G and Control/Command + SHIFT + G.

SCP

The SCP command execution now has the ability to write output to the console.

Lua Debugger

Designer now queries the Lua debug port number in use and passes it into the runtime.

Known Issues

Linux x64 Installer

Users of 32 bit Linux will need to download 64 bit libraries to run the installer.

Require Mac OS X 10.8 or greater

Storyboard Designer requires Mac OS X 10.8 or greater due to the requirement by the Java 8 runtime from Oracle.

Resource Cleanup tool

Images that are not referenced directly, but are pulled in at runtime-based on a Lua script action, will not be identified as used by the Resource Cleanup wizard and may be removed accidentally.

Storyboard Engine

Storyboard Engine continues to move forward with the goal of providing our customers with a broad list of supported targets.

Changes

QNX 7

A QNX 7 swrender and OpenGL runtime are now available.

ST Accordo5

A ST Accordo5 OpenGL Wayland runtime is being shipped.

Runtimes No Longer Shipped

linux-x86-fbdev

linux-fsl-armle-fbdev

linux-fsl-armle-opengles_2.0-imx6

qnx-6_5-armv7le-opengles_2.0

Latest Intel Graphics Drivers

The win32 opengles2.0 runtime is now compiled against the new PowerVR SDK to support the latest Intel graphics drivers.

Canvas

canvas:draw_image now properly scales image when using the OpenGL runtime.

VxWorks

The VxWorks OS is now a supported platform.

Renesas RZA1

Support for the RGA renderer on the RZA1 platform has been added.

iOS

The iOS runtme is now built against the iOS 10 SDK.

Focus Events

Inactive controls no longer receive focus events.

Input Rotation

Input now works correctly when rotating screen on QNX.

Known Issues

ST Accordo5

3D models are not currently supported.

Multi-Sampling

The -orender_mgr,multisample=0 commandline option needs to be passed to the WEC2013 and ST Accordo5 OpenGL runtimes to run correctly.

Texture Memory

"ERROR (1):ERROR: 505 : create_image_texture@2791" relates to running out of texture memory. Since the texture is purged when the image resource gets released you can use the resource manager "image" pool size option, i.e sbengine -oresource_mgr,image=4096 your_app.gapp, to get around this issue. This option enables you to manually set the image pool size. (By default sbengine will use all available memory) When the image pool size is reached, older images are released to create room for newer images.

3D Layer Rotation

3D layer rotation does not respect alpha. When a 3D layer, that has transparency, is rotated it gets rendered with a white fill.

9-Patch

9-Patch images currently can't be rotated.

Linux Wayland Runtimes

The Linux Wayland runtimes do not support keyboards.

Flickering Graphics

Using the Yocto Jethro Linux kernel (3.14) with the boundary devices branch for the nitrogen6x you might encounter flickering graphics. If so, executing the following line resolves the issue ... echo 10 >/sys/devices/soc0/backlight_lvds0.17/backlight/backlight_lvds0.17/brightness

Chapter 41. Release Notes 5.2

Introduction

Welcome to the release of Storyboard 5.2

Storyboard Designer

The Storyboard Suite 5.2 release includes new features, stability and performance enhancements. Storyboard 5.2 will automatically convert Storyboard 4.x and earlier workspaces and project files. Once converted to Storyboard 5.2 format, Storyboard projects are no longer compatible with Storyboard 4 and earlier projects.

Changes

Rich Text Render Extension

Richer text display capability to allow light text formatting (different fonts, size, colors)

Storyboard Connector

A new view in Storyboard Designer allows users to select external events , and generate them from within Designer to a running simulation.

Adobe Artboards

Artboards can now be imported directly into Storyboard Designer as screens.

PSD Reimport Comparison

The reimport will now show both the new image render extension and the old fill render extension in the comparison view. The user will have the choice to keep both or pick one or the other.

Command Line Export Support

Export configurations can now be triggered from the command line.

Exporting Large Projects

Large project exporting has been optimized for speed.

Unused Variables

When unused variables or unresolvable variables are detected, they are now highlighted with a warning marker in the Application View and show up as markers in the Problems View.

Internal Variables

Internal variables that are tied to an event are now displayed in the variables list without having the specific filter selected.

History View

Storyboard Designer now has a history view that allows insight into the undo/redo command stack.

Custom Event Editor

Custom user events now have an advanced editor of their own that offers tighter integration with the Storyboard Connector view and allows direct export to C/C++ header files.

Storyboard Search

When a GDE model file is open in the background, Storyboard search can now take place in the context of the LUA editor.

Known Issues

Ubuntu

Ubuntu 17.10 now uses Wayland as it's default compositor. The Eclipse version that serves as the framework for Storyboard Designer has some issues with Wayland. There are rendering issues when Storyboard Designer first comes up, and it will crash fairly quickly after starting with a "No more handles" error. To work around this, start Ubuntu with the Ubuntu Xorg configuration. To launch this configuration, click your username. Select the gear icon that is displayed below the password box, and then select "Ubuntu xorg". This will start the UI without using Wayland.

3D Model Axis

The 3D model's axis no longer shows up in the direct editing mode.

Designer Notes

HTML generated designer notes cannot be viewed within the Linux version of Storyboard Designer.

3D Control/Layer Rotation

In Storyboard Designer, when a control/layer with transparency is rotated, it is rendered with a white fill.

Storyboard Engine

Storyboard Engine continues to move forward with the goal of providing our customers with a broad list of supported targets.

Changes

Native Win32 Runtime Deprecation

This is a deprecation notice for the Win32 API-based renderer for Windows, Windows Compact 7, and Windows Embedded Compact 2013 platforms. The Win32 API renderer is being replaced with a Win32 software renderer that offers improved performance and rendering capability with other Storyboard platforms.

Windows Compact 7 (WinCompact7) Deprecation

This is a deprecation notice for the Windows Compact 7 platform. If ongoing support of this platform is of particular importance to you, please contact support@cranksoftware.com to discuss the deprecation timeline.

Compressed Storyboard Package (*.sbp) Support

To satisfy international export regulations, the runtime plugin to support SBP archives must be provided separately. E-mail support@cranksoftware.com to receive a plugin for your platform.

File IO falls back to local filesystem if the referenced file does not exist in the .sbp archive.

VxWorks

VxWorks OS is now a supported OS target.

STM32

STM32479I-EVAL is now a supported platform.

Resource Manager

The resource manager unit options for controlling the font and image cache sizes have changed to from kilobytes to bytes and now supports the symbolic qualifiers K and M.

Media Backend

The disconnect action has been exposed as an action in Designer. The media plugin can now send a disconnect to the media backend without exiting.

Scrolling Tables and Layers

Tables and layers no longer scroll when there are no cells or controls outside the current viewport.

Scroll Configuration Parameters

Users can reconfigure the drag threshold and the bounce back time values for their system through the use of command line arguments.

ffmpeg Plugin

The plugin now only starts automatically after it walks the model and finds actions that are part of the media plugin. The -offmpeg, forceload option can be passed to sbengine, forcing the ffmpeg plugin to start.

Frames Per Second Display

The FPS display format on the screen manager has changed and it no longer requires additional levels of verbosity to be output.

Subclipse

When following instructions to install Subclipse to connect to the Crank Public SVN, note that Subclipse is now located at https://dl.bintray.com/subclipse/releases/subclipse/latest/

Known Issues

Flickering Between Screen Transitions

fbdev is single buffered by default, which can cause flickering. Enabling a double buffered framebuffer with -orender_mgr,dblbuffer can resolve the issue.

9-Patch Image Rotation

The rotation of 9-patch images is not supported.

Tables Not Cloneable

Tables are not cloneable using the Lua gre.clone_object() API

iOS Render Manager

We do not currently support multisampling on the iOS runtime.

ffmpeg Plugin

Due to the large ffmpeg size, you can remove if it isn’t needed to save space on resource-constrained target systems.

Chapter 42. Release Notes 5.3

Introduction

Welcome to the release of Storyboard 5.3

Storyboard Designer

The Storyboard Suite 5.3 release includes new features, stability and performance enhancements. Storyboard 5.3 will automatically convert Storyboard 4.x and earlier workspaces and project files. Once converted to Storyboard 5.3 format, Storyboard projects are no longer compatible with Storyboard 4 and earlier projects.

Changes

Major Performance Analysis Work

- Resource allocating object tracking: animations, timers, controls
- Broad range of pre-configured statistics
- Event filtering for content in statistics
- Event filtering for generation at command line
- External runtime performance start/stop event support

Screen Transitions

New change display orientation screen transition action.

Multi-gde

Secondary gdes are now added to the .launch file correctly.

Text Adjustments

- Standard text ellipsis option
- Rich text vertical text alignment

Lua

- RGB color creation API (gre.rgb, gre.torgb)
- Engine integrated logging API (gre.log)
- Clone support extended to tables (and tables in groups)
- Line information for gre function errors and warnings

- Lua GREDOM moved from a library to a plugin

Designer UI Refinements

- Automatic open of generated files (design report)
- Screen transition names better labelled
- Timer names available as drop down selection
- Timer Stop action now available
- Consistent default name (sbengine_model.h) on C/C++ header export
- Design report screen transition reporting overhaul
- Text editing undo operations
- Save As option for component editor
- Alignment controls decoupled from the editor
- Multi-select editing of animation FPS values
- Image rotation center point guidelines

Quality and Efficiency

- Designer compare/merge algorithmic hardening
- Automatic detection for gesture and multi-touch plugin enablement
- Event playback in SBVFS runtime environments

Known Issues

Installer

Starting with MacOS version 10.13.4, users are now notified when they use 32 bit applications for the first time with a pop-up. This will happen when you launch Storyboard Suite Installer.

Designer Notes

HTML generated designer notes cannot be viewed within the Ubuntu 16.04 and 17.10 version of Storyboard Designer. To work around the issue libwebkitgtk-3.0-0 needs to be installed.

```
sudo apt-get install libwebkitgtk-3.0-0
```

Ubuntu

Ubuntu 17.10 now uses Wayland as it's default compositor. The Eclipse version that serves as the framework for Storyboard Designer has some issues with Wayland. There are rendering issues when Storyboard Designer first comes up, and it will crash fairly quickly after starting with a "No more handles" error. To work around this, start Ubuntu with the Ubuntu Xorg configuration. To launch this configuration, click your username. Select the gear icon that is displayed below the password box, and then select "Ubuntu xorg". This will start the UI without using Wayland.

3D Control/Layer Rotation

In Storyboard Designer, when a control/layer with transparency is rotated, it is rendered with a white fill.

Storyboard Engine

Storyboard Engine continues to move forward with the goal of providing our customers with a broad list of supported targets.

Changes

Raspberry Pi

Raspberry Pi swrender and OpenGL runtimes are now shipping

QNX Scaling

For QNX the -orender_mgr,scale option has been deprecated and has been replaced by -orender_mgr,window_w=display_width,window_h=display_height

FPS

The -oscreen_mgr,fps option has been deprecated

Gstreamer

Audio can now be paused using gstreamer.

Native Win32 Runtime Deprecation

This is a deprecation notice for the Win32 API-based renderer for Windows, Windows Compact 7, and Windows Embedded Compact 2013 platforms. The Win32 API renderer is being replaced with a Win32 software renderer that offers improved performance and rendering capability with other Storyboard platforms.

Windows Compact 7 (WinCompact7) Deprecation

This is a deprecation notice for the Windows Compact 7 platform. If ongoing support of this platform is of particular importance to you, please contact support@cranksoftware.com to discuss the deprecation timeline.

ffmpeg Plugin

The plugin now only starts automatically after it walks the model and finds actions that are part of the media plugin. The -offmpeg, forceload option can be passed to sbengine, forcing the ffmpeg plugin to start.

Frames Per Second Display

The FPS display format on the screen manager has changed and it no longer requires additional levels of verbosity to be output.

Subclipse

When following instructions to install Subclipse to connect to the Crank Public SVN, note that Subclipse is now located at https://dl.bintray.com/subclipse/releases/subclipse/latest/

Known Issues

Flickering Between Screen Transitions

fbdev is single buffered by default, which can cause flickering. Enabling a double buffered framebuffer with -orender_mgr,dblbuffer can resolve the issue.

9-Patch Image Rotation

The rotation of 9-patch images is not supported.

iOS Render Manager

We do not currently support multisampling on the iOS runtime.

ffmpeg Plugin

Due to the large ffmpeg size, you can remove if it isn't needed to save space on resource-constrained target systems.

Part IV. Licensing

Table of Contents

Chapter 43. Storyboard Suite Licensing

Storyboard Suite 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`.