OtterUI Documentation

Introduction

Introduction:

OtterUI is a User Interface development solution for embedded systems and interactive entertainment software. It is designed to be both cross-platform and engine-agnostic, allowing developers full control over platform-specific components. It focuses ease of use, rapid iteration, minimal integration and maximum developer control.

OtterUI comes in two parts: the standalone OtterUI Editor and the OtterUI API.

High-Level Features:

- Cross-platform and engine agnostic API (Application Program Interface)

  The OtterUI API is written in standard C/C++ with no third party library dependencies. By giving you full control over platform-specific components (ex: rendering), the API remains engine-agnostic and very flexible, allowing you to integrate and use OtterUI on a wide variety of platforms and engines.

  Drag-and-Drop Visual Editor

  The OtterUI Editor is a fully visual drag-and-drop UI layout and animation tool. It provides immediate visual feedback to allow for rapid iteration and refinement of your user interface.

  Keyframed animations with interactive Timeline

  Animating controls is as simple as placing them in the desired location, and creating a keyframe. OtterUI will take care of the rest : interpolation, placement, interaction, etc.

  Action Lists

  Create actions (sounds or messages) on animations through the Timeline control. Keep your animation and application-event perfectly synchronized, without the need of extra engineering time or effort

  Multiple Resolution support through control anchors

  With OtterUI there's no need to create multiple User Interfaces to account for different resolutions on different platforms. OtterUI's specialized anchors allows a single UI to run on multiple resolutions. Controls and animations will automatically account for the new resolution.

  TrueType Font (TTF) processing and rendering

  The OtterUI Editor will process TrueType Font (TTF) files and create the glyph sheets for your UI. It extracts all of the necessary glyph data and processes it invisibly at runtime, eliminating the need for a custom font engine.

  Unicode support and Custom Glyphs

  In addition to processing TTF fonts, OtterUI also provides support for Unicode and Custom Glyphs. Embed your own special characters and images (emoticons, special buttons, avatars, etc) directly in your text.

Getting Started

Package Overview

Otter is packaged and delivered as a zip file, and contains all the necessary libraries, executables, documentation needed to start development.

It is organized into largely into the following four areas:

API:

/API/inc	*Include directory*
/API/lib/android	*Android libraries*
/API/lib/ios	*iOS libraries*
/API/lib/win32	*Win32 libraries*

Editor:

/Editor	*Otter Editor installation executables (Win32 only)*

Help:

/Help	*Help Documentation*

SampleApp:

/SampleApp	*Sample Applications*
/SampleApp/bin	*Common assets, binary data, working folders, etc.*
/SampleApp/bin/Assets	*Sample App source assets*
/SampleApp/bin/Data	*Sample App exported assets*
/SampleApp/src	*Common Sample Application source*
/SampleApp/Android	*Android Sample Application*
/SampleApp/iOS	*iOS Sample Application*
/SampleApp/Win32	*Win32 Sample Application*
/SampleApp/Unity	*Unity Sample Application (PC and MacOS)*

Supported Platforms

The Otter API was designed and developed as a cross-platform library and relies on minimal external dependencies to reduce porting effort to current and future platforms.

The API comes with precompiled libraries for the following platforms:

Windows:

/lib/win32/OtterMT.lib	*Multi-threaded Release*
/lib/win32/OtterMD.lib	*Multi-threaded DLL Release*
/lib/win32/OtterMTD.lib	*Multi-threaded Debug*
/lib/win32/OtterMDD.lib	*Multi-threaded DLL Debug*

iOS:

/lib/iOS/libOtter.a	*Release*
/lib/iOS/libOtterD.a	*Debug*

Android:

/lib/android/libOtter.a	*Release*
/lib/android/libOtterD.a	*Debug*

Unity:

/lib/unity/OtterC.dll	*PC Plugin*
/lib/unity/OtterC.bundle	*MacOS Plugin*
/lib/unity/iOS/OtterC.a	*iOS Plugin*

Terminology

Control:

A Control is any object that can be seen, animated, and/or interacted with in some way. Controls are the basic building blocks of the User Interface. Example include a sprite image, button, label, etc.

View:

A View is a collection of controls and animations that represents a logical "screen" in a User Interface. An example of a view is the main menu or splash screen in your application. Views can also take the form of message boxes and onscreen HUDs (Heads-Up Displays). Multiple views can become active at the same time, and may overlay one another.

Scene:

A Scene is a collection of Views, and can be loaded or unloaded at runtime as needed. This is particularly useful if your user interface is large and you need to unload multiple views to maintain a small memory footprint.

Animation:

An Animation dictates how a control will be transformed over a period of time. Animations comprise of a set of *keyframes*, and interpolates between each keyframe to produce the desired transformation of the control. Animations have a preset running length and will always continue to run until the last frame, regardless of the actual number of keyframes. For example, if an animation has a length of 100 frames, but contain only two keyframes at frame 0 and frame 50, the animation will not "finish" until 100 frames have been processed and rendered.

Keyframe:

A keyframe represents a single "snapshot" of a control's layout within an animation. Keyframes contain general information such as rotation and translation, but can also store control-specific information like text scale or sprite color.

During animation, keyframes interpolate with one another to produce smooth animations.

Actions:

Actions are performed by animations on specific frames, and is the most basic way an animation may communicate with the application. Examples of actions may include sound playback or sending a string message to the application. Each animation frame can execute an unlimited amount of actions, as specified through the OtterUI Editor.

Samples

A sample application is provided to demonstrate some of the features and benefits of OtterUI and can be found under the /SampleApp directory. The application uses the same OtterUI project, with a common source code, exported to multiple platforms. Platform-specific code is located in the sample application's subdirectories, under the specific platform.

Instructions to build and run the sample application are as follows:

Windows

Requirements:

- Visual Studio 2010
- Latest DirectX SDK

Instructions:

1. Locate /SampleApp/Windows/SampleApp.sln, and open in Visual Studio
2. Set the SampleApp project's working folder to "../bin" (right-click on the SampleApp project -> Properties -> Debugging -> Working Directory)
3. Build and run

iOS

Requirements:

- Latest iOS SDK
- Latest XCode

Instructions:

1. Install the latest version of the iOS SDK and XCode.
2. Locate and open /SampleApp/iOS/SampleApp.xcodeproj in XCode
3. Build and Run in Simulator

Android

Requirements:

- Android SDK (2.2 or higher)
- Android NDK r4 or higher
- Eclipse for Java Developers

Instructions:

1. In Eclipse:
    - File -> New -> Android Project
    - Select Create Project from Existing Source
    - Browse to /SampleApp/Android
    - Hit "Finish"
    - Create a new Android Virtual Device from the AVD Manager (Window -> Android SDK and AVD Manager).
        - Target:

`Android 2.2 - API Level 8 (or higher)`

SD Card Size:

`256mb (or higher)`

Skin:

`Default (HVGA)`

Hardware:

`Abstracted LCD Density = 160`

2. In Cygwin:
    - Browse to /SampleApp/Android/jni
    - Execute:

      ```
      ndk-build
      ```

    - Execute:

      ```
      adb push ../../bin/Data/Android /sdcard/otter
      ```

3. In Eclipse:
    - Run -> Run or Debug

Unity:

Requirements:

- Unity 3.2 or Higher
- iOS SDK 4.2 or Higher and XCode (for iOS / MacOSX Samples)

1. Open Unity and create a new project
2. Import the Unity Package at ./SampleApp/Unity/OtterSample.unitypackage
3. Drag and Drop the `SampleUI.cs` script onto the Main Camera
4. Set the Main Camera's background to **Black**

MacOS:

1. Switch to PC and Mac Standalone platform
    1. Target Platform: `Mac OS X Universal`
2. Open Plugins/OtterSharp/C.cs
    1. Set `Otter.LibrarySettings.LibraryName` to `"OtterC"`
    2. Set `Otter.LibrarySettings.UseCommandBuffer` to `false`

iOS:

3. Switch to iOS Platform
4. Under Player Settings -> "Settings for iOS" -> "Resolution and Orientation" -> Default Orientation = `Landscape Left`
5. Under Player Settings -> "Settings for iOS" -> "Other Settings" -> SDK Version = `iOS Latest`
6. Open Plugins/OtterSharp/C.cs
    1. Set `Otter.LibrarySettings.LibraryName` to `"__Internal"`
    2. Set `Otter.LibrarySettings.UseCommandBuffer` to `true`

Editor

Purpose of the Editor
Where it fits in
Major Features

Setup

Installation:

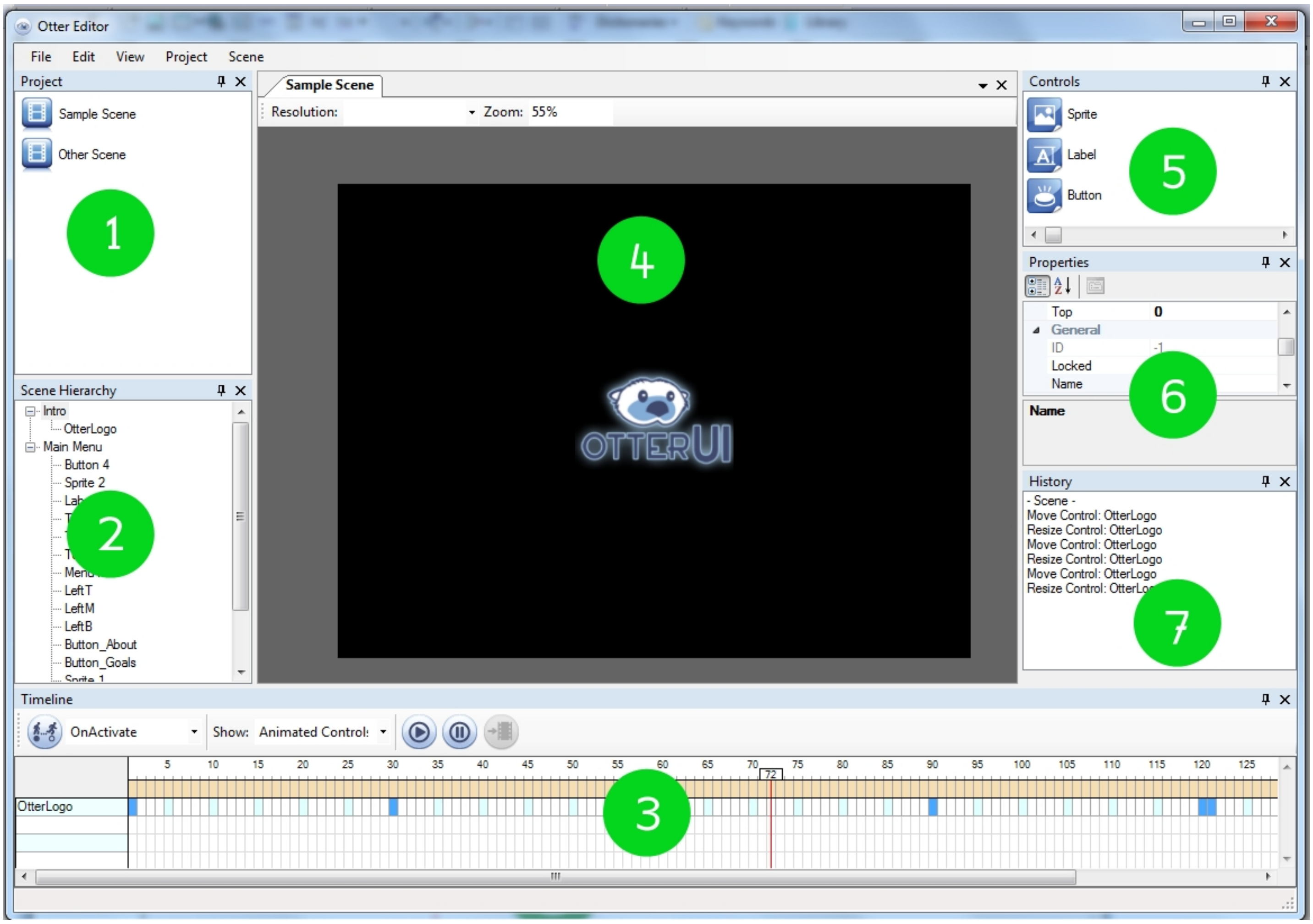
Installation and setup of the OtterUI Editor is very straight-forward. Simply locate and execute the setup file in /Editor, following the on-screen instructions. Once complete, the OtterUI Editor will be fully installed and ready to be used.

Minimum System Requirements:

- Windows XP, Vista or 7
- Microsoft .NET 4.0
- Latest version of DirectX
- Graphics card with VS 3.0 support

Overview

General Layout:



(1) Project View:

Displays and manages the scenes in your project. From this window you can create, delete or rename scenes.

(2) Scene Hierarchy

Displays and manages the view and control hierarchy of the active scene. This window allows you to create, delete and rename views and controls.

(3) Timeline

Manages the animations within a view. This window also allows you to preview animations by moving the current-frame indicator to the appropriate frame.

(4) Scene View

This is the main view of the OtterUI Editor. Within this view you can edit and preview the current scene and view. Multiple scenes can be open at the same time, but only one view in the scene can be viewed at a time.

(5) Controls

Lists the available controls in the OtterUI Editor.

(6) Properties

When a keyframe, control or view is selected the item's properties will be displayed here. Within the property window you can change the values of an item as is appropriate.

(7) History View

This window displays the undo/redo history. Clicking on entries in the list will either progress or revert to associated action.

Tutorials

Setting up a new Project

An OtterUI project manages a set of scene files as well as project-wide settings such as resolutions, platforms, fonts, and so on. In order to start building a User Interface using Otter, you must first create and set up a project.

4. **Create the Project**

To create a project, launch the Otter Editor and from the `File` menu click on `New` and specify where you want to create your new project. It is recommended to create the new project in its own folder, as the Otter Editor will use the location of the project when creating new scenes and other data.

Once the project has been created, it will be opened automatically in the Otter Editor.

5. **Add Platforms**

Once the project has been created, you must add platforms for which you want to build your user interface. Platforms may be as specific or as generic as you want them to be. For the purposes of this tutorial, we will be setting up a platform for a generic big-endian platform:

1. From the `Project` menu click on `Platforms...`
2. Click on the `Add` button found at the bottom of the left pane. A new platform will be created named `Platform 1`.
3. Click on the `Platform 1` entry. The properties pane on the right should now contain the editable values of the platform.
4. Rename your platform by changing the value in the Name field from `Platform 1` to `Big Endian`
5. Change the Endianness field to `Big`

Create as many platforms as you need. At export time you will have the option to export to specific platforms or all at once.

Note: A default platform for PC is provided automatically.

6. **Add Resolutions**

While building user interfaces it is very important to be able to preview your work in the target resolution. To do this, the Otter Editor must be aware of the resolutions you wish to work with. Adding new resolutions is easy:

1. From the `Project` menu click on `Resolutions...`
2. Click on the `Add` button found at the bottom of the left pane. A new resolution will be created.
3. Click on the new resolution, and in the properties pane on the right enter the required values for Width and Height. The resolution name will be updated to reflect the resolution's new values.

Note: A default resolution at 1024x768 is provided automatically.

7. **Add Fonts**

In order to render text labels, you will need some fonts. No font is provided automatically, so it is recommended to add at least one font as soon as possible.

1. From the `Project` menu click on `Fonts...`
2. Click on the `Add` button found at the bottom of the left pane. A new font will be created.
3. Click on the new font, and in the properties pane to the right click on the Filename field and then on the browse button to right. Browse to the desired TTF font file.
4. In the FontSize field specify the desired base font size, ex 36
5. In the Name field, specify a descriptive name for your font

8. **Done!**

Congratulations! Your project is now set up and ready to use.

Scenes

Scenes are simply a container for views, and can be loaded or unloaded at runtime giving you the opportunity to remove unnecessary views from memory if need be. Each scene is saved as a single file in the same location as the main project file. This tutorial assumes that a project has already been created and opened.

9. **Create a new scene**

In the Project window, right click and from the context menu select `Add`. A new scene will be created and saved to disk.

10. **Rename the scene**

To rename a scene, either select the scene and hit `F2`, or right click on it and select `Rename`.

11. **Open the scene**

To open a scene for editing, either double-click on it, or right-click on the scene and hit `Open`. The scene will be opened and ready for editing in the Scene View.

Views

A view is a collection of related controls and animations, and are organized within a single scene.

To create a view, first a scene must be created and opened for edit. See the Scene tutorial for further information. This tutorial assumes that a scene has been opened for editing.

1. **Create a new view**

In the Scene Hierarchy view, right-click and hit `Create View`. A new view will be created with a default name.

You can also rename or delete the view by selecting and right-clicking on it in the Scene Hierarchy.

2. **Select the view for editing**

Once a view has been created, to edit it simply select it in the scene hierarchy. Only one view can be edited at a time.

3. **Resolutions**

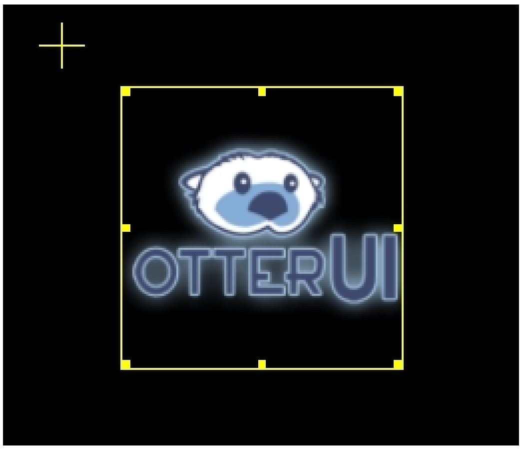
In the top-left corner of the Scene View (see Overview), select the appropriate resolution to preview your view.

4. **Adding controls**

To add a control to the view, simply drag-and-drop an item from the Controls View onto the opened Scene itself. The control will be created at the dropped location and onto the currently selected view.

5. **Manipulating controls**

To manipulate a control (moving, resizing, etc), first select the control itself. Once selected, a yellow selection box will surround the control to indicate the control's bounds and center:



The control selection box consists of the following:

- Control's center (Plus shape)
- Control's bounds (surrounding unfilled box)
- Bounds resize grips (yellow filled boxes)

In the above control, the center is to the top left of the control itself and offset slightly from its bounds. The center also indicates the current position of the control itself.

To move the control, click and drag within the large selection box. This will move the entire control to the desired location.

To resize the control, click and drag on any one of the individual resize grips. Note - resizing the control does not effect its center and therefore does *not* move the control's position!

Control rotation is achieved by typing a rotation angle in the Properties View when the control is selected, under the `Layout` field.

Animation

Each view maintains its own set of animations. Two default animations are created and required by all views:

OnActivate

This animation is automatically played whenever a view becomes active. At the very minimum, the OnActivate animation creates a single keyframe for each and every control. In this way, when a View is activated, all controls are animated and places into a known starting state.

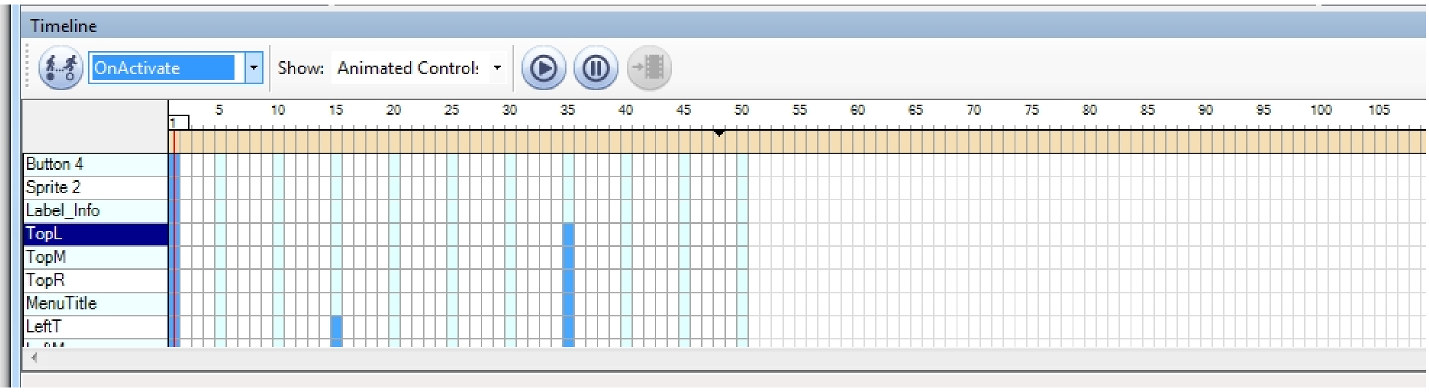
You can add keyframes to the OnActivate if needed, but it is not required. The OnActivate animation cannot be renamed or removed, and individual controls cannot be removed from the animation.

OnDeactivate

The OnDeactivate animation is automatically played whenever a view becomes inactive. It contains, by default, no animation channels or keyframes and it is not required that you animate any controls in OnDeactivate. It serves mostly as a convenience if your view has a generic outro animation that should get played whenever it is deactivated.

Timeline View:

Once you have created controls within a view, you can start animating them. This is done primarily through the Timeline View.



The Timeline displays a single animation at a time, and all of the animated controls within that animation. Each control is animated with a set of animation keyframes. In the above screenshot, the animated controls are listed to the left and their keyframes are indicated in the rows to the right. Each blue box represents a keyframe.

Each animated control is required to have a starting keyframe at frame 1 of the animation.

It is important to note that an animation will *only* animate the controls within that animation - all others will be ignored.

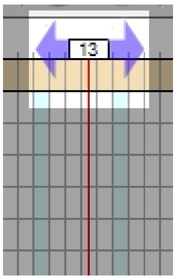
NOTE: The Timeline View will only be active when a view is currently being edited.

Creating an animation:

6. Click on the  button in the Timeline View. This will bring up the Animation Manager.
7. Click `Add`
8. Select the new animation and change the Name field to something descriptive
9. Change the NumFrames field to the number of frames required for this animation.
10. Hit `OK` and select the new animation from the animation dropdown.

Animating a control:

1. Move the timeline's current frame indicator to the target frame by dragging the white box left above the right vertical line:



2. Select a control within the current view
3. Drag the control into position and manipulate its layout properties (color, rotation, etc) to the desired state.
4. Right-click on the control and select `Create Keyframe.` A new keyframe will be created at the currently selected frame. If the control was not previously animated in the animation, it will automatically create and add a starting keyframe at frame 0.

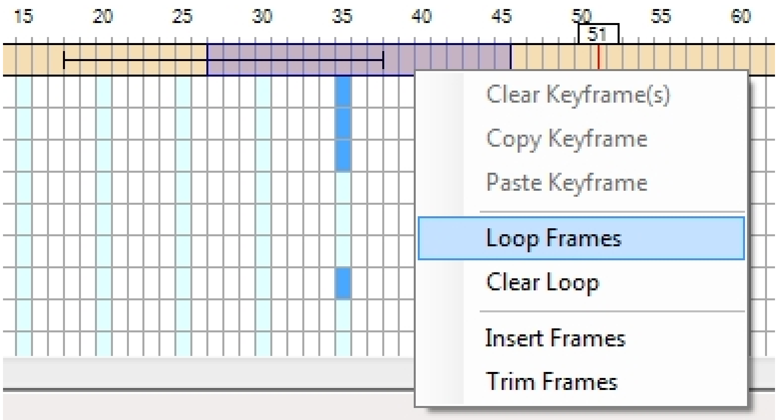
Looping an animation:

An animation can loop a single series of frames. During playback, the loop will be repeated indefinitely until the loop is keyed off, ie exited. The animation will then play to its logical end frame and finish.

To loop frames:

1. Select a series of frames in the main animation channel by click-and-drag selecting frames.
2. With a series of animations selected, right-click and select "Loop Frames"

Similarly, to clear a loop select the looped frames, right-click and select "Clear Loop"



Previewing an animation:

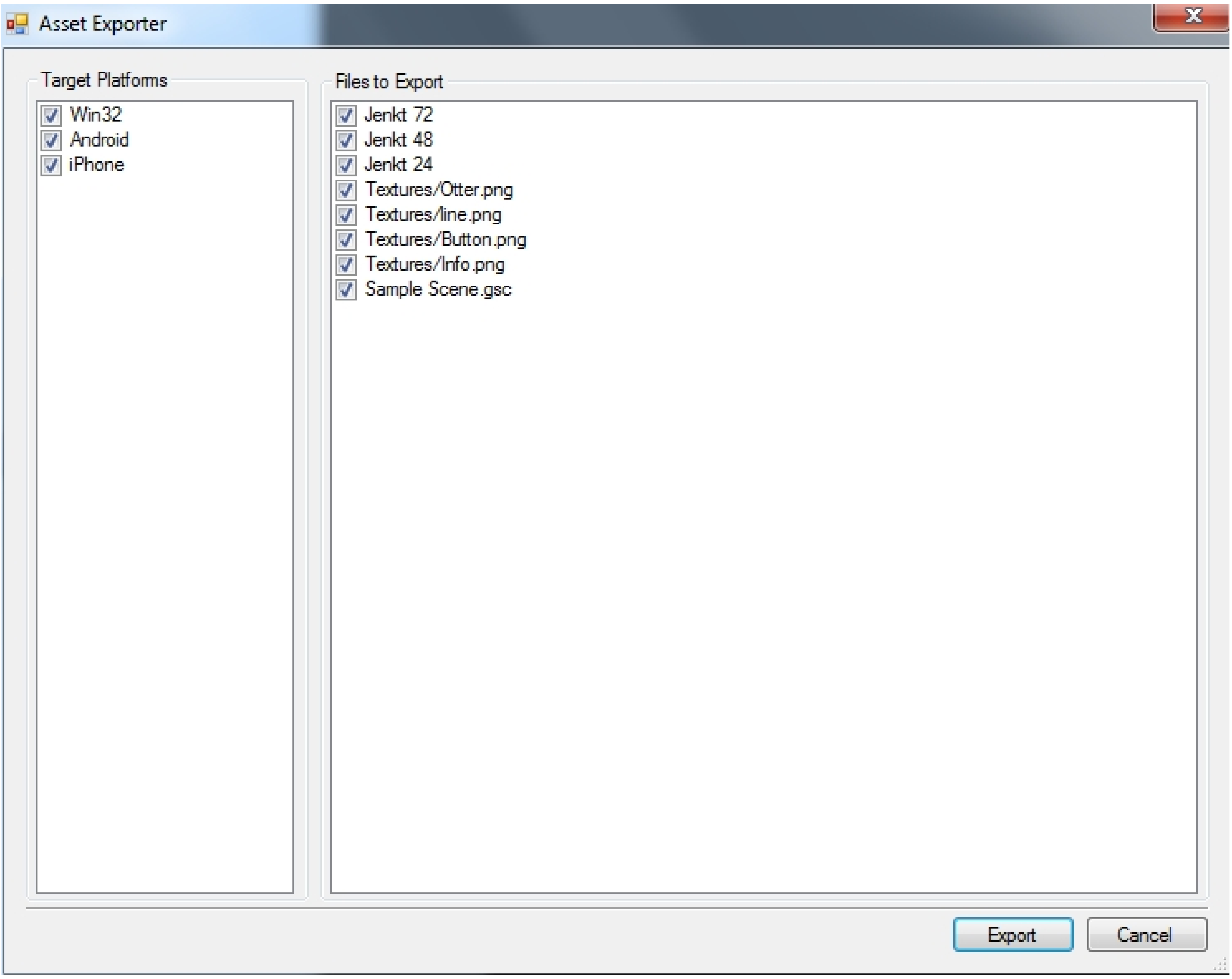
Once a control has been animated, there are two ways to preview the resulting animation: Playing it back in the Otter Editor or dragging the current frame indicator back and forth.

Using animation playback:

To the play the animation, click on the  button. The animation will automatically be played and repeated. The play button automatically changes into a  button. Hit this button to pause the animation at its current frame. If the animation is in a loop, you can exit the loop by hitting the  button.

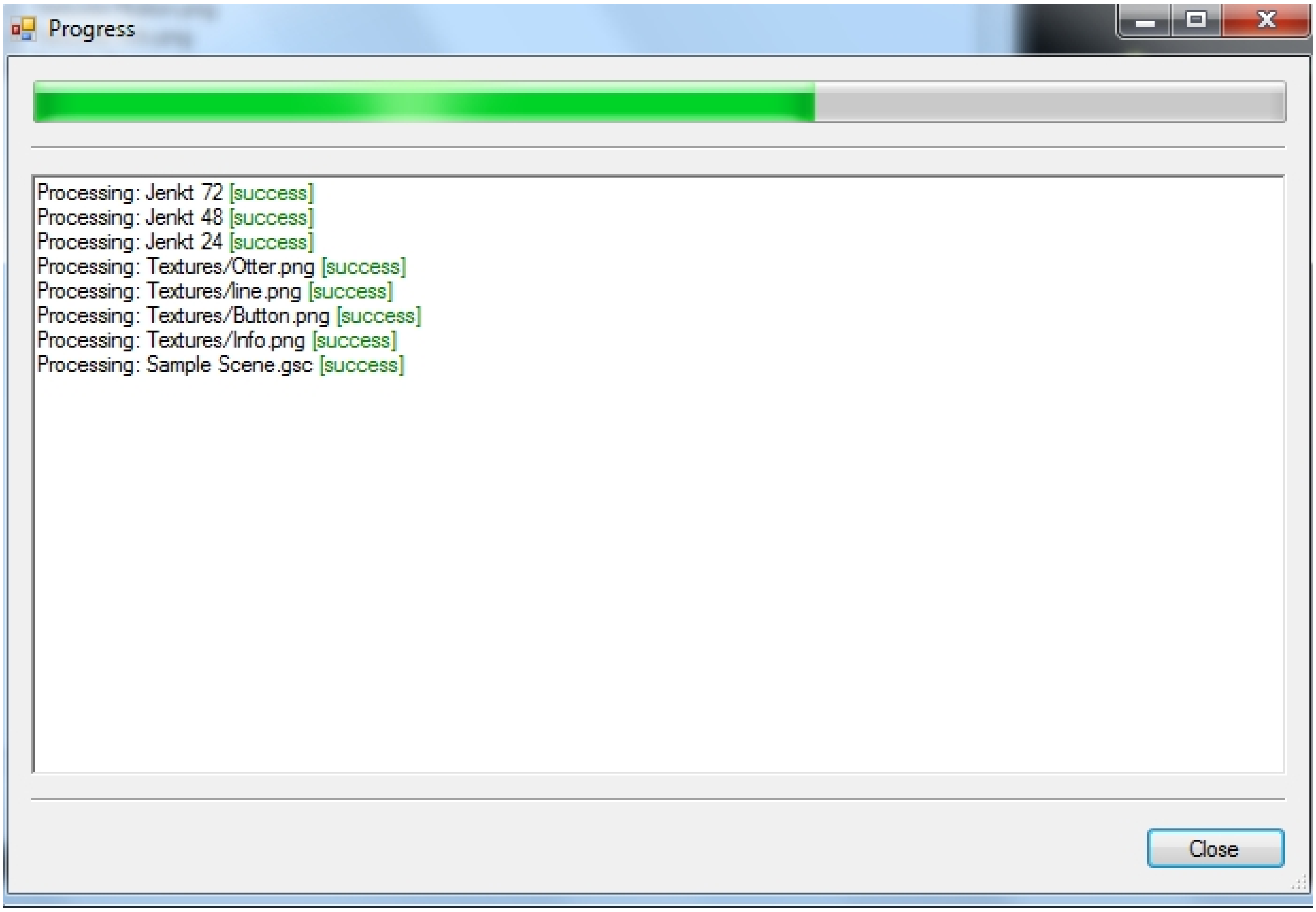
Exporting

The final step in building a UI in Otter is to export it. To export your UI, simply select the scenes you want to export, right-click on them and select `Export`. This will bring up the Asset Exporter:



The Asset Exporter lists the target platforms to export to, as well as the files it will process. Check and uncheck files/targets you want included and excluded in the export process.

Once ready, hit the `Export` button. The export progress window will come up, indicating the current export progress and status:



All files will be exported to the platform's output folder, as specified in the Platform's properties (see Setup and Installation).

Fonts

One of the most important aspects to a solid User Interface is the Font. Otter UI allows you to create fonts from a source TTF and point size, and also create and add your own custom glyphs.

This tutorial will cover the basics of adding a new font as well as adding Unicode characters and custom glyphs.

Adding a new Font

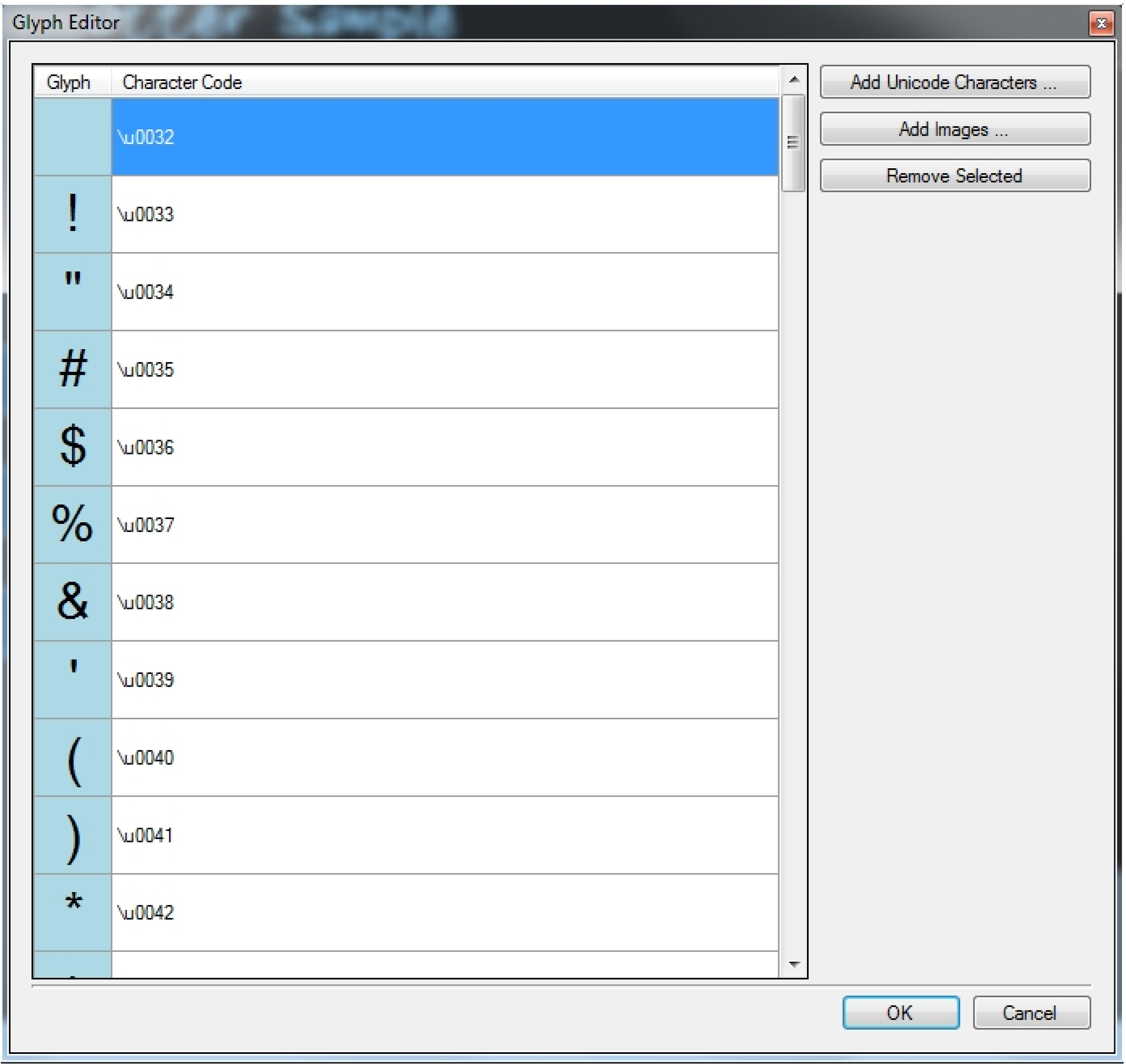
1. From the `Project` menu click `Fonts..`
2. Click on Add. A new font will be created and added to the list of fonts.
3. Select the new font
4. Select the `Name` field, and provide a descriptive name for your new font
5. Click on the `Filename` field in the property grid
6. Click on the browse button, and select the desired TTF file

Your font is now ready to be used.

Adding Unicode characters

In order to support Unicode characters, your font must have the desired characters added ahead of time. This reduces the size of the exporting font texture by using only glyphs that you specify.

1. From the `Project` menu click `Fonts..`
2. Select your desired font
3. Click on the `Glyphs` collection field
4. Click on the browse button to open the Glyph Editor:

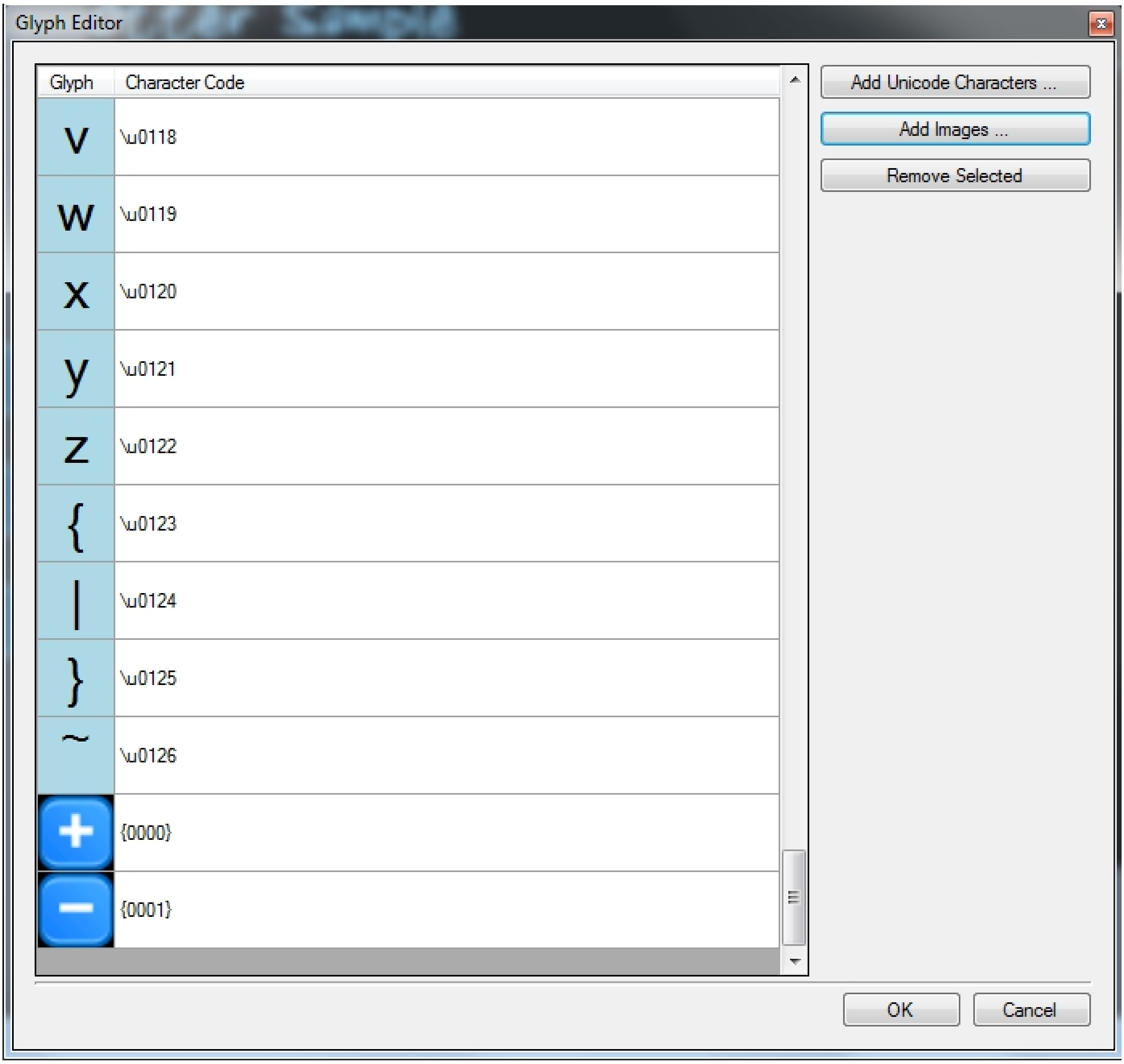


5. Click on Add Unicode Characters...
6. Enter the desired unicode characters in the textbox. Existing and duplicate characters are OK to enter in this box, as they will be trimmed and sorted accordingly.
7. Click OK to close the editor

The font can now use and render the desired characters.

Adding custom glyphs

Adding custom glyphs function much in the same way as adding unicode characters. However, on the Glyph Editor screen click on `Add Images` instead. Once added, the custom images may look something like the following:



To use the new glyphs in a text string (ex, Label Control), use the format as indicated in the Character Code field. For example:

"This is a plus button: {0000}"

The Otter UI engine will parse the {0000} and print out the desired character in its place.

To change the character code of a custom glyph, double-click on its existing character code to start editing the value. All Alpha-numeric characters are accepted, except for { and }, as they are reserved for the parse tokens. The Custom Glyph character codes are also limited to at most four characters. Any extraneous characters will be trimmed.

Actions

Actions are

Messages

Message Actions provide a very simple mechanism of communicating to the application. An example of a message might be to show a popup message on a specific frame of animation, or poll a background process' status.

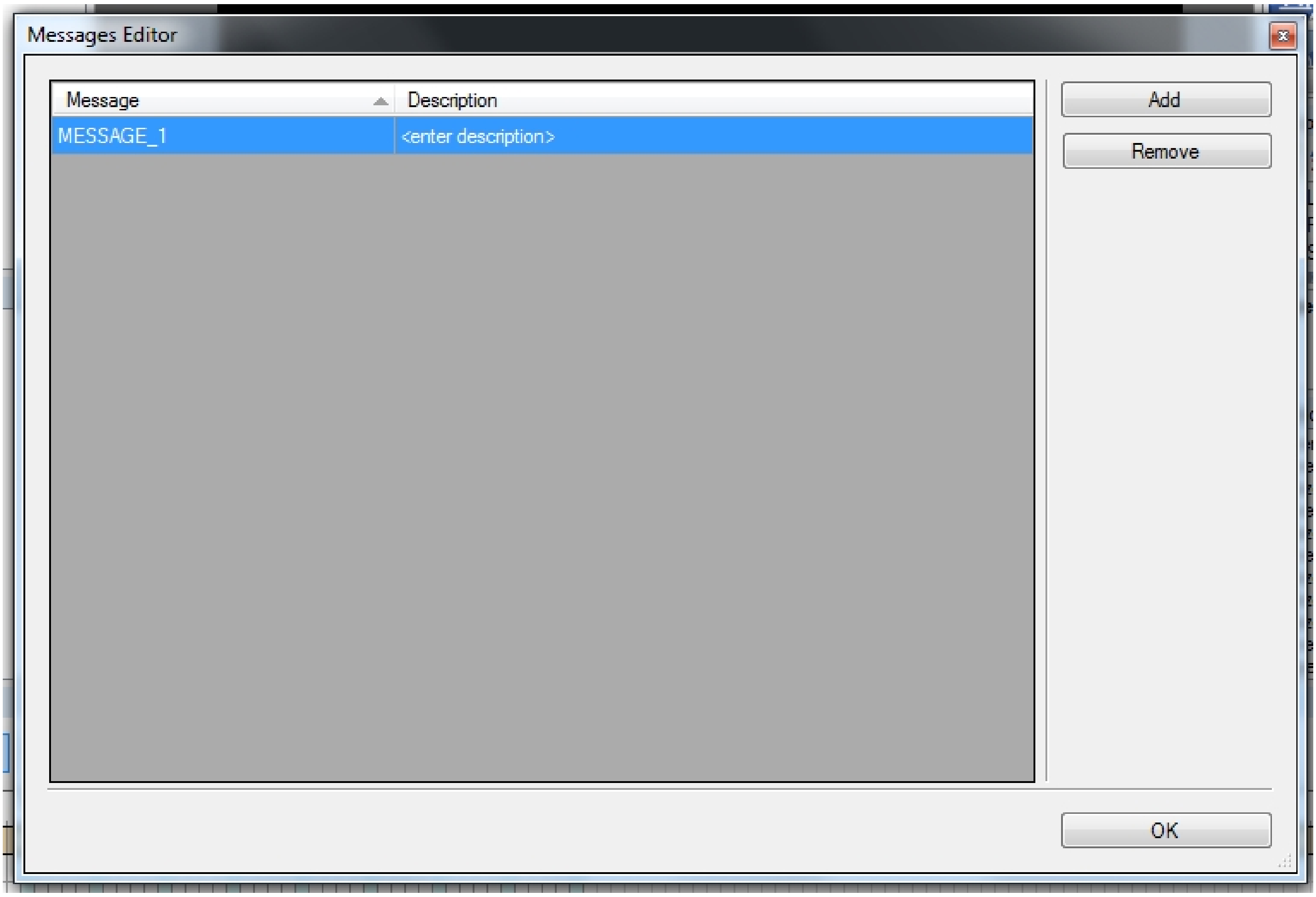
To take advantage of Message Actions you must create one or messages in the OtterUI Editor, add a Message Action to a frame of animation, and then finally process them in the application.

Create Messages in the OtterUI Editor

1. From the `Scene` menu click `Messages...`
2. Click `Add`. A new message will be created and added to the list of messages
3. Double-click on the message text (in the "`Message`" column) and change the message text to something meaningful to your application.

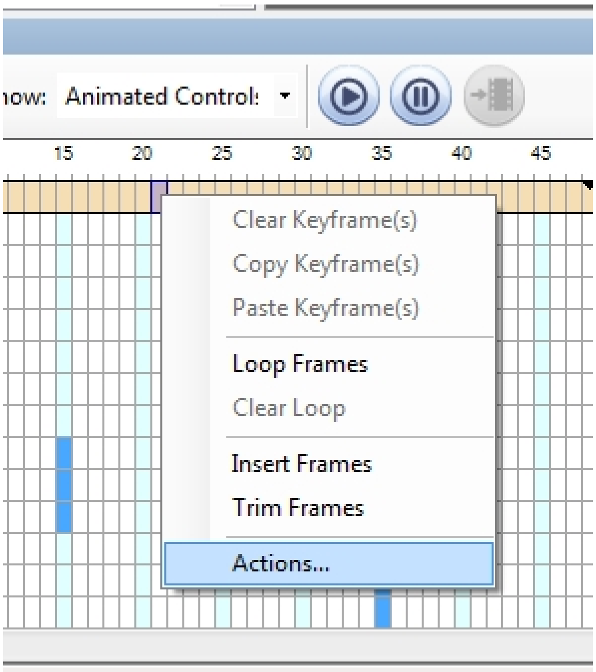
   Please Note: The message text will be reformatted to capitalize the message as well as to convert whitespace to underscores.

4. (Optional) Enter a description for the message by double-clicking in the `Description` column. Enter a meaningful description of the message. This description will not be exported, but rather as a way to describe the purpose of the message and how it should be used by other users.
5. Repeat steps 2 through 4 as needed
6. Hit `OK`



Create a Message Action

1. Right click on the main channel of an animation and select `Actions...`



2. In the `Actions Editor` select `Send Message` from the `Add Actions` dropdown
3. Select the newly created message and in the properties window select `Message` and expand the dropdown box in its value field.
4. Select the appropriate message

Process the Message

Messages are propagated to the application by way of events. Attach an event handler to your View's mOnMessage event and process the arguments accordingly. The Message Action will only be executed if the animation with the Message Action is played. The Message Action will also be executed repeatedly if it is on a frame within a loop section.

See [Event Handling](#) for further information.

Sounds

Sound Playback in OtterUI is accomplished by creating a Sound Action on an animation frame, much the same as you would create a Message Action. Unlike Message Actions, however, Sound Actions use only the ISoundSystem interface to load, unload and play sounds.

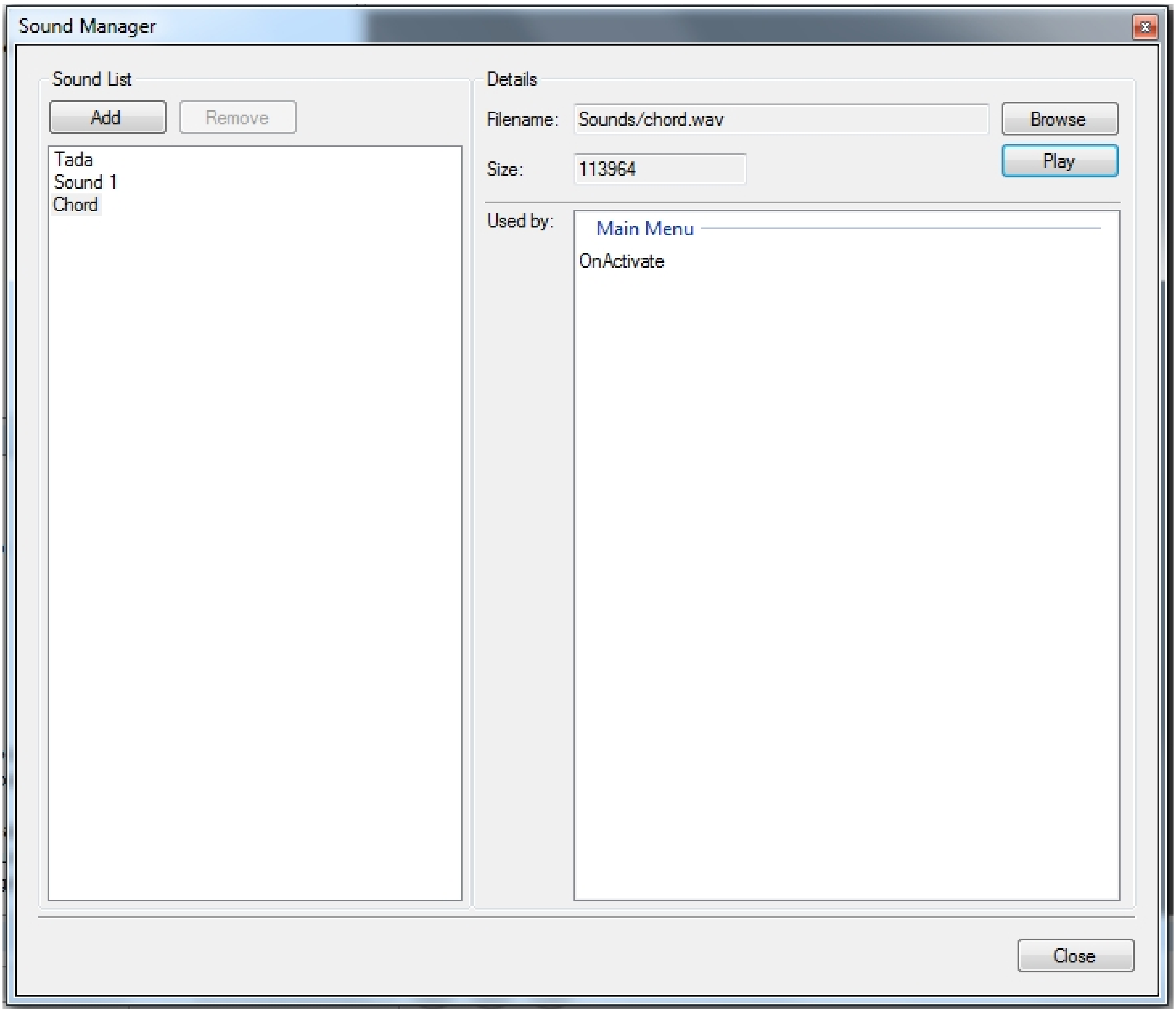
To add and play sounds in OtterUI you must create one our more sounds, add one or more Sound Actions to an animation, and finally implement the ISoundSystem interface.

Create Sounds in the OtterUI Editor

7. From the `Scene` menu click `Sounds...`
8. Click `Add`. A new sound will be created and added to the list of sounds
9. Select the new sound and hit F2 to edit the name. The sound's name must be unique
10. (Optional) Select the sound, and in Details pane and browse to a sound file.

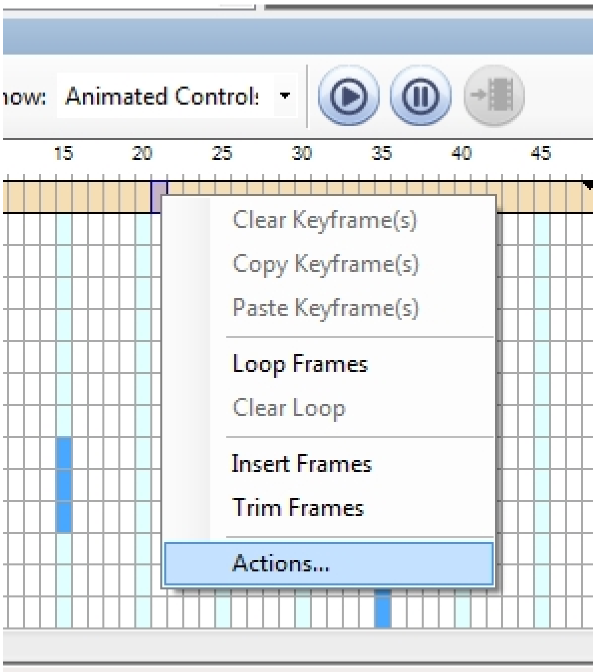
    Note: If a sound file is not specified, the sound's name will be used in identifying the sound at runtime.

11. Repeat steps 2 through 4 as needed
12. Hit `OK`



Create a Sound Action

5. Right click on the main channel of an animation and select `Actions...`



6. In the `Actions Editor` select `Play Sound` from the `Add Actions` dropdown
7. Select the newly created sound action and in the properties window select `Sound` and expand the dropdown box in its value field.
8. Select the appropriate sound to play
9. Edit the `Volume` field to set the playback volume

Implement the ISoundSystem Interface

See the Sound System tutorial for information on how to implement the ISoundSystem interface

API

Set Up

The basic setup of the Otter API is very straight-forward. Create the Otter System object, then update and finally draw it.

1. **Link in the Library**

Libraries:

Platform-specific precompiled binaries are located in `/API/lib`, and are detailed in the Supported Platforms page. Link against the appropriate library of your choice (debug or release), for your platform.

Includes:

To keep things simple, all you need to include is `Otter.h`, located in `/API/inc`

2. **Create the Otter System**

```
gSystem = new Otter::System(2 * 1024 * 1024);
gRenderer = new D3DRenderer();
gFileSys = new SampleFileSystem();
gSystemHandler = new SampleSystemHandler();

gSystem->SetRenderer(gRenderer);
gSystem->SetFileSystem(gFileSys);
gSystem->SetSystemHandler(gSystemHandler);
```

The one parameter to the System constructor indicates the amount of internal memory to reserve for the Otter System. After the system has been created, you must provide a user-implemented Renderer and FileSystem. Without these components, the Otter API will not be able to process files or render anything. See the Renderer, FileSystem, and SystemHandler tutorials for further information.

Sample implementations are provided along with the SampleApp. They are located at:

```
/SampleApp/src/SampleFileSystem.h(cpp)
/SampleApp/src/SampleRenderer.h(cpp)
/SampleApp/src/SampleSystemHandler.h(cpp)
/SampleApp/src/OGLRenderer.h(cpp)
/SampleApp/src/D3DRenderer.h(cpp)
```

3. **Update**

```
gSystem->Update(1.0f);
```

When calling this function, specify how many frames must progressed by the Otter System. A value of "1.0f" indicates that a one full frame of animation will be processed. Otter does not manage an internal framerate, so it is up to you to determine at which framerate your application runs, and build your animations accordingly.

4. **Render**

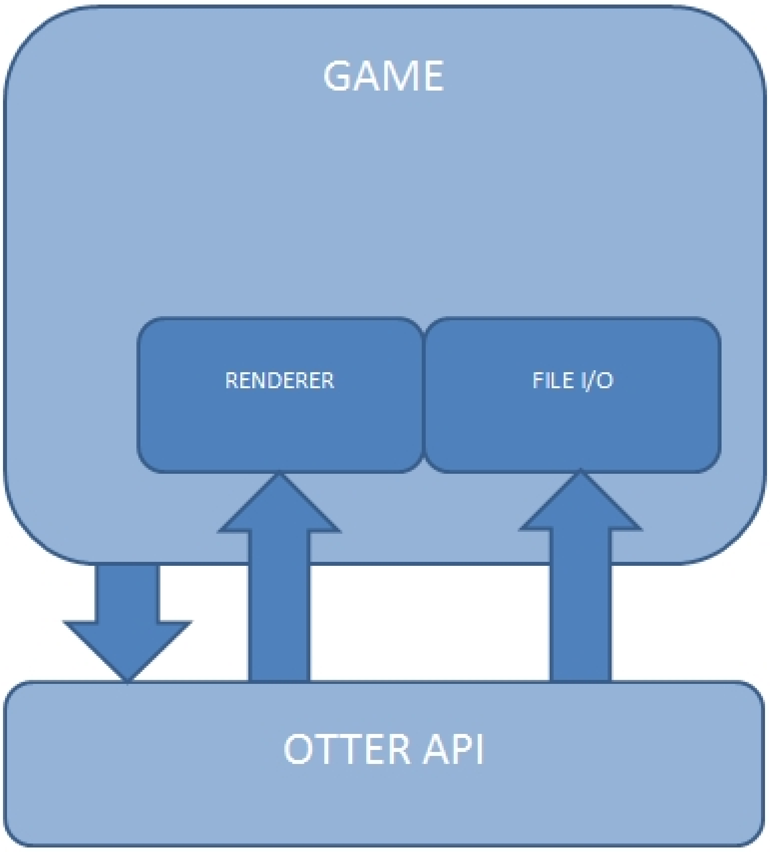
```
gSystem->Draw();
```

Calling `Draw` calls, in turn, the user-provided rendering implementation which renders the UI on screen.

Overview

The Otter API is designed to be cross-platform and engine agnostic, to allow you maximum control over the API in your product. It accomplishes this providing mechanism by which the developer may implement and maintain application-specific components: Rendering, File I/O, etc.

High Level Architecture:



As can be seen, the Otter API does not internally render the UI nor does it perform any low-level File I/O. Instead, it makes calls to user-provided implementations of these components.

Namespace:

The Otter API resides under the `Otter` namespace.

Integration

Event Handling

When the User Interface is active and running, the Otter API may frequently invoke events to communicate the state of the user interface. Examples of such events are:

View has become active
View has become inactive
An animation has started
An animation has ended
Button has been clicked

Event handling is implemented by way of functor listeners, and to handle events you must create and event handler and add it to the listener set. For the purposes of this tutorial, we will add an event handler to the View's OnActivate listener set.

Examples of event handling functions are as follows:

```
class MySampleClass
{
	/* Handles a generic event
	 */
	void MyEventHandler(void* pSender, void* pContext);

	/* Handles an animation event
	 */
	void MyAnimationHandler(void* pSender, uint32 animID);
};
```

The first parameter is always a void pointer to the sender of the event. The type of the second parameter depends on the event's template parameter, as follows:

Event:

```
Event<X> mMyEvent;
```

Method:

```
void Method(void* sender, X param);
```

1. **Create the Event Handler method**

Add a method to any class that will be responsible for handling the event that adheres to the above-mentioned signature. Ex:

```
class MySampleClass
{
	/* Handles the view's OnActivate event
	 */
	void MyOnActivateHandler(void* pSender, void* pContext);
};


/* Handles the view's OnActivate event
 */
void MySampleClass::MyOnActivateHandler(void* pSender, void* pContext)
{
	// ...
}
```

2. **Add Event Handler to the OnActivate ListenerSet**

Once the method has been declared and defined, add it to the event. You can add as many event handlers as is needed, and they will persist for as long as the parent object (ex, a parent View) exists.

```
View* pView = gSystem->GetScene(0)->GetView("MyView");
mView->mOnActivate.AddHandler(this, MySampleClass::MyOnActivateHandler);
```

3. **Remove the Event Handler when done**

When you no longer need to handle an event, you must explicitly remove it from the Event.

```
View* pView = gSystem->GetScene(0)->GetView("MyView");
mView->mOnActivate.RemoveHandler(this, MySampleClass::MyOnActivateHandler);
```

File I/O

When reading and processing files the Otter API defers to a user provided implementation of the IFileSystem interface. This object will be responsible for opening files and closing files, in addition to reading data, seeking within the file, and so on.

Creating a FileSystem

In order to implement your own file system, create a new class that inherits from Otter::IFileSystem and implements the following methods:

```
class SampleRenderer : public Otter::IFileSystem
{
	/* @brief Opens a file
	 */
	virtual void* Open(const char* szFilename, AccessFlag flags) = 0;

	/* @brief Closes the file
	 */
	virtual void Close(void* pHandle) = 0;

	/* @brief Reads data from the file.
	 */
	virtual uint32 Read(void* pHandle, uint8* data, uint32 count) = 0;

	/* @brief Writes data to the file.
	 */
	virtual uint32 Write(void* pHandle, uint8* data, uint32 count) = 0;

	/* @brief Seeks within the file.
	 */
	virtual void Seek(void* pHandle, uint32 offset, SeekFlag seekFlag) = 0;

	/* @brief Returns the size of the file
	 */
	virtual uint32 Size(void* pHandle) = 0;
};
```

Open

Opens the file at the location specified by `szFilename` and access flags specified by `flags`.

If this function succeeds, it must return a `void*` handle to the file that can be used to later identify the file and perform actions on it. The value returned by this method will be passed to all subsequent methods as the `pHandle` parameter.

If an error occurred, or if the file cannot be opened, return `NULL`.

Close

Closes the file identified by `pHandle`.

Read

Reads `count` bytes from the file, and stores the information in `data`. Returns the actual number of bytes read.

Write

Writes `count` bytes from `data` to the file. Returns the actual number of bytes written.

Seek

Seeks to a position within the file. The `offset` parameter's behavior is determined by the value of `seekFlag`.

Size

Returns the size of the file

Using the new FileSystem

Once the new filesystem has been implemented, instantiate it and assign it to the Otter System

```
gFileSys = new SampleFileSystem();
gSystem->SetFileSystem(gFileSys);
```

Renderer

To render the UI, you must provide a renderer that will receive data from the Otter API and present it on screen. The Otter API, at render-time, collects the drawing information of all the on-screen elements and organizes them into batches. These batches are then sent the user-provided renderer to be drawn.

In addition to sending vertices and draw batches, the renderer is also responsible for loading and unloading textures.

Creating a Renderer

In order to implement your own renderer, create a new class that inherits from Otter::IRenderer and implements the following methods:

```
class SampleRenderer : public Otter::IRenderer
{
	/* Loads a texture with the specified id and path
	 */
	virtual void OnLoadTexture(int textureID, const char* szPath);

	/* Unloads a texture with the specified id
	 */
	virtual void OnUnloadTexture(int textureID);

	/* Called when a drawing pass has begun
	 */
	virtual void OnDrawBegin();
```

```
        /* Called when a drawing pass has ended
         */
        virtual void OnDrawEnd();

        /* Commits a vertex buffer for rendering
         */
        virtual void OnCommitVertexBuffer(const Otter::GUIVertex* pVertices, uint32 numVertic

        /* Draws primitives on screen
         */
        virtual void OnDrawBatch(const Otter::DrawBatch& batch);
};
```

These methods will be called by the Otter API during render-time, and it is up to you to ensure they are implemented correctly in your application. A quick overview of the methods are as follows:

OnLoadTexture

This method is called when the Otter API requires a texture to be loaded. The path of the texture is provided as well as its internal ID. After this function completes, the texture will always be referenced by its ID. It is recommended to create a fast mapping between the internal texture ID and the actual texture reference.

OnUnloadTexture

This method is called when the Otter API no longer requires the texture, as specified by the provided texture ID.

OnDrawBegin

This method is called once per frame, and indicates that the Otter API is about to send render information. In this function you would set up the appropriate render states, projection matrices, and so on.

OnDrawEnd

This method is called once all the render information has been sent, and thus concludes rendering for the current frame.

OnCommitVertexBuffer

This method provides a linear vertex buffer that will be used for incoming draw batches. In this function you would update your internal vertex buffers appropriately.

OnDrawBatch

This method may be called multiple times a frame. It draws elements in batches, sorted by texture, transform, and render flags.

Using the new Renderer

Once the new renderer has been implemented, instantiate it and assign it to the Otter System

```
gRenderer = new D3DRenderer();
gSystem->SetRenderer(gRenderer);
```

Renderer

Similar to the Renderer and File System components, the OtterUI API relies on the ISoundSystem interface to load, play and unload sounds.

Creating a Sound System

In order to implement your own Sound System, create a new class that inherits from Otter::ISoundSystem and implements the following methods:

```
class SampleFileSystem : public Otter::ISoundSystem
{
       /* Loads a sound with the specified id and path
              */
       virtual void OnLoadSound(uint32 soundID, const char* szSoundPath);

       /* Unloads a sound with the specified id
              */
       virtual void OnUnloadSound(uint32 soundID);

       /* Plays a sound at the specified volume
              */
       virtual void OnPlaySound(uint32 soundID, float volume);
};
```

These methods will be called by the Otter API at run-time, and it is up to you to ensure they are implemented correctly in your application. A quick overview of the methods are as follows:

OnLoadSound

This method is called when a sound needs to be loaded for a particular view. The sound path as well as its internal ID is provided. Once this function completes, the sound will always be referenced by the internal soundID. It is recommended to create a fast mapping between the internal soundID and actual sound reference.

Note: szSoundPath may refer to the path of the sound file, as it was known during export, or the sound's name if a path was not provided. This is useful if your sound system operates off string identifiers and not actual file paths.

OnUnloadTexture

This method is called when the Otter API no longer requires the texture, as specified by the provided texture ID.

OnDrawBegin

This method is called once per frame, and indicates that the Otter API is about to send render information. In this function you would set up the appropriate render states, projection matrices, and so on.

OnDrawEnd

This method is called once all the render information has been sent, and thus concludes rendering for the current frame.

OnCommitVertexBuffer

This method provides a linear vertex buffer that will be used for incoming draw batches. In this function you would update your internal vertex buffers appropriately.

OnDrawBatch

This method may be called multiple times a frame. It draws elements in batches, sorted by texture, transform, and render flags.

Using the new Renderer

Once the new renderer has been implemented, instantiate it and assign it to the Otter System

```
gRenderer = new D3DRenderer();
gSystem->SetRenderer(gRenderer);
```

Input

Some of the controls in Otter UI require mouse or touch events to allow the user to interact with them. If these controls are used, your application must capture and send these events to the Otter API, which will then in turn process the events appropriately.

To send touch events simply call the following methods on the Otter System object:

```
namespace Otter
{
	/* Main object that manages and maintains all things game-gui related.
	 */
	class System
	{
		// ...

	public:
		/* Points (touches/mouse/etc) were pressed down
		 */
		void OnPointsDown(sint32* pointPairs, sint32 numPairs);

		/* Points (touches/mouse/etc) were released
		 */
		void OnPointsUp(sint32* pointPairs, sint32 numPairs);

		/* Points (touches/mouse/etc) were moved.
		 */
		void OnPointsMove(sint32* pointPairs, sint32 numPairs);

		// ...
	};
}
```

Each method accepts a set of 2-tuples as signed integers and the number of 2-tuples sent. Each 2-tuple is interpreted as an X/Y coordinate pair.

For example, if `numPairs` == 2, then the expected array length of `pointPairs` is 4. The `pointPairs` data is then interpreted as: `{X1, Y1, X2, Y2}`

OnPointsDown

Notifies that `numPairs` of touch points were just pressed down, and are stored in `pointPairs` as a set of 2-tuples. Each 2-tuple represents an absolute location on screen.

OnPointsUp

Notifies that `numPairs` of touch points were just released, and are stored in `pointPairs` as a set of 2-tuples. Each 2-tuple represents an absolute location on screen.

OnPointsMoved

Notifies that `numPairs` of touch points were just moved, and are stored in `pointPairs` as a set of 2-tuples. Each 2-tuple represents an absolute location on screen.

Reference

Button

The Button class responds to mouse/touch inputs and, if clicked, sends an OnClick event to any listeners. The Button control has two states: Up and Down. When it is pressed, it changes to the down state and renders the appropriate the down-state texture as specified in the Otter Editor. When input is released, it reverts back to its normal, or up, state and associated texture.

Methods:

```
Button(Scene* pScene, Control* pParent, const ButtonData* pButtonData);
```

Description:

Constructor

Parameters:

Scene*	pScene	*Parent scene*
Control*	pParentControl	*Parent control*
const ButtonData*	pButtonData	*Button's internal data*

Returns:

None

```
virtual ~Button(void);
```

Description:

Virtual Constructor

Parameters:

None

Returns:

None

Events:

```
Event mOnClick;
```

Sent whenever the button has been clicked.

Handler Parameters:

pSender	*Reference to Button that sent the OnClick event*
pContext	*NULL*

Control

The Control class is a base class from which all other concrete controls derive from, and is the basic building block of the Otter UI.

Methods:

```
Control(Scene* pScene, Control* pParentControl, const ControlData* pControlData);
```

Description:

Constructor

Parameters:

`Scene*`	`pScene`	*Parent scene*
`Control*`	`pParentControl`	*Parent control*
`const ControlData*`	`pControlData`	*Control's internal base data*

Returns:

None

```
virtual ~Control(void);
```

Description:

Virtual destructor

Parameters:

None

Returns:

None

```
void Enable(bool bEnable) { mEnabled = bEnable; }
```

Description:

Enables / Disables the control. If disabled, the control will not draw, update or process input.

Parameters:

`bool`	`bEnable`	*Boolean flag to disable or disable the control*

Returns:

None

```
bool IsEnabled() { return mEnabled; }
```

Description:

Returns whether or not the control is enabled.

Parameters:

None

Returns:

`true` if enabled, `false` if otherwise.

```
const char* GetName();
```

Description:

Retrieves the control's name, as specified in the Otter Editor.

Parameters:

None

Returns:

NULL-terminated string with the control's name.

```
uint32 GetID();
```

Description:

Retrieves the control's unique ID.

Parameters:

None

Returns:

Control's ID as an unsigned integer

```
Scene* GetScene() { return mScene; }
```

Description:

Retrieves the control's parent scene

Parameters:

None

Returns:

Reference to the control's parent scene object.

```
const ControlData* GetData() { return mControlData; }
```

Description:

Retrieves the control's internal base data.

Parameters:

None

Returns:

Reference to the control's internal base data

```
const VectorMath::Vector2& GetPosition();
```

Description:

Retrieves the control's position.

Parameters:

None

Returns:

Vector2 containing the control's position, relative to its parent.

```
void SetPosition(const VectorMath::Vector2& position);
```

Description:

Sets the control's position, relative to its parent.

Parameters:

`const VectorMath::Vector2&`	`position`	*Control's new position, relative to the parent.*

Returns:

None

```
const VectorMath::Vector2& GetCenter();
```

Description:

Retrieves the control's center relative to the control's unrotated top-left corner. For example, a center of (10, 20) indicates that the control's center is 10 units to the right and 20 units below the top-left corner of the control.

Parameters:

None

Returns:

Vector2 containing the control's center.

```
void SetCenter(const VectorMath::Vector2& center);
```

Description:

Sets the control's center relative to the control's unrotated top-left corner.

Parameters:

`const VectorMath::Vector2&`	`center`	*Control's new center, relative to the unrotated top-left*

Returns:

None

```
const VectorMath::Vector2& GetDimensions();
```

Description:

Retrieves the control's dimensions (width and height).

Parameters:

None

Returns:

Vector2 containing the control's dimensions.

```
void SetDimensions(const VectorMath::Vector2& dimensions);
```

Description:

Sets the control's dimensions (width and height)

Parameters:

`const VectorMath::Vector2&`	`dimensions`	*Control's new dimensions*

Returns:

None

```
float GetRotation();
```

Description:

Returns the control's rotation in degrees.

Parameters:

None

Returns:

Control's rotation in degrees.

```
void SetRotation(float rotation);
```

Description:

Sets the control's rotation in degrees

Parameters:

`float`	`rotation`	*Control's new rotation in degrees*

Returns:

None

```
const VectorMath::Matrix4& GetTransform();
```

Description:

Retrieves the control's full transformation matrix.

Parameters:

None

Returns:

Matrix4 representing the control's full transform.

```
Control* GetControl(const char* szControlName);
```

Description:

Retrieves a child control by name

Parameters:

`const char*`	`szControlName`	*Child control's name*

Returns:

Reference to the control with the specified name, if found. `NULL` if otherwise.

```
Control* GetControl(uint32 id);
```

Description:

Retrieves a child control by ID.

Parameters:

`uint32`	`id`	*Child control's ID*

Returns:

Reference to the control with the specified ID, if found. `NULL` if otherwise.

```
Control* GetControl(sint32 x, sint32 y, sint32* lx = 0, sint32* ly = 0);
```

Description:

Retrieves a control by location. Performs a hit-test on the control and all of its children, returning the top-most control that contains the point.

Parameters:

`sint32`	`x`	*Screen X*
`sint32`	`y`	*Screen Y*
`sint32*`	`lx`	*[OUT] If control contains the point, this will be set to the control*
`sint32*`	`ly`	*[OUT] If control contains the point, this will be set to the control*

Returns:

Reference to the top-most child control that contains the point. `NULL` if otherwise.

Label

The Label control is solely responsible for rendering text. In addition to being able to change text color, it can also be aligned vertically and/or horizontally, as well as scaled.

Methods:

```
Label(Scene* pScene, Control* pParent, const LabelData* pLabelData);
```

Description:

Constructor.

Parameters:

`Scene*`	`pScene`	*Parent scene*
`Control*`	`pParentControl`	*Parent control*
`const LabelData*`	`pLabelData`	*Label's internal data*

Returns:

None

```
virtual ~Label(void);
```

Description:

Virtual destructor.

Parameters:

None

Returns:

None

```
void SetText(const String& text)
```

Description:

Sets the label's text.

Parameters:

`const String&`	`text`	*Label's new text value*

Returns:

None

```
void SetText(const char* szText)
```

Description:

Sets the label's text.

Parameters:

`const char*`	`szText`	*Label's new text value*

Returns:

None

```
void SetColor(uint32 color);
```

Description:

Sets the label's color. The format of the color value is dependent on your platform and renderer, ex A8R8G8B8 / R8G8B8A8 / etc.

Parameters:

`uint32`	`color`	*Unsigned 32-bit color value*

Returns:

None

```
uint32 GetColor();
```

Description:

Retrieves the label's current color value.

Parameters:

None

Returns:

Current label color as an unsigned 32-bit value.

```
void SetScale(float scaleX, float scaleY);
```

Description:

Sets the font's X and Y scale. This does not reload or regenerate the font; it simply increases the scale of the rendered glyphs appropriately.

Parameters:

`float`	`scaleX`	*New X font scale*
`float`	`scaleY`	*New Y font scale*

Returns:

None

Scene

The Scene object maintains and manages a set of views.

Methods:

```
Scene(System* pSystem, Renderer* pRenderer, const SceneData* pSceneData);
```

Description:

Constructor

Parameters:

`System*`	`pSystem`	*Otter System object*
`Renderer*`	`pRenderer`	*Internal Otter Renderer*
`const SceneData*`	`pSceneData`	*Scene's internal data*

Returns:

None

```
~Scene(void);
```

Description:

Destructor

Parameters:

None

Returns:

None

```
const SceneData* GetData() { return mSceneData; }
```

Description:

Retrieves the scene's internal data

Parameters:

None

Returns:

Reference to the scene's internal data

```
uint32 GetViewCount();
```

Description:

Retrieves the total number of views in this scene

Parameters:

None

Returns:

Total number of views in this scene

```
View* GetView(const char* szName);
```

Description:

Retrieves a view by name

Parameters:

`const char*`	`szName`	*Views name*

Returns:

Reference to the View with the specified name. `NULL` if otherwise.

```
View* GetView(uint32 index);
```

Description:

Retrieves a view by 0-based index. Index must be in the range `[0, GetViewCount() - 1]`.

Parameters:

`uint32`	`index`	*0-based index of the view*

Returns:

Reference to the View at the specified index. `NULL` if otherwise.

```
void ActivateView(const char* szName);
```

Description:

Activates a view, and automatically plays its `OnActivate` animation.

Parameters:

`const char*`	`szName`	*Name of the view to activate.*

Returns:

None

```
void DeactivateView(const char* szName);
```

Description:

Deactivates a view, and automatically plays its OnDeactivate animation.

Parameters:

None

Returns:

None

```
View* GetActiveView(const char* szName);
```

Description:

Retrieves an active view by name. If the name refers to a view that is either not in the scene or is not active, the function will return `NULL`.

Parameters:

`const char*`	`szName`	*Name of the view*

Returns:

Reference to an active View with the specified name. If the View is not active, returns `NULL`.

```
void LoadViewTextures(const char* szName);
```

Description:

Loads all of the textures of the view with the specified name. This is useful to preload view textures before the view may become active.

Parameters:

`const char*`	`szName`	*Name of the view*

Returns:

None

```
void UnloadViewTextures(const char* szName);
```

Description:

Unloads all of the textures of the view with the specified name.

Parameters:

`const char*`	`szName`	*Name of the view*

Returns:

None

```
Font* GetFont(uint32 fontID);
```

Description:

Retrieves a font by ID

Parameters:

`uint32`	`fontID`	*Font's ID*

Returns:

Reference to the font with the specified ID. If the font could not be found, returns `NULL`.

```
System* GetSystem() { return mSystem; }
```

Description:

Retrieves the Otter System object

Parameters:

None

Returns:

Reference to the Otter System object.

```
void SetResolution(uint32 width, uint32 height);
```

Description:

Sets the internal resolution for this scene. This is mostly used for anchored controls to accommodate varying resolutions.

Parameters:

`uint32`	`width`	*New width*
`uint32`	`height`	*New height*

Returns:

None

Sprite

The Sprite is the most primitive of controls - it simply renders a texture on screen. It does not respond to input events.

Methods:

```
Sprite(Scene* pScene, Control* pParent, const SpriteData* pSpriteData);
```

Description:

Constructor

Parameters:

`Scene*`	`pScene`	*Parent scene*
`Control*`	`pParentControl`	*Parent control*
`const SpriteData*`	`pSpriteData`	*Sprite's internal data*

Returns:

None

```
virtual ~Sprite(void);
```

Description:

Virtual destructor

Parameters:

None

Returns:

None

```
void SetColor(uint32 color);
```

Description:

Sets the sprite's color. The format of the color value is dependent on your platform and renderer, ex A8R8G8B8 / R8G8B8A8 / etc.

Parameters:

`uint32`	`color`	*32-bit color value*

Returns:

None

```
uint32 GetColor();
```

Description:

Retrieves the sprite's current color value

Parameters:

None

Returns:

Current sprite color as an unsigned 32-bit value.

System

The Otter System is responsible for running the Otter UI. It manages memory, scenes, input, and so on.

Methods:

```
System(int memorySize);
```

Description:

Constructor - initializes the Otter System with the specified memory size, measured in bytes. The constructor will allocate `memorySize` bytes from main memory to use for the lifetime of the Otter System. All scenes, views, and so on will be allocated from within this block.

Parameters:

`int`	`memorySize`	*Desired memory size, in bytes*

Returns:

None

```
System(uint8* pMemoryBuffer, int memorySize);
```

Description:

Constructor - initializes the Otter System with a pre-allocated block of memory of a specific size. All scenes, views, and so on will be allocated from within this block.

Parameters:

`uint8*`	`pMemoryBuffer`	*Pre-allocated memory buffer*
`int`	`memorySize`	*Size of the pre-allocated memory buffer, in bytes*

Returns:

None

```
~System(void);
```

Description:

Destructor

Parameters:

None

Returns:

None

```
void SetFileSystem(IFileSystem* pFileSystem);
```

Description:

Sets the File System object. The Otter System will use this object to read from and write to files as necessary.

Parameters:

`IFileSystem*`	`pFileSystem`	*Reference to the user implementation of the IFileSys*

Returns:

None

```
void SetRenderer(IRenderer* pRenderer);
```

Description:

Sets the Renderer object. The Otter System will use this object when rendering.

Parameters:

`IRenderer*`	`pRenderer`	*Reference to the user implementation of the IRender*

Returns:

None

```
void SetSoundSystem(ISoundSystem* pSoundSys);
```

Description:

Sets the SoundSystem object. The Otter System will use this object when loading, unloading and playing sounds.

Parameters:

`ISoundSystem*`	`pSoundSys`	*Reference to the user implementation of the ISoundS*

Returns:

None

```
void OnPointsDown(Point* points, sint32 numPoints);
```

Description:

Notifies the Otter System that a set of points have just been pressed down onto the UI. Points are assumed to be in absolute screen coordinates.

Parameters:

`Point*`	`points`	*Linear array of points*
`sint32`	`numPoints`	*Number of points in the* `points` *array*

Returns:

None

```
void OnPointsUp(Point* points, sint32 numPoints);
```

Description:

Notifies the Otter System that a set of points have just been released from the UI. Points are assumed to be in absolute screen coordinates.

Parameters:

`Point*`	`points`	*Linear array of points*
`sint32`	`numPoints`	*Number of points in the* `points` *array*

Returns:

None

```
void OnPointsMove(Point* points, sint32 numPoints);
```

Description:

Notifies the Otter System that a set of points have just been moved on the UI. Points are assumed to be in absolute screen coordinates.

Parameters:

`Point*`	`points`	*Linear array of points*
`sint32`	`numPoints`	*Number of points in the* `points` *array*

Returns:

None

```
void SetResolution(uint32 width, uint32 height);
```

Description:

Sets the Otter System's internal resolution. This is mostly used for anchored controls to accommodate varying resolutions.

Parameters:

`uint32`	`width`	*New width*
`uint32`	`height`	*New height*

Returns:

None

```
Scene* LoadScene(const char* szPath);
```

Description:

Loads a scene at the given path

Parameters:

`const char*`	`szPath`	*File path to the scene*

Returns:

Reference to the Scene object if loaded. `NULL` if failed.

```
Scene* LoadScene(const uint8* pBuffer, uint32 bufferSize, bool bCopy);
```

Description:

Loads a scene from an in-memory buffer.

Parameters:

`const uint8*`	`pBuffer`	*In-memory buffer containing the scene data*
`uint32`	`bufferSize`	*Length of `pBuffer`, measured in bytes*
`bool`	`bCopy`	*If true, will make an internal copy of the scene data. the data.*

Returns:

None

```
void UnloadAllScenes();
```

Description:

Unloads all scenes.

Parameters:

None

Returns:

None

```
Scene* GetScene(uint32 index);
```

Description:

Retrieves a scene by index. Index must be in the range `[0, GetSceneCount() - 1]`.

Parameters:

uint32	index	*Index of the scene to retrieve*

Returns:

Reference to the Scene* object if found, `NULL` if otherwise.

```
uint32 GetSceneCount();
```

Description:

Retrieves the number of scene currently loaded.

Parameters:

None

Returns:

The number of scenes current loaded.

```
void Draw();
```

Description:

Draws all scenes with active views.

Parameters:

None

Returns:

None

```
void Update(float frameDelta);
```

Description:

Updates the Otter UI with the specified frame delta. A frame delta of 1.0f indicates one full frame of Otter UI animation.

Parameters:

float	frameDelta	*Number of frames to progress in the Otter UI. Must*

Returns:

None

View

A view is simply a collection of controls and animations. By default, it plays two required animations: OnActivate and OnDeactivate when the view becomes active and inactive respectively.

Methods:

```
View(Scene* pScene, const ViewData* pViewData);
```

Description:

Constructor

Parameters:

Scene*	pScene	*Parent scene*
const ViewData*	pViewData	*View's internal data*

Returns:

None

```
virtual ~View(void);
```

Description:

Virtual destructor

Parameters:

None

Returns:

None

```
uint32 PlayAnimation(const char* szName, uint32 startFrame = 0, bool bReverse = false);
```

Description:

Plays an animation of the specified name.

Parameters:

`const char*`	`szName`	*Animation name*
`uint32`	`startFrame`	*Animation's starting frame*
`bool`	`bReverse`	*If true, plays the animation in reverse*

Returns:

If successful, returns the ID of the animation.

```
bool StopAnimation(uint32 animID);
```

Description:

Stops an animation by ID

Parameters:

`uint32`	`animID`	*Animation ID, as returned by* `PlayAnimation`

Returns:

`true` if successful, `false` if otherwise

```
void KeyOffAnimation(uint32 animID);
```

Description:

Flags an animation to continue through a looping section if present, effectively keeping the animation from repeating. This will cause the animation to play through to its logical end and finish.

Parameters:

`uint32`	`animID`	*Animation ID, as returned by* `PlayAnimation`

Returns:

None

```
void Draw(Renderer* pRenderer);
```

Description:

Draws the view with the internal Otter Renderer.

Parameters:

`Renderer*`	`pRenderer`	*Reference to the internal Otter Renderer*

Returns:

None

```
void Update(float frameDelta);
```

Description:

Updates the view with the specified frame delta. A frame delta of 1.0f indicates one full frame of Otter UI animation.

Parameters:

`float`	`frameDelta`	*Number of frames to progress in the Otter UI. Must*

Returns:

None

```
void BringToFront(Control* pControl);
```

Description:

Brings the specified control to the front, so that it is drawn and updated above all other controls

Parameters:

`Control*`	`pControl`	*Reference to the control that will be brought to the fr*

Returns:

None

```
void SendToBack(Control* pControl);
```

Description:

Sends the specified control to the back, so that is drawn and updated below all other controls

Parameters:

`Control*`	`pControl`	*Reference to the control that will be brought to the fr*

Returns:

None

```
void OnPointsDown(sint32* pointPairs, sint32 numPairs);
```

Description:

Notifies the Otter View that a set of points have just been pressed down onto the UI. Points are assumed to be in absolute screen coordinates.

Parameters:

`sint32*`	`pointPairs`	*Linear array of points in the format [X1, Y1, X2, Y2, .* `numPairs * 2`
`sint32`	`numPairs`	*Number of (x,y) points in the* `pointPairs` *array*

Returns:

None

```
void OnPointsUp(sint32* pointPairs, sint32 numPairs);
```

Description:

Notifies the Otter View that a set of points have just been released from the UI. Points are assumed to be in absolute screen coordinates.

Parameters:

`sint32*`	`pointPairs`	*Linear array of points in the format [X1, Y1, X2, Y2, .* `numPairs * 2`
`sint32`	`numPairs`	*Number of (x,y) points in the* `pointPairs` *array*

Returns:

None

```
void OnPointsMove(sint32* pointPairs, sint32 numPairs);
```

Description:

Notifies the Otter View that a set of points have just been moved on the UI. Points are assumed to be in absolute screen coordinates.

Parameters:

`sint32*`	`pointPairs`	*Linear array of points in the format [X1, Y1, X2, Y2, .* `numPairs * 2`
`sint32`	`numPairs`	*Number of (x,y) points in the* `pointPairs` *array*

Returns:

None

VectorMath

Vector2

The Vector2 class represents a simple 2D vector, consisting of only the X and Y coordinates.

Methods:

```
Vector2()
```

Description:

Default constructor. Initializes the vector to (0, 0)

Parameters:

None

Returns:

None

```
Vector2(float x, float y)
```

Description:

Default constructor. Initializes the vector with the specified (X, Y) 2D coordinates.

Parameters:

float	x	*X Coordinate*
float	y	*Y Coordinate*

Returns:

None

Vector3

The Vector3 class represents a simple 3D vector consisting of X, Y and Z coordinates.

Methods:

```
Vector3()
```

Description:

Default constructor. Initializes the vector to (0, 0, 0)

Parameters:

None

Returns:

None

```
Vector3(float x, float y, float z)
```

Description:

Default constructor. Initializes the vector with the specified (X, Y, Z) 3D coordinates.

Parameters:

float	x	*X Coordinate*
float	y	*Y Coordinate*
float	z	*Z Coordinate*

Returns:

None

Vector4

The Vector4 class represents a simple 4D vector consisting of X, Y, Z and W coordinates.

Methods:

```
Vector4()
```

Description:

Default constructor. Initializes the vector to (0, 0, 0, 0)

Parameters:

None

Returns:

None

```
Vector3(float x, float y, float z, float w)
```

Description:

Default constructor. Initializes the vector with the specified (X, Y, Z, W) 4D coordinates.

Parameters:

float	x	*X Coordinate*
float	y	*Y Coordinate*
float	z	*Z Coordinate*
float	w	*W Coordinate*

Returns:

None

Matrix4

The Matrix4 represents a three-dimensions transformation matrix for a 3D vector, including translation.

The Matrix's data is stored as an array of 16 floats, and is organized in row-major order. Entry of row 'r' and column 'c' is stored at index = c + (4 * r).

Methods:

```
Matrix4(void);
```

Description:

Default constructor. Initializes the matrix to the identity matrix.

Parameters:

None

Returns:

None

```
Matrix4 (     float fM00, float fM01, float fM02, float fM03,
              float fM10, float fM11, float fM12, float fM13,
              float fM20, float fM21, float fM22, float fM23,
              float fM30, float fM31, float fM32, float fM33);
```

Description:

Constructor - initializes the matrix with the specified components.

Parameters:

float	*fMRC*	*Matrix entry at row R and column C*

Returns:

None

```
~Matrix4(void);
```

Description:

Destructor

Parameters:

None

Returns:

None

```
Matrix4 operator* (const Matrix4& rkM) const;
```

Description:

Overloaded multiplication operator. Multiplies two matrices together, and returns the result. Operation is from left to right, ie A * B is the transformation of A followed by B.

Parameters:

`const Matrix4&`	`rkM`	*Right hand side operator* argument*

Returns:

Matrix4 that is the result of the transformation of `(*this)` followed by `rkM`

```
Matrix4 Inverse() const;
```

Description:

Returns the inverse of the matrix

Parameters:

None

Returns:

Matrix4 that is the inverse of the current matrix

```
Matrix4 Transpose() const;
```

Description:

Returns the transpose of the matrix

Parameters:

None

Returns:

Matrix4 that is the transpose of the current matrix

```
Vector4 operator*(const Vector4& rkV) const;
```

Description:

Transforms the provided vector with the current matrix, and returns the result. Does not modify the provided vector

Parameters:

const Vector4&	rkV	*Vector to transform*

Returns:

Vector4 that is the result of the provided Vector4 transformed by the current matrix.

```
static Matrix4 Translation(float x, float y, float z);
```

Description:

Creates a translation matrix

Parameters:

float	x	*X Translation*
float	y	*Y Translation*
float	z	*Z Translation*

Returns:

Matrix4 that translates on X, Y and Z units.

```
static Matrix4 RotationX(float fAngle);
```

Description:

Creates a rotation matrix around the X Axis

Parameters:

float	fAngle	*Amount, in radians, to rotate*

Returns:

Matrix4 rotation matrix about the X Axis

```
static Matrix4 RotationY(float fAngle);
```

Description:

Creates a rotation matrix around the Y Axis

Parameters:

float	fAngle	*Amount, in radians, to rotate*

Returns:

Matrix4 rotation matrix about the Y Axis

```
static Matrix4 RotationZ(float fAngle);
```

Description:

Creates a rotation matrix around the Z Axis

Parameters:

`float`	`fAngle`	*Amount, in radians, to rotate*

Returns:

Matrix4 rotation matrix about the Z Axis