# GameJam C++ Engine and Example

## *Overview*

The GameJam C++ Engine is a minimal 3D game engine that provides the following basic features:

- Game object (actor) management, update and rendering
- World camera
- Game object begin / end touch detection
- Touch input monitoring
- Sound effect and music play back
- Text labels
- Timers

The engine is split across the following classes:

- Actor – A basic game object, you can use this class directly for game objects that require no logic, such as inert background objects, but for any kind of object that requires some kind of logic you should derive your own game object from this and implement the Update() method
- Audio – A thin wrapper around the sample Marmalade sound engine which provides easy convenient playback of WAV file sound effects and MP3 audio files for music
- Graphics – A thin wrapper around the Marmalade graphics system, provides features such as initialisation, clean-up, clearing the display and back buffer swapping
- Input – A thin wrapper around the Marmalade pointer system, provides features such as finding screen touch points and touch status
- Label – A basic text label actor that can be used to display text, supports a font, colour, position, scale and vertical / horizontal alignment
- Timer – A simple timer class that can be used to time the duration of events as well as create delays in-game
- World – Responsible for maintaining a list of world objects (actors). You should derive your own World object from this class to provide your own game functionality

The C++ GameJam distribution contains the following folders:

- docs – Contains this document
- engine – The C++ mini game engine
- example – The MaxiCube example game that utilises the mini game engine
- tools – Contains the FBX tool that is used to convert Maya LT exported FBX files to Marmalade format

## *Setting up a Project to use the Engine*

Marmalade manages projects using the MKB file. A MKB file is a text file that contains various sections that define source files, data files, assets that should be deployed, sub projects and other options.

Below is the MKB that is used to generate the MaxiCube example game:

```
#!/usr/bin/env mkb

subprojects
{
    engine
}

files
{
    [Source]
    (source)
    Main.cpp
    MyActor.cpp
    MyActor.h
    MyWorld.cpp
    MyWorld.h
    Resources.cpp
    Resources.h

    [Data]
    (data)
    board.group
    cube.group
}

assets
{
    (data)
    turn.wav
    Serif.ttf
    end_round.mp3
    game_over.mp3

    (data-ram/data-gles1, data)
    board.group.bin
    cube.group.bin
}
```

The project file is split into 3 sections:

- subprojects – In this section we list any sub projects that we want to use in our project. In this case we list the C++ game engine
- files – Lists source and data files that we wish to include in the project. In this case we include the game code source files and our model groups. Note that model groups are included here because they need to be converted to binary format when the x86 debug version of the game is run
- assets – Lists data files that we would like to include when the app is deployed to device. In this case we include the generated model binary data and our audio and font assets

## *The Main Loop*

The main loop for many Marmalade games usually looks something like this:

- Display pre-render (clear screen etc..)
- Monitor input
- Update audio
- Update and draw game
- Display post render (flush graphics system, swap display buffers etc..)
- Yield to operating system (this gives time for OS to take care of general house keeping)
- Repeat from start

Using the provided engine the main loop for most games created with it will look like this:

```
int main()
{
	// Initialise Marmalade utility API
	IwUtilInit();

	// Initialise the graphics system
	g_pGraphics = new Graphics();

	// Initialise the input system
	g_pInput = new Input();

	// Initialise the audio system
	g_pAudio = new Audio();

	// Create a world
	MyWorld*world = new MyWorld();
	world->Init();

	// Loop forever until the user or the OS performs some action to quit the app
	while (!s3eDeviceCheckQuitRequest())
	{
		// Clear the drawing surface
		g_pGraphics->PreRender();

		// Update input system
		g_pInput->Update();

		// Update audio system
		g_pAudio->Update();

		// Update the world
		world->Update();

		// Render the world
		world->Render();

		// Show the drawing surface
		g_pGraphics->PostRender();

		s3eDeviceYield(0);  // Yield to the OS
	}

	// Cleanup
	world->Release();
	delete world;
```

```
        delete g_pAudio;
        delete g_pInput;
        delete g_pGraphics;

        IwUtilTerminate();

        return 0;
}
```

## Touch Input

The Input class provides a thin wrapper around Marmalade's pointer input system s3ePointer. The class provides basic access to touched state and touched position. The main loop that we mentioned previously takes care of the per frame housekeeping enabling you to simply check the position of a touch (m_X, m_Y), the current touch state (m_Touched) and previous touched state (m_PrevTouched). Using a combination of current and previous touch states you detect when the player taps the screen then use the touch position to determine where the user tapped.

## Audio Playback

The Audio class provides a thin wrapper around both the Marmalade sample sound engine for the playback of compressed sound effects such as WAV files as well as playback of larger compressed music files such as MP3.

Playing a sound effect is a simple case of calling Audio::PlaySound(), e.g:

```
g_pAudio->PlaySound("turn.wav");
```

To play music simply call Audio::PlayMusic(), e.g:

```
Audio::PlayMusic("game_over.mp3", false);
```

You can also stop music from playing using Audio::StopMusic()

Note that WAV files need to be saved in mono IMA ADPCM format (not Microsoft format).

## Graphics

The Graphics class contains minimal functionality that deals with initialisation / clean-up of Marmalade's 3D graphics engines IwGx / IwGraphics and font rendering system IwGxFont. It also deals with pre / post rendering activities such as clearing the display and swapping display buffers. You can also set the background colour of the screen using Graphics::setBackgroundColour().

Note that individual actors deal with rendering themselves via the Actor class.

## The World

The World class is responsible for maintaining a list of world objects (actors). You will need to derive your own class from the World class and implement the following 4 methods:

- Init() - Here you will place your world specific initialisation, for example, setting up game objects, setting up initial camera position, setting up initial game variables etc..
- Release() - Here you can clean-up resources and memory that you may have used. Note that any actors that were added to the world will automatically be cleaned up by calling the World::Release() base method
- Update() - Here you will add your custom game logic
- Render() - Here you can add any custom rendering code, for most simple games you will not need to implement this method

A basic derived world would look like this:

```
class MyWorld : public World
{
public:
	void	Init();
	void	Release();
	void	Update();
	void	Render();
};
```

See MyWorld in the MaxiCube example for a custom implementation of World.

## Creating and Adding Actors to the World

The main driving force in your game will be game objects (actors). These provide the focus and interactivity within the game and can be anything from inert background objects such as rocks / trees to fully interactive objects such as cars, opening doors, characters and so on.

You should begin actor creation by firstly deriving your own class from Actor and implementing the following methods:

- Init() - Here you will place your actor specific initialisation, for example, setting the initial state of the actor. You should call Init() directly after creating an instance of your actor
- Release() - Here you can clean-up resources and memory that you may have used. Note that an actor will only be released when the parent world is released or when you call Release() manually
- Update() - Here you will add your custom actor logic
- Render() - Here you can add any custom rendering code, for most simple games you will not need to implement this method

A basic derived actor would look like this:

```
class MyActor : public Actor
{
public:
	void	Init();
	void	Release();
	void	Update();
	void	Render();
};
```

See MyActor in the MaxiCube example for a custom implementation of Actor.

To add an actor to the world for processing and rendering simply call the AddActor(my_actor) method of World.

To remove an actor from the world call RemoveActor(my_actor) method of the World. Note that if you remove an actor from the world manually then you will need to manually Release the actor and delete the actor to free up the memory that it uses.

## *Actor Touch Events*

The World class contains a feature that is useful for many types of games that rely on the player to touch certain objects on the screen.

An actor supports begin and end touch events that are called by the World if the player touches one. The World does this by checking if a ray from the touched screen position intersects with the object. This feature is enabled by default but should be turned off for objects that do not require it using Actor::setTouchable(false).

To respond to actor touch events you should implement the Event_BeginTouch() and Event_EndTouch() event handlers in your derived actor. You can see an example of using touch events in the MyActor class in the MaxiCube example.

The actor touch logic is quite simple and does not take into account overlapping objects.

## *Using Labels to Display Text*

A Label is used to display text on the screen using a specific font and colour. Labels are rendered using 2D screen coordinates so the z value in the labels position is ignored, rotation is also ignored. You can align the label using horizontal and vertical alignment values, these include:

Horizontal alignment values:

- IW_GX_FONT_ALIGN_LEFT - Text will be justified against the left-hand side of the rect
- IW_GX_FONT_ALIGN_CENTRE - Text will be centred horizontally in the rect
- IW_GX_FONT_ALIGN_RIGHT - Text will be justified against the right-hand side of the rect

Vertical alignment values:

- IW_GX_FONT_ALIGN_TOP - Text will be positioned against the top of the rect
- IW_GX_FONT_ALIGN_MIDDLE - Text will be centred vertically in the rect
- IW_GX_FONT_ALIGN_BOTTOM - Text will be positioned against the bottom of the rect

Note that the rect is set by default to the entire area of the screen.

In order to render a label you must assign it a font. Fonts can be created from a true type font (TTF) file using the IwGxFontCreateTTFont() function, e.g:

```
Font = IwGxFontCreateTTFont("Serif.ttf", 14);
```

You can see an example of this in the Resources class in the MaxiCube example.

Below is an example showing how to create a label and add it to the World:

```
Label* label = new Label();
label->Init();
label->setColour(200, 200, 80, 255);
label->setFont(g_pResources->Font);
label->setText("TURNS LEFT");
world->AddActor(label);
```

## *Using Timers*

A simple Timer class has been provided that enables you to create timed events, allowing you to set off events in the future and at regular periods. To create a timer you simply create an instance of the Timer class then set how long you would like the timer to run for. To determine when the the timer times out, you can check it by calling Timer::hasTimedOut(). Below is an example showing how to create and use a timer:

```
// Create a timer
Timer timer1;

// Elsewhere in our code we start a 2 second timer
timer1.setDuration(2000, true);

// Elsewhere in our code we check to see if the timer has expired
if (timer1.HasTimedOut())
{
	// Do something
}
```

## *Working with 3D assets*

Marmalade does not handle FBX files directly, instead you put them through a converter tool called FbxConvertor (located in the tools sub folder) that converts them into Marmalade group and model files. An example showing how to convert the 3D board that is used by the MaxiCube game is shown below:

FbxConverter board.fbx board.group

The above command will output the following files and folders:

- board.group – The 3D board group
- models/board.geo – The 3D board geometry data
- models/board.mtl – The 3D board materials
- models/textures – This folder contains textures that are mapped to the board

Once you have the converted data you should copy it into your projects data folder.

You should then update your project MKB file to add the group file to the files section and the generated binary geometry data to the assets section, e.g:

```
files
{
    [Data]
    (data)
    board.group
}

assets
{
    (data-ram/data-gles1, data)
    board.group.bin
}
```

Note that the file board.group.bin is a binary representation of the board geometry / materials etc.. This is generated for you when you run an x86 Debug version of your project. By adding these files into your project MKB you ensure that they get deployed with the rest of your app when you deploy to a device.

See the MaxiCube example MKB to see how assets have been added to the project.

## *MaxiCube Example Game*

The MaxiCube example has been provided to show a simple fully operational game utilising the mini 3D game engine. To build and run the example, open the example\MaxiCube.mkb project file which will open the project up in Visual Studio (Windows) or Xcode (Mac). Now build and run the project, this will launch the game in the Marmalade simulator.

The aim of the game is to fill the grid with the same symbol. You will be given just enough turns to complete the grid, completing the grid will take you to the next round. To turn a cube simply tap it.