

IwGame Game Engine SDK Programming v0.28

Developed and maintained by Mat Hopwood of Pocketeers Limited (<http://www.pocketeers.co.uk>).
For support please visit <http://www.drmop.com>

This document is protected under copyright to Pocketeers Limited @2011.

Table of Contents

1.0 IwGame.....	
1.1 What is IwGame?.....	
1.2 Installing IwGame.....	
1.3 Usage Rights and Warranties.....	
1.4 Brief How To on Using IwGame.....	
1.5 IwGame Concepts.....	
1.5.1 Standard Class Definitions.....	
1.5.2 Getters and Setters.....	
1.5.3 Init / Release.....	
1.5.4 Singletons.....	
1.5.5 Event Notification.....	
1.5.6 XOML Mark-up Language.....	
1.5.7 Local and Global Resources.....	
1.6 Example Code and Tutorials.....	
2.0 CIwGame Object – The Eye in the Sky.....	
2.1 Introduction.....	
2.2 Implementing our own CIwGame.....	
3.0 CIwGameScene Object – A Place for Actors to Play.....	
3.1 Introduction.....	
3.2 Creating a Scene.....	
3.3 Virtual Canvas.....	
3.4 Scene Object Management.....	
3.5 Scene Extents and Clipping.....	
3.6 Scene Camera.....	
3.7 Potential Colliders and Collision Handling.....	
3.8 Current Scene Concept.....	
3.9 Actor Management.....	
3.10 Scene Naming and Finding Scenes.....	
3.11 Scene Layers.....	
3.12 Scene Origin.....	
3.13 Scene Visibility and Active State.....	
3.14 Scene Transitions.....	
3.15 Creating a Scene from XOML.....	
3.16 Animating Scene Components.....	
3.17 Creating a Custom Scene.....	
4.0 CIwGameActor Object – Sprites With Brains.....	
4.1 Introduction.....	
4.2 Creating Actors.....	
4.3 Creating a CIwGameActorImage.....	
4.4 Actor Lifetimes.....	
4.5 Actor Naming and Finding Actors.....	
4.6 Actor Types.....	
4.7 Moving, Rotating and Spinning Actors.....	
4.8 Attaching a Visual and an Animation Timeline.....	
4.9 Changing an Actors Colour.....	
4.10 Obeying Scene Extents.....	

4.11 Actor Layering.....	
4.12 Scene Visibility and Active State.....	
4.13 Resetting Actors.....	
4.14 Collision Checking.....	
4.15 Creating an Actor from XOML.....	
4.16 Animating Actor Components.....	
4.17 Creating a Custom Actor.....	
5.0 CIwGameString – String Building Without Fragmentation.....	
5.1 Introduction.....	
5.2 Basic String Building.....	
5.3 Comparing Strings.....	
5.4 Stream Style Searching.....	
5.5 Other Useful String Tools.....	
6.0 CIwGameFile – File System Access.....	
6.1 Introduction.....	
6.2 Loading a Local File.....	
6.3 Saving a Local File	
6.4 Loading a Memory Based File.....	
6.5 Loading a Remote File.....	
6.6 Other Useful File Tools.....	
7.0 CIwGameInput – I Need Input.....	
7.1 Introduction.....	
7.2 Checking Availability.....	
7.3. Single and Multi-touch Touches.....	
7.4 Working with Touches.....	
7.5 Checking Key / Button States.....	
7.6 On Screen Keyboard.....	
7.7 Accelerometer Input.....	
7.8 Compass Input.....	
7.9 Input and the Marmalade Emulator.....	
7.10 Other Useful Utility Methods.....	
8.0 CIwGameTimer – Time and Timers.....	
8.1 Introduction.....	
8.2 Getting the Current Time.....	
8.3 Creating and Using Timers.....	
9.0 IwGameHttp – Playing Outside the Box.....	
9.1 Introduction.....	
9.2 The HTTP Manager CIwGameHttpManager.....	
9.3 IP Addresses and User-Agents.....	
9.4 POST and GET.....	
9.5 Setting Up Headers.....	
9.6 Performing a GET.....	
9.7 Performing a POST.....	
10.0 CIwGameAudio – Say No To Silent Movies.....	
10.1 Introduction.....	
10.2 The Audio Manager CIwGameAudio	
10.3 Adding Audio Resources.....	
10.3.1 Adding Sound Effects.....	

10.3.2	Creating a Resource Group.....	
10.4	Playing and Modifying Sound Effects.....	
10.5	Playing Streamed Music.....	
11.0	CIwGameImage – The Art of Game.....	
11.1	Introduction.....	
11.2	Creating an Image from a Resource.....	
11.2.1	Adding Images.....	
11.2.2	Creating a Resource Group.....	
11.3	Creating an Image from Memory.....	
11.4	Creating an Image from a Web Resource.....	
11.5	Creating Images from XOML.....	
12.0	CIwGameSprite – A Sprite for Life.....	
12.1	Introduction.....	
12.2	Sprite Manager.....	
12.3	Creating a Bitmapped Sprite.....	
12.4	Creating our own Custom Sprites.....	
13.0	CIwGameAnim – Life and Soul of the Party.....	
13.1	Introduction.....	
13.2	Animation Frame Data.....	
13.3	Animations and the Resource Manager.....	
13.4	Animation Instances.....	
13.5	Animation Targets and Target properties.....	
13.6	Animation Timeline's.....	
13.7	Resource Manager and Timelines.....	
13.8	Working with Animations.....	
13.9	Creating a Basic Animation in Code.....	
13.9	Creating a Basic Animation in XOML.....	
13.10	Creating an Image Animation.....	
15.0	CIwGameAds – Wanna Make Some Money?.....	
15.1	Introduction.....	
15.2	Setting up and Updating IwGameAds.....	
15.3	Ad Types.....	
15.4	Requesting Ads.....	
15.5	Working with CIwGameAdsView.....	
15.6	Ad Animators.....	
15.7	CIwGameAdsMediator – The Art of Ad Mediation.....	
15.8	Supported Ad Providers.....	
16.0	CIwGameXml – Cross Platform Data.....	
16.1	Introduction.....	
16.2	Setting the Memory Pooling.....	
16.2	Loading an XML file.....	
16.3	Working with Nodes.....	
16.4	Node and Attribute Iteration.....	
16.5	Attribute Query.....	
16.6	Creating an XML file.....	
17.0	CIwGameDataIO – Stream Style Access to Memory.....	
17.1	Introduction.....	
17.2	Input Streams.....	

17.3	Output Streams.....	
18.0	CIwGameResource – Getting a Handle on the Beast.....	
18.1	Introduction.....	
18.2	Creating a Resource Group.....	
18.3	Resource Management.....	
18.4	Global Resources.....	
19.0	CIwGameXOML – Welcome to the Easy Life.....	
19.1	Introduction.....	
19.2	Loading a XOML file.....	
19.3	Types of XOML Data.....	
19.3.1	Resource Groups.....	
19.3.2	Images.....	
19.3.3	Animations.....	
19.3.4	Timeline.....	
19.3.5	Actors.....	
19.3.6	Scenes.....	
19.3.7	Shapes.....	
19.3.8	Box2dMaterials.....	
19.4	XOML Workflow.....	
19.5	XOML's future.....	
20	Box2D Physics – Lets Bounce.....	
20.1	Introduction.....	
20.2	Box2dMaterial's.....	
20.3	Box2D World.....	
20.4	Box2D Bodies.....	
20.5	Box2D Collision.....	

1.0 IwGame

1.1 What is IwGame?

IwGame is a little more than what one would class as a traditional game engine, it also includes (or will soon include) much support for none game related functionality such as reading the camera, compass, sending / receiving data between web servers and requesting / showing ads etc.. IwGame is more of an object orientated extension of the Marmalade SDK designed and built to lessen the learning curve and to provide a lot of out-of-the-box functionality that is great for game development. The main aim of IwGame is to allow game developers to get started on a game immediately without having to worry about things such how to read the camera or perform gets and posts to send and retrieve data from web servers.

IwGame is an evolving open source SDK that enables developers to create games using tried and tested industry standard game programming techniques that are widely used across the industry today.

IwGame is built upon the incredibly powerful cross platform Marmalade SDK, enabling unparalleled “native” peddle to the metal game and application development across a wide range of platforms including Apple iPhone, iPad, Google Android (Phone and Tablet), Samsung Bada, Blackberry Playbook, Symbian, Palm WebOS, LG-TV, Windows and Mac OS.

In order to use IwGame you will need a copy of the the Marmalade SDK which you can download from <http://www.madewithmarmalade.com>

IwGame is currently maintained at <http://www.drmp.com/index.php/iwgame-engine/>

Whilst Pocketeers Limited does provide limited free support via www.drmp.com and www.pocketeers.co.uk, we can provide 24/7 dedicated paid support via email, telephone and Skype.

IwGames current feature set includes:

- Native and cross platform
- Free and open source
- Fully documented
- Support for multiple platforms (iPhone, iPad, Android, Bada, BBX, Symbian, Windows Mobile, mobile Linux, LG-TV, Windows and Mac OS)
- Layered sprites / sprite management (CIwGameSprite)
- Extensible actor / scene / camera system for organising game and game objects
- Box2D integration into the scene and actor system as well as XOML
- Advanced extensible animation system based on time lines with smooth key frame interpolation

IwGame Game Engine SDK Programming by Pocketeers Limited

- Multi-channel audio and music playback
- Touch and multi-touch support
- Button, keyboard, accelerometer and compass support
- Powerful extensible XML based mark-up system (XOML) – Enables definition of IwGame objects from XML script such as Scenes, Actors, Images, Animations, Timelines, Shapes, Physical materials and ResourceGroups. Support for custom tags also
- Queued POST / GET http communications
- Global and scene local resources, including automated clean-up
- Support for resource groups
- Auto handles resizing to any size / aspect ratio display using virtual canvasses
- Auto handles frame rate variations to produce smooth animation
- IwGameAd engine which mediates ad collection, display and click processing across multiple ad providers (12 providers supported to date). Ad system uses an animation system that is producing 3%-8% CTR
- Support for streamed IO style access to memory
- Support for simple, fast and memory efficient XML serialisation using memory pooling
- On demand GIF, PNG and JPEG image loading over HTTP
- PNG image format saving
- String builder support reducing memory fragmentation when dealing with strings
- Platform agnostic File I/O
- Timers and other utilities

Future planned features include:

- Optimised particle system
- Effects system for adding post render effects / scene transitions etc..
- Web streamed resources (allowing you to change and upgrade your game without having to resubmit it to app stores)
- Video playback wrapper (IwGameVideo)
- User interface system (IwGameUI) including support for Native UI
- Location services integration (IwGameLocation)
- Integration with the Box2D physics engine
- Camera and image reel access (CIwGameImaging)
- In-app purchasing integration using IwGameMarket classes
- Social services integration (Facebook, Twitter, IwGameSocial)
- Analytics integration (IwGameAnalytics)
- Web based game editor

1.2 Installing IwGame

Firstly, download the latest version of the IwGame SDK from <http://www.drmpop.com/index.php/iwgame-engine/>

If you haven't already installed the Marmalade SDK then grab a copy from <http://www.madewithmarmalade.com>

IwGame is currently provided as a Marmalade library project and test bed project giving examples of how it should be used.

The IwGame zip archive that you downloaded contains the following important folders:

```
IwGame
  Docs - Contains documentation
  h - Contains IwGame engine headers
  Libs - Contains IwGame dependencies such as zlib and libpng
  Source - The full source to IwGame
  TestBed - A test bed app that shows IwGame usage and integration
```

To open up the test bed app go to IwGame\TestBed and open the IwGame_TestBed.mkb Marmalade project file, this will open up the test bed project in Visual Studio / XCode.

If you take a look at the solution explorer you will see a IwGame filter / folder. If you drill down you will be able to see the source files and the Marmalade project file that make up the engine.

Now lets take a look at the IwGame_TestBed.mkb project file to find out how to integrate IwGame into your project.

From the MKB we can see that we only need to add two commands to integrate IwGame into our project:

```
options
{
    module_path="../../IwGame"
}

subprojects
{
    IwGame
}
```

These commands will tell Marmalade where to locate the project.

Once you have updated your project MKB with the above changes, run your MKB file again to regenerate your project. You can of course place the IwGame engine anywhere you like, just remember to change the module path.

Note that if you already have something in `module_path` then you can quite easily tag IwGame into the path like this:

```
module_path="module_path="$MARMALADE_ROOT/modules/third_party;../IwGame"
```

1.3 Usage Rights and Warranties

IwGame and associated classes and components are provided “as is” and “without” any form of warranty. Yours, your employers, your companies, company employees, your clients use of IwGame is completely at your own risk. We are not obligated to provide any kind of support in any shape or form.

You are free to use IwGame in your projects in part or in whole as long as the header comments remain in-tact. Whilst you are not obliged to mention your usage of IwGame in your products it would be great and beneficial to let us know as we can publicise your product on our blog and other web sites / services.

You may not claim the IwGame engine or its documentation as your own work or package it up and include it in any kind of premium Middleware product without express prior written notice from an executive of Pocketeers Limited with the correct authority to do grant authorisation. The only exception to this rule is provided to the makers of the Marmalade SDK.

Also note that IwGame utilises zlib, libpng and Box2D, so please take note of those licenses.

Our aim is to promote IwGame as a viable cross platform game engine built on top of the Marmalade SDK, so any publicity will serve to not only help increase public awareness of IwGame in the development community but also increase awareness of the Marmalade SDK, who made this all possible.

If you would like to let us know that you are using IwGame in your products then please get in touch with us at admin@pocketeers.co.uk. We also appreciate any comments or feedback.

1.4 Brief How To on Using IwGame

At this stage I think its important for you to have a basic understanding of how to use IwGame as this will help you to better understand the rest of this material.

The basic purpose of IwGame is to enable developers to get a game up and running quickly, with IwGame taking care of all the basic and mundane tasks.

Initially you would create you own game class derived from CIwGame then implement your own initialisation and clean-up routines in YourGame::Init() and YourGame::Release()

Once you have a basic game class up and running you begin adding scenes to your game. You can think of scenes as different views / environments with their own purpose. For example, you may have a scene that contains your main game area and another scene that contains your in-game overlays / HUD. You may also have a number of scenes that make up your user interface (options screen, menu, in-app purchases, ad display etc..). You can then add separate entities to each scene using actors. You can think of actors as smart sprites, they contain logical and visual components. Actors can be anything from simple in-game objects and effects to something more complex such as user interface button

As of v0.27 a new mark-up language was added to allow developers to create scenes, actors, images, animations, animation time lines etc.. using a very simple human readable XML style language. XOML is similar in style and function to Microsoft's XAML and Adobe's MXML mark-up languages, so if you are familiar with these then you have a good head start.

1.5 IwGame Concepts

It will make explaining how to use IwGame much easier if we explain a few concepts that relate to IwGame's development up front.

1.5.1 Standard Class Definitions

We like to write neat and tidy code and have come up with a standard system to make classes nice and readable. A class is usually split into 5 sections:

- Section 1 – Public enums, typedefs and static data
- Section 2 – Properties provide public access to class private data via getters and setters. A property can be thought of as a private class variable that you want to allow the outside world to change but in a safe manner.
- Section 3 – Private / protected types, class data and methods (used by the class internally only)
- Section 4 – Construction / destruction and initialisation.
- Section 5 – Public access class functionality

1.5.2 Getters and Setters

A long time ago I used to have many classes with lots of public variables and as you would guess many odd and difficult to track bugs to go along with that style of software development. Eventually I came across the idea of hiding all of my variables away (even those that I wanted to be public) and used methods to access them instead. These methods are called getters and setters and we use them extensively throughout IwGame

1.5.3 Init / Release

I personally took the decision a long time ago to not use constructors and destructor's much because they discourage object re-use. Instead I like to move all class initialisation code out of the constructor and place it into my class Init() method, same goes for the destructor and the Release() method. This allows me to re-initialise and tear down a class without having to actually new or delete it. This can prove very useful in many situations such as object pooling to reduce memory fragmentation, where the idea is to preserve the classes memory and re-use it.

1.5.4 Singletons

We use singletons extensively throughout IwGame because a) they provide global access to a particular system b) they ensure that only one instance of that system can be created at runtime reducing the chances of conflict and c) They are convenient and easy to use

We use the `CDEFINE_SINGLETON(class_name)` in our header file to define a class as a singleton and `CDECLARE_SINGLETON(class_name)` to declare the singleton methods. You will also note that we use a macro such as:

```
#define GAME Game::getInstance()
```

This macros allows us access the singleton using a more readable shorthand, for example:

```
GAME->Init();
```

Which looks better and more readable than:

```
Game::getInstance()->Init();
```

1.5.5 Event Notification

We use two primary methods of handling event notification:

- C style callbacks - Many of IwGames classes allow the user to be notified of specific events occurring or to allow the user to tag in their own specific functionality when certain events occur. For example, when an animation is started, stopped or has looped the user can be notified by setting the appropriate handler for the animation
- C++ virtual methods – Where we find it appropriate / convenient (generally when we expect the user to derive their own class from one of our base classes) we provide virtual and in some cases pure virtual event methods that the user should implement in their own derived version of the class. For example, IwGameScene implements a number of virtual event notification methods `NotifySuspending()`, `NotifyResuming()`, `NotifyLostFocus()` and `NotifyGainedFocus()` which should be implemented by the derived scene class

1.5.6 XOML Mark-up Language

XOML (Xml Object Modelling Language) is an XML style language designed and developed specifically by Pocketeers Limited to allow developers to design and layout IwGame scenes, set up actors, build complex animations, load resource group's etc.. XOML was chosen as it is an easy to understand human readable cross platform text based format

1.5.7 Local and Global Resources

Resources can be anything from images and animations to Marmalade resource collections. Resources can be local to a scene or global. Local resources are only accessible to objects within a scene and will be destroyed with the scene when it gets destroyed. Global resources are global to the game object and are accessible to all scenes and objects contained within those scenes. Global objects are destroyed when the game is destroyed. The distinction is important when taking into account resource names. Two resources of the same type should not share the same name within the same scene.

1.6 Example Code and Tutorials

The current IwGame distribution comes with an example application that shows many of the features of IwGame. IwGame is still very much in its infancy and it matures we will include new sample projects as well as tutorials on our blog at www.drmop.com

2.0 CIwGame Object – The Eye in the Sky

2.1 Introduction

CIwGame is basically the kind of eye-in-the-sky controller of the game engine and can be thought of as the main loop of the game engine. CIwGame takes care of many things including:

- Initialisation, processing and cleanup of many systems including graphics, input, resource manager and audio etc..
- Managing and updating scenes
- Notifying the user of scene change events
- Rendering scenes
- Tracking frame speed

You never actually create an instance of CIwGame, instead you derive your own version of the class from CIwGame like this:

```
#define      GAME      Game::getInstance()

class Game : public CIwGame
{
    CDEFINE_SINGLETON(Game)
public:
protected:
    /// Properties
public:
    /// Properties end
protected:
public:
    void          Init(bool enable_http);
    void          Release();
    bool          Update();
    void          Draw();
    void          Save();
    void          Load();
};
```

You then implement Init(), Release(), Update() and Draw() methods to provide your own initialisation, clean-up, per frame update and per frame rendering code.

Note that the games Init() method takes a parameter called enable_http. If you plan on using anything relating to the HTTP manager, such as ads, web file / image access then you will need to pass true to the game object during initialisation.

2.2 Implementing our own CIwGame

Lets now take a quick look at a bare bones implementation of the above methods:

```
CDECLARE_SINGLETON(Game)

void Game::Init(bool enable_http)
{
    CIwGame::Init(enable_http);

    // TODO: Insert your own initialisation here
}

void Game::Release()
{
    // TODO: Insert your own cleanup here

    CIwGame::Release();
}

bool Game::Update()
{
    if (!CIwGame::Update())
        return false;

    // TODO: Insert your own custom game update functionality here

    return true;
}

void Game::Draw()
{
    CIwGame::Draw();

    // TODO: Insert your own custom game rendering functionality here
}

void Game::Save()
{
    // TODO: Insert your own game save functionality
}

void Game::Load()
{
    // TODO: Insert your own game load functionality
}
```

Note that if you utilise IwGames systems then it is very unlikely that you will need to add additional rendering code to Game::Draw().

At its heart, CIwGame contains a collection of game scenes (CIwGameScene's) that in turn drive actors and cameras to provide your games functionality (more on these classes later).

IwGame Game Engine SDK Programming by Pocketeers Limited

CIwGame enables you to add, remove and search for scenes within the game as well as set the currently active scene using the following methods:

```
void          addScene(CIwGameScene *scene);
void          removeScene(CIwGameScene* scene);
void          removeScene(unsigned int name_hash);
CIwGameScene* findScene(unsigned int name_hash);
CIwGameScene* findScene(const char* name);
CIwGameScene* getScene(int index);
void          clearScenes();
void          changeScene(CIwGameScene *new_scene);
bool          changeScene(unsigned int name_hash);
CIwGameScene* getCurrentScene();
void          BringSceneToFront(CIwGameScene* scene);
```

Note that all visible scenes will be rendered every game frame and usually only the current scene will be updated.

3.0 CIwGameScene Object – A Place for Actors to Play

3.1 Introduction

Its easier to think about game development if we think in terms of the movie business, we all watch movies and programmes on the TV which makes it easy to relate to.

A movie usually consists of a number of scenes that contains the environment, actors and cameras. At the moment, IwGame only supports actors and cameras (environments may be added at a later date).

A CIwGameScene is responsible for the following functionality:

- Setup and handling of the virtual canvas
- Managing, updating and cleaning up actors
- Managing, updating, rendering and cleaning up sprites
- Managing and clean up of animation data
- Managing and clean up of animation timelines
- Managing and clean up of Marmalade resource groups
- Managing and clean up of images
- Managing and clean up of physics materials and shapes
- Clipping sprites against the scenes visible extents
- Updating the camera
- Tracking actors that can potentially collide
- Transitions between scenes
- Updating the scenes animation timeline
- Instantiating itself from XOML
- Updating physics

Its also worth a major note at this point that any actors that live inside a scene will be transformed by the same transform as the scene, so if you move, scale or spin the scene then the contained actors will also move, scale and spin with it. The same can also be said about the base colour of the scene

3.2 Creating a Scene

Creating a scene is very simple, as the following code shows:

```
CIwGameScene* game_scene = new CIwGameScene();
game_scene->Init();
game_scene->setName("GameScene");
game_scene->setVirtualTransform(VIRTUAL_SCREEN_WIDTH, VIRTUAL_SCREEN_HEIGHT, 0, true,
false);
changeScene(game_scene);
```

In the above code snippet we allocate a new `CIwGameScene` object and called its `Init()` method to set up scene internals. We give the scene a name so we can find it later then set the virtual transform (see next section for explanation of the virtual canvas). Finally we ask the game to change the current scene to our newly created scene (`CIwGame::ChangeScene()`)

You can create as many scenes as you like and add them to the game, just remember only one can be the active scene, but all visible scenes will be rendered.

I would like to add some additional information regarding `CIwGameScene::Init()`, its prototype looks like this:

```
int Init(int max_collidables = 128, int max_layers = 10);
```

As you can see, the method actually takes two parameters (defaults are applied so you can go with the defaults if you like). The two additional parameters are defined as:

- `max_collidables` – This the maximum size of the collidable objects list and should be as large as the maximum number of objects that can possibly collide in your scene. For example, if you have 100 objects that are marked as collidable then you can set this value to 100
- `max_layers` – Scenes support object layering, the default number of layers is set to 10, but you can change this value here

If you would like to prevent the internal Box2D physics engine update then set the scenes `WorldScale` to 0:

```
game_scene->setWorldScale(0, 0);
```

This stops the internal Box2D physics update. It can be re-enabled by changing `WorldScale` to a valid value.

3.3 Virtual Canvas

Targeting a large selection of different phones, tablets and other devices with a variety of screen sizes and aspect ratios is a bit of a nightmare when it comes to game development. Luckily the CIwGameScene class takes care of this for us. A scene is quite a clever object in that it can render itself to any sized / configuration display using the virtual canvas concept. A virtual canvas is basically our own ideal screen size that we want to render to. The scene will scale and translate its visuals to fit our canvas onto the devices display allowing us to get on with developing our game using a static resolution. Lets take a look at the prototype for setting a scenes virtual canvas:

```
void CIwGameScene::setVirtualTransform(int required_width, int required_height,
float angle, bool fix_aspect = false, bool lock_width = false);
```

And an explanation of its parameters:

- required_width – The width of the virtual canvas
- required_height – The height of the virtual canvas
- angle – Angle of the virtual canvas
- fix_aspect – Forces the rendered canvas to lock to the devices aspect ratio (you may see letter boxing or clipping)
- lock_width – Calculates scale based on screen_width / required_width if set true, otherwise scale based on screen_height / required_height.

3.4 Scene Object Management

When it comes to coding I am quite bone idle and hate having to track things such as allocating actors, images / memory for animations etc,. I want something that will let me allocate these types of objects, assign them then forget about them. CIwGameScene contains a SpriteManager, and a ResourceManager (will cover these in detail later) that takes care of the things that I don't want bothering with during development. In addition, if I remove a scene from the game CIwGameScene will clean the lot up for me when it gets destroyed.

This automated clean-up does come at a very small price however, if you want to add a resource group to the scene then you will need to do that through the scene itself. Here's an example:

```
// Create and load a resource group
CIwGameResourceGroup* level1_group = new CIwGameResourceGroup();
level1_group->setGroupName("Level1");
level1_group->setGroupFilename("Level1.group");
level1_group->Load();
// Add the resource group to the scenes resource manager
game_scene->getResourceManager()->addResource(level1_group);
```

As you can see we need a reference to the scene so that we can get to the resource manager. Not too painful, but thought it best to make you aware of this small caveat.

3.5 Scene Extents and Clipping

Scenes have an extents area that can be defined which defines the area in which actors can be, actors that are outside of the scenes extents are wrapped around to the other side of the scene. This behaviour can be enabled / disabled on a per actor basis. You can set the scenes extents by calling:

```
void CIwGameScene::setExtents(int x, int y, int w, int h);
```

Scenes also allow you to define a visible clipping area, pixels from the scene that fall outside that area will not be drawn. You can set the clipping area of a scene by calling:

```
void CIwGameScene::setClippingArea(int x, int y, int w, int h);
```

3.6 Scene Camera

A scene has a camera associated with it that allows the user to pan around the scene as well as rotate and scale the view. All actors within the scene will move / rotate and scale in relation to the camera. It is possible to switch cameras within a scene, although you will need to manage the lifetime of these cameras, the scene will only manage the currently attached camera (so do not delete it a camera if it is assigned to the scene. You can assign a camera to the scene using:

```
void CIwGameScene::setCamera(CIwGameCamera* camera);
```

3.7 Potential Colliders and Collision Handling

As a scene processes actors it will build a list of references to all actors that are marked as possibly colliders, once all actors have been processed the scene will walk the list of potential colliders and call their ResolveCollisions() method to give each actor a chance to handle its own collisions. Note that the scene does NOT handle collision detection and response, actors themselves should take care of that.

Note that actors that have a physics material attached are under the control of the Box2D physics engine and collision is handled separately (more on this in the actors section)

3.8 Current Scene Concept

Because the game consists of multiple scenes and only one scene can have the focus at any one time we use the concept of the current scene. Whilst all scenes are visible (unless made hidden) and rendered they are not all processed. By default the only scene that is processed is the current scene. It is however possible to force a scene to be processed even when it does not have the focus by calling `CIwGameScene::setAllowSuspend(false)`. This will prevent the scene from being suspended when another scene is made the current scene.

Scenes can exist in two states, suspended or operational (resumed). Suspending scenes are not processed but are still rendered. When a new scene is switched to using `CIwGame::changeScene(CIwGameScene* new_scene)` the old scene will be put into a suspended state and the new scene will be resumed. IwGame will notify you when either of these events occur via the following methods:

```
virtual void NotifySuspending(CIwGameScene* new_scene)    // This event is called when this
scene is being suspended
virtual void NotifyResuming(CIwGameScene* old_scene)      // This event is called when this
scene is being resumed
virtual void NotifyLostFocus(CIwGameScene* new_scene)    // This event is called when this
scene has just lost focus
virtual void NotifyGainedFocus(CIwGameScene* old_scene)  // This event is called when this
scene has just gained focus
```

In order to receive these events you should implement them in your derived scene class, e.g.:

```
void MyGameScene::NotifySuspending(CIwGameScene* new_scene)
{
    // Add code to handle the scene being suspended
}
```

Note that you do not need to derive your own scene class from `CIwGameScene` as long as you are happy with the generic functionality that it provides, however you will not have access to the suspend / resume functionality as C style callbacks are not used in this instance.

3.9 Actor Management

The main reason that scenes exist is to facilitate actors. Once an actor is created and added to a scene the scene handles their update, rendering and clean up. CIwGameScene contains the following methods for adding, removing and searching for actors within a scene:

```
void          addActor(CIwGameActor *actor);
void          removeActor(CIwGameActor* actor);
void          removeActor(unsigned int name_hash);
CIwGameActor* findActor(const char* name);
CIwGameActor* findActor(unsigned int name_hash);
CIwGameActor* findActor(int type);
void          clearActors();
```

3.10 Scene Naming and Finding Scenes

IwGame is designed to prevent what I like to call “unreliable references”. To me an unreliable reference is a reference to another object that can disappear at any moment without the object that references it knowing about it, this can lead to some pretty nasty bugs that are incredibly difficult to track down. So instead of simply keeping a reference to an object we keep a name.

IwGame uses object naming quite extensively for major system such as actors and scenes. The idea is that if we want to speak to a scene from somewhere outside that scenes instance we simply find it by name using CIwGame::findScene(). Once found we can grab a pointer to it and access it. The wonderful thing about this system is that if the scene has disappeared when we call findScene() a NULL pointer will be returned signifying that it no longer exists, allowing our calling code to determine what to do about it (as opposed to simply crashing or doing something even worse such as writing all over memory that its not supposed to).

The naming system does add a little overhead to our game but not a great deal as searches are done using string hashes instead of string comparisons. The tiny overhead is definitely worth the buckets of tears that you can potentially save from days of tracking down hard to find bugs.

3.11 Scene Layers

Actors within a scene can be depth sorted using layers. Each actor has its own layer number which decides where it will appear within the scenes depth tree. Actors on higher layers will appear over actors on lower layers. Actors on the same layer will be drawn in the order in which they were added to the layer, so later actors will be drawn on top of earlier added actors.

Note that the layering system is not strictly part of the scene engine, instead it is handled by the sprite manager contained within a scene, but for the purpose of easy access is accessible through the scene.

3.12 Scene Origin

A scenes origin is set at 0, 0, which is the centre of the virtual canvas (usually centre of screen) for the default transform.

3.13 Scene Visibility and Active State

Scenes can be made visible or invisible by calling `CIwGameScene::setVisible()`. You can also query the visibility of a scene using `CIwGameScene::isVisible()`. When a scene is invisible it is not rendered.

Scenes can also be made active or inactive by calling `CIwGameScene::setActive()`. You can also query the active state of a scene by calling `CIwGameScene::isActive()`. When a scene is active it is processed.

Note that a scenes active and visibility states are independent, an inactive scene will still be rendered and an invisible scene that is active will still be processed.

3.14 Scene Transitions

Coming soon....

3.15 Creating a Scene from XOML

Scenes can be created declaratively using XOML mark-up, making scene creation much easier and more readable. Below shows an example of a scene declared using XOML:

```
<Scene Name="GameScene" CanvasSize="320, 480" FixAspect="true" LockWidth="false"
Colour="0, 0, 255, 255" AllowSuspend="false">
</Scene>
```

The scene tag supports many different attributes that determine how a scene is created and how it behaves. A description of these tags are listed below:

- Name – Name of the scene (string)
- CanvasSize – The virtual canvas size of the screen (x, y 2d vector)
- FixAspect – Forces virtual canvas aspect ratio to be fixed (boolean)
- LockWidth – Forces virtual canvas to lock to width if true, height if false (boolean)
- Extents – A rectangular area that describes the extents of the scenes world (x, y, w, h rect)
- AllowSuspend – Determines if the scene can be suspended when other scenes are activated (boolean)
- Clipping – A rectangular area that represents the visible area of the scene (x, y, w, h rect)
- Active – Initial scene active state (boolean)
- Visible – Initial scene visible state (boolean)
- Layers – The number of visible layers that the scene should use (integer)
- Colliders – The maximum number of colliders that the scene should support (integer)
- Current - If true then the scene is made the current scene (boolean)
- Colour – The initial colour of the scene (r, g, b, a colour)
- Timeline – The time line that should be used to animate the scene
- Gravity – Box2D directional world gravity (x, y 2d vector)
- WorldScale – Box2D world scale (x, y 2d vector)

3.16 Animating Scene Components

Scenes allow an animation time line to be attached to them that animates various properties of the scene. The following properties are currently supported:

- Camera Position – Cameras current position
- Camera Angle– Cameras current angle
- Camera Scale– Cameras current scale
- Colour – Scenes current colour
- Clipping – Scenes current clipping extents
- Visible – Scenes current visible state
- Timeline – The currently playing timeline

Any of these properties can be set as an animation target

3.17 Creating a Custom Scene

Whilst CIwGameScene can suffice for most tasks, you may find that you need to create your own type of scene that has functionality specific to your game or app. You begin the creation of a custom scene by deriving your own scene class from CIwGameScene then overloading the following methods to provide implementation:

```
virtual int   Init(int max_collidables = 128, int max_layers = 10);
virtual void  Update(float dt);
virtual void  Draw();
```

Here's a quick example:

```
class MyGameScene : public CIwGameScene
{
public:
    MyGameScene() : CIwGameScene() {}
    virtual ~MyGameScene();

    virtual int   Init(int max_collidables = 128, int max_layers = 10)
    {
        CIwGameScene::Init(max_collidables, max_layers);
    }
    virtual void  Update(float dt)
    {
        CIwGameScene::Update(dt);
    }
    virtual void  Draw()
    {
        CIwGameScene::Draw();
    }
};
```

We have provided a very basic implementation of Init(), Update() and Draw() which call the base CIwGameScene class methods so we keep its functionality in-tact.

You can take the implementation one step further (or maybe two) by implementing both the IIwGameXomlResource and IIwGameAnimTarget interfaces to allow instantiation of your custom class from XOML and to allow your class to be a target for animation time lines.

Firstly lets take a look at XOML enabling your custom scene class. To get IwGame to recognise your class whilst parsing XOML files you need to do a few things:

- Derive your class from IiwGameXomlResource and implement the LoadFromXoml method
- Create a class creator that creates an instance of your class then add this to the XOML engine

Lets start by taking a look at step 1.

Because we have derived our class from CIwGameScene we already have the support for step 1. However we would like to insert our own custom attribute tags so we need to make a few changes.

Lets take a look at our new class with those changes:

```
class MyGameScene : public CIwGameScene
{
public:
    // Properties
protected:
    float Gravity;
public:
    // Properties End
    void setGravity(float gravity) { Gravity = gravity; }
    float getGravity() const { return Gravity; }
public:
    MyGameScene() : CIwGameScene(), Gravity(10.0f) {}
    virtual ~MyGameScene();

    virtual int Init(int max_collidables = 128, int max_layers = 10)
    {
        CIwGameScene::Init(max_collidables, max_layers);
    }
    virtual void Update(float dt)
    {
        CIwGameScene::Update(dt);
    }
    virtual void Draw()
    {
        CIwGameScene::Draw();
    }

    // Implementation of IIwGameXomlResource interface
    bool LoadFromXoml(IIwGameXomlResource* parent, bool load_children,
CIwGameXmlNode* node)
    {
        if (!CIwGameScene::LoadFromXoml(parent, load_children, node))
            return false;

        // Add our own custom attribute parsing
        for (CIwGameXmlNode::_AttribIterator it = node->attribs_begin(); it != node->attribs_end(); it++)
        {
            unsigned int name_hash = (*it)->getName().getHash();

            if (name_hash == CIwGameString::CalculateHash("Gravity"))
            {
                setGravity((*it)->GetValueAsFloat());
            }
        }

        return true;
    }
};
```

Our new class now basically supports a Gravity attribute that we will eventually be able to set in XOML using something like:

```
<MyGameScene Name="GameScene" Gravity="9.8">
</MyGameScene>
```

However, before we can do that we need to let the XOML system know about our new type of class (MyGameScene), so it can be instantiated when the XOM parser comes across it. To do this we need to create a creator:

```
class MyGameSceneCreator : public IIwGameXomlClassCreator
{
public:
    MyGameSceneCreator()
    {
        setClassName("MyGameScene");
    }
    IIwGameXomlResource* CreateInstance(IIwGameXomlResource* parent) { return new
MyGameScene(); }
};
```

The creator basically defines the tag name "MyGameScene" and returns an instance of the MyGameScene class when CreateInstance() is called.

To get the XOML system to recognise our creator we need to add it to the XOML parsing system using:

```
// Add custom MyGameScene to XOML system
Iw_GAME_XOML->addClass(new MyGameSceneCreator());
```

Now XOML integration is out of the way, lets take a quick look at enabling our class as an animation target.

To enable a class as an animation target we derive it from `IwGameAnimTarget` and implement the `UpdateFromAnimation()` method. Luckily we derived our `MyGameScene` class from the `CIwGameScene` class which already provides this functionality. Lets take a quick look at how we extend the animation update method to account for animating our gravity variable.

```
bool          UpdateFromAnimation(CIwGameAnimInstance *animation)
{
    if (CIwGameScene::UpdateFromAnimation(animation))
        return true;

    // Add our own custom animating property
    unsigned int element_name = animation->getTargetPropertyHash();

    if (element_name == CIwGameString::CalculateHash("Gravity"))
    {
        CIwGameAnimFrameFloat* frame = (CIwGameAnimFrameFloat*)animation-
>getCurrentData();
        setGravity(frame->data);
        return true;
    }

    return false;
}
```

We added the above code to our `MyGameScene` class definition. We begin by calling the base `UpdateFromAnimation()` method so we can keep the existing animation properties of the scene. We then add our own custom check for the Gravity variable. If the animation property matches Gravity then we set the gravity to the provided interpolated value.

4.0 CIwGameActor Object – Sprites With Brains

4.1 Introduction

Whilst our title comparison suggests that actors are simply sprites with brains they have the potential to be much more.

Going back to comparison in the scene introduction section, actors play a pivotal role in our scenes, each actor having its own unique role and visual appearance. Actors are the building block of the game, they provide the actual unique functionality and visuals that make up the game as a whole. They can provide any type of functionality from a simple bullet fleeting across the screen to something as complex as a dynamic machine that modifies its behaviour and appearance based upon data streamed from a web server.

A CIwGameActor is a very generic object that provides quite a lot of functionality out of the box. The idea is for developers to create their own actor types from the base CIwGameActor class then implement their own custom functionality within its Update() method. The basic functionality provided by CIwGameActor includes:

- Support for actor pooling to help reduce memory fragmentation
- Unique names so they can be searched
- Actor types
- Position, velocity and velocity damping
- Angle, angular velocity and angular velocity damping
- Scale and Colour
- Layers
- Active and visible states
- A visual that represents it on screen
- Animation timeline that can be attached to the visual
- Collision size / rectangle
- Wrapping at scenes extents
- Instantiation itself from XOML
- Animation timeline update
- A Box2D physical body consisting of a material and shape
- Box2D collision category, mask and group

Note that any changes made to the actor will automatically be applied to the actors visual.

As IwGame progresses more actor types with additional functionality will be created to create more out of the box style game objects (plug-in actors if you will). For the moment the following actors have been created for you:

- CIwGameActorImage – This object represents a basic image based actor which has an associated image and animation.

A word of warning, do not forget to call the based classes Init(), Reset(), Update(), UpdateVisual() methods from your own derived classes or the underlying functionality will not be provided.

4.2 Creating Actors

Creating an actor is very simple as the following code shows:

```
// Create player actor
MyActor* actor = new MyActor();
if (actor == NULL)
    return NULL;

actor->Init();
actor->setName("Player1");
actor->setPosition(x, y);

// Add player actor to the scene
scene->addActor(actor);
```

In the above code we create a basic MyActor object, which is a class that I created derived from CIwGameActor giving us the base CIwGameActor functionality. However, adding this code into a game wouldn't actually see anything as we have not assigned a visual element to the actor. CIwGameActor does not handle the creation of a visual for you, instead it handles the rendering and update of a visual and its animations.

To get developers started with actors we included the CIwGameActorImage that will create a basic image based actor that supports animation.

If you require your actor to support Box2D physics then you should either define the Box2DMaterial and Shape in XOML or if creating manually then call:

```
InitBody(Scene, shape, material, &Position, Angle, com.x, com.y);
```

This can be called before or after CIwGameActor::Init()

4.3 Creating a CIwGameActorImage

Creating an image based actor is a little more complicated, lets take a look at some example code:

```
// Create a new instance
ActorPlayer* actor = new ActorPlayer();
if (actor == NULL)
    return NULL;

// Create player actor
actor->setScene(scene);
actor->Init(image, 36, 40);
actor->setPosition(x, y);

// Add player actor to the scene
scene->addActor(actor);
```

Creation is very similar to creating a basic CIwGameActor with the additional complication of having to pass an image to the actors Init() method.

Looking at the above code we create an ActorPlayer, which is a class that I created derived from CIwGameActorImage as we want some basic image functionality.

We then call the actors its Init() method to set up actor internals. We give the actor a name so that we can find it later then set its world position to the centre of the scene. Finally we add the actor to the scene.

You will notice that ActorPlayer's Init() method has quite a few parameters. When we call Init(...) we are actually calling CIwGameActorImage::Init(...) and passing along all the details shown in the code above which includes an image that will represent our actor (or more usually an image atlas), and the width and height of the visual on screen (in virtual canvas coordinates). Internally CIwGameActorImage will create a sprite to display our actor.

The end product of the above code is an actor that can be seen, moved around, scaled, rotated etc..

Now lets take a look at a slightly more complicated example that creates an image based actor that uses an animation time line (more details on time line's later):

```
// Create a new instance
ActorPlayer* actor = new ActorPlayer();
if (actor == NULL)
    return NULL;

// Create an animation timeline to hold our image animation
CIwGameAnimTimeline* timeline = new CIwGameAnimTimeline();

// Create and set up our face animation
CIwGameAnimInstance* face_anim = new CIwGameAnimInstance();
face_anim->setAnimation(anim);
face_anim->setTarget(actor, "SrcRect");
timeline->addAnimation(face_anim);
timeline->play();

// Create player actor
actor->setScene(scene);
actor->Init(image, 36, 40);
actor->setTimeline(timeline);
actor->setPosition(x, y);

// Add player actor to the scene
scene->addActor(actor);
```

I have marked the changes from the previous example.

The first set of changes deals with creating a time line object then creating the instance of an animation and adding that to the time line. This process allows the actor to track and update its own animations

In the last change we simply assign the time line to the actor, the actor will now take care of playing the animation and updating the actors visual with animation changes.

4.4 Actor Lifetimes

Actors will persist within the scene until a) the scene is deleted b) you explicitly remove them or the recommended method c) they remove themselves. An actor can easily remove and delete itself from the scene by returning false from its Update() method. Here's an example:

```
bool ActorPlayer::Update(float dt)
{
    // If fade timer has timed out then delete this actor
    if (FadeTimer.HasTimedOut())
    {
        return false; // returning false tells the scene that we no need to be removed
    }

    // Calculate our opacity from time left on fade timer
    int opacity = FadeTimer.GetTimeLeft() / 2;
    if (opacity > 255) opacity = 255;
    Colour.a = opacity;

    return CIwGameActorImage::Update(dt);
}
```

4.5 Actor Naming and Finding Actors

As mention previously for scenes, actors also named objects, each instance of an object that you wish to query should have its own unique name (per scene) so that it can be located and modified at a later date.

You can find an actor in a particular scene using:

```
CIwGameActor* actor = scene->findActor("Player1");
if (actor != NULL)
{
    // Do somethinig with the actor
}
```

There are three ways to locate actors within a scene:

```
CIwGameActor*    findActor(const char* name);
CIwGameActor*    findActor(unsigned int name_hash);
CIwGameActor*    findActor(int type);
```

These allow you to search for actor by string, hash or type. Note that searching by type will return the first and only the first instance of that particular actor type. This is very useful if you want to find a unique actor type, for example the player.

4.6 Actor Types

When developing games I find it incredibly useful to assign different types of actors different type ID's, this allows me to optimise many area of my code such as collision checks. Carrying a type ID for each actor also comes in handy when you want to know the types of actor that you are interacting with.

You can set and get the actors type ID using:

```
void      setType(int type)
int       getType() const
```

4.7 Moving, Rotating and Spinning Actors

Actors come with a very basic physics system that allows movement via velocity and angular velocity, actors can also be scaled. CIwGameActor provides the following basic functionality to handle these features:

```
void      setPosition(float x, float y)
CIwFVec2  getPosition()
void      setAngle(float angle)
float     getAngle()
void      setVelocity(float x, float y)
CIwFVec2  getVelocity()
void      setVelocityDamping(float x, float y)
void      setAngularVelocity(float velocity)
float     getAngularVelocity() const
void      setAngularVelocityDamping(float damping)
void      setScale(float scale)
float     getScale() const
```

Note that velocity and angular velocity damping is a reduction factor that is applied each game frame to slow down objects linear and angular velocities. Their default values are 1.0f which provides no damping, setting this value to less than 1.0f will dampen velocity whilst setting it to a value greater than 1.0f will enhance velocity.

Also note that changing position or angle will not effect velocity.

If the actor was created with Box2D physics enabled then you can also use the supplied force application methods.

4.8 Attaching a Visual and an Animation Timeline

For our actor to become visible on screen we need to assign it a visual component. If you are rolling your own actor and don't go the CIwGameActorImage route then you will need to create and assign your own visual component to the actor.

To assign a visual to an actor you would call:

```
void          setVisual(CIwGameSprite* visual)
```

Now when the scene renders the actor it will attempt to render the visual. I want to mention at this point that as far as IwGame is concerned a visual is an object type that derived from a CIwGameSprite (we will cover this later), but for now we will just say that a sprite as far as IwGame is concerned is anything that can be displayed, be it a simple image or a complex piece of SVG.

And where you find visuals you will usually find some kind of animation. The actor class supports attachment of CIwGameAnimTimeline which is basically a collection of animations (we will cover this in more depth later). To assign a time line we call:

```
void          setTimeline(CIwGameAnimTimeline* timeline) { Timeline = timeline; }
```

4.9 Changing an Actors Colour

Each actor has its own independent colour (including opacity). All actors are set to a default colour of white and full opacity. To change the colour of an actor you can call:

```
void          setColour(CIwColour& colour)
```

Note that an actors colour will be combined with its parents base colour.

4.10 Obeying Scene Extents

By default an actor would merrily travel across the game scene and beyond its extents into oblivion and out of range coordinates, this can cause a bit of a mess for the underlying math and rendering routines. To prevent actors from going off into oblivion we can tell them to wrap around to the other side of the scene if they hit its extents boundary. To force actors to wrap around at the boundaries of the scene we call setWrapPosition(true):

```
void          setWrapPosition(bool enable)
bool          getWrapPosition() const      {
```

4.11 Actor Layering

We touched on layering earlier when we talking about layering in scenes. All actors within a scene exist (visually) on a layer. The layer determines the order in which the actors are rendered with lower layers appearing below higher layers. The maximum layer that an actor can exist on is determined by the scene that it lives in. To change the layer that an actor appears on and to retrieve its current layer we use:

```
void      setLayer(int layer)
int       getLayer() const
```

4.12 Scene Visibility and Active State

You can query an actors visibility state and set its visibility state using:

```
void      setVisible(bool visible)
bool      isVisible() const
```

You can query an actors active state and set its active state using:

```
void      setActive(bool active)
bool      isActive() const
```

Note that when an actor is made inactive it will also become invisible. However making an actor invisible will not make it inactive.

4.13 Resetting Actors

Because actors can be part of an object pooling system and may not get re-initialised when re-used, we provide the functionality to reset them to a default state. This allows developers to re-use objects and not worry about the previous state of the object. Just remember to call the underlying `CIwGameActor::Reset()` method from your own `Reset()` method to ensure that the actor is completely reset.

4.14 Collision Checking

Right now IwGame does not carry out collision checks for you, instead it calls back each actor in the scene after the scene has been updated to give each possible colliding object a chance to check and respond to collisions. To take advantage of this functionality you need to implement the following handler in your derived actor class:

```
virtual void ResolveCollisions() = 0;
```

A basic actor to actor collision method is included in CIwGameActor to allow actors to test for overlap based on the size set by setCollisionRect();

When a collision does take place, actors can notify each other by calling:

```
virtual void NotifyCollision(CIwGameActor* other) = 0;
```

Here's a quick example showing how to use the system:

```
void ActorPlayer::ResolveCollisions()
{
    // Walk the scenes actors
    for (CIwGameScene::_Iterator it = Scene->begin(); it != Scene->end(); ++it)
    {
        // Only test collision against ball type actors
        if ((*it)->getType() == ActorType_Ball)
        {
            // Check for physical collision
            if (CheckCollision(*it))
            {
                // Norify ourselves that we collided with ball actor
                NotifyCollision(*it);
                // Notify ball actor that we collided with it
                (*it)->NotifyCollision(this);
            }
        }
    }
}
```

4.15 Creating an Actor from XOML

Actors can be created declaratively using XOML mark-up, making actor creation much easier and more intuitive. Below shows an example of an actor declared using XOML:

```
<MyActor Name="Player1" Position="0, 0" Size="100, 100" Angle="45" SrcRect="0, 0, 36, 40" Image="Sprites" Timeline="Player1Intro2" />
```

The basic actor tag supports many different attributes that determine how an actor is created and how it behaves. A description of these tags are listed below:

- Name – Name of the scene (string)
- Type – A numerical type that can be used to identify the type of this actor (integer)
- Position– Position in the scene (x, y 2d vector)
- Velocity – Initial velocity of the actor (x, y 2d vector)
- VelocityDamping – The amount to dampen velocity each frame (x, y 2d vector)
- Angle – The orientation of the actor (float)
- AngularVelocity – The rate at which the orientation of the actor changes (float)
- AngularVelocityDamping – The amount of rotational velocity damping to apply each frame (float)
- Scale – The scale of the actor (float)
- Colour – The initial colour of the actor (r, g, b, a colour)
- Layer – The scenes visible layer that the actor should appear on (integer)
- Active – Initial actor active state (boolean)
- Visible – Initial actor visible state (boolean)
- Collidable – Collidable state of actor (boolean)
- CollisionSize – The circular size of the actor (float)
- CollisionRect – The rectangular collision area that the actor covers (x, y, w, h rect)
- WrapPosition – Of truethen the actor will wrap at the edges of the canvas (boolean)
- Timeline – The time line that should be used to animate the actor
- Box2dMaterial – Sets the physical material type used by the Box2D actor
- Shape – Box2D fixture shape for the Box2D actor
- COM – Centre of mass of Box2D body (x, y 2d vector)
- Sensor – Can be used to set the Box2D actor as a sensor (boolean)
- CollisionFlags – The Box2D body collision flags (category, mask and group)

For actors that are derived from CIwGameActorImage we have the following additional properties:

- Image – The image that is to be used as the actors visual (string)
- Size – The on screen visible size of the actor (x, y 2d vector)
- SrcRect – The position and source of the source rectangle in the image atlas (x, y, w, h rect). Used for panning the portion of a sprite atlas shown allowing frame based animation.

Note that unlike scenes you cannot create an Actor or ActorImage directly as their corresponding CIwGameActor and CIwGameActorImage classes are abstract, so you must derive your own actor class. More on this later.

In addition, actors must be declared inside a scene tag element as they must have a parent scene and cannot be declared as resources.

4.16 Animating Actor Components

Actors allow an animation time line to be attached to them that animates various properties of the actor. The following properties are currently supported:

- Position – Actors current position
- Velocity – Actors current velocity
- Angle – Actors current angle
- AngularVelocity – Actors current angular velocity
- Scale – Actors current scale
- Colour – Scenes current colour
- Layer – Actors current visible layer
- Visible – Actors current visible state
- Timeline – The currently playing timeline

For actors that are derived from CIwGameActorImage we have the following additional properties:

- SrcRect – Actors current bitmapped visual source rectangle
- Size – Actors visible size on screen

Any of these properties can be set as an animation target

4.17 Creating a Custom Actor

Whilst CIwGameScene can be instantiated and used as-is, CIwGameActor and CIwGameActorImage are abstract and cannot. The actor system is designed this way as the developer is meant to create their own custom actor types that provide bespoke functionality that is specific to their game.

You begin the creation of a custom actor by deriving your own actor class from either CIwGameActor or CIwGameActorImage then overloading the following methods to provide implementation:

```
virtual void    Init();
virtual bool    Update(float dt);
virtual bool    UpdateVisual();
virtual void    ResolveCollisions() = 0;
virtual void    NotifyCollision(CIwGameActor* other) = 0;
```

Here's a quick example:

```
class MyActor : public CIwGameActor
{
public:
    MyActor() : CIwGameActor() {}
    ~MyActor() {}

    void        Init()
    {
        CIwGameActor::Init();
    }

    bool        Update(float dt)
    {
        if (!CIwGameActor::Update(dt))
            return false;

        // Here we put our actor specific implementation

        return true;
    }

    bool        UpdateVisual()
    {
        if (!CIwGameActor::UpdateVisual())
            return false;

        // Here we put our actor specific rendering code (if any is needed)

        return true;
    }

    void        ResolveCollisions() {}
    void        NotifyCollision(CIwGameActor* other) {}
}
```

```
};
```

We have provided a very basic implementation of Init(), Update() and UpdateVisual() which call the base CIwGameActor class methods so we keep its functionality in-tact.

We also provide a none functional implementation of ResolveCollisions() and NotifyCollision() as these are required methods

You can take the implementation one step further by implementing both the IIwGameXomlResource and IIwGameAnimTarget interfaces to allow instantiation of your custom actor class from XOML and to allow your class to be a target for animation time lines.

Firstly lets take a look at XOML enabling your custom actor class. To get IwGame to recognise your class whilst parsing XOML files you need to do a few things:

- Derive your class from IIwGameXomlResource and implement the LoadFromXoml method
- Create a class creator that creates an instance of your class then add this to the XOML engine

Lets start by taking a look at step 1.

Because we have derived our class from CIwGameActor we already have the support for step 1. However we would like to insert our own custom attribute tags so we need to make a few changes.

Lets take a look at our new class with thiose changes:

```
class MyActor : public CIwGameActor
{
public:
    // Properties
protected:
    int                NumberOfEyes;
public:
    void                setNumberOfEyes(int num_eyes)    { NumberOfEyes = num_eyes; }
    float              getNumberOfEyes() const          { return NumberOfEyes; }
    // Properties End
public:
    MyActor() : CIwGameActor() {}
    ~MyActor() {}

    void                Init()
    {
        CIwGameActor::Init();
    }

    bool                Update(float dt)
    {
        if (!CIwGameActor::Update(dt))
            return false;
    }
}
```

```

        // Here we put our actor specific implementation
        return true;
    }

    bool    UpdateVisual()
    {
        if (!CIwGameActor::UpdateVisual())
            return false;

        // Here we put our actor specific rendering code (if any is needed)

        return true;
    }

    void    ResolveCollisions() {}
    void    NotifyCollision(CIwGameActor* other) {}

    // Implementation of IIwGameXomlResource interface
    bool    LoadFromXoml(IIwGameXomlResource* parent, bool load_children,
CIwGameXmlNode* node)
    {
        if (!CIwGameActor::LoadFromXoml(parent, load_children, node))
            return false;

        // Add our own custom attribute parsing
        for (CIwGameXmlNode::_AttribIterator it = node->attribs_begin(); it != node->attribs_end(); it++)
        {
            unsigned int name_hash = (*it)->getName().getHash();

            if (name_hash == CIwGameString::CalculateHash("NumberOfEyes"))
            {
                setNumberOfEyes((*it)->GetValueAsInt());
            }
        }

        return true;
    }
};

```

Our new class now basically supports a new NumberOfEyes attribute that we will eventually be able to set in XOML using something like:

```
<MyActor Name="AlienCritic" Position="100, 100" Size="100, 100" NumberOfEyes="3" />
```

However, before we can do that we need to let the XOML system know about our new type of class (MyActor), so it can be instantiated when the XOML parser comes across it. To do this we need to create a XOML class creator:

```
class MyActorCreator : public IIwGameXomlClassCreator
{
public:
    MyActorCreator()
    {
        setClassName("MyActor");
    }
    IIwGameXomlResource* CreateInstance(IIwGameXomlResource* parent) { return new
MyActor(); }
};
```

The creator basically defines the tag name "MyActor" and returns an instance of the MyActor class when CreateInstance() is called.

To get the XOML system to recognise our new creator we need to add it to the XOML parsing system using:

```
// Add custom MyActor to XOML system
IW_GAME_XOML->addClass(new MyActorCreator());
```

Now XOML integration is out of the way, lets take a quick look at enabling our class as an animation target.

To enable a class as an animation target we derive it from IIwGameAnimTarget and implement the UpdateFromAnimation() method. Luckily we derived our MyActor class from the CIwGameActor class which already provides this functionality. Lets take a quick look at how we extend the animation update method to account for animating our NumberOfEyes variable.

```
bool                UpdateFromAnimation(CIwGameAnimInstance *animation)
{
    if (CIwGameActor::UpdateFromAnimation(animation))
        return true;

    // Add our own custom animating property
    unsigned int element_name = animation->getTargetPropertyHash();

    if (element_name == CIwGameString::CalculateHash("NumberOfEyes"))
    {
        CIwGameAnimFrameFloat* frame = (CIwGameAnimFrameFloat*)animation->
>getCurrentData();
        setNumberOfEyes((int)frame->data);
        return true;
    }

    return false;
}
```

We added the above code to our MyActor class definition. We begin by calling the base `UpdateFromAnimation()` method so we can keep the existing animation properties of the actor. We then add our own custom check for the `NumberOfEyes` variable. If the animation property matches `NumberOfEyes` then we set the number of eyes to the provided interpolated value.

5.0 ClwGameString – String Building Without Fragmentation

5.1 Introduction

Strings are used extensively throughout game development, making it an incredibly important subject. We use strings for everything from naming objects to presenting interactive text to the player.

String building can be a nightmare for memory managers as constantly rebuilding strings causes many memory allocations and deallocations fragmenting the available memory into small hard to use units.

A string builder is a class that allows you to build a string using a predefined sized buffer or at the very least a buffer that can be resized. ClwGameString supports the following features:

- String builder functionality
- Named strings
- String concatenation and resizing
- String building from integers, floats and boolean types
- String comparison
- Stream style string searching
- Find strings between markers
- Character replacement
- HTML decoding
- URL encoding / decoding
- Change of case

5.2 Basic String Building

Strings can be created from raw text, integers, floats and boolean variables as shown below:

```
CIwGameString string("Hello");           // Creation from raw text
CIwGameString int_string(1234);           // Creation from an integer
CIwGameString int_string(100.234f);       // Creation from a float
CIwGameString int_string(true);           // Creation from a boolean
```

Strings can also be concatenated:

```
CIwGameString string("Hello");
string += ". How you doing";
```

If you are creating a string and you know that it will require quite a number of concatenations then you should set its initial size to prevent memory resizing, here's an example:

```
CIwGameString string;
string.allocString(1024); // here we preallocate 1024 bytes of memory for the string
string += "Hello!";
string += " How you doing.";
string += " I'm great thanks, how are you?";
string += " Fantastico!";
```

5.3 Comparing Strings

There are 5 ways to compare a string or part of a string:

```
bool      operator== (const CIwGameString& op);
bool      operator== (const char* op);
bool      operator== (unsigned int hash);
bool      Compare(const char* pString, int len) const;
bool      Compare(int start, const char* pString, int len) const;
```

The most optimal way to compare two strings is to compare two CIwGameString objects with auto hashing enabled on both, this will involve only a basic check of both strings hashes to see if they match. You do however need to enable auto hashing on both strings before you compare them, e.g.:


```
CIwGameString string1("String1");
CIwGameString string2("String1");

string1.setAutoHash(true);
string2.setAutoHash(true);
if (string1 == string2)
{
}
```

If a string is set to auto hashing then the new hash value will be recalculated every time the string is changed. For performance it is best to disable auto hashing then enable it when the string has finished building.

5.4 Stream Style Searching

CIwGameString is set up to allow stream like searching whereby your last searched position will be saved, allowing you to carry out additional searches from where the last search left off. This type of string searching is incredibly useful when it comes to parsing areas of memory. The following methods can be used:

```
int    Find(const char* string);           // Simple string search
int    FindNext(const char* string, int len); // Searches from last find position
for test string
int    FindNext(const char* string);       // Searches from last find position
for test string
void    FindReset();                      // Resets the find position to start
of string
int    StepFindIndex(int amount);          // Adjust the find position by the
specified
int    getFindIndex()                     // Gets the current find index
int    GetNextMarkedString(char start_mark, char end_mark, int &offset); // Returns
a string marked by start and end marker characters
```

5.5 Other Useful String Tools

CIwGameString contains a few additional utility methods to help make various tasks easier:

```
void    Replace(char chr, char with);
int     Contains(char c) const;
void    ReplaceHTMLCodes();
void    URLEncode(const char* str);
void    URLDecode();
void    ToUpper();
void    ToLower();
bool    SplitFilename(CIwGameString& filename, CIwGameString& ext);
bool    GetFilenameExt(CIwGameString& ext);
```

- Replace() - Replaces all occurrences of char “chr” with char “with” in a string
- Contains() – Returns true if a string contains the specified character
- ReplaceHTMLCodes() – Replaces HTML style codes such as & with their ASCII equivalents
- URLEncode() – Encodes a string as URL encoded
- URLDecode() – Decodes a URL encoded string
- ToLower() – Converts a string to all lower case
- ToUpper() – Converts a string to all upper case
- SplitFileName() - Splits a string into file name and extension strings
- GetFilenameExt() - Extracts a file names extension as a string

6.0 CIwGameFile – File System Access

6.1 Introduction

Veering off course a little, I'm not going to cover some of the more lower level features of IwGame as many of them are required to gain a complete understanding on how IwGame works.

IwGame encapsulates Marmalade's file system neatly into a single class called CIwGameFile. This class enables the following features:

- Auto file closing when the file object goes out of scope
- Reading and writing of local files
- Reading and writing of memory based files
- Blocking and none blocking reading of files from an external source such as a web site / server
- File name splitting
- File type retrieval

6.2 Loading a Local File

Loading a local file is very simple, as is shown in the following example:

```
// Here we declare a string, open a file then read some data into it
CIwGameString data;
CIwGameFile file;
if (file.Open("\\my_data.txt", "rb"))
{
    int len = file.getFileSize();
    data.allocString(len);
    data.setLength(len);
    file.Read((void*)data.c_str(), len);
}
```

6.3 Saving a Local File

Saving a local file is also a very simple, as is shown in the following example:

```
// Here we declare a string, open a file then write the string to it
CIwGameString data("Hello storage, how you doing?");
CIwGameFile file;
if (file.Open("\\my_data.txt", "wb"))
{
    file.Write((void*)data.c_str(), data.GetLength());
}
```

6.4 Loading a Memory Based File

Loading a memory file is just as easy as opening a local file, as is shown in the following example:

```
// Here we declare a string, open a file then read some data into it from memory
CIwGameString data;
CIwGameFile file;
if (file.Open(my_data_in_memory, my_data_length))
{
    int len = file.getFileSize();
    data.allocString(len);
    data.setLength(len);
    file.Read((void*)data.c_str(), len);
}
```

6.5 Loading a Remote File

The format of loading a file for remote is the same as local file loading

```
// Here we declare a string, open a remote file then read some data into it
CIwGameString data;
CIwGameFile file;
if (file.Open("http://www.myserver.com/my_data.txt", NULL, true))
{
    int len = file.getFileSize();
    data.allocString(len);
    data.setLength(len);
    file.Read((void*)data.c_str(), len);
}
```

With a few differences. The first difference is the very noticable filename change to that of a web address. The second more subtle difference is the inclusion of a 3rd parameter to Open() which tells the method to block until the complete file has been downloaded or an error occurs.

With modern games the user expects action on the screen most of the time, so sitting loading your assets from a server with no visual update would not be a great idea, ruling blocking remote file loading out for anything more than a few files. A better alternative is to use asynchronous file downloading that is none blocking, allowing the game loop to proceed whilst your assets load.

Loading a remote file using a none blocking method can be achieved as shown below:

```
int32 WebFileRetrievedCallback(void* caller, void* data)
{
    CIwGameFile* file = (CIwGameFile*)caller;

    // file->getContent() and file->getContentLength() contain the data and data size

    delete file;

    return 0;
}

// Initiate a none blocking file download
CIwGameFile* image_file = new CIwGameFile();
image_file->setFileAvailableCallback(WebFileRetrievedCallback, NULL);
image_file->Open("http://www.battleballz.com/bb_icon.gif", NULL, false);
```

Examining the above code we can see that we set up a callback so that we get notified when our file has been downloaded. Next we initiate the file download but this time passing "false" as our blocking parameter to ensure that the download does not block the main thread.

If you don't fancy setting up a callback, you can poll the CIwGameFile instead to see if the file has been retrieved using:

```
bool CIwGameFile::isFileAvailable()
```

6.6 Other Useful File Tools

CIwGameFile also contains a few additional useful tool type methods:

```
static void GetComponents(const char* file_path, CIwGameFilePathComponents& components);  
static bool GetFileType(const char* file_path, CIwGameString& type);  
static bool isHttp(const char* file_path, int path_len);
```

- GetComponents() – Splits a path into its separate drive, path, name and extension components
- GetFileType() – Returns the file type of the supplied file name
- isHttp() – Checks a file name to see if it uses the http protocol

7.0 CIwGameInput – I Need Input

7.1 Introduction

A game wouldn't really be much of a game if the user could not interact with it. IwGame provides the CIwGameInput singleton class to manage all game input. CIwGameInput manages the following types of input:

- Single and multi-touch input
- Button and key states
- On screen keyboard input
- Accelerometer
- Compass

Access to input methods are provided via the IW_GAME_INPUT macro, for example:

```
if (IW_GAME_INPUT->getTouchCount() > 0)
{
}
```

If you are using CIwGame then you do not need to worry about initialising, updating or cleaning up the input system, however if you are rolling your own solution then you will need to take care of these steps yourself, here's a quick example showing how to do this:

```
// Initialise the input system
CIwGameInput::Create();
IW_GAME_INPUT->Init();

// Main loop
while (1)
{
    // Update input system
    IW_GAME_INPUT->Update();
}

// Shut down the input system
IW_GAME_INPUT->Release();
CIwGameInput::Destroy();
```

7.2 Checking Availability

As IwGame is designed to work across multiple platforms you should check to ensure that a particular input system is available before you use it. Here's a quick example showing how to check that the pointer input is available:

```
// Check to see that the pointer is available
if (IW_GAME_INPUT->isPointerAvailable())
{
    // Check to see if any touches have been made
    int num_touches = IW_GAME_INPUT->getTouchCount();
}
```

IwGame provides a number of methods to check for particular input systems availability:

```
bool    isPointerAvailable()    // Returns availability of the pointer
bool    isKeysAvailable()      // Returns availability of keys
bool    isOSKeyboardAvailable() // Returns availability of on screen keyboard
bool    isAccelerometerAvailable() // Returns availability of accelerometer
bool    isCompassAvailable()   // Returns true if compass is available
```

7.3. Single and Multi-touch Touches

CIwGameInput supports single and multi-touch events, allowing you to check for multiple simultaneous touches. However many devices do not support multi-touch events so a method has been provided to determine multi-touch support:

```
bool    isMultiTouch()          // Returns multitouch capability
```

If you are developing a game or app that relies on multi-touch then you should implement a fall back method that will work with single touch devices. Touch modes is a good solution that can help mirror multi-touch functionality by putting the pointer into different modes, such as move, scale, rotate etc.. and allow the user to switch between them.

No matter if you are using single or multi-touch functionality retrieving touches is done in very much the same way.

7.4 Working with Touches

CIwGameInput provides methods that enable you to detect and collect touch data. The usual process is to determine if any touches have been made by calling IW_GAME_INPUT->getTouchCount() and then take a look at the touches list to see what touch events occurred. Here's an example:

```
// Check to make sure that the pointer is available
if (IW_GAME_INPUT->isPointerAvailable())
{
    // Get the total number of current touches
    int num_touches = IW_GAME_INPUT->getTouchCount();
    if (num_touches != 0)
    {
        // Check list of touches to see which are active
        for (int t = 0; t < MAX_TOUCHES; t++)
        {
            // Get the touch data
            CIwGameTouch* touch = IW_GAME_INPUT->getTouch(t);
            if (touch->active)
            {
                // Do something with the touch
            }
        }
    }
}
```

Note that getTouch() returns the CIwGameTouch struct for the touch at the specified index. CIwGameTouch looks like this:

```
struct CIwGameTouch
{
public:
    int      x, y;          // Touch position
    bool     active;        // Touch active state
    int      id;            // ID of touch - The system tracks multiple touches by assigning
                           // each one a unique ID
};
```

If you want to track a touch to monitor its status then you should store its ID and use CIwGameInput::getTouchByID(id) to find it again later.

7.5 Checking Key / Button States

As you expand your list of supported devices for your products you will discover that devices come in all sorts of different configurations, some will even have hard keyboards / keypads and buttons. For example, the Samsung Galaxy pro has a full QWERTY keyboard and almost all Android devices have hardware buttons for menu, home and back.

To query the state of a key / button (buttons are mapped to keys) you call the following methods of CIwGameInput:

```
bool    isKeyDown(s3eKey key)           // Tests if a key is down
bool    isKeyUp(s3eKey key)            // Tests if a key is up
bool    wasKeyPressed(s3eKey key)       // Tests if a key was pressed
bool    wasKeyReleased(s3eKey key)      // Tests if a key was released
```

Each method takes an s3eKey as input, a full list of possible keys can be found in s3eKeyboard.h.

Note that to detect the back and menu buttons you should check both s3eKeyBack / s3eKeyAbsBSK and s3eKeyMenu / s3eKeyAbsASK respectively.

7.6 On Screen Keyboard

As most devices do not have hardware keyboards an on screen keyboard is the only method of inputting text into the device. IwGameInput provides access to this functionality via the showOnScreenKeyboard():

```
const char* showOnScreenKeyboard(const char* prompt, int flags = 0, const char* default_text = NULL);
```

Calling this method will display a modal on screen keyboard with the provided prompt text and using the supplied default text (pass NULL if you do not require default text). Flags provides a hint to the system to let it know what type of keyboard you want to display to the user, possible values are:

- S3E_OSREADSTRING_FLAG_PASSWORD - A Password entry keyboard
- S3E_OSREADSTRING_FLAG_EMAIL - An email address entry keyboard
- S3E_OSREADSTRING_FLAG_URL - A web URL entry keyboard
- S3E_OSREADSTRING_FLAG_NUMBER - A Numeric entry keyboard

Passing 0 for flags will use the default keyboard.

Once the on screen keyboard has been dismissed the entered text will be returned as a string.

7.7 Accelerometer Input

An accelerometer is a device usually found inside phones and tablets that measures acceleration. This is great for gaming as you can use the actual angle or speed at which the user tilts their device to affect game play. For example, you could for example use the accelerometer to allow the player to navigate a ball around a maze or maybe determine how hard the player wants to hit a ball. However the accelerometer does have limitations. If the users phone is perpendicular to the floor then changes in reading may not be registered.

Accelerometer hardware is usually quite power hungry so in order to use it you need to start it using:

```
IW_GAME_INPUT->startAccelerometer();
```

And when not in use you can turn it off using:

```
IW_GAME_INPUT->stopAccelerometer();
```

Per frame update of the accelerometer is automatically taken care of by CIwGameInput.

To read the current position of the accelerometer you call:

```
CIwVec3 accelerometer_pos = IW_GAME_INPUT->getAccelerometerPosition();
```

Because the user can potentially start a game with the phone held at any angle, reading accelerometer readings are best made from a frame of reference. This is usually the initial position that the user is holding the device at when they start the game. To set the reference point for the accelerometer call:

```
IW_GAME_INPUT->setAccelerometerReference();
```

This will set the reference point for offset reads to the current position of the users phone. You may want to display a short instructions screen at this point that informs the user how to hold the phone.

To read the accelerometer position with respect to the reference point call:

```
IW_GAME_INPUT->getAccelerometerOffset();
```

7.8 Compass Input

The digital compass is a device that uses the Earth's ambient magnetic field to determine the orientation of the users phone. This allows you to measure the angle of the device and the direction in which its pointing.

Like the accelerometer hardware the compass is usually quite power hungry so in order to use it you need to start it using:

```
IW_GAME_INPUT->startCompass();
```

And when not in use you can turn it off using:

```
IW_GAME_INPUT->stopCompass();
```

Per frame update of the compass is automatically taken care of by CIwGameInput.

To read the current orientation and heading of the compass you call:

```
CIwVec3 compass_heading = IW_GAME_INPUT->getCompassHeading();  
int compass_direction = IW_GAME_INPUT->getCompassDirection();
```

7.9 Input and the Marmalade Emulator

The Marmalade SDK simulator will allow you to simulate multi-touch functionality in your application but you firstly need to enable it. To enable this functionality you need to:

- Go to the simulator menu and select Configuration Pointer
- Tick “Report multi-touch available” and “enable multi-touch simulation mode”

Now that you have enabled multi-touch simulation you can use the middle mouse button to place touches. You can move the touches around by holding the middle mouse button down over the placed touch and move it. To remove a multi-touch touch, simply click the middle mouse button over the touch again.

The PC keyboard provides more than adequate emulation of a real device keyboard. In addition certain keyboard keys act as buttons, for example the F1 key will simulate the menu button press on Android and F3 will simulate the back button.

7.10 Other Useful Utility Methods

The CIwGameInput class provides some additional utility functionality that can speed up development:

```
bool        hasTapped()
bool        isTouching()
CIwVec2     getTouchedPos()
CIwVec2     getDragDelta()
bool        isBackPressed()
void        resetBackPressed()
bool        isMenuPressed()
void        resetMenuPressed()
```

- hasTapped() - Returns true the user has tapped on the display
- isTouching() - Returns true if the user is touching the display
- getTouchPos() - Returns the position on the display that the user is touching
- getDragDelta() - Returns the number of pixels the user last moved their finger across the display
- isBackPressed() - Returns true if the user is pressing the back button
- resetBackPressed() - Resets the back button pressed status
- isMenuPressed() - Returns true if the user is pressing the menu button
- resetMenuPressed() - Resets the menu button pressed status

8.0 CIwGameTimer – Time and Timers

8.1 Introduction

Time plays a very important role in app and game development. Time allows us to perform useful tasks such as time events, fire off events at regular intervals and stabilise animations etc..

CIwGameTimer provides a software based timer mechanism for timing events as well a static method for retrieving the current time in milliseconds. Timers will not automatically fire off events when they expire, instead they have to be polled.

Timers provide additional functionality for calculating how much time is left on the timer as well as how much time has expired since the timer was started.

Timers don't really have much of an overhead so you can create as many as you like.

8.2 Getting the Current Time

To retrieve the current time in milliseconds CIwGameTimer provides a static method:

```
uint64 GetCurrentTimeMs()
```

8.3 Creating and Using Timers

Creating a timer is a simple case of declaring or allocating a CIwGameTimer then setting it off going. To check the timer you then poll it to check to see if it has timed out. Here's an example:

```
// Create a timer that expires after 10 seconds
CIwGameTimer BusyTimer;
BusyTimer.setDuration(10000);

// Check to see if the timer has timed out
if (BusyTimer.HasTimedOut())
{
}
```

Timers can be reset, stopped and started using Reset(), Stop() and Start().

A few additional utility methods are also included in the CIwGameTimer class:

```
bool        hasStarted()
bool        hasStopped()
void        setAutoReset(bool auto_reset)
uint64      GetElapsedTime()
uint64      GetTimeDiff(uint64 this_time)
uint64      GetTimeDiff()
uint64      GetTimeLeft()
```

- hasStarted() - Returns true if the timer was started
- hasStopped() - Returns true if the timer has stopped
- setAutoReset() - If true the timer will automatically restart itself when it runs out
- GetElapsedTime() - Returns the amount of time elapsed since the timer was started
- GetTimeDiff(time_diff) - Returns the time difference between the supplied time and the start of the timer
- GetTimeLeft() - Returns the amount of time left on the timer

9.0 IwGameHttp – Playing Outside the Box

9.1 Introduction

Modern apps and games are no longer limited to the confines of their local memory, they now have the power to play outside their limited box and interact with other external systems such as web sites and web services. For example, posting the players latest achievements to Facebook and Twitter feeds or playing an interactive game with friends on different devices running different operating systems.

IwGame provides access to the outside world via IwGameHttp. IwGameHttp provides the following functionality:

- Queues requests between your game and a web server
- Supports POST and GET requests
- Calculates user-agent based on platform / device and obtains IP address
- Error handling and reporting

IwGameHttp consists of a number of classes:

- CIwGameHttpHeader – Specifies headers that can be sent with POST and GET requests
- CIwGameHttpPostData – Specifies data that can be sent with a POST request
- CIwGameHttpRequest – Represents an HTTP request
- CIwGameHttpManager – The main HTTP manager that queues requests between the game and a web server

9.2 The HTTP Manager *CIwGameHttpManager*

CIwGameHttpManager is a singleton class that acts as the mediator between your game and an external web service. All requests sent to the HTTP manager are queued and processed in the order in which they are created. Requests are sent out one at a time, the next request will wait until the previous request returns or times out.

Access to the HTTP manager methods are provided via the IW_GAME_HTTP macro, for example:

```
IW_GAME_HTTP_MANAGER->AddRequest(&AdRequest);
```

Because not all games will have a need for HTTP communications, CIwGameHttpManager will not automatically be created and updated for you unless you request it when creating the main IwGame object.

You can also handle this yourself as shown below:

```
// Initialise the http manager
CIwGameHttpManager::Create();
IW_GAME_HTTP_MANAGER->Init();
```

the HTTP manager will need to be updated every game frame inside your main loop as shown below:

```
while (game_running)
{
    // Do game related stuff

    // Update http manager
    IW_GAME_HTTP_MANAGER->Update();
}
```

And finally the HTTP manager must be cleaned up on app exit

```
// Clean up http manager
IW_GAME_HTTP_MANAGER->Release();
CIwGameHttpManager::Destroy();
```

9.3 IP Addresses and User-Agents

Many web services require some method of identifying the mobile device that is accessing them. Identification usually comes in the form of the user-agent and IP address.

The user-agent is a header that browsers send to web server when they request data from it. From the user-agent the web server can determine what type of device and operating system the device has as well as what language the user is using

Some web services also require the users IP address. An IP address is a unique address that identifies the users mobile device on the internet. IP addresses can be local (to the network they are on, such as a home or office network) or remote. If your device is connected to the net over Wi-Fi then it will very likely be assigned a local IP address. If the devices is connected via the carrier then your device will have a remote IP address assigned to it by your carrier. This is a very important distinction to take note of as some services will not or cannot serve content to devices with local IP addresses.

Once you have initialised the CIwGameHttpManager you can retrieve the user-agent and IP address by calling:

```
CIwGameString&    getUserAgent()    // Returns current user-agent
CIwGameString&    getIPAddress()    // returns current IP address
```

If you would like to supply your own custom user-agent and / or IP address then you can do that also using:

```
void    setUserAgent(const char* user_agent)
void    setIPAddress(const char* ip_address)
```

9.4 POST and GET

POST and GET are two methods of communicating with a web server over HTTP. A POST is used to send data to the server whilst a GET is usually used to retrieve data from a server. You can actually think of POST and GET as commands that are sent to a server, with POST implying that you want to modify the state of the server in some way by sending data to it, whilst GET implies that you want to simply retrieve data and not modify it.

However, most web services support both GET and POST, allowing GET to modify the state of the server as data “can” be passed to the server as part of the URL. You have probably seen many long URL's of the form http://www.someserver.com/add_this.php?name=mat&socks=3&trousers=10&method=update (don't click the URL it means nothing). Notice how we are passing the items socks=3, trousers=10,method=update to the server. One major problem with using GET where POST should be used is data caching. Many servers now employ data caching techniques to reduce server load (they no longer have to go to the back end database and perform expensive SQL queries, instead they cache the web page or some data relating to the web page). So no matter what parameters you pass you could be returned cached data!

GET is however very convenient to use, you simply append all of your variables and values separated by ampersands onto the end of the URL.

Performing a POST on the other hand is only slightly more complicated. Instead of appending all your data onto the URL you package the data up into a string and set the data as the HTTP requests POST body. You then tell the server what type of data it is going to get by setting the Content-Type header and then tell it the size of the data by setting the Content-Length header.

IwGame supports both headers and post data via the CIwGameHTTPHeader and CIwGameHttpPostData classes. More on these classes later.

No matter which method you choose POST or GET, both will return a response from the web server, which can be read by calling CIwGameHttpRequest::getContent();

9.5 Setting Up Headers

In the world of HTTP communications, headers play a very important role, they carry around information that tells the web server and the device lots of important information, such as the user-agent used, the accepted types of data (MIME types), cookies, the size of the POST body etc. in fact, you can pass anything you like. A good summary of the standard header types can be found at http://en.wikipedia.org/wiki/List_of_HTTP_header_fields .

9.6 Performing a GET

To perform a GET you simply create a CIwGameHttpRequest and fill it in as shown below:

```
// Our GET completed callback
int32 GetCompletdCallback(void* caller, void *data)
{
    CIwGameHttpRequest* request = (CIwGameHttpRequest*)caller;

    // request->getContent() and request->getContentLength() contains the
    // request data and data length

    IW_GAME_HTTP_MANAGER->RemoveRequest(request); // Remove request from http manager
queue
    delete request; // Delete the request

    return 0;
}

CIwGameHttpRequest* Request = new CIwGameHttpRequest();
Request->setGET(); // Tell the manager that we want to do a GET
// Set the GET URL
Request->setURI("http://www.someserver.com/add_this.php?
name=mat&socks=3&trousers=10&method=update");
// Set a callback so we know when the request completes
Request->setContentAvailableCallback(&GetCompletdCallback, NULL);
// Set the user-agent header
Request->SetHeader("User-Agent", UserAgent.c_str());
// Queue our request
IW_GAME_HTTP_MANAGER->AddRequest(Request);
```

The callback GetCompletdCallback() is called when the HTTP manager retrieves the data, you can use getContent() to read the retrieved data. Note that you must also remove the request from the http manager queue to prevent the queue from becoming cluttered with previously processed requests.

9.7 Performing a POST

Performing a POST is strikingly similar with only a few additional changes.

To perform a POST you simply create a CIwGameHttpRequest and fill it in as shown below:

```
// Our POST completed callback
int32 PostCompletedCallback(void* caller, void *data)
{
    CIwGameHttpRequest* request = (CIwGameHttpRequest*)caller;

    // request->getContent() and request->getContentLength() contains the
    // request data and data length

    IW_GAME_HTTP_MANAGER->RemoveRequest(request); // Remove request from http manager
queue
    delete request;    // Delete the request

    return 0;
}

CIwGameHttpRequest* Request = new CIwGameHttpRequest();
Request->setPOST(); // Tell the manager that we want to do a POST
// Set the POST URL
Request->setURI("http://www.someserver.com/add_this.php");
// Set a callback so we know when the request completes
Request->setContentAvailableCallback(&PostCompletedCallback, NULL);
// set the POST body
Request->setBody("name=mat&socks=3&trousers=10&method=update");
// Set the user-agent header
Request->SetHeader("User-Agent", UserAgent.c_str());
// Set the POST body content MIME type as application/x-www-form-urlencoded
Request->SetHeader("Content-Type", "application/x-www-form-urlencoded");
// Set the body content length as a string
Request->SetHeader("Content-Length", CIwGameString(Request-
>getBody().GetLength()).c_str());
// Queue our request
IW_GAME_HTTP_MANAGER->AddRequest(Request);
```

The callback PostCompletedCallback() is called when the HTTP manager retrieves the data, you can use getContent() to read the retrieved data. Note that you must also remove the request from the http manager queue to prevent the queue from becoming cluttered with previously processed requests.

10.0 CIwGameAudio – Say No To Silent Movies

10.1 Introduction

Games would be pretty boring if they had no sound effects or music. Audio has always played an important role in games since the first games appeared many years ago. IwGame provides access to sound effect and streamed music playback via the CIwGameAudio singleton.

CIwGameAudio has the following features:

- Compressed WAV software sound effect playback
- MP3 playback via the devices media engine
- Support for multiple simultaneous sound effects
- Control over volume and pitch

IwGame contains a number of classes for dealing with audio playback:

- CIwGameSound – Represents a sound effect
- CIwGameSoundCollection – Represents a collection of sound effect
- CIwGameAudio – The game audio manager

If you are using CIwGame then you do not need to worry about initialising, updating or cleaning up the audio system, however if you are rolling your own solution then you will need to take care of these steps yourself, here's a quick example showing how to do this:

```
// Initialise audio system
CIwGameAudio::Create();
IW_GAME_AUDIO->Init();

// Main loop
while (1)
{
    // Update audio
    IW_GAME_AUDIO->Update();
}

// Shut down audio
IW_GAME_AUDIO->Release();
CIwGameAudio::Destroy();
```

Sound effects are compressed using 8 and 16 bit ADPCM IMA (not Microsoft's version).

10.2 The Audio Manager *CIwGameAudio*

CIwGameAudio is a singleton class that is responsible for playing sound effects and streamed music

Access to the audio manager methods are provided via the IW_GAME_AUDIO macro, for example:

```
IW_GAME_AUDIO->PlaySound("explosion");
```

CIwGameAudio provides the following methods:

```
CIwGameSound* PlaySound(const char* name);
CIwGameSound* PlaySound(unsigned int name_hash);
void          StopSound(const char* name);
void          StopSound(unsigned int name_hash);
void          StopAllSounds();
void          PauseAllSounds();
void          ResumeAllSounds();

bool          PlayMusic(const char* name); // Plays music from a local file
bool          PlayMusic(void* buffer, uint32 buffer_length, uint32 repeat_count); //
Plays music from a memory buffer
void          StopMusic();
void          PauseMusic();
void          ResumeMusic();
bool          isMusicPlaying();
```

Sound effects are played by name whilst music is played by file name or via a memory buffer (for pre-loaded audio).

CIwGameAudio contains a sound collection that stores all of the sound specs for all sound effects within the game. A sound collection is populated from a Marmalade resource group (more on this later)

10.3 Adding Audio Resources

10.3.1 Adding Sound Effects

Sound effects are loaded and tracked by the Marmalade resource system. To give our audio manager access to them we need to load the group containing the audio and assign it to the CIwGameAudio's sound collection. Here's an example showing the process:

```
// Load audio resource group into the resource manager
IwGetResManager()->LoadGroup("Audio.group");

// Set up audio
CIwResGroup* AudioGroup = IwGetResManager()->GetGroupNamed("Audio");
IW_GAME_AUDIO->setGroup(AudioGroup);
```

Our sound effects are now available and ready to be played.

Note that when you change the current audio resource group being used by CIwGameAudio, all sound effects will be stopped and all previous sound effects will be deleted and replaced with the new set.

10.3.2 Creating a Resource Group

Marmalade supports a resource grouping system via a resource manager called IwResManager(). This manager loads groups of resources in the using .group files. A typical resource group for our sound effects would like this:

```
CIwResGroup
{
    name "Audio"

    // Sound sample WAV files
    "./explosion.wav"

    // Create sound specs (can be thought of as sound materials)
    CIwSoundSpec
    {
        name          "explosion"          # The name we want to use to refer to this
sound effect in out code
        data          "explosion"          # The WAV file name (without .wav
        vol           0.7                  # Default volume to be played at
        loop          false                # Do we want this sound effect to play
forever?
    }

    // Create a sound group to contain all of our sound specs
    CIwSoundGroup
    {
        name          "sound_effects"      # Name of our sound group
        maxPolyphony  8                    # Maximum sounds that can be played
simultaneously
```



```
killOldest    false           # Tell system not to stop the oldest sound
effects frmo playing if we run out of channels
addSpec       "explosion"      # Add the explosion sound spec to our sound
group
}
}
```

The above resource group script basically creates a resource group named “Audio”, which we later access in our code by name via:

```
// Set up audio
CIwResGroup* AudioGroup = IwGetResManager()->GetGroupNamed("Audio");
```

It then creates a sound specification for our explosion sound effect. Finally we create a new sub group called “sound_effects” that contains all of our sound specifications.

If you want to learn more about Marmalade's resource management system then take a look at our blog on the object at <http://www.drmp.com/index.php/2011/10/01/marmalade-sdk-tutorial-marmalades-resource-management-system/>

We also have an additional blog directly related to creating audio resource groups at

<http://www.drmp.com/index.php/2011/10/07/quick-and-easy-audio-and-music-using-s3eaudio-and-iwsound/>

Once a resource group has been created you need to add it to the assets section of your mkb project file like so:

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

Note that audio.group.bin gets built from our Audio.group file when the x86 Debug build of our game is ran. So remember to run the game on the emulator before deploying to a device to ensure that the latest version of this file is created.

10.4 Playing and Modifying Sound Effects

Once a group of sound effects are assigned to the audio managers using `IW_GAME_AUDIO-->setGroup()`, all sound effects within the resource group will be created and added to the audio managers internal sound collection, at this point in time each sound effect will be instantiated as a `CIwGameSound`.

When we play a sound effect using the following code:

```
CIwGameSound* sound = IW_GAME_AUDIO->PlaySound("explosion");
```

A `CIwGameSound` object is returned from `PlaySound()`. This object allows us to control the sound after it has been started.

If we take a quick look at `CIwGameSound` functionality we see that it has a number of methods available to modify and check the status of the sound being played:

```
void    Play()
void    Stop()
void    SetVolume(float vol)
void    SetPitch(float pitch)
bool    isPlaying()
```

These methods perform the following function:

- Play – Plays the sound effect
- Stop() - Stops the sound effect from playing
- SetVolume() - Sets the volume of the sound effect. A value of 1.0f represents full volume
- SetPitch() - Sets the pitch of the sound effect. A value of 1.0f represents normal playback pitch
- isPlaying() - Returns true if the sound is currently playing

Sound effects can be stopped, paused and resumed en-mass using the following `CIwGameAudio` methods:

```
void    StopAllSounds();
void    PauseAllSounds();
void    ResumeAllSounds();
```

10.5 Playing Streamed Music

Streamed audio playback is usually used to play music that sits in the background of the game to give it atmosphere. IwGame utilises the media player on the device to playback audio from either storage or a pre-loaded memory buffer.

However, before music can be played back, we must firstly check that the device supports the codec that we used to encode the music we are attempting to play. Here is a quick example showing how to check and play an MP3 file using CIwGameAudio:

```
// Check to see if MP3 codec is supported
if (IW_GAME_AUDIO->isMusicCodecSupported(S3E_AUDIO_CODEC_MP3))
{
    // Play some music
    IW_GAME_AUDIO->PlayMusic("music.mp3");
}
```

Below is a list of all possible codecs:

```
S3E_AUDIO_CODEC_MIDI    // MIDI files
S3E_AUDIO_CODEC_MP3     // MP3 files
S3E_AUDIO_CODEC_AAC     // Raw AAC files
S3E_AUDIO_CODEC_AACPLUS // AAC plus files
S3E_AUDIO_CODEC_QCP     // QCP files
S3E_AUDIO_CODEC_PCM     // PCM files
S3E_AUDIO_CODEC_SPF     // SPF files
S3E_AUDIO_CODEC_AMR     // AMR files
S3E_AUDIO_CODEC_MP4     // MP4 or M4A files with AAC audio
```

Please note that these codecs may change with future versions of the Marmalade SDK and hence the IwGame engine so please consult the s3eAudioCodec enum located in s3eAudio.h for a full and up to date list.

Streamed music can also be stopped, paused and resumed. You can also check to see if music is playing using the following methods of CIwGameAudio:

```
void    StopMusic();
void    PauseMusic();
void    ResumeMusic();
bool    isMusicPlaying();
```

11.0 CIwGameImage – The Art of Game

11.1 Introduction

Lets face it, its possible to play a game without audio, we've all played a game with the audio turned down and it was still playable, in fact some games have truly awful audio and sound better with the audio switched off. However, try playing a game with all the graphics switched off, hmm, don't think that is going to work.

The brunt of any game is bore by the graphics engine, its usually where most of the games processing is taking place. Images are pivotal to any game engine (2D game engines at least), even vector graphic based games cache their vector based offering as images to increase performance.

IwGame provides a class specifically for loading and dealing with images called CIwGameImage.

CIwGameImage and associated classes are simple yet powerful supporting the following features:

- Image loading (PNG, GIF and JPEG) from Marmalade resource groups
- Image loading (PNG, GIF and JPEG) from a memory buffer
- Image loading (PNG, GIF and JPEG) from a file located locally or on the web (blocking and none blocking)
- Creation of PNG's from image data that can be saved or sent to a web server
- Management of collections of images using CIwGameResourceManager
- Instantiation from XOML

11.2 Creating an Image from a Resource

11.2.1 Adding Images

Images are loaded and tracked by the Marmalade resource system. When we create an image we assign it a name and a resource group. Here's an example showing the process:

```
// Load our graphics resource group into the resource manager
CIwResGroup* group = IwGetResManager()->LoadGroup("Graphics.group");
CIwGameImage* image = new CIwGameImage();
image->Init("sprites", group);

// Add the image to global resource manager
IW_GAME_GLOBAL_RESOURCES->getResourceManager()->addResource(image);
```

Note that if you are using scenes then you can use the scenes image manager instead of creating and managing your own. e.g.

```
// Add an image to the scenes resource manager
game_scene->getResourceManager()->addResource(image);
```

Either way, our image is now available and ready to use, well, almost.

The underlying 2D image will not actually be created until it has been loaded / uploaded to video RAM. There are two ways to accomplish this:

- You can call `CIwGameImage::Load()` on the image to force it to be loaded / uploaded.
- Load on demand – The image will be loaded / uploaded when it is first accessed (for example when the sprite manager attempts to draw a sprite that uses the image)

11.2.2 Creating a Resource Group

Marmalade supports a resource grouping system via a resource manager called `IwResManager()`. This manager loads groups of resources in the using `.group` files. A typical resource group for our graphics would look like this:

```
CIwResGroup
{
    name "Graphics"

    // Graphics
    "./sprites.png"
}
```

The above resource group script basically creates a resource group named “Graphics”, which we later access in our code by name via:

```
// Add an image to image manager
CIwResGroup* group = IwGetResManager()->LoadGroup("Graphics.group");
```

The resource group contains our `sprites.png` image file.

If you want to learn more about Marmalade's resource management system then take a look at our blog on the object at <http://www.drmop.com/index.php/2011/10/01/marmalade-sdk-tutorial-marmalades-resource-management-system/>

Once a resource group has been created you need to add it to the assets section of your mkb project file like so:

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

Note that graphics.group.bin gets built from our Graphics.group file when the x86 Debug build of our game is ran. So remember to run the game on the emulator before deploying to a device to ensure that the latest version of this file is created.

11.3 Creating an Image from Memory

We can create an image from a file in memory using the following method CIwGameImage:

```
bool Init(void* memory_file, int memory_file_size);
```

This method of image creation gives you the option to create images from local files or images that you have downloaded from an external source such as a web server. Here is an example showing how to create an image from a memory buffer:

```
// Download an image file from the web
CIwGameFile* image_file = new CIwGameFile();
image_file->Open("http://www.battleballz.com/test_image.jpg", NULL, true);

if (image_file->isFileAvailable() && image_file->getError() ==
CIwGameFile::ErrorNone)
{
    // Create an image from the downloaded JPEG file
    CIwGameImage* image = new CIwGameImage();

    // Initialise the image with the image file data in memory
    image->setName("test_image");
    image->Init(image_file->getContent(), image_file->getContentLength());
}
delete image_file;
```

Note that when creating an image from a memory buffer the image is automatically loaded so Load() does not need to be called.

11.4 Creating an Image from a Web Resource

We can create an image from a file located externally on a web server using:

```
void Init(const char* filename);
```

This method of image creation gives you the option to create images from local files or images that you have downloaded from an external source such as a web server. The image will not be loaded until Load(bool blocking) has been called. If you load an image using none blocking then you will need to check its loaded status by calling:

```
if (Image->getState() == CIwGameImage::CIwGameImage_State_Loaded)
```

11.5 Creating Images from XOML

IwGame allows you to create An image using XOML mark-up language. Here's a quick example:

```
<Image Name="Buddy" Location="http://www.battleballz.com/bb_icon.gif" Preload="true"
Blocking="false" />
```

This creates an image named buddy from a GIF file located on the the web. Note that it is preloaded but will not block the main loop whilst loading. Instead the sprite that is rendering the image will not appear on screen until the image has been fully downloaded.

We can also declare images that are located in Marmalade resource group files. Here is an example:

```
<ResourceGroup Name="Level1" GroupFile="Level1.group" Preload="true" />
<Image Name="Sprites" Location="Level1" Preload="true" />
```

Here we create a resource group called level1 then create an image called Sprites, passing the group name of the group that contains the “Sprites” image. Note that the image name and the name of the Marmalade resource must match or the resource may not be found.

Please note that images created inside a scene will be local to the scene and will be deleted when the scene is destroyed.

12.0 ClwGameSprite – A Sprite for Life

12.1 Introduction

Images wouldn't be much use unless we could move them around the screen, rotate, them, flash them and perform a whole host of other cool effects on them. A sprite is a visual element that can be moved around the display and rotated, scaled etc.. For the purpose of this document a sprite is not just an image based visual, it could be any kind of visual including a line or even a complex piece of vector graphics.

As explained in our actor discussion earlier, an actor is made of two parts a) a logical component and b) a visual component. The visual part of an actor is basically a sprite.

If you decided to use the predefined `CIwGameActorImage` class to derive your actors from then you will have had an image based sprite created for you automatically. If not then we have some extra work to do.

At the moment we have two types of sprite:

- `CIwGameSprite` – A basic generic sprite type that you can use to create your own types of sprites. It offers no rendering functionality and you are required to implement the base `Draw()` method to provide the custom rendering functionality.
- `CIwGameBitmapSprite` – A generic bitmapped based sprite that provides functionality for rendering an image based sprite

Sprites have the following parameters than can be dynamically changed:

- Width and height – Visual width and height on screen
- Position – A position on the screen, relative to the parent sprite manager
- Angle – Orientation in degrees (`IW_ANGLE_2PI == 360` degrees)
- Scale – Scale of sprite (`IW_GEOM_ONE == 1.0f`)
- Colour – Colour and opacity of the sprite
- Visible – Visibility
- Pooled – Determines if the sprite is part of a sprite pool
- Layer – Visible layer that the sprite lives on

During rendering a sprites visual transform is built based upon the sprites position, angle and scale as well as its parent sprite managers transform. To optimise sprite rendering a sprites visual transform is only updated if either the position, rotation or scale changes. It will also be updated if the parent sprite managers transform is modified.

12.2 Sprite Manager

The sprite manager (CIwGameSpriteManager) is the system that takes care of tracking and rendering sprites. When sprites are created they are added to a sprite manager. The sprite managers geometric transform will be used as the base of the sprites transform, so any translation, rotation or scaling that is applied to the sprite managers transform will also be applied to all sprites that it is managing.

The sprite manager handles layers of sprites to allow sprites to be rendered using depth. Sprites on lower layers appear below sprites on higher layers. You can define the total number of layers available to the sprite manager upon its creation using:

```
void Init(int max_layers = 10);
```

The default number of layers is set to 10, although you can change this to any value within reason. Note that if you are using the sprite manager that is created and maintained by a scene then the scene will determine the number of layers that a sprite manager has. The scene will also be in charge of updating and rendering the sprite manager.

12.3 Creating a Bitmapped Sprite

A bitmapped sprite is a basic image based sprite with support for image atlases using a source rectangular area that defines which portion of the assigned image is rendered; an image atlas is a large image that contains sub images.

A bitmapped sprite can render just a small portion of the large image allowing repositioning of the source rectangle within the image to create frame based animations.

Creating a bitmapped sprite is very simple as can be seen from the following example:

```
// Allocate a sprite
CIwGameBitmapSprite* sprite = new CIwGameBitmapSprite();

// Set its image
sprite->setImage(image);

// Set its display size
sprite->setDestSize(width, height);

// Add sprite to the sprite manager so it can be managed and drawn
Scene->getSpriteManager()->addSprite(sprite);
```

The variable “image” shown in the above code is a CIwGameImage that you would have previously created. The “Scene” variable is a scene that you will have already previously created. If you do not add a sprite via the scenes sprite manager then you must create and manage your own.

12.4 Creating our own Custom Sprites

IwGame's sprite system is powerful in the sense that its extensible. You can create a multitude of different custom sprite types and the sprite manager will update and render them for you (obviously you will need to provide the rendering functionality in your own derived sprite class). Lets take a quick look at how we would create our own sprite class by taking a look at the implementation of CIwGameSpriteImage:

Firstly lets look at our class layout:

```
class CIwGameBitmapSprite : public CIwGameSprite
{
    // Properties
protected:
    CIwGameImage* Image;           // Bitmapped image that represents this sprite
    int SrcX, SrcY;               // Top left position in source texture
    int SrcWidth, SrcHeight;      // Width and height of sprite in
source texture
public:
    void setImage(CIwGameImage* image)
    {
        Image = image;
    }
    CIwGameImage* getImage() { return Image; }
    void setSrcRect(int x, int y, int width, int height)
    {
        SrcX = x;
        SrcY = y;
        SrcWidth = width;
        SrcHeight = height;
    }
    void setSrcRect(CIwGameAnimImageFrame* src)
    {
        SrcX = src->x;
        SrcY = src->y;
        SrcWidth = src->w;
        SrcHeight = src->h;
    }
    int getSrcWidth() const { return SrcWidth; }
    int getSrcHeight() const { return SrcHeight; }
    // Properties End
public:
    CIwGameBitmapSprite() : CIwGameSprite(), Image(NULL), SrcX(0), SrcY(0), SrcWidth(0),
SrcHeight(0) {}
    virtual ~CIwGameBitmapSprite() {}

    void Draw();
};
```

Here we add properties to define the visual appearance of our sprite such as an image and a source rectangular area within that image that defines which portion of the image we want to display. We added a few setters and getters to access our new data.

Note how in the constructor we call the base CIwGameSprite constructor and set some default values for or class variable.

Next we will take a look at the Draw() method, only major method that we need to implement:

```
void CIwGameBitmapSprite::Draw()
{
    // Do not render if not visible or no image assigned
    if (Image == NULL || !Visible || Colour.a == 0)
        return;

    // If transform has changed then rebuild it
    if (TransformDirty)
        RebuildTransform();

    // Set this transform as the active transform for Iw2D
    Iw2DSetTransformMatrix(Transform);

    // Set colour of sprite
    Iw2DSetColour(Colour);

    // Render the sprite (centered)
    int x = -(Width / 2);
    int y = -(Height / 2);
    Iw2DDrawImageRegion(Image->getImage2D(), CIwSVec2(x, y), CIwSVec2(Width, Height),
        CIwSVec2(SrcX, SrcY), CIwSVec2(SrcWidth, SrcHeight));
}
```

Again, quite simple straight forward code. Firstly we perform a little error checking then rebuild the visual transform if it has changed. Next we set the Iw2D transform so that anything we render will be transformed by our sprites transform. Finally we render the sprite using its centre point as its origin using Iw2D (We use the centre point as the origin to allow the sprite to rotate and scale around its centre).

13.0 ClwGameAnim – Life and Soul of the Party

13.1 Introduction

There are plenty of great games out there that do not feature any form of animation. However, most modern games do feature animations to enhance their look and feel. Lets take “Angry Birds” as an example, Angry Birds uses animations all over the place, such as the animating birds and pigs, even the menus contain flashing and sweeping animations.

When we initially decided to implement an animation system we decided that it had to be very flexible and support any type of animation, be it animating the position of an actor, animating the graphical frames of an actor or even animating a list of commands for an actor. From IwGame's point of view, animation refers to any variable or set of variables that can change over time.

IwGameAnim currently supports the following features:

- Time based frame and interpolated named animations
- Delayed and looped animations
- Support for boolean, float, vectors (2d, 3d and 4d), rect (for image frames), string and custom frame data types
- Resource manager that tracks and manages sets of animations (scene local and global)
- Animation data re-use using animation instances
- Animation time lines to support multiple simultaneous animations that can target object specific properties such as the colour of a scene or the position of an actor
- Time line manager that tracks and manages sets of animation time lines
- Callback notifications for animation instance started, stopped and looped events
- Animations and animation time lines can be defined and attached to actors and scenes using XOML

The animation is split into the following classes:

- CIwGameAnimFrame – Base animation frame class
- IIwGameAnimTarget – Interface for classes that act as an animation target
- CIwGameAnim – A basic animation
- CIwGameAnimCreator – Class used to instantiate an animation from XOML
- CIwGameAnimManager – Manages a collection of animations
- CIwGameAnimInstance – An instantiated running animation
- CIwGameAnimTimeline – A collection of running animation instances
- CIwGameAnimTimelineCreator – Class used to instantiate a collection of running animations from XOML
- CIwGameAnimTimelineManager – Manages a collection of timelines

13.2 Animation Frame Data

Animation frame data is the backbone of all animations, it represents the actual discrete state of a variable at a given point in time (called a key frame). An animation frame consists of two components:

- Frame data – Frame data is the value or values of a specific variable at a specific point in time
- Frame time – The time at which the frame has its value (in seconds)

The base class for all animation frame data types is the `CIwGameFrameData`. This class provides a means for the developer to implement their own custom animation frame data types. The following animation frame data types are already implemented:

- `CIwGameAnimFrameBool` – Boolean variables such as actor visibility
- `CIwGameAnimFrameFloat` – Floating point variables such as angle and scale
- `CIwGameAnimFrameVec2` – Floating point 2 parameter variables such as 2D position and velocity
- `CIwGameAnimFrameVec3` – Floating point 3 parameter variables such as 3D position and velocity
- `CIwGameAnimFrameVec4` – Floating point 4 parameter variables such as colour
- `CIwGameAnimFrameRect` – Integer 4 parameter variables such as image source rectangles
- `CIwGameAnimFrameString` – String based parameters such as narrative text, commands or even other timeline's

13.3 Animations and the Resource Manager

The resource manager is used to manage a group of resources (animations and animation time lines included). The idea is that you create a group of animations and time lines then add them to the resource manager and then forget about them. The resource manager will take care of cleaning them up when the scene or game is destroyed. The main game object contains its own global resource manager whilst all scenes contain their own local resource manager.

Note that when animations are created via XOML mark-up they are added to the resource manager. Animations that are created inside a scene will be added to the scenes resource manager, whilst animations created outside of the scene will be added to the global resource manager.

To find an animation within a scenes resource manager you would use could such as:

```
CIwGameAnim* anim = (CIwGameAnim*)scene->getResourceManager()->findResource("Player1Anim",  
CIwGameXomlNames::Animation_Hash);
```

Note that if the animation is not found within the scene then the system will automatically search the global resource manager for the animation. You can prevent this global search by passing false

as the third parameter to findResource().

13.4 Animation Instances

You do not play a CIwGameAnim directly, instead you create an instance of it using CIwGameAnimInstance:

```
// Create and set up an animation
CIwGameAnimInstance* face_anim = new CIwGameAnimInstance();
face_anim->setAnimation(anim);
face_anim->setTarget(actor, "SrcRect");
```

Note the following line of code:

```
face_anim->setTarget(actor, "SrcRect");
```

This line of code tells the animation to modify the “SrcRect” (the actors image atlas position) property of the actor object, causing the image to change.

Animation instances can be played, paused, resumed, stopped and restarted. You can also tell an animation to delay playing for a finite period of time. Methods of CIwGameAnimInstance are shown below:

```
int          getRepeatCount()
void         setRepeatCount(int repeat_count)
float        getCurrentTime()
bool         isFinished()
CIwGameAnimFrame* getCurrentData()
float        getDelay()
void         setDelay(float delay)
bool         isPlaying()
bool         isPaused()
bool         isDelayed()
void         restart()
void         play()
void         stop()
void         pause()
void         setStartedCallback(CIwGameCallback callback)
void         setStoppedCallback(CIwGameCallback callback)
void         setLoopedCallback(CIwGameCallback callback)
```

13.5 Animation Targets and Target properties

IwGameAnim uses the concept of animation targets and animation target properties. An animation target is basically a class that contains a variable or group of variables that can be targeted for modification by an animation. The actual variable or variables that are targeted are called target properties. An example target would be an actor and example target property would be the actors position. When you create an instance of an animation you set the target and target property that the animation will modify using code similar to that shown below:

```
face_anim->setTarget(actor, "SrcRect");
```

Any class can be used as a target for animation as long as it derives from the IwGameAnimTarget interface and implements the the following pure virtual method:

```
virtual bool UpdateFromAnimation(CIwGameAnimInstance *animation) = 0;
```

When the animation updates it will call back this method passing in its data asking the method to update the state of the object. Both scene and actor classes already have this functionality implemented.

Lets take a quick look at how the CIwGameScene class has implemented this method:

```
bool CIwGameScene::UpdateFromAnimation(CIwGameAnimInstance *animation)
{
    unsigned int element_name = animation->getTargetPropertyHash();

    if (Camera != NULL)
    {
        if (element_name == CIwGameXomlNames::Position_Hash)
        {
            CIwGameAnimFrameVec2* frame = (CIwGameAnimFrameVec2*)animation-
>getCurrentData();
            Camera->setPosition(frame->data.x, frame->data.y);
            return true;
        }
        else
        if (element_name == CIwGameXomlNames::Angle_Hash)
        {
            CIwGameAnimFrameFloat* frame = (CIwGameAnimFrameFloat*)animation-
>getCurrentData();
            Camera->setAngle(frame->data);
            return true;
        }
        else
        if (element_name == CIwGameXomlNames::Scale_Hash)
        {
            CIwGameAnimFrameFloat* frame = (CIwGameAnimFrameFloat*)animation-
>getCurrentData();
            Camera->setScale(frame->data);
            return true;
        }
    }
}
```

```

    }
}
if (element_name == CIwGameXomlNames::Colour_Hash)
{
    CIwGameAnimFrameVec4* frame = (CIwGameAnimFrameVec4*)animation-
>getCurrentData();
    Colour.r = (uint8)frame->data.x;
    Colour.g = (uint8)frame->data.y;
    Colour.b = (uint8)frame->data.z;
    Colour.a = (uint8)frame->data.w;
    return true;
}
else
if (element_name == CIwGameXomlNames::Clipping_Hash)
{
    CIwGameAnimFrameRect* frame = (CIwGameAnimFrameRect*)animation-
>getCurrentData();
    ClippingArea.x = (uint8)frame->data.x;
    ClippingArea.y = (uint8)frame->data.y;
    ClippingArea.w = (uint8)frame->data.w;
    ClippingArea.h = (uint8)frame->data.h;
    return true;
}
else
if (element_name == CIwGameXomlNames::Visible_Hash)
{
    CIwGameAnimFrameBool* frame = (CIwGameAnimFrameBool*)animation-
>getCurrentData();
    IsVisible = frame->data;
    return true;
}

return false;
}

```

The logic is quite simple, we check the name of the property that was passed in by the animation instance then check that against known property names of the class. If it matches then we move the animation data from the animation instance into our classes local variable.

As you can see implementing your own custom animation targets is a simple case of:

- Deriving your class from `IIwGameAnimTarget`
- Implement the `UpdateFromAnimation(CIwGameAnimInstance *animation)` method

13.6 Animation Timeline's

An animation time line is basically a way to group together multiple animation instances and play, pause, stop and resume them all together. The general idea is that you create an animation time line then create animation instances and add them to the time line. You then attach the time line to your destination object, be that a scene or an actor. The animation system will then take care of the rest for you. Here is an example showing how to create and use a time line:

```
// Find our face animation
CIwGameAnim* face_anim = (CIwGameAnim*)scene->getResourceManager()-
>findResource("FaceAnim", CiwGameXomlNames::Animation_Hash);

// Create an animation timeline to hold our image animation
CIwGameAnimTimeline* timeline = new CIwGameAnimTimeline();

// Create and set up our animation instance
CIwGameAnimInstance* instance = new CIwGameAnimInstance();
instance->setAnimation(face_anim);
instance->setTarget(actor, "SrcRect");
timeline->addAnimation(instance);
timeline->play();

// Attach timeline to the actor
actor->setTimeline(timeline);
```

Defining and attaching animations is much easier and more intuitive if done declaratively using XOML mark-up. More on this this later

Note that when you attach an animation time line to a scene or an actor, the scene / actor will automatically take over updating it for you.

Time lines can be played, paused, resumed and stopped. All animation instances within the animation time line will be affected. Methods of CIwGameAnimTimeline are shown below:

```
void          setName(const char* name)
unsigned int  getNameHash()
void          addAnimation(CIwGameAnimInstance* anim)
void          removeAnimation(CIwGameAnimInstance* anim)
CIwGameAnimInstance* findAnimation(const char* name)
CIwGameAnimInstance* findAnimation(unsigned int name_hash)
void          play()
void          stop()
void          pause()
void          restart()
int           getAnimationCount()
void          setTargetElement(IIwGameAnimTarget* target)
```

13.7 Resource Manager and Timelines

The resource manager is generally responsible for managing the lifetimes of animation time lines, in particular those created from XOML mark-up.

Each scene has its own local resource manager and the global game class also contains a global resource manager.

13.8 Working with Animations

The general work flow when working with the animation system has two potential paths:

- Manual definition – You manually create all of the animation frames, animation classes, instances, timelines etc
- XOML definition – You load a XOML file that contains the definitions, find the time lines and attach

The first method is more difficult as it does require creating and setting up animation classes yourself. Here's the basic flow for manual animation setup:

- Create a CIwGameAnim object and give it a meaningful name
- Create and set-up animation frames
- Add animation frames to the CIwGameAnim object
- Add the CIwGameAnim object to the scene or global CIwGameAnimManager
- Later when you need to create an instance of the animation, search the animation manager for the animation by name
- Create a CIwGameAnimTimeline object
- Create a CIwGameAnimInstance object and attach the CIwGameAnim object
- Set up any additional parameters for the time line object
- Add the time line object to the scene or global time line manager (not essential as scenes and actors can clean them up for you)
- Set the actor or scenes time line using setTimeline()
- Call the time line's play() method to start the time line playing

The second method using XML is much simpler:

- Create a XOML file
- Add Animation and Timeline definitions
- If your scene / actor is also defined in XOML then you simply set the Timeline property to the name of the timeline and you are done. If not then continue onto the next step
- Search the scenes timelines for our named time line
- Set the actor or scenes time line using setTimeline()
- Call the time line's play() method to start the time line playing

13.9 Creating a Basic Animation in Code

IwGame supports a number of different types of animation as previously explained. In our example we are going to create a basic floating point animation that animates the rotation of an actor object.

```
// Create an animation
CIwGameAnim* anim = new CIwGameAnim();
anim->setName("TestAnim");
anim->setDuration(3);
anim->setType(CIwGameAnimFrame::FT_Float);

// Create and add frames
CIwGameAnimFrameFloat* frame = new CIwGameAnimFrameFloat();
frame->Time = 0;
frame->data = 0;
anim->addFrame(frame);
frame = new CIwGameAnimFrameFloat();
frame->Time = 1;
frame->data = 45;
anim->addFrame(frame);
frame = new CIwGameAnimFrameFloat();
frame->Time = 2;
frame->data = 90;
anim->addFrame(frame);

// Create a time line
CIwGameAnimTimeline* timeline = new CIwGameAnimTimeline();
timeline->setName("Timeline1");

// Create animation instance
CIwGameAnimInstance* instance = new CIwGameAnimInstance();
instance->setAnimation(anim);
instance->setDelay(0);
instance->setRepeatCount(0);
instance->setTarget(actor, "Angle");

// Play the timeline
timeline->play();
```

As you can see it is a little verbose, which is why we recommend using XOML mark-up for defining animations in particular.

13.9 Creating a Basic Animation in XOML

XOML is IwGame's mark-up language that can be used to define and set-up actors, scenes, resource, animations and other game elements. Here is an example showing how to set up the previous example using XOML syntax:

```
<xml>
  <Animation Name="TestAnim" Type="float" Duration="3" >
    <Frame Value="0" Time="0.0" />
    <Frame Value="45" Time="1.0" />
    <Frame Value="90" Time="2.0" />
  </Animation>
  <Timeline Name="Timeline1" AutoPlay="true">
    <Animation Anim="TestAnim" Target="Angle" Repeat="0" StartAtTime="0"/>
  </Timeline>
</xml>
```

As you can see it is incredibly easy to set up animations and time line's using XOML syntax.

13.10 Creating an Image Animation

We are now going to take a look at creating an image based animation that we can attach to an actor:

```
// Create an animation
CIwGameAnim* anim = new CIwGameAnim();
anim->setName("TestAnim");
anim->setDuration(0.8f);
anim->GenerateAtlasFrames(8, 36, 40, 0, 0, 512, 40, 512, 0.1f);

// Create a time line
CIwGameAnimTimeline* timeline = new CIwGameAnimTimeline();
timeline->setName("Timeline1");

// Create animation instance
CIwGameAnimInstance* instance = new CIwGameAnimInstance();
instance->setAnimation(anim);
instance->setDelay(0);
instance->setRepeatCount(0);
instance->setTarget(actor, "SrcRect");

// Play the timeline
timeline->play();
```

Setting up an image based animation is a little simpler because the `CIwGameAnim::GenerateAtlasFrames()` method() automates the generation of frames for us.

However in XOML will still need to list all frames:

```
<Animation Name="TestAnim" Type="rect" Duration="0.8" >
  <Frame Value="0, 0, 36, 40" Time="0.0" />
  <Frame Value="0, 40, 36, 40" Time="0.1" />
  <Frame Value="0, 80, 36, 40" Time="0.2" />
  <Frame Value="0, 120, 36, 40" Time="0.3" />
  <Frame Value="0, 160, 36, 40" Time="0.4" />
  <Frame Value="0, 200, 36, 40" Time="0.5" />
  <Frame Value="0, 240, 36, 40" Time="0.6" />
  <Frame Value="0, 280, 36, 40" Time="0.7" />
</Animation>
<Timeline Name="Timeline1" AutoPlay="true">
  <Animation Anim="TestAnim" Target="SrcRect" Repeat="0" StartAtTime="0"/>
</Timeline>
```

Helper tags will be added at a future date to automate the creation of atlas image animations and frames that follow patterns.

15.0 ClwGameAds – Wanna Make Some Money?

15.1 Introduction

On certain platforms, trying to get users to part with “the cost of less than a cup of coffee” for all your hard work is seemingly impossible. A popular alternative to paid apps are free ad supported apps, where the app is given away for free and revenue is generated by users clicking on ads displayed within the app. Before we get into the coding side of things there are a few important things you need to know about ads and the ad industry in general.

An ad's potential to make you money is measured in CPC (cost per click) or eCPM (cost per 1000 impressions). I prefer to deal in CPC as its a much clearer measuring stick to determine the value of particular ads and ad providers. Unfortunately the industry hides behind eCPM because it is vague and easy to mask, however its all we have to measure ad performance so we will speak in terms of eCPM.

eCPM is calculated based on how much you have or will have earned based on an average 1000 ads shown to users. Values can vary from as low as \$0.10 to as high as \$5.00. So as you can see it is going to take a LOT of impressions to make any decent money, but it is better than your app sat on some app store making nothing because users don't even know its there. Free apps tend to get much more exposure and users are much more likely to download them because they are free.

The next term you need to become familiar with is CTR (click through rate). This is the ratio of ads served to your app and the number of people that have actually clicked on your ads. So for example, lets say you get 10,000 ads served to your app in one day and 500 users click on ads, this will make you CTR $(500 * 100) / 10,000 = 5\%$. Click through rates can typically range from 0.1% to 10.0%, with the average being around 1%. At the moment ads running using IwGame's animating ads system are getting the following CTR's:

- Android – 3.2%
- iPhone – 8%
- iPad – 3%
- Samsung Bada - 4%

As you can see by adding some style to your ads you can vastly exceed the standard 1% CTR rate that most static ads obtain.

The last term that you should need to become familiar with is fill rate. This is the ratio of ad requests (termed requests) made from your app to ads actually sent back (termed impressions) to the user from their ad network. This varies based on the device running your app, the country and language of the user and the ad providers size and popularity. To calculate fill rate you divide the number of impressions by the number of requests. So for example, lets say you get 10,000 ad requests in one day but only 5,000 impressions then you fill rate is $(5000 * 100) / 10,000 = 50\%$. Average fill rates vary massively across platform, provider and country. iPhone and Android platforms tend to have very high fill rates (typically 90%+) because they are popular and everyone

wants to advertise their wares on these platforms. Platforms such as Samsung Bada and Blackberry Playbook experience much lower fill rates. Bada fill rates (can be as low as 20%,) simply because they aren't quite as popular.

One way to combat low fill rates is to use multiple ad providers. If one ad provider does not return an ad for your platform then request an ad from the next provider. (usually referred to as ad mediation) You may need to integrate 4-5 ad providers to ensure a 100% fill rate and maximise your apps earning potential on some platforms.

Ok, now that we have that out the way, lets take a look at IwGameAds.

IwGameAds is a cross platform unified ad API that allows you to make requests from a number of ad providers. It also goes one step further and provides a view that can display retrieved banner based ads and register clicks. IwGameAds provides the following functionality:

- Asynchronously request text, image and html based ads from a variety of ad providers
- Automatic extraction of ad data, such as image url, click url and text
- Provision for targetted ads by age, gender, location etc..
- Integrated ads view that can display animating ads to the user as well as detect clicks and launch the external URL / app that deals with the click destination
- Cache and display multiple ads from multiple providers
- Ad mediation using the priority based IwGameAdsMediator class

IwGameAds currently supports [Inner-active](#), [AdFonic](#), [VServ](#), [Mojiva](#), [Millennial Media](#) and [AdModa](#) ad providers. More will be added as they are confirmed as working.

CIwGameAds and CIwGameAdsView are both singletons that provide easy global access from anywhere in your game. CIwGameAds is not dependent upon CIwGameAdsView so can roll your own ad view system if you want to.

15.2 Setting up and Updating IwGameAds

If you are going to use the provided ads view then the process is setting up and updating IwGameAds is very simple as the view will take care of initialising and updating the ads system:

```
// Create ad view
CIwGameAdsView::Create();

// Initialise with Application ID (you get this from your ad provider)
IW_GAME_ADS_VIEW->Init("Your App ID");

// Set ad request interval in seconds
IW_GAME_ADS_VIEW->setNewAdInterval(30);

// Set ad provider (if you want ads to be automaticall collected at an interval)
IW_GAME_ADS_VIEW->setAdProvider(CIwGameAds::InnerActive);

// Set total number of ads visible in the ads view
IW_GAME_ADS_VIEW->setNumAdsVisible(1);
```

If you not want ads to be requested automatically at a set interval then you should not use `setNewAdInterval()`. Calling `setAdInterval(0)` will disable auto collection of ads.

And updating the view in your main loop would look like this:

```
// Update the ads view
IW_GAME_ADS_VIEW->Update(1.0f);
// draw the ads view
IW_GAME_ADS_VIEW->Draw();
```

Note that if you would like the ads view to scale to fit your scenes transform then you can pass the scene as a parameter to `Draw()`.

Finally cleaning up the ads view is done like this:

```
// Clean up the ads view
IW_GAME_ADS_VIEW->Release();
CIwGameAdsView::Destroy();
```

As you can see its a very simple 3 stage integration. If however you have decided to roll your own ads view then you will need to set up `CIwGameAds` yourself:

To initialise the ads system you should implement the following:

```
// Create ad view
CIwGameAds::Create();
IW_GAME_ADS->Init();
// Set the Application ID (you get this from your ad provider)
IW_GAME_ADS->setApplicationID(id);
```

You need to update the ads system each game loop using:

```
// Update the ads system
IW_GAME_ADS->Update();
```

And finally clean up the ad system when done with it using:

```
IW_GAME_ADS->Release();
CIwGameAds::Destroy();
```

15.3 Ad Types

CIwGameAdsView currently supports the rendering of image based banner ads only. It will not render text or html based ads. However you tell the ads system which types of ad you are willing to accept. You can tell the ad system to return these types of ads using:

```
void          setTextAds(bool text_ads)
void          setHtmlAds(bool html_ads)
```

Note that not all ad providers may offer this functionality so the system uses it more as a hint.

15.4 Requesting Ads

Requesting an ad using CIwGameAdsView is very simple as the following piece of code will show:

```
// Request an Inner-active ad
IW_GAME_ADS_VIEW->RequestNewAd(CIwGameAds::InnerActive);
```

Once the ad is available the ads view will display it to the user.

If you are rolling your own ad view then requesting ads looks very much the same.

```
// Request an Inner-active ad
IW_GAME_ADS->RequestAd(CIwGameAds::InnerActive);
```

The slightly more complicated process is detecting when the ad has arrived. The code below shows how to detect the arrival of a new ad and how to get at the ad information:

```
// Check to see if new ad has arrived
if (IW_GAME_ADS->isAdAvailable())
{
    CIwGameAds::eIwGameAdsError error = IW_GAME_ADS->getError();
    if (IW_GAME_ADS->getError() <= MinError)
    {
        // No error - You can now access the ads information
        CIwGameAd& ad = IW_GAME_ADS->getAd();
    }
    else
    {
        // Error occurred - error contains an error
    }

    // Allow the next ad request
    IW_GAME_ADS->setAdAvailable(false);
}
```

Your own implementation will also need to take care of displaying ads and responding to clicks.

15.5 Working with CIwGameAdsView

CIwGameAdsView is a convenient ad view that will retrieve ads, display them using animations and monitor and respond to ad clicks. We've already explained how to create an update an ads view, but we haven't yet covered some of the cooler features of the view. Here is a list of features:

- Can display multiple ads
- Can display ads anywhere on screen
- Can rotate and scale ads
- Can change the colour and opacity of ads
- Can hide and show ads
- Can animate ads

When you create the CIwGameAdsView you can set how many ads you would like the view to display using:

```
void setNumAdsVisible(int count)
```

You can also set the position, scale, rotation, colour and visibility of each separate ad as well add animators using the following methods:

```
void setVisible(int index, bool visible)
void setPosition(int index, int x, int y)
void setScale(int index, float scale)
void setAngle(int index, float angle)
void setColour(int index, int r, int g, int b, int a)
void addAnimator(int index, CIwGameAdsViewAnimator* animator)
```

Using the above methods you can provide a whole range of animations to your ads

15.6 Ad Animators

To test out our animating ads idea we created a basic class called CIwGameAdsViewAnimator. This class provides an easy way to quickly get ads animating. This class is provided mainly as an example showing how to animate ads yourself.

The ad view animator assumes that the ad is going to be in one of 4 states:

- AnimIn – The ad is coming into view
- AnimOut – The ads is going out of view
- AnimStay – The ad is at rest but visible to the user
- AnimDone – The ad is finished

The ad view will allow you to add animations for the in, out and stay phases.

Here's an example that shows a typical sweep in over one second, stay on screen for seven seconds and then sweep back out over one second.

```
// Create and attach an animator that fades the ad in over 1 second, pauses for 7
// seconds and then fades the ad back out
CIwGameAdViewAnimator* anim = new CIwGameAdViewAnimator();
anim->Init();
anim->setAdViewDataIndex(0);
anim->setCanvasSize(width, height);
anim->setInAnim(CIwGameAdViewAnimator::AnimFadeIn, 1000);
anim->setOutAnim(CIwGameAdViewAnimator::AnimFadeOut, 1000);
anim->setStayDuration(7000);
IW_GAME_ADS_VIEW->addAnimator(0, anim);
```

You can add multiple animations to the same ad as shown in the following example:

```
// Create and attach an animator that sweeps the ad in from the right the over 1.2
// seconds, pauses for 7 seconds and then sweeps back out
anim = new CIwGameAdViewAnimator();
anim->Init();
anim->setAdViewDataIndex(0);
anim->setCanvasSize(width, height);
anim->setRestingPosition(0, -height / 8);
anim->setInAnim(CIwGameAdViewAnimator::AnimRightSweepIn, 1200);
anim->setOutAnim(CIwGameAdViewAnimator::AnimRightSweepOut, 1200);
anim->setStayDuration(7000);
IW_GAME_ADS_VIEW->addAnimator(0, anim);

// Create and attach an animator that scales the ad in over 1.5 seconds, pauses for 7
// seconds and then scales back out
anim = new CIwGameAdViewAnimator();
anim->Init();
anim->setAdViewDataIndex(0);
anim->setCanvasSize(width, height);
anim->setInAnim(CIwGameAdViewAnimator::AnimScaleIn, 1500);
anim->setOutAnim(CIwGameAdViewAnimator::AnimScaleOut, 1500);
anim->setStayDuration(7000);
IW_GAME_ADS_VIEW->addAnimator(0, anim);

// Create and attach an animator that rotates the ad in over 1 second, pauses for 7
// seconds and then rotates back out
anim = new CIwGameAdViewAnimator();
anim->Init();
anim->setAdViewDataIndex(0);
anim->setCanvasSize(width, height);
anim->setInAnim(CIwGameAdViewAnimator::AnimSpinIn, 1000);
anim->setOutAnim(CIwGameAdViewAnimator::AnimSpinOut, 1000);
anim->setStayDuration(7000);
IW_GAME_ADS_VIEW->addAnimator(0, anim);
```

By combining animators you can produce some interesting effects.

15.7 CIwGameAdsMediator – The Art of Ad Mediation

Integrating a single ad provider into your product is not a great idea if you want to maximise your products earning potential. When ad providers give fill rate figures of over 90%, they leave out the fact that the fill rate they advertise is based upon their best performing platforms and countries. An ad provider will base eCPM on how many ads were delivering to your app and not how many ad requests were made from your app, which can be VERY misleading. Some platforms such as Bada have fill rates as low as 20%, whilst some countries such as China have even lower fill rates.

To get around this problem you need to use ad mediation. This is the process of going through a list of prioritised ad providers requesting an ad, if the ad provider does not provide an ad then request an ad from the next provider. Keep doing this until you get an ad that you can use.

IwGame provides automated ad mediation using the CIwGameAdsMediator. To use the mediator you create a CIwGameAdsmediator, populate it with ad party's then attach it to the CIwGameAds object as shown below:

```
// Create ad mediator and attach it to the main ad object
CIwGameAdsMediator* ad_mediator = new CIwGameAdsMediator();
IW_GAME_ADS->setMediator(ad_mediator);

// Create Inner-active ad party and add to the mediator
CIwGameAdsParty* party = new CIwGameAdsParty();
party->ApplicationID = "Your inner-active App ID";
party->Provider = CIwGameAds::InnerActive;
ad_mediator->addAdParty(party);

// Create AdModa ad party and add to the mediator
party = new CIwGameAdsParty();
party->ApplicationID = "You AdModa App ID";
party->Provider = CIwGameAds::AdModa;
ad_mediator->addAdParty(party);

// Create AdFonic ad party and add to the mediator
party = new CIwGameAdsParty();
party->ApplicationID = "Your AdFonic App ID";
party->Provider = CIwGameAds::AdFonic;
ad_mediator->addAdParty(party);
```

As you can see, incredibly simple. The process of attaching the ad mediator will ensure that the ad object will attempt to collect ads from all ad parties should previous parties fail to collect an ad. You do not need to make any changes to the ad view, the system is completely automated.

15.8 Supported Ad Providers

The following ad providers are currently supported:

- [Inner-active](#)
- [AdFonic](#)
- [Vserv](#) – Also provides support for [InMobi](#), [BuzzCity](#), [JumpTap](#), [ZestAdz / Komli Mobile](#) and [Inner-active](#)
- [Mojiva](#)
- [Millennial Media](#) – Also provides support for [AdMob](#), [Amobee](#), [JumpTap](#) and [Mojiva](#)
- [AdModa](#)

We have plans on adding support for the following ad providers:

- MobFox
- InMobi
- Madvertise

If you know of any additional ad providers that support a REST based API then please let us know at admin@pocketeers.co.uk

16.0 CIwGameXml – Cross Platform Data

16.1 Introduction

IwGame comes with a basic XML parser that has the following features:

- Load and save XML files
- Very small code footprint
- Very quick parser
- Uses memory pooling for tags, attributes and values, reducing memory fragmentation

CIwGameXml does however have limitations such as no support for XML schemas and will allow you to do some things that normal XML parsers will not allow you to do.

The XML engine is split into the following classes:

- CIwGameXmlAttribute – A nodes named attribute and value
- CIwGameXmlNode – A named node containing a collection of attributes
- CIwGameXmlParser – The main parser object that loads and parsers the XML file

16.2 Setting the Memory Pooling

Before you can use the Xml parser you must set-up the Xml parser memory pools. These pools pre-allocate space for a group of nodes, attributes and tags before parsing a file. Once the pools are set-up they can be re-used by all future parse calls. Below is an example showing how to set up the initial XML parser pools:

```
CIwGameXmlParser::PoolsInit();
```

This will allocate 2048 nodes, 2048 tags and 4096 attributes.

- Nodes – Represents actual XML nodes found in the document
- Tags – Represent the total number of actual XML tags found in document. The tag pool is used to pre-parse the XML document and speed up parsing
- Attributes – Represents the total number of actual XML node attributes found in the document

So if you have an XML document that contains 1000 total nodes and each node contains at most 2 attributes then you could pass the following values to PoolsInit():

```
CIwGameXmlParser::PoolsInit(1000, 1000, 2000);
```

16.2 Loading an XML file

To load an XML file, create an instance of CIwGameXmlParser and call Parse() to parse the data, like shown below:

```
// Reset XML parser pools memory
CIwGameXmlParser::PoolsReset();

// Load the xml file
CIwGameXmlParser* xml = new CIwGameXmlParser();
if (xml->Parse("./Scene1.xml") == eXMLParserError_None)
{
    // Save the xml file back out
    xml->GetRoot()->Save("test1.xml");
}
```

We firstly reset the XML parser pool memory, if we did not reset this then we would soon run out of pool memory space.

Next we create an instance of the XML parser object then call Parse() passing in the name of the XML file. If there was no parsing errors then we save the file back out to check that it worked.

Three versions of Parse() are available:

eXMLParserError	Parse(const char* filename);
eXMLParserError	Parse(CIwGameFile *file);
eXMLParserError	Parse(CIwGameDataInput* data);

These allowing parsing of a named file, an IwGame file and a data input stream.

16.3 Working with Nodes

The parser provides a number of useful methods that you can use to get nodes from the parsed data:

```
CIwGameXmlNode*      GetRoot()
CIwGameXmlNode*      GetFirstNamedNode(CIwGameXmlNode *parent, const char* node_name)
CIwGameXmlNodeList*  GetNamedNodes(CIwGameXmlNode *parent, const char* node_name)
```

- **GetRoot** – Returns the root node of the loaded XML data
- **GetFirstNamedNode** – Searches the complete XML node structure from the specified parent node and returns the first node who's name matches the supplied node name
- **GetNamedNodes** – Searches the complete XML node structure from the specified node and returns all nodes who's names match the supplied node name

Once you have a node you can begin querying the nodes value and attributes. To get the nodes value you can use the following methods:

```
const CIwGameString&  GetValue()
int                   GetValueAsInt() const;
float                 GetValueAsFloat() const;
bool                  GetValueAsBool() const;
```

For debug purposes you can also print the nodes attributes and complete hierarchy using the following methods:

```
void                  PrintAttributes();
void                  PrintTree();
```

You can search a nodes child nodes using the following methods:

```
CIwGameXmlNode*      GetFirstNode();
CIwGameXmlNode*      GetFirstNamedNode(const char* nodeName, unsigned int name_hash);
void                  GetNamedNodes(const char* nodeName, unsigned int name_hash,
CIwGameXmlNodeList* nodes);
```

Note that these methods take a hashed string value as node names instead of a string.

CIwGameXmlNode also provides methods for saving its structure to a file:

```
int          SaveAttributes(CIwGameFile* file);  
int          SaveTree(CIwGameFile* file);  
int          Save(const char* filename);
```

Querying attributes can be done using the following methods:

```
int          GetAttributeCount();  
CIwGameXmlAttribute* GetAttribute(const char* name);  
CIwGameXmlAttribute* GetAttribute(int index);
```

The string version of GetAttribute() will return the named attribute whilst the integer version returns the attribute at the specified index (0 index based)

And finally methods have been provided for building nodes:

```
void          SetName(const char* name, int len)  
void          SetValue(const char* value, int len)  
void          AddChild(CIwGameXmlNode* node)  
void          AddAttribute(CIwGameXmlAttribute *attribute)  
void          AddAttribute(CIwGameString& name, CIwGameString& value)  
void          AddAttribute(CIwGameString& name, const char* value)
```

16.4 Node and Attribute Iteration

CIwGameXmlNode provides iterator based access to both child nodes and attributes:

```
typedef CIwList<CIwGameXmlNode*>::iterator      _Iterator;
typedef CIwList<CIwGameXmlAttribute*>::iterator _AttribIterator;
_Iterator begin()
_Iterator end()
_AttribIterator attrs_begin()
_AttribIterator attrs_end()
```

These types and methods allow you to easily walk the nodes child nodes and attributes. Below is an example showing how to walk a nodes child nodes and each nodes attributes:

```
// Walk the child nodes
for (CIwGameXmlNode::_Iterator nodes = node->begin(); nodes != node->end(); ++nodes)
{
    // Walk the nodes attributes
    for (CIwGameXmlNode::_AttribIterator attrs = (*nodes)->attrs_begin();
    attrs != (*nodes)->attrs_end(); ++attrs)
    {
    }
}
```

16.5 Attribute Query

CIwGameXmlAttribute provides an extensive set of methods for querying attribute values and converting the data to different formats. Below is a list of all methods:

CIwGameString&	GetValue() { return Value; }
int	GetValueAsInt() const;
float	GetValueAsFloat() const;
bool	GetValueAsBool() const;
bool	GetValueAsPoint(CIwFVec2 &point);
bool	GetValueAsPoint3(CIwFVec3 &point);
bool	GetValueAsPoint4(CIwFVec4 &point);
bool	GetValueAsColour(CIwColour &colour);
bool	GetValueAsRect(CIwRect &rect);
CIwGameXmlStringList*	GetValueAsList();
int	GetValueAsListOfInt();
int	GetValueAsListOfFloat();
int	GetValueAsListOfBool();

The list value retrieval methods use a pooled memory system to reduce constant allocation and deallocation of memory, so please ensure that you store off the values retrieved into the pool buffers

before calling the list methods again or data will be written over. To see how the list pool buffers are used lets take a quick look at `GetValueAsRect()`:

```
bool CIwGameXmlAttribute::GetValueAsRect(CIwRect& rect)
{
    if (GetValueAsListOfInt() != 4)
    {
        return false;
    }
    rect.x = g_IntListPool[0];
    rect.y = g_IntListPool[1];
    rect.w = g_IntListPool[2];
    rect.h = g_IntListPool[3];

    return true;
}
```

As you can see, when we call `GetValueAsListOfInt()` a global buffer called `g_IntListPool` is filled with the values that are returned.

Global data is messy and we don't like it however so this will change in the future, but the implementation will remain the same.

16.6 Creating an XML file

XML is very useful when it comes to representing data in a platform independent manner. It's also very useful when it comes to serialising game state and other data to storage as it can be saved in a good structured format.

To create an XML file you need to create a root XML node then add further named child nodes that contain values and attributes that contain your data. Below shows a quick example:

```
// Reset XML pool data
CIwGameXmlParser::PoolsReset();

// Create root XML node
CIwGameXmlNode* root = CIwGameXmlParser::AllocNode();
root->SetName("xml", 3);

// Create and add a settings child node to the root
CIwGameXmlNode* settings_node = CIwGameXmlParser::AllocNode();
settings_node->SetName("Settings", 8);
root->AddChild(settings_node);

// Create and add a FullVersion node to the settings node
CIwGameXmlNode* value_node = CIwGameXmlParser::AllocNode();
value_node->SetName("FullVersion", strlen("FullVersion"));
value_node->SetValue("true", 4);
settings_node->AddChild(value_node);

// Save the XML file
settings_node->Save("./Settings.xml");
```

The above code will generate the following XML file:

```
<?xml version="1.0"?>
<Settings>
  <FullVersion>true</FullVersion>
</Settings>
```

Note that the above code also uses the XML memory pooling system so you do not need to worry about cleaning up after you are done creating the node structure.

17.0 CIwGameDataIO – Stream Style Access to Memory

17.1 Introduction

From the perspective of IwGame a stream is data that can be read from or written to. When data is read from a stream, the position at which we read the next lot of data moves to the end of the previously read data. Streams have a stream pointer / position which marks the place where new data will be read or written.

IwGame provides two simple classes that allow you to treat memory as though it was a stream. This can come in very useful when serialising binary data or parsing data located in a memory buffer.

Two classes are provided which provide support for input and output:

- CIwGameDataInput – Input stream access to a memory buffer
- CIwGameDataOutput – Output stream access to a memory buffer

17.2 Input Streams

To create an input stream we create a CIwGameDataInput object, initialise it then load some data into it:

```
// Create and initialise an input stream of 1024 bytes
CIwGameDataInput *stream = new CIwGameDataInput();
stream->Init(1024);

// Read some data into the stream
if (!file->Read(stream->getData(), size))
    return -1;

// Do something with the data
....
```

You can now access the stream using the following methods:

```
int        Skip(int nNumBytes);
int        SetPos(int nPos);
int        GetPos() const
int        GetLength() const
bool       IsEOF() const
int        Remaining() const
char       GetByte();
float      GetByteAsFloat();
char       GetUByte();
int        GetUByteAsInt();
float      GetUByteAsFloat();
```

```

    int      GetBytes(char* pData, int nCount);
    int      GetBytes(short* pData, int nCount);
    int      GetBytes(int* pData, int nCount);
    int      GetBytes(float* pData, int nCount);
    int      GetUBytes(char* pData, int nCount);
    int      GetUBytes(int* pData, int nCount);
    int      GetUBytes(float* pData, int nCount);
    int      GetChars(char* pData, int nCount);
    short    GetShort();
    short    GetShortSwab();
    float    GetShortAsFloat();
    int      GetUShort();
    float    GetUShortAsFloat();
    int      GetShorts(short* pData, int nCount);
    int      GetShorts(int* pData, int nCount);
    int      GetShorts(float* pData, int nCount);
    int      GetUShorts(short* pData, int nCount);
    int      GetUShorts(int* pData, int nCount);
    int      GetUShorts(float* pData, int nCount);
    int      GetInt();
    int      GetIntSwab();
    int      GetInts(int* pData, int nCount);
    int      GetInts(float* pData, int nCount);
    float    GetFloat();
    int      GetNextString(CIwGameString *pString);
    int      GetNextString(char *pString, int max_len);
    int      GetNextQuotedStringAsint(int *pNum);
    bool     GetNextQuotedString(CIwGameString *pString);
    bool     GetNextQuotedString(char *pString, int max_len);
    bool     GetNextOccuranceOf(char find);
    int      GetNextTag(char tag_start_char, char tag_end_char, int range, int
&start_pos);
    int      GetNextTagOrValue(char tag_start_char, char tag_end_char, int range,
int &start_pos);
    int      FindString(char* pString, int str_len) const;
    int      SkipToWhitespaceInTag();
    int      SkipToNoneWhitespaceInTag();
    int      SkipToCharInTag(char chr);
    int      GetLineNumber(int pos) const;
    int      CalculateChecksum();
    int      StripXMLComments();

```

As you can see there are a lot of methods for accessing streams, some of the less obvious methods are described below:

- Skip – Moves the stream data pointer
- SetPos / GetPos – Move / get the current stream pointer position
- IsEOF – Return true if the end of the stream was reached
- Remaining – Returns the total number of bytes remaining in the stream
- GetLineNumber – Counts the number of carriage returns to determine the approximate text line number of the current stream position

For an example of CIwGameDataInput usage, take a look at the source for CIwGameXml as this class uses streams extensively for parsing XML data.

17.3 Output Streams

To create an output stream we create a `CIwGameDataOutput` object, initialise it then write some data into it:

```
// Create and initialise an input stream of 1024 bytes
CIwGameDataOutput *stream = new CIwGameDataOutput();
stream->Init(1024);

// Write some data into the stream
stream->Set(10);           // Write an integer
stream->Set(1.2f);         // Write a floating point value
stream->Set("Hello", 5);   // Write some text
```

You can now access the stream using the following methods:

```
int      Skip(int nNumBytes);
int      SetPos(int nPos);
int      GetPos() const
int      GetLength() const
bool     IsEOF() const
int      Remaining() const
void     Set(char data);
void     Set(short data);
void     Set(int data);
void     Set(float data);
void     Set(char* pData, int nCount);
void     Set(short* pData, int nCount);
void     Set(int* pData, int nCount);
void     Set(float* pData, int nCount);
void     Set(CIwGameString* pString);
int      CalculateChecksum();
```

- **Set** – Writes data to the stream
- **CalculateChecksum** – Calculates a simple checksum for the entire stream (up to the current stream position)

18.0 CIwGameResource – Getting a Handle on the Beast

18.1 Introduction

The Marmalade SDK groups collections of resources into groups (CIwResGroup). You can find out more details about Marmalade resource groups and their creation at <http://www.drmp.com/index.php/2011/10/01/marmalade-sdk-tutorial-marmalades-resource-management-system/>.

IwGame provides a basic wrapper around a Marmalade resource group called CIwGameResourceGroup which allows groups to be loaded and destroyed when IwGame groups are instantiated and deleted. CIwGameResourceGroup also allows integration of groups into the XOML mark-up system.

CIwGameResource consists of a number of classes:

- CIwGameResourceGroup – A basic resource group
- CIwGameResourceGroupCreator – Class uses to instantiate a resource group from XOML
- CIwGameGlobalResources – A global resource manager, managing resource groups, animations, images and timelines on an application level

18.2 Creating a Resource Group

IwGame has two ways to create a resource group:

- Manually create a CIwGameXomlResourceGroup and assign it a name and filename then attach it to either the global CIwResourceManager or a scenes CIwGameXomlResourceManager
- Declare a ResourceGroup in XOML

To create a resource group manually:

```
// Create a resource group
CIwGameResourceGroup* group = new CIwGameResourceGroup();
group->setGroupName("Level1");
group->setGroupFilename("Level1.group");

// Add the resource group to the global resource manager
IW_GAME_GLOBAL_RESOURCES->getResourceManager()->addResource(group);
```

Note that at this point the resource group is only created and has not been loaded. You can call `CIwGameResourceGroup::Load()` to load the group into the Marmalade resource group manager system. Alternatively you can allow the group to be loaded on first access to the group. That is when the first call to `CIwGameResourceGroup::getResourceGroup()` is called.

To create a resource group from XOML we would declare the following:

```
<ResourceGroup Name="Level1" GroupFile="Level1.group" Preload="true" />
```

As you can see its much easier to declare resource groups using XOML. In addition, if you declare the resource group within scene tags then the resource group will become local to that scene, which will allow the resource group to be cleaned up when the scene is destroyed.

The `CIwGameResourceGroupCreator` class is responsible for instantiating a `CIwGameResourceGroup` from XOML.

18.3 Resource Management

Life is much easier when we leave the management of various classes to someone else. In this case we use the CIwGameXomlResourceManager to manage a collection of resource groups. Each scene that is created contains its own resource manager and the global resource system also contains one.

Any type of IIwGameXomlResource derived resource can be added to the resource manager, including Marmalade resource groups, images, animations, timelines, shapes, Box2dMaetrialas and custom resources.

To create your own resource type you should:

- Derive your class from IIwGameXomlResource
- Implement LoadFromXoml(IIwGameXomlResource* parent, bool load_children, CIwGameXmlNode* node);
- In your classes constructor set the class type name using setClassType() for example setClassType("ResourceGroup"); - Generally this represents the common base class for all objects of this type.
- Create a class creator derived from IIwGameXomlClassCreator

The resource manager supports the following methods:

```
void      addResource(IIwGameXomlResource* resource);
void      removeResource(IIwGameXomlResource* resource);
void      removeResource(unsigned int name_hash, unsigned int type_hash);
IIwGameXomlResource* findResource(unsigned int name_hash, unsigned int type_hash, bool
global_search = true);
IIwGameXomlResource* findResource(const char* name, unsigned int type_hash, bool
global_search = true);
IIwGameXomlResource* findResource(const char* name, const char* type, bool
global_search = true);
void      clearResources();
```

The resource type name is the name that you assigned to your class with setClassType()

18.4 Global Resources

CIwGameGlobalResources represents a collection of resources that are meant to be accessible across the entire game and for the life time of the game. A CIwGameGlobalResources contains a CIwGameXomlResourceManager which a list of resources of varying types.

These types of resources are used across the whole game engine and are globally available.

The CIwGameGlobalResources class is a singleton that is available via the IW_GAME_GLOBAL_RESOURCES macros.

To get to the contained resources you should call one of the following methods to gain access to its manager:

CIwGameXomlResourceManager*

getResourceManager()

19.0 ClwGameXOML – Welcome to the Easy Life

19.1 Introduction

I like to make code that is extensible but I also like to make life easier for myself without sacrificing too much versatility, so my coding style constantly strives towards balancing ease of use to extensibility. I've done a lot of Silverlight / WPF coding in the passed which is where I came across Microsoft's XAML mark-up language. I later came across Adobe Flex's MXML mark-u language and decided that mark-up language was the way to go with game and app development. Being able to define my whole UI, game scenes, animations and even game logic using simple readable XML seemed like the best way to go.

IwGame adds some of the great functionality found in XAML / MXML allowing you to declare much of what will appear in your game using XOML (XML Object Modelling Language).

The major advantages of using XOL include:

- Saves a whole bunch of typing
- Design and layout scenes and actors using mark-up
- Design resource groups and images using mark-up
- Design complex animations and time line animations
- No recompiling required to test changes
- Add new content without resubmitting to the app stores
- Add content that is streamed from a server

As you can see, we have some pretty neat advantages to switching over to using a mark-up language

Lets take a quick look at an example XOML file:

```
<?xml version="1.0"?>
<xml>
  <ResourceGroup Name="Audio" GroupFile="Audio.group" Preload="true" />
  <ResourceGroup Name="Level1" GroupFile="Level1.group" Preload="true" />
  <Image Name="Sprites" Location="Level1" Preload="true" />
  <Image Name="Buddy" Location="http://www.battleballz.com/bb_icon.gif" Preload="true"
Blocking="false" />
  <Animation Name="PlayerImageAnim" Type="rect" Duration="0.8" >
    <Frame Value="0, 0, 36, 40" Time="0.0" />
    <Frame Value="0, 40, 36, 40" Time="0.1" />
    <Frame Value="0, 80, 36, 40" Time="0.2" />
    <Frame Value="0, 120, 36, 40" Time="0.3" />
    <Frame Value="0, 160, 36, 40" Time="0.4" />
    <Frame Value="0, 200, 36, 40" Time="0.5" />
    <Frame Value="0, 240, 36, 40" Time="0.6" />
    <Frame Value="0, 280, 36, 40" Time="0.7" />
  </Animation>
```

```

    <Animation Name="SpinAnim1" Type="float" Duration="8" >
        <Frame Value="0" Time="0.0" />
        <Frame Value="90" Time="2.0" />
        <Frame Value="180" Time="4.0" />
        <Frame Value="270" Time="6.0" />
        <Frame Value="360" Time="8.0" />
    </Animation>
    <Animation Name="ScaleAnim1" Type="float" Duration="4" >
        <Frame Value="0.1" Time="0.0" />
        <Frame Value="1.0" Time="1.0" />
        <Frame Value="1.5" Time="2.0" />
        <Frame Value="1.6" Time="3.0" />
        <Frame Value="1.65" Time="4.0" />
    </Animation>
    <Animation Name="ColourAnim1" Type="vec4" Duration="4" >
        <Frame Value="255, 255, 255, 0" Time="0.0" />
        <Frame Value="255, 255, 255, 200" Time="1.0" />
        <Frame Value="255, 255, 255, 255" Time="2.0" />
        <Frame Value="255, 255, 255, 200" Time="3.0" />
        <Frame Value="255, 255, 255, 0" Time="4.0" />
    </Animation>
    <Animation Name="PlayerStates" Type="string">
        <Frame Value="State1" Time="0" />
        <Frame Value="State2" Time="0.5" />
        <Frame Value="State3" Time="1.0" />
        <Frame Value="State4" Time="1.5" />
    </Animation>
    <Timeline Name="Scene1Anim" AutoPlay="true">
        <Animation Anim="SpinAnim1" Target="Angle" Repeat="1" StartAtTime="0"/>
        <Animation Anim="ScaleAnim1" Target="Scale" Repeat="1" StartAtTime="0"/>
    </Timeline>

    <Scene Name="GameScene" CanvasSize="320, 480" FixAspect="true" LockWidth="false"
Current="true" Colour="128, 128, 128, 255" Timeline="Scene1Anim">
        <Timeline Name="Player1Intro" AutoPlay="true">
            <Animation Anim="PlayerImageAnim" Target="SrcRect" Repeat="0" StartAtTime="0"/>
            <Animation Anim="SpinAnim1" Target="Angle" Repeat="4" StartAtTime="10"/>
            <Animation Anim="ColourAnim1" Target="Colour" Repeat="10" StartAtTime="2"/>
        </Timeline>
        <TestActor Name="Player1" Position="-100, 0" Size="100, 100" Angle="45" SrcRect="0,
0, 36, 40" Image="Sprites" Timeline="Player1Intro" />
        <TestActor Name="Player2" Position="0, 0" Size="100, 100" Angle="-45" SrcRect="0, 0,
36, 40" Image="Sprites" />
        <TestActor Name="Player3" Position="0, 100" Size="100, 100" Angle="0" SrcRect="0, 0,
64, 64" Image="Buddy" />
    </Scene>

    <Scene Name="GameScene2" CanvasSize="320, 480" FixAspect="true" LockWidth="false"
Colour="0, 0, 255, 255" AllowSuspend="false">
        <Timeline Name="Player1Intro2" AutoPlay="true">
            <Animation Anim="PlayerImageAnim" Target="SrcRect" Repeat="0" StartAtTime="1"/>
            <Animation Anim="SpinAnim1" Target="Angle" Repeat="4" StartAtTime="10"/>
        </Timeline>
        <TestActor Name="Player1" Position="0, 0" Size="100, 100" Angle="45" SrcRect="0, 0,
36, 40" Image="Sprites" Timeline="Player1Intro2" />
        <TestActor Name="Player2" Position="100, 0" Size="100, 100" Angle="-45" SrcRect="0,
0, 36, 40" Image="Sprites" />
        <TestActor Name="Player3" Position="100, 100" Size="100, 100" Angle="0" SrcRect="0,
0, 64, 64" Image="Buddy" />
    </Scene>

```

```
</xml>
```

It may look like quite a bit of XML but the above would take many hundreds of lines of code to instantiate all of these classes and set up their data.

XOML is also extensible in that you can provide your own custom tags.

19.2 Loading a XOML file

Loading a XOML file is incredibly simple:

```
// Load a test XOML file  
IW_GAME_XOML->Process(this, "Scene1.xml");
```

The first parameter to the Process() method represents the parent class that you want to load all of the XOML data into. For example, if you load a XOML file that contains scenes then you should pass the CIwGame derived game object so that the created scenes will be added to the main game object. On the other hand if the XOML file contains a simple list of global resources then you can pass NULL. This kind of versatility allows you to split your game definitions across multiple files.

Its worth noting at this point that you need to ensure that IwGame::Init() has been called before using the XOML system as IwGame::Init() sets up important systems that are used by the XOML system.

19.3 Types of XOML Data

19.3.1 Resource Groups

Resource groups provide grouped resources to the game engine. A resource group is basically a Marmalade resource group with a unique name that the engine can use to identify it. Here are a few examples of declaring a resource group in XOML:

```
<ResourceGroup Name="Audio" GroupFile="Audio.group" Preload="true" />
<ResourceGroup Name="Level1" GroupFile="Level1.group" Preload="true" />
```

Valid attributes:

- Name – The name of the resource group
- GroupFile – The name of the resource group file
- Preload – If true then the resource group will be loaded into the Marmalade resource manager as soon as it is declared. If false then the resource group will be loaded on first access.

A resource group can be declared inside a scene or outside a scene. Scene declared resources will be local to the scene and objects within that scene.

19.3.2 Images

In order to use an image (even if stored inside a resource group) we must first declare it in XOML:

```
<Image Name="Sprites" Location="Level1" Preload="true" />
```

Or for a web based image:

```
<Image Name="Buddy" Location="http://www.battleballz.com/bb_icon.gif" Preload="true"
Blocking="false" />
```

Valid attributes:

- Name – Image name
- Location – The web address of the image for web based images or the group name that contains the image
- Preload – If true then the image is loaded as soon as it is instantiated, otherwise it is loaded on first access
- Blocking – If true then the image will be loaded and main loop processing will stop until the image is loaded. If false then the image will be loaded asynchronously. Sprites that use unloaded image will not be displayed until the image has been fully loaded.

Images can be declared inside a scene or outside a scene. Scene declared images will be local to the scene and objects within that scene.

19.3.3 Animations

IwGame's animation system is very powerful but simple to use when couple with XOML. An example some of the types of animation that you can create declaratively is shown below:

```
<Animation Name="PlayerImageAnim" Type="rect" Duration="0.8" >
  <Frame Value="0, 0, 36, 40" Time="0.0" />
  <Frame Value="0, 40, 36, 40" Time="0.1" />
  <Frame Value="0, 80, 36, 40" Time="0.2" />
  <Frame Value="0, 120, 36, 40" Time="0.3" />
  <Frame Value="0, 160, 36, 40" Time="0.4" />
  <Frame Value="0, 200, 36, 40" Time="0.5" />
  <Frame Value="0, 240, 36, 40" Time="0.6" />
  <Frame Value="0, 280, 36, 40" Time="0.7" />
</Animation>
<Animation Name="SpinAnim1" Type="float" Duration="8" >
  <Frame Value="0" Time="0.0" />
  <Frame Value="90" Time="2.0" />
  <Frame Value="180" Time="4.0" />
  <Frame Value="270" Time="6.0" />
  <Frame Value="360" Time="8.0" />
</Animation>
<Animation Name="ColourAnim1" Type="vec4" Duration="4" >
  <Frame Value="255, 255, 255, 0" Time="0.0" />
  <Frame Value="255, 255, 255, 200" Time="1.0" />
  <Frame Value="255, 255, 255, 255" Time="2.0" />
  <Frame Value="255, 255, 255, 200" Time="3.0" />
  <Frame Value="255, 255, 255, 0" Time="4.0" />
</Animation>
<Animation Name="PlayerStates" Type="string">
  <Frame Value="State1" Time="0" />
  <Frame Value="State2" Time="0.5" />
  <Frame Value="State3" Time="1.0" />
  <Frame Value="State4" Time="1.5" />
</Animation>
```

To create an animation we open an Animation tag, set the animations attributes then add Frame tags that declare the actual animation frames that make up the animation.

Valid Animation attributes:

- Name – The name of the animation
- Type – The type of animation data, currently supports:
 - bool - Boolean
 - float - Floating point
 - vec2 - 2D vectors
 - vec3 - 3D Vectors
 - vec4 - 4D vectors
 - rect - Rectangles (no interpolation between frames and used mainly by the image animation system for atlas frame based animations)
 - string - Strings

- Duration – The length of the animation in seconds (floating point)

Animations can be declared inside a scene or outside a scene. Scene declared animations will be local to the scene and objects within that scene.

19.3.4 Timeline

Time lines are IwGame's way of grouping together many different types of animations into a time line, then applying them to specific properties of specific objects, providing a robust and powerful animating scenes and actors. An example time line is shown below:

```
<Timeline Name="Player1Intro" AutoPlay="true">
  <Animation Anim="PlayerImageAnim" Target="SrcRect" Repeat="0" StartAtTime="0" />
  <Animation Anim="SpinAnim1" Target="Angle" Repeat="4" StartAtTime="10" />
  <Animation Anim="ColourAnim1" Target="Colour" Repeat="10" StartAtTime="2" />
</Timeline>
```

To create a time line we open an Ttimeline tag, set the timeline's attributes then add Animation tags that declare the actual animations that make up your time line.

Once declared a time line can be attached to any object that supports time line's (a scene or actor for example)

Valid timeline attributes:

- Name – The name of the timeline
- AutoPlay – If true then the animation will start playing as soon as it is loaded. If false then you must set the time line playing manually

Valid child animation attributes:

- Anim – The name of the previously declared animation
- Target – The target element that the animation will modify such as Position, Angle, Velocity etc..
- Repeat – The total number of times to repeat the animation (0 represents play forever)
- StartAtTime – Allows the animation to be delayed, value provided is in seconds (floating point)

19.3.5 Actors

Actors are the back bone of the IwGame engine so it seems fitting that they should be included in the XOML system. There is a small caveat when it comes to declaring actors in XOML and that is that you cannot directly declare an actor or image actor because they abstract class types and cannot be instantiated. Instead you create your own actor derived from one of the actor classes then add it to the XOML system so you can declare it in XOML. Lets say for example that you have already added two types of actors to the XOML system, you can then declare instances of those actors inside a scene like this:

```
<Scene Name="GameScene">
  <PlayerActor Name="Player1" Position="-100, 0" Size="100, 100" Angle="45"
SrcRect="0, 0, 36, 40" Image="Sprites" Timeline="Player1Intro" />
  <AlienActor Name="Baddy1" Position="0, 0" Size="100, 100" Angle="-45" SrcRect="0, 0,
36, 40" Image="Sprites" />
  <AlienActor Name="Badedy" Position="0, 100" Size="100, 100" Angle="0" SrcRect="0, 0,
64, 64" Image="Buddy" />
</Scene>
```

In the above XOML we declared a basic scene then declared 3 actors, one PlayerActor and two AlienActors. See section 4.17 for a walk-through on how to create a custom Actor that you can declare using XOML.

Valid CIwGameActor derived attributes:

- Name – Name of the scene (string)
- Type – A numerical type that can be used to identify the type of this actor (integer)
- Position– Position in the scene (x, y 2d vector)
- Velocity – Initial velocity of the actor (x, y 2d vector)
- VelocityDamping – The amount to dampen velocity each frame (x, y 2d vector)
- Angle – The orientation of the actor (float)
- AngularVelocity – The rate at which the orientation of the actor changes (float)
- AngularVelocityDamping – The amount of rotational velocity damping to apply each frame (float)
- Scale – The scale of the actor (float)
- Colour – The initial colour of the actor (r, g, b, a colour)
- Layer – The scenes visible layer that the actor should appear on (integer)
- Active – Initial actor active state (boolean)
- Visible – Initial actor visible state (boolean)
- Collidable – Collidable state of actor (boolean)
- CollisionSize – The circular size of the actor (float)
- CollisionRect – The rectangular collision area that the actor covers (x, y, w, h rect)
- WrapPosition – Of truethen the actor will wrap at the edges of the canvas (boolean)
- Timeline – The time line that should be used to animate the actor
- Box2dMaterial – Sets the physical material type used by the Box2D actor
- Shape – Box2D fixture shape for the Box2D actor
- COM – Centre of mass of Box2D body (x, y 2d vector)

- Sensor – Can be used to set the Box2D actor as a sensor (boolean)
- CollisionFlags – The Box2D body collision flags (category, mask and group)

Additional attributes available to actors derived from CIwGameActorImage:

- Image – The image that is to be used as the actors visual (string)
- Size – The on screen visible size of the actor (x, y 2d vector)
- SrcRect – The position and source of the source rectangle in the image atlas (x, y, w, h rect).
Used for panning the portion of a sprite atlas shown allowing frame based animation.

19.3.6 Scenes

XOML is very powerful system to the point that you can declare and instantiate complete scenes from XOML including all of their resources and actors without even writing a single line of code. Below is an example showing declaration of a couple of scenes using XOML:

```
<Scene Name="GameScene" CanvasSize="320, 480" FixAspect="true" LockWidth="false"
Current="true" Colour="128, 128, 128, 255" Timeline="Scene1Anim">
  <Timeline Name="Player1Intro" AutoPlay="true">
    <Animation Anim="PlayerImageAnim" Target="SrcRect" Repeat="0" StartAtTime="0"/>
  </Timeline>
  <PlayerActor Name="Player1" Position="-100, 0" Size="100, 100" Angle="45"
SrcRect="0, 0, 36, 40" Image="Sprites" Timeline="Player1Intro" />
  <AlienActor Name="Baddy1" Position="0, 0" Size="100, 100" Angle="-45" SrcRect="0, 0,
36, 40" Image="Sprites" />
  <AlienActor Name="Baddy2" Position="0, 100" Size="100, 100" Angle="0" SrcRect="0, 0,
64, 64" Image="Buddy" />
</Scene>

<Scene Name="GameScene2" CanvasSize="320, 480" FixAspect="true" LockWidth="false"
Colour="0, 0, 255, 255" AllowSuspend="false">
  <Timeline Name="Player1Intro2" AutoPlay="true">
    <Animation Anim="PlayerImageAnim" Target="SrcRect" Repeat="0" StartAtTime="1"/>
    <Animation Anim="SpinAnim1" Target="Angle" Repeat="4" StartAtTime="10"/>
  </Timeline>
  <TestActor Name="Player1" Position="0, 0" Size="100, 100" Angle="45" SrcRect="0, 0,
36, 40" Image="Sprites" Timeline="Player1Intro2" />
  <TestActor Name="Player2" Position="100, 0" Size="100, 100" Angle="-45" SrcRect="0,
0, 36, 40" Image="Sprites" />
  <TestActor Name="Player3" Position="100, 100" Size="100, 100" Angle="0" SrcRect="0,
0, 64, 64" Image="Buddy" />
  <MyActor Name="Player3" Position="100, 100" Size="100, 100" NumberOfYes="3" />
</Scene>
```

Valid scene attributes:

- Name – Name of the scene (string)
- CanvasSize – The virtual canvas size of the screen (x, y 2d vector)
- FixAspect – Forces virtual canvas aspect ratio to be fixed (boolean)
- LockWidth – Forces virtual canvas to lock to width if true, height if false (boolean)
- Extents – A rectangular area that describes the extents of the scenes world (x, y, w, h rect)
- AllowSuspend – Determines if the scene can be suspended when other scenes are activated (boolean)
- Clipping – A rectangular area that represents the visible area of the scene (x, y, w, h rect)
- Active – Initial scene active state (boolean)
- Visible – Initial scene visible state (boolean)
- Layers – The number of visible layers that the scene should use (integer)
- Colliders – The maximum number of colliders that the scene should support (integer)
- Current - If true then the scene is made the current scene (boolean)
- Colour – The initial colour of the scene (r, g, b, a colour)
- Timeline – The time line that should be used to animate the scene

- Gravity – Box2D directional world gravity (x, y 2d vector)
- WorldScale – Box2D world scale (x, y 2d vector)

19.3.7 Shapes

XOML provides the ability to create shapes of a variety of types, below are some examples:

Below shows an example of creating a box shape called floor with a width of 320 and a height of 20:

```
<Shape Name="Floor" Type="box" width="320" height="20" />
```

Below shows an example of creating a circle shape called Alien with a radius of 100 units:

```
<Shape Name="Alien" Type="circle" radius="100" />
```

Below shows an example of creating an arbitrary shaped polygon called Platform1 that consists of 5 points:

```
<Shape Name="Platform1" Type="polygon">  
  <Point Value="-100, -100" />  
  <Point Value="-100, -200" />  
  <Point Value="100, -100" />  
  <Point Value="100, 100" />  
  <Point Value="-100, 100" />  
</Shape>
```

Shapes can be used for any purpose but mainly for providing body shapes to actors that use the Box2D physics engine.

19.3.8 Box2dMaterials

Box2dMaterial's allow you to specify the physical properties of an actor that is under control of the Box2D physics engine. Below shows a few examples:

```
<Box2DMaterial Name="Floor" Type="static" Density="1.0" Friction="1.0" Restitution="0.5"
/>
<Box2DMaterial Name="BounceyBall" Type="dynamic" Density="1.0" Friction="0.9"
Restitution="0.1" />
```

Supported attributes include:

- Name – Name of the physics material
- Type – Type of physics material (values can be static, dynamic and kinematic)
- Density – The objects density (default is 1.0)
- Friction – The coefficient of friction (default is 1.0)
- Restitution - The coefficient of restitution (default is 0.1)

19.4 XOML Workflow

How is XOML meant to be used? Well that's up to you, I'm pretty sure you already have hundreds of cool ideas bouncing around your mind right now. That said we do have some recommended usage and work flow ideas.

The idea is to define all of your different types of actors and / or scenes derived from IwGame base scenes and actors then add the creators to the XOML system. This allows you to declare them in XOML.

You should very few if not a single global resource XOML file that contains all of your global resources. This gets loaded first when you boot the game.

You then create separate XOML files that contain either single scenes or scenes grouped by functionality.

Try to keep scene specific resources within scenes so they can be freed up when the scene is no longer needed, but try to balance resource group loading times with resource re-use.

The animation system provided by XOML is very powerful so try to use it wherever appropriate.

Start looking at your game as being driven by XOML rather than XOML being a simple data format.

19.5 XOML's future

XOML was initially designed to simplify the mundane and time consuming tasks involved in creating scene layouts and animations etc, but luckily it turned into something much more. Over time XOML will evolve into even more with future with plans for the following additions:

- Data binding
- Declaration of user interfaces
- Support for actions / events and action targets
- Support for styling
- More IwGame class support such as support for video, data files, camera streaming, audio time lines and more..

20 Box2D Physics – Lets Bounce

20.1 Introduction

I need to admit that physics is fun. Didn't quite think like that at school, but games that use physics simulation to control game objects and the environment seem to have that cool factor that almost always wow gamers. Take a look at the likes of Angry Birds to see how adding a little physics can really bring a game to life. I would hazard a guess that without Box2D style physics, games such as Angry Birds would be not have been anywhere near as popular.

As of v0.28 IwGame supports Box2D physics. Actors that are set up with a collision shape and physics material will be put under control of the scenes Box2D world controller.

Actors support a number of physical attributes including:

- Physical shape (defined by a CIwGameShape)
- Physical material (defined by a CIwGameBox2dMaterial)

Scenes currently support:

- World scale – This is the world to pixel scaling factor required by Box2D as Box2D does not work in pixel coordinates
- Gravity – Direction and speed of gravity

Generally you will declare a scene in XOML, set the Gravity and WorldScale attributes then add shapes and Box2dMaterial's that define the shape and physical properties of your game actors. Finally you will add actors to the scene that reference the shapes and materials declared earlier. Example XOML is shown below:

```
<Scene Name="GameScene3" CanvasSize="320, 480" FixAspect="true" LockWidth="false" Colour="255, 255, 255, 255"
AllowSuspend="false">
  <Box2dMaterial Name="Solid" Type="static" Density="1.0" Friction="1.0" Restitution="0.1" />
  <Box2dMaterial Name="BounceyBall" Type="dynamic" Density="1.0" Friction="0.9" Restitution="0.6" />
  <Shape Name="PlayerShape" Type="box" width="60" height="60" />
  <Shape Name="Wall" Type="box" width="20" height="320" />
  <Shape Name="Floor" Type="box" width="320" height="20" />
  <Timeline Name="Player1Intro1" AutoPlay="true">
    <Animation Anim="PlayerImageAnim" Target="SrcRect" Repeat="0" StartAtTime="1"/>
  </Timeline>
  <TestActor Name="Floor" Position="0, 200" Size="320, 20" Angle="0" SrcRect="0, 0, 32, 32" Image="Block" Shape="Floor"
Box2dMaterial="Solid" CollisionFlags="1, 1, 1" />
  <TestActor Name="LeftWall" Position="-140, 0" Size="20, 480" Angle="0" SrcRect="0, 0, 32, 32" Image="Block"
Shape="Wall" Box2dMaterial="Solid" CollisionFlags="1, 1, 1" />
  <TestActor Name="RightWall" Position="140, 0" Size="20, 480" Angle="0" SrcRect="0, 0, 32, 32" Image="Block"
Shape="Wall" Box2dMaterial="Solid" CollisionFlags="1, 1, 1" />
  <TestActor Name="Player1" Position="-50, -200" Size="60, 60" Angle="0" SrcRect="0, 0, 36, 40" Image="Sprites"
Timeline="Player1Intro1" Shape="PlayerShape" Box2dMaterial="BounceyBall" CollisionFlags="1, 1, 1" />
  <TestActor Name="Player2" Position="0, -150" Size="60, 60" Angle="0" SrcRect="0, 0, 36, 40" Image="Sprites"
Timeline="Player1Intro1" Shape="PlayerShape" Box2dMaterial="BounceyBall" CollisionFlags="1, 1, 1" />
  <TestActor Name="Player3" Position="50, -200" Size="60, 60" Angle="0" SrcRect="0, 0, 36, 40" Image="Sprites"
Timeline="Player1Intro1" Shape="PlayerShape" Box2dMaterial="BounceyBall" CollisionFlags="1, 1, 1" />
  <TestActor Name="Player4" Position="0, -1000" Size="60, 60" Angle="0" SrcRect="0, 0, 36, 40" Image="Sprites"
Timeline="Player1Intro1" Shape="PlayerShape" Box2dMaterial="BounceyBall" CollisionFlags="1, 1, 1" />
</Scene>
```

In the above scene we declare two materials, one for our solid environment pieces (floor and walls) and then another for our bouncy players.

Next we define a shape for our player, a shape for our floor and a shape for our walls.

Lastly, we create actors that represent our floor, left wall, right wall and 4 player actors, assigning the correct shapes and materials.

20.2 Box2dMaterial's

Box2dMaterial's allow you to specify the physical properties of an actors body that is under control of the Box2D physics engine. A Box2D material is represented by the CiwGameBox2dMaterial class and supports the following methods:

<code>void</code>	<code>setBodyType(b2BodyType type)</code>
<code>b2BodyType</code>	<code>getBodyType()</code>
<code>void</code>	<code>setDensity(float density)</code>
<code>float</code>	<code>getDensity()</code>
<code>void</code>	<code>setFriction(float friction)</code>
<code>float</code>	<code>getFriction()</code>
<code>void</code>	<code>setRestitution(float restitution)</code>
<code>float</code>	<code>getRestitution()</code>

These methods allow you to set the type of physical body, density and coefficients of friction and restitution

20.3 Box2D World

All objects that are under control of the Box2D physics system require attachment to the Box2D world. The Box2D world is represented by the `CiwGameBox2dWorld` class.

This class contains the actual Box2D `b2World` object as well as a few additional useful properties:

```

    b2World*      World;                // Physical world
    CIwFVec2      WorldScale;           // Scaling between physical and viusal
worlds (set to 0, 0 to disable physics update)
    int           VelocityIterations;   // Number of internal iterations used when
computing velocities
    int           PositionIterations;   // Number of internal iterations used when
computing positions
    CIwFVec2      Gravity;              // Scene gravity

```

These properties can be modified and queried using the following methods:

```

void           setWorldScale(float x, float y)
CIwFVec2       getWorldScale()
b2World*       getWorld()
void           setVelocityIterations(int count)
int            getVelocityIterations()
void           setPositionIterations(int count)
int            getPositionIterations()
void           setGravity(float x, float y);
CIwFVec2       getGravity()

```

In addition methods are provided for converting between Box2D world and IwGame scene canvas pixel coordinates:

```

float          WorldToPixelX(float x)
float          WorldToPixelY(float y)
float          PixelToWorldX(float x)
float          PixelToWorldY(float y)

```

To initialise a Box2D world you call `InitWorld()` and to update the Box2D world each frame you call `UpdateWorld()`.

20.4 Box2D Bodies

A Box2D body is a basically the physical representation of an object in the Box2D world. Bodies contain fixtures of various shapes and sizes that have particular attributes such as friction, density and restitution.

A Box2D body is defined by the `CiwGameBox2dBody` class which contains the `b2Body` object, a single fixture (multiple fixtures will be supported at a later date) a material and a shape.

To create a body you call the `InitBody()` method:

```
void InitBody(CiwGameBox2dWorld* world, CIwGameShape* body_shape,
CIwGameBox2dMaterial* body_mat, CIwFVec2* pos, float angle, float com_x, float com_y);
```

- world – The box2d world
- body_shape – The shape of the body
- body_mat – The material of the body
- pos – Position of the body in the Box2D world
- angle – Rotation of the body in the Box2D world
- com_x – Centre of mass x value
- com_y – Centre of mass y value

All co-ordinates are in pixels

A number of additional functions are included that allow you to apply various forces to the body:

```
void ApplyForce(float force_x, float force_y, float world_pos_x, float
world_pos_y);
void ApplyForceToCenter(float force_x, float force_y);
void ApplyTorque(float torque);
void ApplyLinearImpulse(float impulse_x, float impulse_y, float world_pos_x,
float world_pos_y);
void ApplyAngularImpulse(float impulse);
```

All values are in Box2D world coordinates

20.5 Box2D Collision

The CIwGameBox2dCollidable class manages collision begin and end events and tracks a list of all other Box2D objects that are currently in contact with the object.

Objects that implement CIwGameBox2dBody will automatically get this collision handling functionality as CIwGameBox2dBody is derived from a CIwGameBox2dCollidable which enables it to receive begin and end collision contact events.

When a collision occurs between two objects the Box2D collision system will call addCollisionStarted() adding each of the objects to each others collision started list. When a collision stops the system will call addCollisionEnded() which will remove the corresponding objects from each others collision started lists and add each of the objects to each others collision ended list. Iterators have been provided to walk the list of collision started and ended lists:

```
typedef CIwList<CIwGameBox2dCollidable*>::iterator _CollisionIterator;  
_CollisionIterator      started_begin()  
_CollisionIterator      started_end()  
_CollisionIterator      ended_begin()  
_CollisionIterator      ended_end()  
_CollisionIterator      incontact_begin()  
_CollisionIterator      incontact_end()
```

These allow you to check the current collision started and ended lists to check if a collision has taken place or stopped taking place.

Methods have also been provided to clear the collision lists:

```
void      clearCollisionsStarted()  
void      clearCollisionsEnded()  
void      clearAllCollisions()
```