AppEasy

Mobile Game and App Development the Easy Way

Developed and maintained by Pocketeers Limited (http://www.pocketeers.co.uk). For community support please visit http://www.appeasymobile.com

Version 1.5.0

Table of Contents

1.0 Introduction

1.1 What is AppEasy?

AppEasy is a cross platform game and application development system that enables developers to rapidly create applications for desktop and mobile devices easily using a mixture of XML mark-up language (XOML) and Lua. No compilers, compiling, or IDE's or sophisticated programming knowledge is required to develop games and applications using AppEasy. Applications created with AppEasy will run across al supported platforms without any changes, so no maintaining multiple code bases. You can think of AppEasy development as very similar to HTML5 / Javascript, Silverlight or Adobe Flex development but utilising full native acceleration and an API that is specific to game and app development.

1.2 What is XOML?

XOML is an XML based language developed by Pocketeers Limited to facilitate rapid production of games and apps across different platforms, particularly in the mobile field. XML is a standard human readable data interchange format that is used widely to exchange documents that contain structured data.

XOML is a lot more than a simple XML format however, it is what we at Pocketeers like to call "active XML". Active XML is XML with many more features, in particular features that make the document functional.

XOML documents instead of representing simple data represent a whole host of additional elements such as scenes, game objects, resources, physics materials, user interface elements, variables, bindings, programs, commands, event's, actions and more.

What does XOML look like? Lets take a look at a few XOML snip its to find out (don't worry if you do not understand any of the following, more information will be provided next):

```
<!-- Create a text label -->
<Label Position="10, 10" Text="Hello World" Font="trebuchet" />
```

The above example creates a text label that displays the phrase "Hello World" using the trebuchet font.

```
<!-- Create a variable -->
<Variable Name="my_variable" Type="float" Value="15" />

<!-- Create an array variable -->
<Variable Name="my_variable" Type="arrayint" Size="5" Value="1, 2, 3, 4, 5" />
```

The above example show how to create two variables. The first is a simple floating point (decimal) number which is assigned the value of 15. The second example shows how to create an array of numbers and fills that array with the values of 1, 2, 3, 4, 5

```
<!-- Create an animation -->
<Animation Name="Animation1" Duration="5" Type="vec2">
    <Frame Value="0, 0" Time="0" />
    <Frame Value="50, 50" Time="2.5" />
    <Frame Value="0, 0" Time="5" />
</Animation>
```

The above example shows a slightly more complicated example that creates an animation, notice how the animation frames are defined inside the Animation tag

If you have ever used Microsoft XAML (used in Silverlight and WPF) or Adobe MXML then you will feel quite at home with XOML. If not then the ins and outs of XOML shall soon become clear.

How do you pronounce XOML? XOML is pronounced as zommel.

1.3 How To Write XOML

We have a rough idea of what XOML is from the previous chapter. Now lets take a look at how to create XOML.

Firstly lets break down a simple XOML statement into its constituent parts and examine each part closer. We will take a look at our first previous example that involves creating a text label that we can see on screen:

```
<!-- Create a text label -->
<Label Position="10, 10" Text="Hello World" Font="trebuchet">
```

The first line contains a comment because it is surrounded by `<!-- -->`
Any text within a XOML document that is surrounded by comment markers will be ignored and classed as a simple comment for the reader. Comments are used to add more information to parts of the XOML file to help the reader understand what is going on.

The second line of our above example contains a "tag" along with a number of "attributes" (we also refer to attributes as properties) and "values". All XML documents consist of a collection of tags, a tag represents a specific piece of data and always begins with an open tag sign "<" and ends with a close tag sign ">" as shown below:

```
<i_am_a_tag>
```

Also note that all tags must include a closing tag to let the reader know that it has come to the end of that tag definition. A closing tag prepends the "/" character to the closing tag > sign. Lets take a look how this looks:

```
<i_am_a_tag>        -- Open tag
</i_am_a_tag>       -- Close tag
```

Tags are generally enclosed within other tags to form a hierarchical document, for example:

```
<Person>
    <Name>Bob Doe</Name>
    <Age>21</Age>
    <Height>76</Height>
</Person>
```

In this example we create a tag called Person that contains the child tags Name, Age and Height. Note that we included additional data about each property by adding

values to the Name, Age and Height tags which are “Bob Doe” for Name, 21 for Age and 76 for Height.
We can also represent the above XML data using tag attributes as shown below:

```
<Person Name="Bob Doe" Age="21" Height="76" />
```

As you see this method looks much cleaner and less verbose. We have added the Name, Age and height to the Person tag as “attributes”. The value assigned to each attribute is called the attributes value and is always enclosed withn quotation marks “”), for example, Name=”Bob Doe”, the attribute is “Name” and the value is “Bob Doe”. Note that if an attribute is defined then it must also have a value, although you can supply an empty value, for example Name=””

You may be wondering at this point why we have not written the above tag like this:

```
<Person Name="Bob Doe" Age="21" Height="76">
</Person>
```

XML provides a short cut for tags that do not contain any children, allowing us to dispense with the </Person> closing tag and simply prep-end a forward slash to the closing tag character so that it becomes />. This is another example of making XML more readable and less verbose.

Now that we know more about how to write XML, lets take a look at our first example again:

```
<Label Position="10, 10" Text="Hello World" Font="trebuchet">
```

Our tag is called “Label” and it contains three attributes along with values. The attribute / value pairs are read as:

- Position = 10, 10
- Text = Hello World
- Font = trebuchet

This XOML command tells the reader to create a Label at Position 10 pixels across the screen and 10 pixels down the screen that displays the text “Hello World” using the font called “trebuchet”.

A XOML document is a collection of many of such tags like those outlined in the previous examples. The document is used to define all components that will appear in your game or app as well as how each of those components react / interact opening up a world of endless possibilities.

1.4 Editing XOML

In order to write XOML documents it is highly recommended that you use a modern XML document editor that supports syntax highlighting (adds colouring to comments, tags, attributes and values) as well as tag tree collapse / expansion to aid readability and navigation especially in larger documents. We also suggest that you use an XML document editor that supports XML schema. An XML schema file has been supplied called XOML.xsd which is a file that explains to the XML editor how XOML can be used. Installing this schema to your XML editor will give you great features such as:

- XOML validation - If you have typed something that isn't standard XOML then the editor will tell you so helping to prevent errors
- Auto-complete - This feature allows you to start typing a tag name and the editor will suggest valid tags that you can use as you type. Ths is a great feature for preventing tag spelling mistakes
- Hover help - Hovering over a tag / attribute will display informative help about that particular tag or attribute

We highly recommend Microsoft Visual Studio 2010 Express as a XOML editor as it is free and it supports an XML editor with a schema. You can download Visual Studio Express from http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express

Another great free XML editor is Notepad++ which is very lightweight and includes syntax highlighting, tree collapsing and a few other good tools, although it does not support XML schema. You can download Notepad++ from http://notepad-plus-plus.org/

If you prefer editing XML graphically then you may want to try XML Notepad 2007, which is a free download from http://www.microsoft.com/en-us/download/details.aspx?id=7973 . This editor is lightweight, supports syntax highlighting, schema validation etc..

There are also a number of commercial XML editors including:

XML Spy by Altova - http://www.altova.com/xml-editor/
XML Blueprint 8 - http://www.xmlblueprint.com/
Liquid XML Studio - http://www.liquid-technologies.com/Xml-Studio.aspx

1.4.1 Installing the XOML Schema (Microsoft Visual Studio)

AppEasy ships with an XML schema file called XOML.xsd which is located in the Docs folder. To install this schema in Visual Studio, open up an XML file and right click anywhere on the page then select properties from the pop up menu. In the properties view select the … icon next to the Schema option to bring up the Schemas dialog. If this is the very first time that you have installed the schema to Visual Studio then click the Add button and select the XOML.xsd file and click open. The schema will now appear in the list of schemas. To tell the current document to use the XOML schema, select the drop down in the "Use" column and select the "Use this schema" option. You should now have intellisense, XML validation and auto-complete functionality. You will need to reselect the schema for each XOML file in your project.

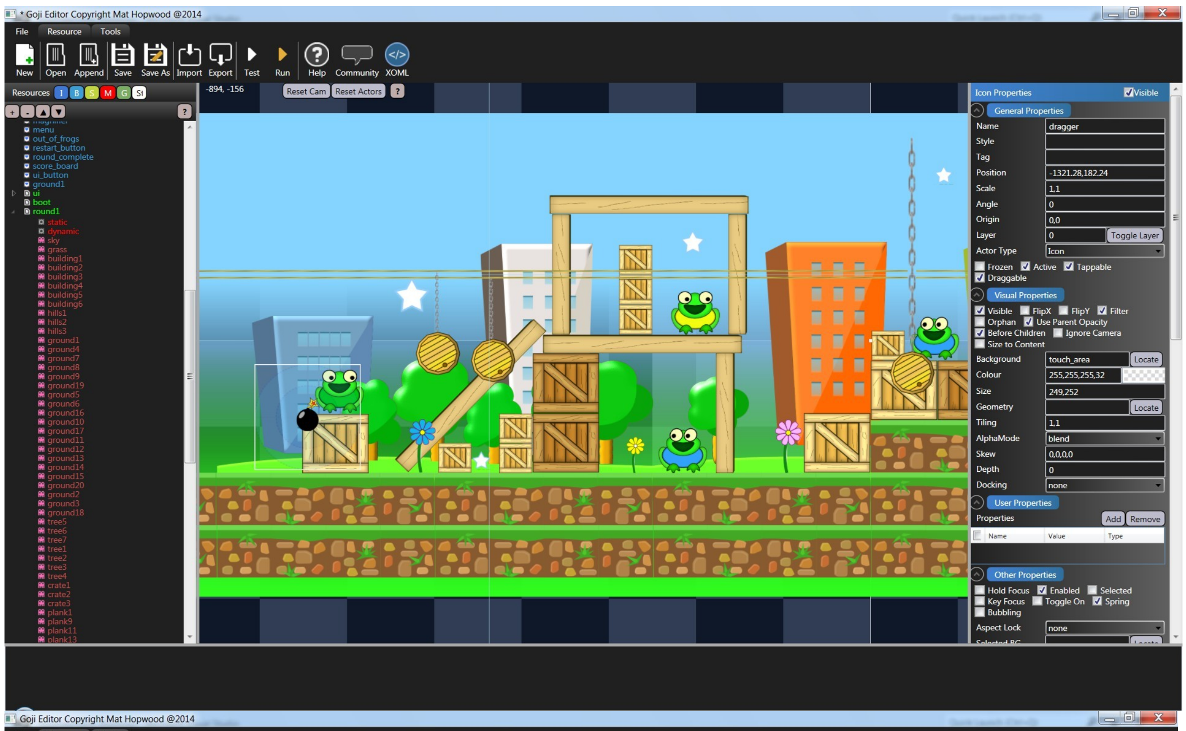
1.4.2 Adding Schema to your XOML files

You can automate the inclusion of the XML schema by adding the following to the XML definition in your XOML file:

```
<xml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="c:\AppEasy\Docs\XOML.xsd">
```

1.5 Goji Editor

The Goji game and app editor also supports export directly to the XOML format, which is much eaiser than writing XML. It also comes with a built in code editor and the ability to run and test directly from the editor, including viewing colour highlighted debug output. Shots of the Goji Editor are shown below:



Icon Properties ✓ Visible

General Properties

Name	dragger
Style	
Tag	
Position	-1321.28,182.24
Scale	1,1
Angle	0
Origin	0,0
Layer	0
Actor Type	Icon

☐ Frozen ✓ Active ✓ Tappable
✓ Draggable

Visual Properties

✓ Visible ☐ FlipX ☐ FlipY ✓ Filter
☐ Orphan ✓ Use Parent Opacity
✓ Before Children ☐ Ignore Camera
☐ Size to Content

Background	touch_area
Colour	255,255,255,32
Size	249,252
Geometry	
Tiling	1,1
AlphaMode	blend
Skew	0,0,0,0
Depth	0
Docking	none

User Properties

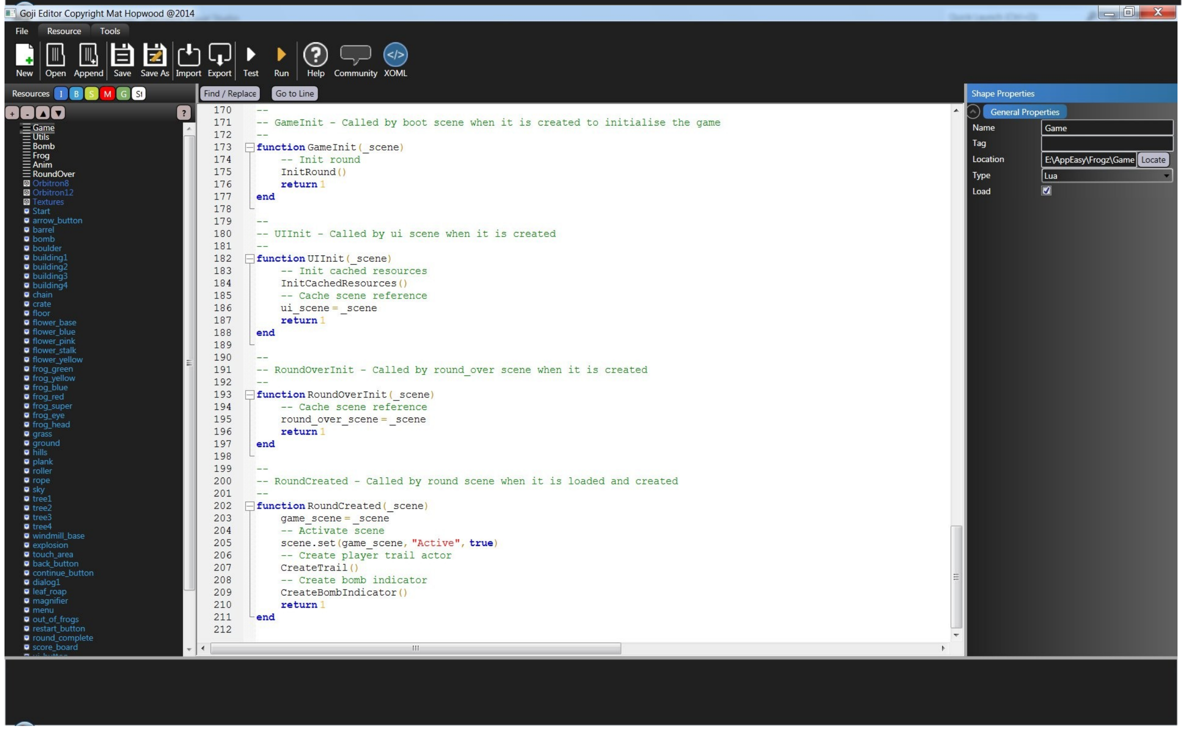
Properties

Name	Value	Type

Other Properties

☐ Hold Focus ✓ Enabled ☐ Selected
☐ Key Focus ☐ Toggle On ✓ Spring
☐ Bubbling

Aspect Lock	none



```
--
-- GameInit - Called by boot scene when it is created to initialise the game
--
function GameInit(_scene)
    -- Init round
    InitRound()
    return 1
end

--
-- UIInit - Called by ui scene when it is created
--
function UIInit(_scene)
    -- Init cached resources
    InitCachedResources()
    -- Cache scene reference
    ui_scene = _scene
    return 1
end

--
-- RoundOverInit - Called by round_over scene when it is created
--
function RoundOverInit(_scene)
    -- Cache scene reference
    round_over_scene = _scene
    return 1
end

--
-- RoundCreated - Called by round scene when it is loaded and created
--
function RoundCreated(_scene)
    game_scene = _scene
    -- Activate scene
    scene.set(game_scene, "Active", true)
    -- Create player trail actor
    CreateTrail()
    -- Create bomb indicator
    CreateBombIndicator()
    return 1
end
```

Shape Properties

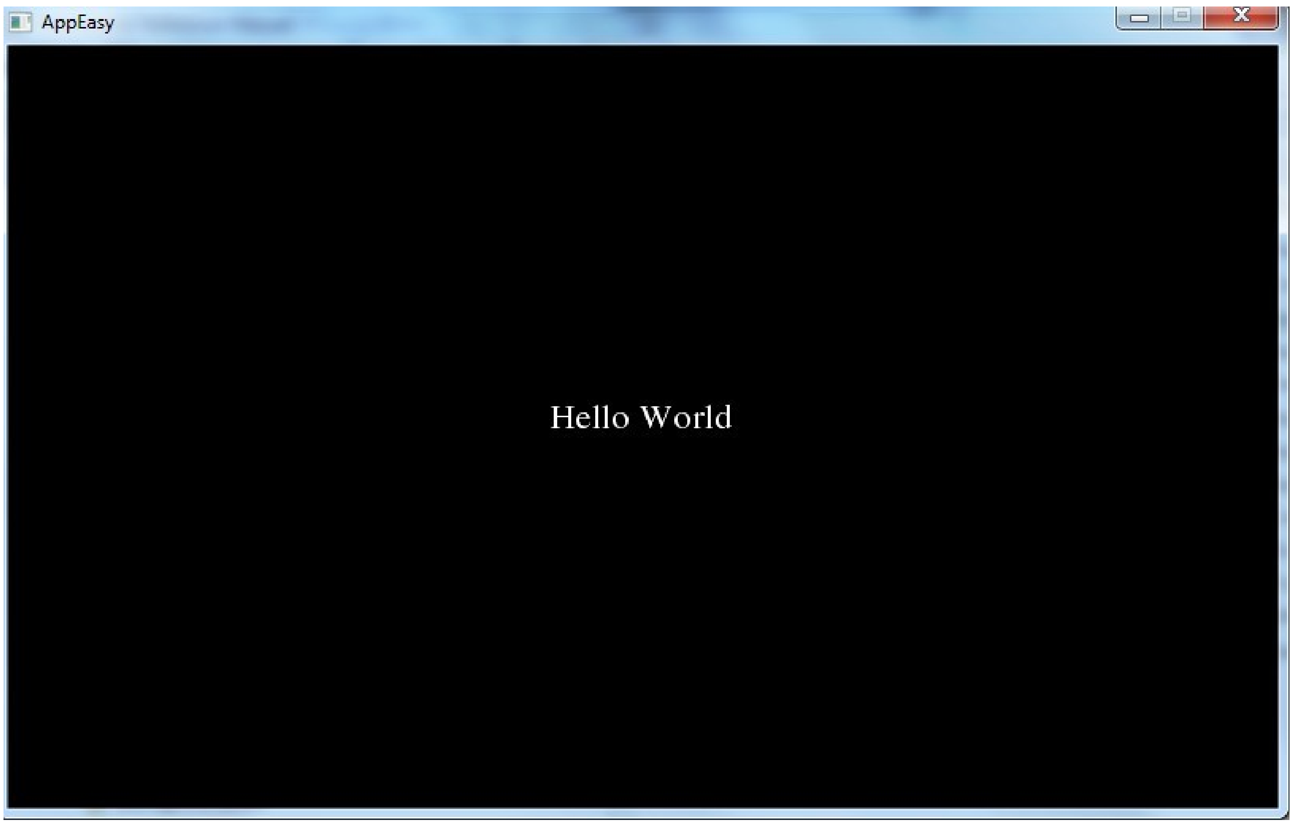
General Properties

Name	Game
Tag	
Location	E:\AppEasy\Frogz\Game
Type	Lua
Load	✓

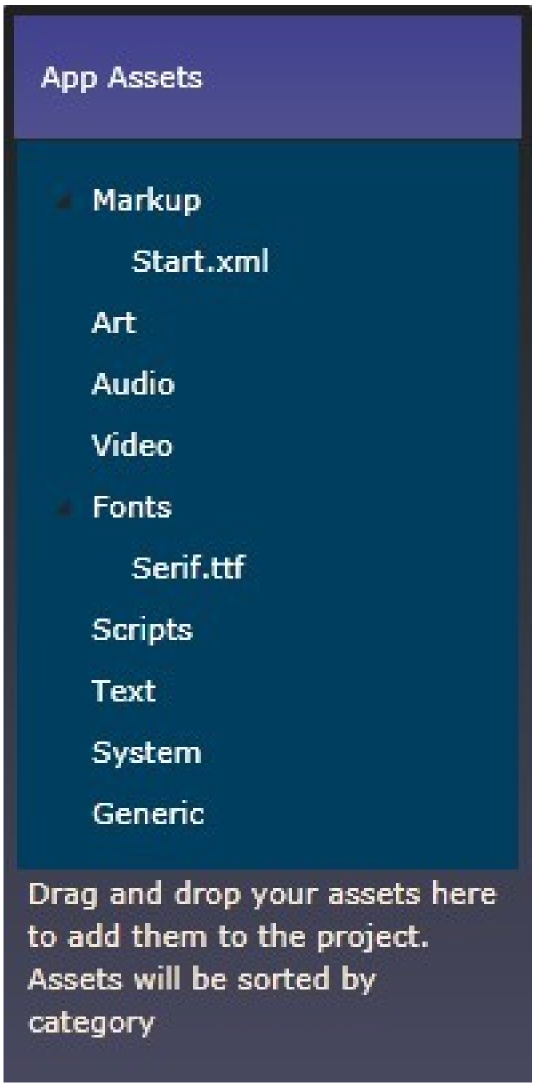
The Goje editor ernables cmplete game and app production without the need to write XML.

1.6 Hello World

Its customary to begin any kind of programming style tutorial with a Hello World example. Our Hello World example is basically the minimal amount of XOML or script that is required to display the phrase “Hello World” on the screen. To open up and test the Hello World example run the AppEasy Project Manager and click the “Open” button to open a project. If you have not yet seen the AppEasy Project Manager then please see the AppEasy Project Manager documentation. Select the HelloWorld example folder (c:/AppEasy/Examples/HelloWorld) then select the HelloWorld.xml project file to open the project. To run the project click the “Test” button. Running the project will show the following on screen:



Now lets take a look at the code for this project. In the AppEasy Project Manager take a look at the assets section (the column to the left) – see image below.



Drag and drop your assets here to add them to the project. Assets will be sorted by category

You will notice that Mark-up and Fonts both have a small arrow to their left (this shows you that the section is collapsed and that you need to click the arrow to expand the section to see the assets that are contained within). If you expand both of these sections then you will see that the Mark-up section contains the Start.xml file and the Fonts section contains Serif.ttf. **The Start.xml file is a special file as far as AppEasy is concerned, it is the file that is loaded by default to begin execution of your app**. Serif.ttf is a true type font that is loaded when the Hello World app runs and is used to render text using the Serif font style. Now double click Start.xml to open the file and lets have a look at what's inside.

The mark-up contained within Start.xml is shown below:

```
<?xml version="1.0"?>
<xml>
    <!-- Load the serif true type font  -->
    <Font Name="serif" Location="Serif.ttf" PointSize="10" />

    <!-- Create a scene to hold our game / app objects -->
    <Scene Name="GameScene" Current="true" >

        <!-- Create a label object -->
        <Label Font="serif" Text="Hello World" Size="-100, -100" />

    </Scene>
</xml>
```

All XOML documents are enclosed within the following mark-up:

```
<?xml version="1.0"?>
<xml>

    <!-- XOML goes here -->

</xml>
```

The header <?xml version="1.0"?> identifies the document as an XML document, whilst the <xml> tag and </xml> close tag act as the "root" tag element. The root tag is there because an XML document should have only one single root tag, all other tags are children of the root tag.

Now lets break down the above XOML to see what it does.

```
    <!-- Load the serif true type font  -->
    <Font Name="serif" Location="Serif.ttf" PointSize="10" />
```

Here we create a font called "serif" from the true type font file called "Serif.ttf". This file is part of the assets that you can see in the assets section of AppEasy. This file could quite easily be located elsewhere such as on a web server. If for example, our font was located at www.mydomain.com/assets/Serif.ttf then our font declaration would change to:

```
    <Font Name="serif" Location="http://www.mydomain.com/assets/Serif.ttf" PointSize="10" />
```

If you do not fully understand this then do not worry as we will be dealing with asset locations in more depth in a later chapter.

The remaining attribute in our font declaration sets the point size, this is the size at which the font will be rendered, the larger the point size the larger the default size of the text displaying using the font.

The next task our XOML carries out is to create a scene:

```
    <!-- Create a scene to hold our game / app objects -->
    <Scene Name="GameScene" Current="true" >
    </Scene>
```

For now you can think of a scene as the screen or a container for your game or app objects. All game / app objects must be placed inside a scene for them to be visible and to allow interaction with them. In our example we create a scene called GameScene and tell AppEasy to make the scene the currently active scene.

Lastly, we create a label that displays text inside our scene:

```
    <!-- Create a label object -->
    <Label Font="serif" Text="Hello World" Size="-100, -100" />
```

We instruct the label to use the serif font that we previously declared using the Font tag. We also supply the text that we would like displayed, in this case the text is “Hello World”. Finally we instruct the label to use the full size of the display by setting its size to -100, -100 (negative sizes are used to proportionally size objects based on either the screen or parent container size, more on this later).

Note that any decent XML editor will show typo errors like these helpnig you to avoid simple mistakes when writing XOML.

1.7 Logs, Errors and Warnings

As XOML is running it displays a log of what it is currently doing. For example when a XOML file is loaded it displays all of the tasks that it is currently performing, for example:

```
Info: XOML - loadxoml - file="Common.xml"  - File: Start.xml line 5
Info: XOML - style - name="GenericSceneStyle"  - File: Common.xml line 6
Info: XOML - image - name="sprites" location="sprites.png" preload="true" format="RGBA_5551"  -
File: Common.xml line 15
Info: CzImage::Init() - Size = 117561
Info: CzImage::Init() - Width = 1024
Info: CzImage::Init() - Height = 1024
Info: XOML - image - name="buttons" location="buttons.png" preload="true" format="RGBA_5551"  -
File: Common.xml line 16
Info: CzImage::Init() - Size = 252800
Info: CzImage::Init() - Width = 1024
Info: CzImage::Init() - Height = 1024
Info: XOML - image - name="background" location="bg1.png" preload="true" format="RGBA_4444"  -
File: Common.xml line 17
Info: CzImage::Init() - Size = 135059
Info: CzImage::Init() - Width = 1024
Info: CzImage::Init() - Height = 1024
Info: XOML - brush - name="building1" image="background" type="image" srcrect="0, 0, 286, 484"  -
File: Common.xml line 22
Info: XOML - brush - name="building2" image="background" type="image" srcrect="286, 0, 286, 406"
- File: Common.xml line 23
Info: XOML - brush - name="building3" image="background" type="image" srcrect="572, 0, 286, 329"
- File: Common.xml line 24
Info: XOML - brush - name="building4" image="background" type="image" srcrect="286, 406, 264,
504"  - File: Common.xml line 25
```

Generally if XOML has trouble with something that you have passed to it the warning or error will be shown here. Major errors will usually halt execution whereas warnings will allow execution to continue. If running your XOML app from the AppEasy simulator then you can view the error log by hitting the View Log button. Lets take a look at small part of the above log is showing us:

```
Info: XOML - loadxoml - file="Common.xml"  - File: Start.xml line 5
```

The above line shows us that the XOML parser has started to load file Commono.xml XOML file at line 5 of Start.xml. Lets take a ;ook at the XOMLcode that created the above output:

```
        <LoadXOML File="Common.xml" />
```

Now lets take a look at a typical warning:

```
Warning: Timeline - Could not find named animation in the scene or in the global resources -
quick_fade_out - File: Hud.xml line 38
```

This output is warning us that the animation used by the timeline called scene_fade_in was not found. Lets take a look at the XOML:

```
<Animation Name="fade_i" Type="vec4" Duration="5">
	<Frame Value="64,128,200,0" Time="0.0" Easing="quadin"/>
	<Frame Value="255,255,255,255" Time="2.0"/>
</Animation>

<Timeline Name="scene_fade_in"AutoPlay="true">
	<Animation Anim="fade_in" Target="Colour" Repeat="1" StartAtTime="0"/>
</Timeline>
```

The timeline fade_in was not declared in this instance because it was incorrectly named fade_i.

1.8 Examples

AppEasy comes with a large collection of examples that help to teach you how to use various parts of AppEasy. These examples include:

0. Anchors – Shows how to change actor visual anchor point
1. Scene Animation – Shows how to animate a scene
2. Scene Clipping – Shows how to change the clipping extents of a scene
3. Scene Augmentation – Shows how to add functionality to a scene after it has already been declared
4. Scene Events – Shows how to use and handle scene events
5. Scene Extents – Shows how to change a scenes extents
6. Scene Layers – Shows how to layer scenes
7. Scene Panning – Shows how to use touch camere panning within a scene
8. Scene Physics – Shows how to set up basic physics in a scene
9. Actor Children – Shows how actor hierarchies work
10. Actor Collision Flags – Shows how to use collision flags to mask collisions between actors
11. Actor Connectors – Shows how to use the connector actor to visually connect actors
12. Actor Docking – Shows how actors can be docked to arrange them effectively on different sized screens
13. Actor Layers – Shows how actors can be layered to add depth sorting
14. Actor Modifiers – Shows how to use actor modifiers to add additional functionality to actors
15. Actor Particles – An example that shows how to use the particles system. This example builds up to 1000 animating particles at different depths over a short period of time.
16. Actor Physics – Shows how to apply physics to actors
17. Advanced Collision – Shows how to query and interact with an actors list of start / end collision contacts
18. Basic Animation – Shows how to create basic animation and attach it to an actor
19. Basic Data Bindings – Shows how to basic data bindings. This example binds properties of an actor to different variables
20. Bitmap Animation – Shows how to create a bitmap animation from a sprite atlas and attach it to an actor
21. Complex Program – Shows how to create a slightly complex program using commands that are executed in sequence or in parallel
22. Conditional Actions – Shows how to execute actions based on certain conditions being met

23. Conditional Images – Shows how to load different image sets based on conditions such as the screen size
24. DataGrid Bindings – Shows how to create a grid user interface component and bind XML data to it
25. Dynamic WebView – This example shows how to write html directly to a web view using actions creating dynamic web content. It also shows how to use actions to navigate web pages / sites.
26. Files – Shows how to load files and bind the loaded data to a label
27. Game Of 10 – A game written completely in XOML, including logic
28. Hang Man – A hang man game written in XOML and Lua
29. Rock Paper Scissors – A rock paper scissors written completely in XOML
30. Physics Joints – Demo that shows how to use a variety of different Box2D joints to connect actors together
31. Music Playback – Shows how to play and stop music using actions
32. Overlapping Actors Test s– A simple example that shows how to use script to test for overlapping actors
33. Persistent Variables – An example that shows how variables can be made persistent across different app sessions
34. Program Loops – Shows how to create and use loops in programs
35. Resources removal by Group – Shows how to remove resources from the global resource manager
36. Sound Effects – Shows how to play sound effects
37. Styles – Shows how to use styles to style a group of actors
38. Templates – Shows how to use templates to create multiple actors at the same time
39. Riled Textures – Demo that shows how to use texture tiling to create effects
40. Complex UI – A more complex UI example that shows a large complex virtual user interface
41. UI Canvas – Shows how to use the Canvas UI container control
42. UI Grid (Manual)– Shows how to use the Grid UI container control to host pre-set UI elements
43. UI Grid (auto generated) – Shows how to create a UI Grid component automatically from data. Also shows how to add and remove elements, change the data source and template
44. UI ImageView – Shows how to create an ImageView container UI control. The ImageView container supports pinch zoom / pan
45. UI Listbox (manual) – Shows how to use the Listbox container UI control to host pre-set UI/ Also shows how to add / remove elements as well as change orientation
46. UI Listbox (auto generated) – Shows how to use the Listbox container UI control to host dynamic UI. Also shows how to change data source and

templates

47. UI StackPanel – Shows how to use the StackPanel UI container control
48. UI TabBar – Shows how to use the TabBar UI container control to host different views of information
49. UI Textbox – Shows how to use the Textbox UI component to allow the user to enter information
50. UI WrapPanel – Shows how to use the WrapPanel container UI component
51. Video Playback – Shows how to display video
52. Video Cam Streaming – Shows how to map video camera input to actors
53. XMLVariables – Shows how to use XML variables in XOML to pull specific information from XML files
54. Http Test – Shows how to download an image from the web and map it to an actor
55. Simple Car Physics – A simple demo that shows how to build a car type physical object
56. Simple Bridge Physics – A simple dwmo that shows how to build a bridge physical object that carries cars
57. Video Cam Snapshots – Shows how to take snaps from the video cam and store / retrieve them
58. Market Test – Shows how to use in-app purchasing
59. Multi-touch test – Shows how to handle multiple touches
60. User Properties – Shows how to use scene and actor user properties to add additional data that can be shared between XOML and Lua
61. Accelerometer – Shows how to use the accelerometer
62. Compass – Shows how to use the compass
63. Ads – Shows how to display inner-active and leadbolt banner ads
64. RemoteReq – Shows how to post data to a web server and retrieve a response
65. Keyboard – Shows how to use the keyboard
66. Create Resources – Shows how to instantiate XOML objects from Lua tables and strings of XOML
67. Timers – Shows how to use XOML timers in XOML and Lua
68. Polygon Sprites – Shows how to create geometries and assign them to actors to give them unique shape
69. SQLite Test – An example that shows the creation and interaction with an SQL database
70. Facebook – An example that shows how to carry out tasks such as log users in to Facebook, collect users info, friends lists and their details, post to the users wall etc..

1.9 Supported Platforms

Currently XOML is supported on the Marmalade platform. Versions of XOML will be made available natively on the following additional platforms in 2013/2014:

- Windows XP/Vista/7/8
- Mac OS
- Android
- Apple iOS
- Windows Phone 8
- BlackBerry OS 10
- PlayStation Vita

2.0 Anatomy of a XOML App

In this section we will take a quick look at the parts that go into making a XOML app, an overview if you will. Do not worry if you do not fully understand all of it, we will be looking into each section in more depth later on. Feel free to return to this section again when you reach the end of this document.

2.1 Assets

All elements that an AppEasy app are made from are called assets. These assets can be split into a number of categories:

- Mark-up - Mark-up describes how the app is laid out on screen and how it functions. Mark-up can be located in a single file or across multiple files
- Artwork - Artwork refers to any assets in your project that are graphical. This could be anything from a background in JPEG format to a character animation sprite atlas in PNG format
- Audio - Audio refers to the sound effects and music that are played back by your app
- Video - Refers to any video files that can be played back
- Fonts - True type fonts that will be used to render particular styles of text
- Scripts - Scripts are files written in a scripting language such as Lua that add extra none standard features to your app
- Text - Basic text files
- Generic - All other assets are classed as generic and do not fit into any particular category

When AppEasy generates your app, it combines all of your included assets and packages them into an app that can be installed to a mobile device or submitted to app stores for sale.

When your app first runs either on PC or on an actual device it loads the Start.xml XOML file and executes any tags in that file. Start.xml is where you place all of the XOML that introduces your app. To ease development and to help you cut your app into more easily manageable and re-usable sections a XOML file can include other XOML files which are loaded and parsed along with the original XOML file

Its generally a good idea to build your app using multiple XOML files, separating screen specific functionality into their own files. For example, a typical game may have a set of XOML files like shown below:

- Start.xml - This file loads and displays your game or apps welcome screen, waits for the user to tap the screen then loads the main menu.
- MainMenu.xml - This file loads and displays the main menu where the user can choose to do things such as start a new game, view a tutorial etc..
- LevelSelect.xml - This file loads and displays a level select screen that allows the user to select which level to play
- Level1.xml - This file loads and plays level 1 of your game
- Level2.xml - This file loads and plays level 2 of your game
- GameOver.xml - This file loads and displays the game over screen showing the player their score and other stats

Note that you do not have to follow this style of development but we have found it very useful to separate functionality into separate XOML files.

You may also want to consider separating out specific commonly used functionality into its own file and re-use this across multiple files. For example, you may want to separate out the XOML that defines your main player character into its own XOML file called Player.xml. This file can then be loaded into each of your level files.

2.2 XOML Parsing & Debugging

The XOML file reader parses XOML from the top of the page and does not stop until it reaches the last tag, unless a serious error occurs in which case the parser will stop and display an error. The XOML parser may also come across minor errors in which case it will output warnings but will not stop. You should try to resolve most warning messages to ensure that your app runs without issue. The Window simulator test build will also output a debug.txt file into its build folder, you can open this file from the App Easy Project Manager by clicking the “View Log” button. If your app does not behave as expected then this is the first place to look for potential problems.

2.3 The Scene / Actor Paradigm

XOML uses a scene / actor paradigm to efficiently organise your apps screens and objects. You can think of a scene as roughly a screen full of information (or a stage) and actors as the interactive objects that are contained within those scenes. An app can contain many scenes during its lifetime, some may exist throughout the whole app whilst others are loaded, displayed and destroyed as and when required. An app can load and display many scenes at the same time overlaying them in a specific way.

Scenes contain actors (usually many at once) that can interact with one another in specific ways. For example, a pool game scene could contain actors that represent the edges of the table and pockets as well as an actor for each of the balls. It may also contain labels and other such user interface components to display score / hi-score, menu buttons etc..

In summary a scene is a container for actors and actors represent the individual game or app components that make your game or app what it is.

Lets take a quick look at how to create a basic scene using XOML:

```
<?xml version="1.0"?>
<xml>
    <!-- Create a scene to hold our game / app objects -->
    <Scene Name="GameScene" Current="true" >

        <!-- Actors go here -->

    </Scene>
</xml>
```

Here we create a scene called GameScene and tell the system to make it the current scene. Because many scenes can be active and visible at the same time (lets say for example, the level foreground scene, the level background scene and the player stats scene) the system uses the concept of “current scene” to manage which scene can have the current input focus. The input focus is the ability for a scene to receive input events such as user taps / drags, key presses etc..(just like when you switch from one window to another on your computer, you change the focus between different applications) In the above example we make our “GameScene” the current scene that receives input focus by adding Current=”true” to the scene tag.

2.4 Resources

An app wouldn't be much if it did not have some resources to work with. XOML can utilise a selection of different types of resources including:

- Images - Image files such as JPEG's and PNG's
- Fonts - True type fonts
- Sound effects - Compressed WAV files (usually used to inform the user something has happened such as user tapped a button or the player character hit something)
- Music - MP3 music (usually used as background music for games)
- Video - MPEG4 / 3GPP movie files
- Animations - Animation data (used to animate actors and scenes)
- Timelines – Targeted collections of instances of animations
- Brushes - Brushes can be used to represent portions of images as well as 9-patch images
- Shapes and Materials - Shapes and materials are used by the physics system
- Geometries – Geometric shapes that can be assigned to actors to change their shape
- Styles and Templates - Styles are used to style actors and scenes, whilst templates are used to create blocks of re-usable XOML
- Cameras - Cameras allow you to adjust the way you look at a scene and its contents
- RemoteReq – A Remote request definition. These an be used to communicate with a web server / web service
- VideoCam – Allows streaming of devices video cam input to an image which can then be mapped to actors
- Scripts – Lua and other scripts
- Actions – Actions carry out a variety of tasks and can be called by any XOML event
- Programs and Commands – Programs and commands enable you to put together complex logic that can be ran at a regular interval
- Variables – Variables can be used to hold data and can be bound to actors and scenes
- Files – File resources can hold any type of data and can be used to pull local and remote data
- Timers –A timer is a resource that can be used to fire off events at some point in the future either on a single occasion or at regular intervals

Lets take a brief look at examples of creating some of these types of resources (do not worry if you do not understand these examples yet, each resource will be addressed in full in a later chapter):
Create an image example:

```
    <Image Name="ui_image" Location="ui.png" />
```

Create a font example:

```
    <Font Name="serif" Location="Serif.ttf" PointSize="10" />
```

Create a sound effect example:

```
    <Sound Name="click_sound" Location="click_sound.wav" />
```

Create a video example:

```
    <Video Name="welcome_video" Location="welcome.mp4" Codec="MPEG4" />
```

Create an animation example:

```
    <Animation Name="scene_zoom_in" Type="float" Duration="3" >
        <Frame Value="5.0"  Time="0.0"/>
        <Frame Value="1.0"  Time="3.0"/>
    </Animation>
```

Create a brush example:

```
    <Brush Name="button_brush" Image="ui_image" SrcRect="320, 210, 48, 48" Type="image" />
```

Create a shape example:

```
    <Shape Name="player_shape" Type="box" Width="87" Height="56" />
```

Create a physics material example:

```
    <Box2dMaterial Name="bouncey" Type="dynamic" Density="1.0" Friction="0.3"
       Restitution="0.6" />
```

Create a style example:

```
    <Style Name="GenericSceneStyle">
        <Set Property="CanvasSize" Value="800, 480" />
        <Set Property="CanvasFit Value="best" />
    </Style>
```

Create a template example:

```
    <Template Name="AsteroidTemplate">
        <Icon Name="Asteroid$base$" Size="$sizex$, $sizey$" Background="AsteroidBrush" />
    </Template>
```

Create a camera example:

```
    <Camera Name="Cam" Position="0, 0" Angle="0" Scale="1.0" />
```

Create a remote request:

```
    <RemoteReq Name="Request1" URL="http://www.mywebservice.com" Data="name=Mat"
              OnResponse="GotData" OnError="Error" Variable="ReqVar1" />
```

Create a VideoCam resource:

```
    <VideoCam Name="Cam1" Target="Image1" Quality="low" Resolution="low" Start="true" />
```

Create a timer example:

```
    <Timer Name="imer1" Duration="1.5" Repeat="10" OnRepeat="GlobalActions1"
AutoStart="true" AutoDelete="true"/>
```

2.5 Resource Scope

Resource scope refers to the ability to access a particular resource depending upon where it was declared. Resources can have three types of scope:

- Actor local – The resource is specific to an actor and cannot be generally accessed by anything else. Actions and timeline resources that are declared inside actor tags are by default local to that actor and will be released when the actor is destroyed. You cannot search the global or scene resource managers for these types of resources
- Scene local - The resource is specific to a scene and cannot be accessed by objects outside that scene. Resources that are declared inside scene tags are by default local to that scene and will be released when the scene is destroyed. For large games and apps this is very useful as memory constrained devices such as mobile phones cannot always fit everything in memory at once.
- Global - Global resources can be accessed from anywhere in your app and from any scene. Resources that are declared outside of a scene are automatically added to the global resource system

Resource scope brings up the important subject of “name conflicts”. If you add a resource to the system that already exists then you will create a name conflict because that resource already exists. However, name conflicts are only limited to the “type of resource”. For example, you can quite happily add an Image named Background and a Brush named Background because the two types of resources differ. However adding two images called Background will cause a name conflict.

XOML resolves the name conflicts issue by deleting the existing resource and replacing it with the newly created resource.

If you add a resource to a scene that has the same name and type as a resource that is declared globally then the global version will be ignored during resource searches.

2.6 Resource Tagging

Sometimes it is needed to load a group of resources into the global resource space so they can be accessed across different scenes, however you do not necessarily want them to remain there for the duration of your apps life time. All resources can be tagged with a name so that they can be removed en-mass based on their tag name. Here is an example of a group of tagged resources:

```
<Image Name="ui_image" Location="ui.png" Tag=”Group1” />
<Font Name="serif" Location="Serif.ttf" PointSize="10" Tag=”Group1” />
<Sound Name="click_sound" Location="click_sound.wav" Tag=”Group1” />
```

The above 3 resources all share the same resource tag name of Group1. At a later stage you can remove all of these resources by removing them by their group tag name. We will look at how we remove resources later on.

2.7 Events and Actions

XOML is an event driven system which means that when something happens such as the user taps on a button or two game objects collide then the button / objects get to know about it. Scenes and actors both support a number of different events that they can listen for and respond to. For example our button actor can listen for the OnTapped event which gets fired when the user taps the actor. When this event occurs the actor can respond to it by calling an actions list and performing a group of actions. Different types of actors support different types of events (more on this later).

XOML supports a large number of actions. An action is basically a command that tells the system to do something such as change a variable, change an actors property, start an animation, play a sound effect etc..

The events and actions system is the backbone of XOML interactivity and it enables very complex applications to be built without the need for additional scripting languages.

2.8 Styles and Templates

Its always good and efficient to do more with less and that is the function of styles and templates. A style allows you to provide a set of pre-defined attributes to an actor or a scene which saves you repeatedly specifying the same information time and time again, not to mention even more time if you decide that you need to change the attributes of a whole group of scenes or actors. If you are familiar with HTML then you can consider styles to be something similar to styles that are defined in style sheets. Lets take a look at defining a style for a scene:

```
<Style Name="GenericSceneStyle">
    <Set Property="CanvasSize" Value="800, 480" />
    <Set Property="CanvasFit" Value="best" />
</Style>
```

Here we create a style called GenericSceneStyle which sets two default attributes:

- CanvasSize to the value of "800, 480"
- CanvasFit to the value of "best"

Now we create a scene that utilises this style:

```
<Scene Name="GameScene" Style="GenericSceneStyle">
```

Applying the style to the GameScene produces the following XOML internally:

```
<Scene Name="GameScene" CanvasSize="800, 480" CanvasFit="best">
```

It doesn't look like we have saved much by way of typing in this example, but in a more realistic example you would save much typing. In addition you could change many scenes or actors by simply changing the style.

Templates offer a much greater control over XOML re-use as templates can store a large collection of XOML tags that can be re-purposed using template parameters. Lets take a look at a simple example:

```
<Template Name="AsteroidTemplate">
    <Icon Name="Asteroid$base$" Size="$sizex$, $sizey$" Background="AsteroidBrush" />
</Template>
```

Here we define an AsteroidTemplate that contains an Icon (an actor that has a bitmap associated with it). Note how we supply the default definition of the Icon including Name, Size and Background attributes inside the template. The template parameters are each enclosed within double dollar signs $$, which include $base$, $sizex$ and $sizey$ and must be in lower case. At a later stage we will instantiate multiple asteroids from this template as follows:

```
        <FromTemplate Template="AsteroidTemplate" base="0" sizex="80" sizey="80" />
```

The parameters that are supplied with the FromTemplate tag are substituted into the template allowing creation of custom XOML.

Note that templates can also be instantiated from script, commands and actions.

2.9 Animations

Lets face it all games and apps look much better when something is going on on the screen. Imagine Angry Birds if the birds and pigs just sat static and did nothing, or if you tapped a button in an app and there was no visual feedback to let you know that you had tapped the button. Apps with no animation are boring!

XOML solves this problem by providing a way of creating animations incredibly easy using mark-up. XOML uses what is called the frame approach to animation. The frame based approach is animation using samples at specific points in time (key frames). For example we could say that object A should be off the screen at time 0, but in the middle at time 5 seconds. Lets take a quick look at an example of an animation that would be used to create this type of behaviour:

```
    <!-- Create animation to move actor from off screen to middle of screen -->
    <Animation Name="move_to_middle" Type="vec2" Duration="5" >
        <Frame Value="-1000, 0"   Time="0" />
        <Frame Value="0, 0"       Time="5" />
    </Animation>
```

In the above animation we declare a vec2 animation with two frames (vec2 is short for 2D vector which refers to a position with an x and y point) and a duration of 5 seconds (it lasts 5 seconds). Two frames that make up the animation are declared inside the Animation tag.

Animations can be applied to both actors and scenes using Timelines.

2.10 Variables and Data Binding

There will come a point during the development of more complex apps to store information such as the players score or how many lives they have remaining. To facilitate this kind of functionality XOML offers variables. Variables can store any data including whole numbers (integers), floating point (decimal numbers), 2d, 3d, 4d vectors, colours, strings, boolean, conditions, arrays and xml.

Lets take a look at a simple variable declaration in XOML:

```
        <Variable Name="lives_left" Type="int" Value="3" />
```

This piece of XOML declares a variable called lives_left that ts of type integer (whole number) and assigns it the value of 3.

You can access this variable from other parts of XOML using conditions, data binding, actions, scripts or by passing the variable name to various commands or actions.

Data binding is the binding of a variable to the property of an actor. For example, we could declare a variable that holds a player actors position and bind it to a player actor, changing the variable will also change the actors position.

We will look into all of these concepts later in further chapters.

2.11 Modifiers

Modifiers are set units of specific functionality that modify the behaviour of an actor or scene. For example, we have a modifier that we can attach to an actor which allows it to monitor and react to collision events. At the moment not many modifiers are available but many will be added as XOML matures.

2.12 Programs and Commands

A program is a list of commands that are generally executed one after the other in a set sequence. A program can be used to define a set group of commands that should be carried out each time the app updates (each game frame) and / or in response to certain actions. Programs allow you to add simple structured conditional logic to your XOML without the need for additional script languages.

2.13 Scripts

Whilst XOML is a powerful mark-up language in its own right, it does have some limitations when it comes to defining complex logic for more complex apps and games. To eliminate this problem, XOML supports the loading and calling of scripts written in simple script languages such as Lua. Lets take a look at a simple Lua script example:

```
function Scene_OnTick(_object)

	local pos = actor.get(_object, "PositionX")
	pos = pos + 1;
	if (pos > 200) then
		pos = -200;
	end
	actor.set(_object, "PositionX", pos)

end
```

This short example is a function that is called each time the scene updates itself (approximately 30 to 60 times per second). The example scrolls the scene slowly to the left.

2.14 Communicating with a Web Server

XOML provides many ways of communicating with external data that is located on a web server / web site. Files, images, audio and fonts can for example reside outside the app on some web site and can be loaded on demand by the XOML app. This is a very easy to use automated system that provides very powerful control over what resources you ship with your apps and which you can host on a server and change at any time.

However, its often useful to be able to communicate with a web service or web service, such as send information about the user or the app. For example, your app may store its users data on the server in a database. XOML offers the ability to use both HTTP POST and GET to send and retrieve data to and from a web server.

Heres a short example:

```
<RemoteReq Name="Request1" URL="http://www.mywebservice.com" Data="name=Mat"
 OnResponse="GotData" OnError="Error" Variable="ReqVar1">
	<Header Name="Content-Type" Value="application/x-www-form-urlencoded" />
</RemoteReq>
```

This example creates a remote request definition that calls mywebservice.com and passes the data “name=Mat”. The inner header tag sets the Content-Type header. The server will later send a response which is written into the variable ReqVar1 and calls the GotData actions list allowing you to perform actions when the data is received. If an error occurs then Error actions list will be called instead.

Note that a RemoteReq is not automatically sent when it is created. You need to actually call the request from an action or program command, e.g.:

```
<Actions Name="GetData">
    <Action Method="RemoteReq" P1="Request1" />
</Actions>

<Program Name="Program1" AutoRun="true">
    <Command Method="remote_req" P1="Request1" />
</Program>
```

A request will remain in memory for as long as the RemoteReq resource is available. A RemoteReq can be re-used.

2.15 User Properties

Actors and scenes both carry many properties that define their look and behaviour. However, there will be many instances where you would like to define your own specific properties that you can use to track various attributes that are specific to a specific instance of an actor or scene. For example, you may want to track the total number of baddies in a scene or the number of lives the player has left. Whilst this can quite easily be accomplished with variables, it becomes cumbersome when you have many objects of the same type but each requires very specific data that's unique to each instance. For example, if we have an actor that represents a baddy character that can carry a specific weapon and has specific spells that it can use, clearly using variables would not be quite as efficient. Lets take a look at an example of how we would represent this in XOML:

```
<IconName="Baddy1"Background="BaddyBrush">
    <UserProperties>
        <Property Name="Weapon" Type="string" Value="machine gun" />
        <Property Name="WeaponRounds" Type="int" Value="100" />
        <Property Name="Spell1" Type="string" Value="fire ball" />
        <Property Name="Spell2" Type="string" Value="ice block" />
        <Property Name="Spell3" Type="string"Value="frost burn" />
        <Property Name="Spell4" Type="string "Value="arcane blast" />
    </UserProperties>
</Icon>
```

These properties can later be modified and accessed from XOML and script.

2.16 Geometries

Geometries are collections of shapes that can be assigned to actors to change their visible shape, so you are not limited to simple rectangular shaped actors. Geometries can consist of a single polygon with any number of vertices, a list of triangles or a list of of quads. As well as stipulating the geometric shape of the actor, geometries also support specification of UV texture coordinates, per vertex RGBA colour information and vertex face index lists. This type of system enables you to create very complex shapes built from a single or multiple shapes that fully support hit detection and clipping. Lets take a look at a quick example of a simple geometry that defines two triangles:

```
<Geometry Name="Geoms1" Vertices="-200,-200, 200,200, -200,200, -400,-400, -200,-400,
-200,0" Type="TriList" />
```

This can then be assigned to an actor:

```
<Icon Name="Sprite1" Position="200,0" Background="bg1" Geometry="Geoms1" />
```

The actor now takes on the shape of the supplied geometry Geoms1

A tool has been supplied that enables you to convert SVG data to geometries

3.0 Scenes - A Place for Actors to Play

3.1 Introduction

Sometimes understanding concepts are much easier when compared to something that one is already familiar with. In this case we will compare the making of apps and games using XOML to the making of a movie. In the movie business a scene (or set) is a place where the action takes place, it contains all of the actors, the cameras and scenery. Each actor has its own specific instructions, behaviours and purpose.

To create a scene in XOML we use the scene tag, supplying attributes and values for those attributes that define how the scene will look and behave. Lets take a look at a basic scene example:

```
<?xml version="1.0"?>
<xml>
    <!-- Create a scene to hold our game / app actors -->
    <Scene Name="AppScene" Current="true" CanvasSize=”800, 480” >

        <!-- Actors go here -->

    </Scene>
</xml>
```

Here we create a scene called AppScene that has a canvas size of 800x480 pixels. Once a scene is set-up we can begin adding the actors and other content that form our game or app.

3.2 Scene Properties

Now that we know how to create a scene lets take a look at the extensive list of properties that scenes support:

General Properties:

- Name (string) - The name of the scene is a very important property and should be chosen so that it does not conflict with names of other scenes that are present at the same time. It is possible to have multiple scenes with the same name, but only one of those scenes can be created. It is possible to destroy a scene and create a new scene of the same name. A scene name should also be memorable as you may want to refer to the scene from other areas of XOML or even script.

- Type (whole number) - The type of scene is a simple number that can be used to identify specific scene types. For example, you could create multiple scenes that are all related and assign them all the same type.
- Extents (x, y, width, height) - A rectangular area that describes the extents of the scenes world
- AllowSuspend (boolean) - Determines if the scene can be suspended when other scenes are activated
- Clipping (x, y, width, height) - A rectangular area that represents the visible area of the scene
- ClipStatic (boolean) – When true camera transform is not applied to the clipping rect
- Active (boolean) - The active state of a scene determines if the scene is processing its contents. If not active then scene processing will stop and any content that is contained within the scene will also stop processing
- Visible (boolean) - If a scene is not visible then it will be hidden from view, although the scene and its contents will still be processed.
- Layers (number, read-only) - The number of visible actor layers that the scene should use (10 is default value)
- Current (boolean) - If true then the scene is made the current scene
- Camera (camera name) - Current camera, getting this property will return the camera object
- Batch (boolean) - Tells the system to batch sprites for optimised rendering, default is disabled
- AllowFocus (boolean) - If set to true then this scene will receive input focus events that the current scene would usually receive exclusively. This is useful if you have a HUD overlay that has functionality but it cannot be the current scene as the game scene is currently the current scene
- Style (string) - The style can be used to set properties from a common set of properties known as a style (se Style tag). Note that the style can be set only when the scene is declared. This property cannot be read or written.
- UserProperties (object, read-only) – The scenes user properties list
- TargetX (actor name, write-only) – Sets the actor that the camera will be used to track its x-axis
- TargetY (actor name, write-only) – Sets the actor that the camera will be used to track its y-axis
- Bindings (bindings name) – Defines a bindings list that binds variables to scene properties. Reading this property will return a bindings object
- TimeScale (number) – Changes the speed at which an attached timeline is played. This property cannot be set when initially declaring the scene
- ScriptEngine (script engine type) – The current script engine type

Visual Properties:

- Position (x, y) – The position of the scene in virtual canvas coordinates
- PositionX (number) – The X position of the scene in virtual canvas coordinates
- PositionY (number) – The Y position of the scene in virtual canvas coordinates
- Angle (degrees) – Orientation of the scene
- Scale (number) – The scale of the scene (default is 1.0)
- CanvasSize (x, y) - The virtual canvas size of the screen
- CanvasFit (none, best, both, width, or height) - The type of method to use when scaling the canvas to the devices native screen resolution
- CanvasOrigin (top, left, topleft or centre) - Where to locate the canvas origin
- Layer (whole number) - The visual layer that this scene should be rendered on (Valid values are 0 to 9)
- Colour (r, g, b, a) - The colour of the scene, Each component red, green, blue and alpha should be between the value of 0 and 255 (0 is no colour, whilst 255 is full colour)
- Opacity (opacity) - The opacity of the scene (how see-through the scene appears). Opacity value ranges from 0 (invisible) to 255 (fully visible)
- Timeline (timeline) - The time line that should be used to animate the scene. Reading this property will return a timeline object

Event Properties:

- OnSuspend (actions list, write-only) - Provides an actions group that is called when the scene is suspended
- OnResume (actions list, write-only) - Provides an actions group that is called when the scene is resumed
- OnCreate (actions list, write-only) - Provides an actions group that is called when the scene is created
- OnDestroy (actions list, write-only) - Provides an actions group that is called when the scene is destroyed
- OnKeyBack (actions list, write-only) - Provides an actions group that is called when the user presses the back key
- OnKeyMenu (actions list, write-only) - Provides an actions group that is called when the user presses the menu key
- OnOrientationChange (actions list, write-only) - Provides an actions group that is called when the user changes the devices orientation
- OnTick (actions list, write-only) - Provides an actions group that is called every time the scene is updated (30 to 60 times per second)

- OnDeviceSuspended (actions list, write-only) - Provides an actions group that is called when the app is suspended
- OnDeviceResumed (actions list, write-only) - Provides an actions group that is called when the app is resumed
- OnDeviceVolumeUp (actions list, write-only) - Provides an actions group that is called when the volume up button was pressed
- OnDeviceVolumeDown (actions list, write-only) - Provides an actions group that is called when the volume down button was pressed
- OnDeviceNetworkOnline (actions list, write-only) - Provides an actions group that is called when the network goes online
- OnDeviceNetworkOffline (actions list, write-only) - Provides an actions group that is called when the network goes offline
- OnPan (actions list, write-only) - Provides an actions group that is called when the user pans the camera in the scene

Physical Properties:

- Physics (boolean) - Swicthes physics on and off, note that if the scene must have been declared with physics enabled to have any affect
- Gravity (x, y) - Box2D directional world gravity
- WorldScale (x, y) - Box2D world scale
- DoSleep (boolean) - If set to true then actors that utilise physics will be allowed to sleep when they are not moving / interacting. Can only be set when scene is declared
- Physics (boolean, read-only) - Enables or disables physics processing in the scene. Can only be set when scene is declared
- PhysicsTimestep (number) – Sets how fast the physics engine updates the physics world in seconds. Setting this to a value of 0 will use a variable time step based on the current frame rate of the app. Default value is 1/30.

3.3 Scene Virtual Canvas

Targeting a large selection of different phones, tablets and other devices with a variety of different screen sizes and aspect ratios can be difficult to manage. Luckily scenes can take care of this for you. A scene is quite clever 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 app using a static resolution. Lets take a look at an example that creates a scene with a virtual canvas of 800x480:

```
<?xml version="1.0"?>
<xml>
    <!-- Create a scene to hold our game / app actors -->
    <Scene Name="AppScene" Current="true" CanvasSize=”800, 480” CanvasFit=”best” >

        <!-- Actors go here -->

    </Scene>
</xml>
```

Using the AppScene above regardless of whatever size screen we run our app on our content will be scaled to fit an 800x480 pixel display resolution because we specified a CanvasSize of “800x480”. We also told the scene to use its best judgement when scaling the scene by specifying CanvasFit=”best”. As an example if we run our app on an iPhone retina display at 960x640 pixels and using a canvas fit method of best fit then our app will be scaled to best fit the 960x640 resolution, which would be 960x576 pixels. The scene would be centred on the screen leaving a little space at the top and bottom of the display.

Scenes support a number of ways of fitting the the virtual canvas to the device screen including:

- none - No scaling is performed. This is ideal if you want a 1:1 pixel ratio and mostly used by apps and games that use proportional sizing and positioning.
- both - The scene is scaled to fit the exact size of the display ignoring aspect ratio (aspect ratio is the ratio of the screens width to the screens height). The problem with this method of scaling is that the scene will be stretched on displays with different aspect ratios which can make certain apps look odd.
- best - The scene is scaled to best fit the devices display size whilst maintaining aspect ratio. This method of scaling prevents stretched graphics, however it can leave gaps to the top and bottom or left and right
- width / height - This method of scaling is the same as best except scaling is locked to either the width or the height.

Scenes also provide a way to allow you to specify how coordinates are interpreted allowing you to place content relative to either the centre, left, top-left or top of the screen. The scenes origin is by default set to the centre of the scene, but it can be changed by supplying the CanvasOrigin property when declaring the scene, e.g.:

```
    <Scene Name="AppScene" Current="true" CanvasOrigin=”topleft” >
```

This scene places actors relative to the top-left hand corner of the scene instead of relative to the centre. However, we find that having the scenes origin at the centre is much better and feels more natural when designing apps an games for a variety of screen sizes as content can be allowed to extend from the centre and beyond the scenes size

The default canvas size for a scene is the devices native display resolution

3.4 Scene Layers and Actor Layering

Because an app can contain multiple scenes that are visible at the same time, its sometimes required to sort the order in which these scenes are displayed. For example, you may have a game that has a background layer, a game object layer and a foreground layer. The visibility ordering of these scenes will be displayed in the opposite order to which you create them. For example, if you create the foreground scene and then the background scene afterwards then the background scene will be drawn on top of the foreground scene, which is the wrong way around. You could remedy this problem by adding the background scene first, but sometimes this is not practical. To eliminate the problem, scenes can be placed on a layer of depth by setting the Layer=”layer number” (layer numbers 0 to 9 are supported with 9 being the highest layer) property of the scene. Scenes that are placed on the same layer will be sorted in the order they were added, with more recent additions appearing on top.

Each scene also determines how actors are sorted in a scene (we will discuss actors in great depth later). We can define how many layers are available for the actor system to use by adding the Layers=”number of layers” property to the scene definition. The default is 16 which is fine for most applications.

An example has been provided that demonstrates scene layering (see SceneLayers example). Lets take a quick look at an extract of the XOML for this example:

```
<!-- Create scene 1 on highest layer -->
<Scene Name="Scene1" Layer="9">
</Scene>


<!-- Create scene 2 on middle layer -->
<Scene Name="Scene2" Layer="5">
</Scene>

<!-- Create scene 3 on lowest layer -->
<Scene Name="Scene3" Layer="0">
</Scene>
```

Here we create 3 scenes and assign each one a different layer.

3.5 Current Scene

An app generally consists of a number of different scenes that all contain their own objects. However only one scene can normally receive input. This scene is known as the current or focus scene. When you declare a scene it can be made to be the current scene by adding Current=”true” to the scene definition. By comparison, you could think of all of the different windows on your PC or Mac as separate scenes, but only one of those windows can usually accept user input at one time.

When a scene is made current it is brought to the front of any scenes that are on the same layer (more on layers later).

It is also possible to mark scenes so that more than one scene can receive input events at the same time. This can be useful if you have two scenes on lower layers that need to also receive input. For example, you may have a heads-up display that has functional buttons on the highest layer and the game scene on a lower layer that also needs to receive input events. To mark a scene to receive input events you need to add the AllowFocus=”true” attribute to your scene definition.

3.6 Suspending and Resuming Scenes

By default all active scenes in your app will be processed continuously. However this is not always required and for apps that contain a lot of active scenes it can slow down your app. Scenes can be suspended so that they stop processing (they will still be visible). In fact, scenes that allow suspending / resuming are automatically suspended when you change the current scene to another scene. To enable suspending / resuming for a scene you should set its AllowSuspend=”true” property. When a scene is suspended or resumed it will fire a suspend or resume event which allows you to perform actions (more on this later).

Scenes can be suspended and resumed using the following actions:

- ChangeScene – The previous scene will be suspended and the new scene will be resumed (provided that the scenes have AllowSuspend="true" defined)
- SuspendScene – Suspends the scene even if suspend / resume is disabled for the scene
- ResumeScene – Resumes the scene even if suspend / resume is disabled for the scene

3.7 Scene Extents

Scenes can be as large or as small as you like, they can even be many times larger than the devices screen size. This enables you to create content that exists in a much larger space than the screen can display. You can then allow the user to pan around the scene to view more of the content. The size of your scenes world can be set using the Extents property which accepts 4 values which represent the rectangular area that objects can exist in. Actors can be made so that when they hit the edge of the scenes extents they re-appear on the opposite side (much like asteroids in the old Atari game Asteroids does), this is called wrapping. An example called SceneExtents has been provided that shows an actor moving across the scene, it wraps back around to the opposite end of the scene when it hits the scenes extents. Lets take a quick look at the XOML for this example:

```
    <!-- Create scene with narrow extents -->
    <Scene Name="Scene1" Extents="-100, -100, 200, 200" Current="true">

        <!-- Create a label object -->
        <Label Font="serif" Background="Button1Brush" BackgroundColour="255, 80, 80, 255"
Text="Wrapping" WrapPosition="true" OnTick="Update">
            <Actions Name="Update">
                <Action Method="AddProperty" Param1="Position" Param2="2, 2" />
            </Actions>
        </Label>

    </Scene>
```

This example introduces some concepts that you have not yet met such as actions and events. You do not need to worry about these for now as we are only focusing on the bold sections. Note how we have set the scenes extents using Extents=”-100, -100, 200, 200”. This collection of 4 value represent the left, top, width and height of the scenes extents.

Also note that our Label has been given a new property called WrapPosition which is set to true. This tells the Label actor to wrap its position when it hits the edges of the scenes extents.

3.8 Scene Clipping

When a scene is smaller than the devices display size its sometimes useful to be able to ensure that anything that is drawn within the scene is not drawn outside the scene. To accomplish this we need to clip content that falls outside the scenes area of coverage. To tell a Scene to clip its contents you add the Clipping property which takes 4 values (left, top, width, height). An example has been provided called SceneClipping which demonstrates how to apply scene clipping. Lets take a quick look at this example:

```
    <!-- Create scene with narrow extents -->
    <Scene Name="Scene1" Extents="-100, -100, 200, 200" Clipping="-100, -100, 200, 200"
Current="true">
    </Scene>
```

In this example we limit the scenes extents and set the clipping area to -100, -100, 200, 200.

By default the scenes clipping area will move around with the scene, although you can modify this behaviour to force the clipping area to remain in place on screen. To force clipping to remain in place add ClipStatic=”true” to your scene definition.

3.9 Scene Events

An event is something that occurs in your app that you may or may not want to know about and act upon. For example, when the scene is first created the OnCreate event is raised. Using actions (explained later) you can react to these events and modify your apps behaviour.

Scenes support an events system that enables you to listen to and react to certain events occurring in the scene. You can listen to events by adding an OnEvent property to the scene. Lets take a look at the list of events that can occur in a scene:

- OnSuspend - This event is raised when the scene is suspended. A scene is suspended if it is the current scene and another scene becomes the current scene (for scenes that support suspend / resume) or when a SuspendScene action is called
- OnResume - This event is raised when a scene is resumed (become the current scene) or when a ResumeScene action is called
- OnCreate - This event is raised when the scene is first created and gives you the opportunity to handle post scene creation logic. You could use this event to for example play a sound or hide another scene etc..
- OnDestroy - This event is raised when the scene is about to be destroyed, this gives you the opportunity to clean-up certain elements
- OnKeyBack - This event is raised when the user pressed the back button
- OnKeyMenu - This event is raised when the user pressed the menu button
- OnOrientationChanged - This event is raised when the devices orientation changes an allows yo to for example modify the scenes layout.
- OnTick - This event handler is raised every time the scene is updated (30 to 60 times per second)
- OnDeviceSuspended (actions list, write-only) - Provides an actions group that is called when the app is suspended
- OnDeviceResumed (actions list, write-only) - Provides an actions group that is called when the app is resumed
- OnDeviceVolumeUp (actions list, write-only) - Provides an actions group that is called when the volume up button was pressed
- OnDeviceVolumeDown (actions list, write-only) - Provides an actions group that is called when the volume down button was pressed
- OnDeviceNetworkOnline (actions list, write-only) - Provides an actions group that is called when the network goes online
- OnDeviceNetworkOffline (actions list, write-only) - Provides an actions group that is called when the network goes offline

- OnPan (actions list, write-only) - Provides an actions group that is called when the user pans the camera in the scene

The SceneEvents example shows an example on how to handle the scenes OnTick event.

3.10 Scene Animation Properties

Scenes can be animated using animations defined in XOML (we will look at animation in the animation chapter later). All of the properties defined in section 3.2 can be animated with exception to those properties that are marked as read-only or can only be set when the scene is initially declared.

The SceneAnimation example has been provided that shows how to apply animation to a scene

3.11 Scene Modifiable Properties

Scenes can be queried and modified after their creation using actions, command and scripts. All of the properties defined in section 3.2 can be modified with exception to those properties that are marked as read-only or can only be set when the scene is initially declared. Most properties can be read with exception to:

- Any of the event handlers such as OnTick
- TargetX to track its x-axis
- TargetY

It either doesn't make sense to read these properties or they are not stored in a manner that allows them to be easily retrieved.

3.12 Scene Rendering

A scene has no actual visible component, instead it renders all of its contained actor objects. All actor objects are moved, rotated and scaled by the same amount that the scene is. For example, if you move the scene to the left, all contained actors will also moved to the left. If you rotate the scene to the right 45 degrees then all actors also rotate 45 degrees to the right. In addition, all scene actors will be scaled by the scenes colour and opacity settings, so if you fade down the scene all actors will also fade down with it.

Scenes support a method of optimisation known as batch rendering. This is the process of combining actors together at render time to render them all in one go, which can substantially speed up rendering for scenes that contain many actors. To force a scene to use batch rendering add Batch=”true” to the scene definition. However, scenes that are marked as batch enabled will not sort their contained actors properly with the likes of fonts, so you will need to use actor layers to force the sorting.

When a scene is hidden from view, all of its actors will also be hidden. When a scene is deleted all of its actors are also deleted.

3.13 Scene Cameras

A camera is a view into the scene from a specific position, angle and scale. In order to move, rotate and scale a scene a camera should be created and attached to it. To create a camera you use the Camera XOML tag:

```
<!-- Create a camera -->
<Camera Name="Camera1" />
```

To attach the camera to the scene we add the Camera attribute to the scene definition:

```
<Scene Name="Scene1" Current="true" Camera="Camera1">
```

When we move, scale or rotate a scene we are actually modifying the camera view within the scene. When we move the camera objects tend to move in the opposite direction, for example moving the camera left moves the scene actors to the right. If you think about how a real camera works, when you move the camera in one direction the view seen by the camera moves in the opposite direction.

Its possible to create a number of different cameras and switch between them to offer different views into the scene.

Cameras offer a great out of the box feature called touch panning, which enables the user to pan the camera around a scene on the x and y axis by dragging their finger around the scene. Take a look at the ScenePanning example for an example showing how to use touch panning.

Lets take a look at what properties the Camera tag supports:

- Name (string) - Cameras resource name
- Position (x, y) - Start position of the camera
- Angle (degrees) - Rotation of the camera
- Scale (scale factor) - Scale of the camera (can be used for zoom effects)
- TouchPanX / TouchPanY (boolean) - Setting true causes the camera to move around when the user drags their finger on the screen using velocity obtained from the speed of the users drag. Separate X and Y axis panning can be enabled / disabled.
- VelocityDamping (x, y) - Amount of damping to apply to the cameras velocity (x, y) each frame. This value when set to a value of less than 1.0, 1.0 will cause the camera to slow down over time.
- IgnoreActors (boolean) - When set to true and touch panel is enabled, dragging a finger over an actor will still be classed as a touch pan drag. This option basically allows the user to touch pan the scene regardless of what the user touches on the screen.
- TargetX (actor name, write-only) – An actor that the camera will be used to track its x-axis position
- TargetY (actor name, write-only) – An actor that the camera will be used to track its y-axis position
- FollowSpeed (x, y) – The speed at which the camera should track the specified actor(s). Higher values will catch the camera up to the actors target position faster. Setting either value to 0 will result in the camera locking to the zeroed target axis
- Tag (string) - Group tag

Some of these properties can be modified via the scene such as Position, Angle and Scale as changes to these properties go straight to the attached camera. You can set all properties for a camera via LUA using the camera.set() function.

3.14 Scene Augmentation

A scene once declared in XOML can later be updated / augmented with additional XOML elsewhere. For example, lets say that you declare some common scene that contains a basic background and some other elements that are common across a number of screens. You can later load the scene and then augment it by declaring the scene again supplying the additional elements inside the newly declared scene:

```
<Scene Name="CommonScene" ............ >
    <Original_Element1 />
    <Original_Element2 />
    <Original_Element3 />
</Scene>
```

Now declare a 2nd scene with the same name:

```
<Scene Name="CommonScene">
    <Extra_Element1 />
    <Extra_Element2 />
    <Extra_Element3 />
</Scene>
```

In memory the scene now looks like this:

```
<Scene Name="CommonScene" ............ >
    <Original_Element1 />
    <Original_Element2 />
    <Original_Element3 />
    <Extra_Element1 />
    <Extra_Element2 />
    <Extra_Element3 />
</Scene>
```

For a working example of augmented scenes take a look at the SceneAugmentation example.

3.15 Scene Physics

XOML scenes can run actors that are under the influence of the Box2D physics systems. The scene system allows you to specify some scene global information about the physics simulation. The following scene global properties can be set by adding the relevant tags to the scene definition:

- Gravity (x, y) - Box2D directional world gravity. Box2D uses directional gravity which means you can have gravity act in any direction.
- WorldScale (x, y) - Box2D world scale. This value determines how your visible world scales to the Box2D world (default is 10, 10)
- DoSleep (boolean) - If set to true then actors that utilise physics will be allowed to sleep when they are not moving / interacting, this can help speed up scenes that contain many actors that are under Box2D control.
- Physics (boolean) - Enables or disables physics processing in the scene, enabled by default. Disable in scenes that do not use physics to maximise performance.
- PhysicsTimestep (number) – Sets how fast the physics engine updates the physics world in seconds. Setting this to a value of 0 will use a variable time step based on the current frame rate of the app. Default value is 1/30.

To see how scene physics is used please take a look at the ScenePhysics example.

3.16 User Properties

User properties give you the opportunity to add additional custom properties to the scene that can be accessed from XOML and Lua.

```
<Scene Name="Scene1">
    <!--Create user properties list that is assigned to the scene-->
    <UserPropertiesName="GameProperties">
        <PropertyName="Counter"Type="int"Value="0"/>
        <PropertyName="MyAngle"Type="float"Value="0"/>
    </UserProperties>
</Scene>
```

And a short example showing how to access the properties from XOML:

```
    <Actions Name="MyActions">
         <Action Method="AddUserProp" P1="Counter" P2="1" P4="Scene1" />
         <Action Method="SetUserProp" P1="MyAngle" P2="180" P4="Scene1" />
    </Actions>
```

4.0 Actors

4.1 Introduction

In the Scenes Introduction chapter we spoke about scenes and compared them to the scenes of a movie set. Carrying on from our movie business analogy you can think of Actors as the actors and scenery objects that make up the movie scene with each actor having its own function and purpose within the scene. Actors are the central focus of all XOML development as they provide the actual app or game functionality and interaction that makes your game or app what it is. Actors are created inside a scene and belong to that scene. When the scene is destroyed its actors will also be destroyed.

XOML comes with many pre-defined actor types that each have their own specific purpose. For example, we have Text Actors that display text, Image Actors that display images, Icon actors that act as buttons and many more. We will cover every type of actor in this chapter although user interface actors will be covered in more depth in the User Interface chapter.

Actors can also be augmented to give them additional functionality. Actors can be augmented in a number of ways including adding modifiers to them which modify their behaviour, they can respond to events with actions which in turn can affect the scene and other actors and they can run XOML programs or call functions in script languages to create more complex behaviours. Actors can also take full advantage of the built in physics and animation systems to truly bring them to life.

If you have looked at any of the scene examples whilst reading the section covering Scenes then you should already be somewhat familiar with the Label actor and hopefully you already have a feel for how actors are used. We will now go into Actors in more detail to help you fully understand how they work and their importance in XOML development. Lets take a quick look at an actor definition in XOML to see what it looks like:

```
<!-- Create a label object -->
<Label Font="serif" Text="Hello World" Position="10, 20" Size="-100, -100" />
```

The Label is basically an actor that can display an image background with text overlaid on top of it. The label is positioned at x=10, y=20 and has a size that is 100% the screen width and 100% the screen height and displays the phrase “Hello World” using the serif font. We will now look at the different types of actors that are available in XOML.

4.2 The Many Faces of Actors

XOML provides a large array of different types of actors, but all actors are derived from 2 basic types of actors:

- ActorImage - This type of actor enables you to display images or more commonly portions of images within a scene
- ActorText - This type of actor enables you to display text using a font within a scene

All other actors that can be created are derived from these two types of actors.

Now lets take a brief look at all of the other types of actors that XOML can use:

- ActorImage - Basic image actor that can display an image or brush
- ActorText - Basic text actor that can display text using specific fonts
- ActorParticles - An actor that can generate particles used for special / spot effects
- Icon - An Image actor that can also be used as a button or check box
- Label - An image actor that contains a text actor, can also be used as buttons
- VideoOverlay - An image actor that can display video content
- TextBox - A label actor that allows text entry via on screen keyboard
- Slider - Image actor that can be used as a slider control
- ListBox - A complex actor that displays a list of child actors that can be selected / toggled etc
- Grid - A complex actor that displays a grid / data grid and allows selection of cells
- Image View / Text View - Displays an image or text area that can be pinch zoomed and panned
- Web View - Displays web content
- Tab Bar - A complex actor that can be used to create navigation between different views
- Canvas, StackPanel and WrapPanel - Image actors that act as containers that arrange content in specific ways

As you can see there is a large selection of actors. However we will only be covering ActorImage and ActorText actors in this section. The remaining actors will be covered by the user interface chapter.

4.3 The Basic Actor

All actors share some common base functionality and properties which we will discuss in detail during this chapter. We will begin by looking at common properties that are shared across all of the different types of actors:

General Properties:

- Name (string) - Name of the actor, used to refer to the actor from scripts and such. Note that no two actors in the same scene should share the same name.
- Style (style) - Provides a style that this actor should use to style its properties
- Type (number) - A numerical type that can be used to identify the type of this actor (default is 0)
- Active (boolean) - Active state (default is true), actors that are not active will not be processed
- Visible (boolean) - Visible state (default is true), actors that are not visible will not be shown on screen.
- UserProperties (object, read-only) – The scenes user properties list
- Tappable (boolean) - If true then this actor will receive touch events when touched (default is true)
- Draggable (boolean) - When set to true the user can drag the actor around the world with their finger (default is false)
- Timeline (timeline) - The time line that should be used to animate the actor
- TimeScale (number) - Speed at which to play back any attached timelines (can only be set from actions, commands and Lua)
- Bindings (bindings list) - Name of the bindings set that will be bound to this actors properties
- Binding (binding) - A simple binding that is bound to a specific property of the actor
- UserData (number) – A user data, can be used to store anything
- Scene (scene, read-only) – The scene that the actor lives in
- Destroyed (boolean (read-only)) – Returns the destroyed state of the actor. When an actor is killed it is not deleted until the end of the current frame. You can use this property to determine whether or not the actor will be available next frame.

Visual Properties:

- Anchor (string) – Defines where the visual anchor point lies. Possible values are topleft and centre, centre being the default. When an actors anchor is set to centre it will be positioned using its visual centre, if set to topleft then it will be positioned using the top-left corner of the visual. Note that topleft is not currently supported by actors that are placed inside list boxes, grids etc..
- Position (x, y) - The actors position within the scene (default is 0, 0)
- PositionX (number) - The actors position on the x-axis
- PositionY (number) - The actors position on the y-axis
- PositionOrg (x, y (write-only)) – Sets the position relative to the original position of the actor when it was created. Can only be set via property setters
- PositionOrgX (number (write-only)) - Sets the x-axis position relative to the original x-axis position of the actor when it was created. Can only be set via property setters
- PositionOrgY (number (write-only)) - Sets the y-axis position relative to the original y-axis position of the actor when it was created. Can only be set via property setters
- PercPos (boolean) - When set to true positions are specificity as a percentage of the devices screen size (default is false)
- Angle (degrees) - The angle of the actor (default is 0)
- Origin (x, y) - Sets the offset around which the actor will rotate and scale (default is 0,0)
- Depth (number) - Depth of the actor in 3D (larger values move the sprite further away, default is 1.0 for parent actors and 0.0 for child actors)
- Scale (x, y) - The x and y axis scale of the actor (default is 1, 1, which represents no scaling)
- ScaleX, ScaleY (number) - The separate x and y axis scale of the actor
- Colour (r, g, b, a)- The colour of the actor, Each component red, green, blue and alpha should be between the value of 0 and 255 (0 is no colour, whilst 255 is full colour, default)
- Opacity (opacity) - The opacity of the actor (how see-through the actor appears). Opacity value ranges from 0 (invisible) to 255 (fully visible, default)
- Layer (number) - The visible layer that the actor should appear on (maximum layer number is limited by the number of actor layers defined by the scene)
- Docking (dock position) - When set will dock the actor to an edge of the screen or canvas, valid values are top, left, right, bottom, topleft, topright, bottomleft and bottomright
- ScreenDocking (boolean) – Enables / disables screen docking
- HoldFocus (boolean) – If true then the actor will hold the focus and not release

it
- Margin (left, right, top, bottom) - The amount of space to leave around the actor when placed in a container or docked
- UseParentOpacity (boolean) - When set to true this actor will scale its own opacity by its parents opacity (default is true)
- IgnoreCamera (boolean) – If set to true then this actor will ignore the cameras transformation staying in place when when the camera moves, scales or rotates (default is false)
- Orphan (boolean) – If set to true then this actor will ignore its usual parent-child hierarchy when sorting by layer. This enables child actors to be sorted by layer with parent actors (default is false)
- Geometry (geometry-name) – Sets the geometry that will be used to render the actors shape

Physical Properties:

- Velocity (x, y) - Initial velocity of the actor
- VelocityDamping (x, y) - The amount to dampen velocity each frame, values of less than 1.0 will slow the actor down over time, values of greater than 1.0 will speed the actor up over time.
- AngularVelocity (number) - The rate at which the orientation of the actor changes in degrees per second
- AngularVelocityDamping (number) - The amount of rotational velocity damping to apply each frame
- WrapPosition (boolean) - If true then the actor will wrap at the edges of the canvas
- Box2dMaterial (material) - Sets the physical material type used by the Box2D actor. Actors also support multiple materials via the Fixtures inner tag
- Shape (shape, write-only) - Box2D fixture shape for that represents this actor during collisions. Actors also support multiple shapes via the Fixtures inner tag. Setting the shape from an action or script will remove all existing shapes from the actor, adding to the shape property will add additional shapes to the actor
- COM (x, y) - Centre of mass of Box2D body. Actors also support multiple COM's via the Fixtures inner tag
- Sensor (boolean) - Can be used to set the actor as a sensor (default is false). Actors also support multiple sensor specifications via the Fixtures inner tag
- CollisionFlags (category, mask, group) - Box2D collision flags. Actors also support multiple collision flags via the Fixtures inner tag
- Collidable (boolean) – If true then collision flags are used to determine

collision, else collision is disabled
- Torque (torque, write-only) - Applies a torque to the actor
- AngularImpulse (impulse, write-only) - Applies an angular impulse to the actor
- Force (x, y, world_x, world_y, write-only) - Applies the force x, y to the actor at the world position world_x, world_y
- LinearImpulse (x, y, world_x, world_y, write-only) - Applies the linear impulse x, y to the actor at the world position world_x, world_y
- BodyAwake (boolean) – The awake status of the actors physical body. A body goes to a sleep state when it comes to rest

Event Properties:

- OnBeginTouch (actions list, write-only) - When the user begins to touch this actor it fires this event and calls the supplied actions list. Actor also supports OnBeginTouch2 to OnBeginTouch5 representing the 2nd to 5th touches on multi-touch devices.
- OnEndTouch (actions list, write-only) - When the user stops touching this actor it fires this event and calls the supplied actions list. Actor also supports OnEndTouch2 to OnEndTouch5 representing the 2nd to 5th end touches on multi-touch devices.
- OnTapped (actions list, write-only) - When the user taps this actor it fires this event and calls the supplied actions list. Actor also supports OnTapped2 to OnTapped5 representing the 2nd to 5th taps on multi-touch devices.
- OnCreate (actions list, write-only) - When this actor is first created it fires this event and calls the supplied actions list
- OnDestroy (actions list, write-only) - When the actor is about to be destroyed it fires this event and calls the supplied actions list
- OnOrientationChange (actions list, write-only) - When the devices orientation changes it fires this event and calls the supplied actions list
- OnCollisionStart (actions list, write-only) - When two actor that have Box2D physics enabled start to collide it fires this event and calls the supplied actions list (only if the actor has the iw_notifycollision modifier attached)
- OnCollisionEnd (actions list, write-only) - When two actor that have Box2D physics enabled stop colliding it fires this event and calls the supplied actions list (only if the actor has the iw_notifycollision modifier attached)
- OnTick (actions list, write-only) - Provides an actions group that is called every time the scene is updated (30 to 60 times per second)
- Bubbling (boolean) - When set to true touch events can bubble up from child actors

Miscellaneous Properties:

- GridPos (x, y) - Grid cell in which to place the actor in a grid
- LinkedTo (actor) - Name of actor that this actor links to (advanced)

4.4 Image Actors

Image actors form the backbone of most of the games and apps developed with XOML. An image actor is an actor that represents itself on screen using an image or portion of an image. All of the user interface actors are derived from an image actor. Image actors have the following specific properties:

- Brush (brush) - Specifies a brush that is used to define the image and source rectangle
- Image (image) - The image that is to be used as the actors visual (deprecated, use brushes instead, with exception to particle actors)
- Size (x, y) - The world size of the actor
- SrcRect (x, y, width, height) - 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. (deprecated, use brushes instead, with exception to particle actors)
- Tiling (x, y) – Sets the texture coordinate tiling factor for this actor (default is 1.0, 1.0), if x ot y are > 1.0 then Tiling will also be set to true
- Tiled (boolean) – When set to true, texture coordinates that fall outside the normal UV range will be tiled, otherwise they will be clamped (default is false)
- FlipX (boolean) - If true then this actor is horizontally flipped
- FlipY (boolean) - If true then this actor is vertically flipped
- Skew (top, bottom, left, right) - Four parameter skewing, which allows the actor to be skewed in four different directions
- BeforeChildren (boolean) - When set to true this actor will be rendered before its children, otherwise it will be rendered afterwards (default is true)
- Filter (boolean) - When set to true this actor will rendered using filtering (default is true)
- AlphaMode (alpha_mode) - Sets the mode to use when mixing transparency (alpha) from the image. AlphaMode can be one of none, half, add, sub and blend (default mode is blend)
- AspectLock (lock_mode) - Locks the aspect ratio of the actor to fit to the screens aspect ratio lock_mode can be one of x, y or none (default is none)

Notes:

- An image actor requires an Image or a Brush to be defined
- Because an image actor inherits from a basic actor, it inherits all of the basic actors properties as well as those properties shown above.

Lets take a look at an image actor definition in XOML:

```
<ActorImage Brush=”Brush1” Position="10, 20" Size="-100, -100" />
```

4.5 Text Actors

Text actors whilst can be created directly are usually used in conjunction with some other actor. For example user interface controls usually host a text actor on an image actor that represents the background to the user interface control. Lets take a look at text actor properties:

- Font (font) - Name of font to use to draw the text
- Rect (x, y, width, height) - The area that the text should be drawn inside of. If not provided then thearea will be calculated based on the screen size or the parent actor
- Text (string) - String to display
- AlignH (centre, left, right, write-only) - Horizontal alignment, default is centre
- AlignV (middle, top, bottom, write-only) - Vertical alignment, default is middle
- Wrap (boolean, write-only) - If true then text is wrapped onto next line if too long to fit on one line, if not then text will overhand its container
- Skew (top, bottom, left, right) - Four parameter skewing, which allows the actor to be skewed in four different directions
- BeforeChildren (boolean) - When set to true this actor will be rendered before its children, otherwise it will be rendered afterwards (default is true)
- Filter (boolean) - When set to true this actor will rendered using filtering (default is true)
- AlphaMode (alpha_mode) - Sets the mode to use when mixing transparency (alpha) from the actor. AlphaMode can be one of none, half, add, sub and blend (default mode is blend)
- AutoHeight (boolean) - When set to true the height of the text actor will be recalculated to make it large enough to fit its text content. For example, if you set the original Rect to only hold a single line but the text takes 3 lines then the actor will be resized to fit all 3 lines of text (default is false)
- TextSize (x, y, read-only) - Returns the area covered by the text

Notes:

- A font must be specified
- Because a text actor inherits from a basic actor, it inherits all of the basic actors properties as well as those properties shown above.

Now lets take a quick look at a text actor definition in XOML:

```
<ActorText Font="serif" Text="Hello World" Position="10, 20" />
```

4.6 Actor Hierarchies

Actors can be declared inside others actors to form an hierarchy. In this hierarchy actors that are declared inside other actors are called child actors, whilst the container actor is called the parent actor. You will see hierarchical actors used extensively throughout XOML, especially in regards to the user interface system. This child / parent system enables the creation of complex multi-part actors that are built from many actors.

When you place child actor inside a parent actor the child's actor is modified in a number of ways:

- Position, scale, rotation and depth become relative to the parent actor, so if the parent actor moves around then the child actors will follow. So for example if you set the child actors position to 0, 0 then it will be centered at the parent. If the parent rotates or scales then the child will also rotate and scale by the same rate as the parent
- The opacity of child actors will be scaled by the parent actors opacity if UseParentOpacity="true" was specified in the child actors definition. For example, if you set the parents opacity to half then all child actors will also appear at half opacity
- When a parent actor is hidden or made visible then all child actors will also be made hidden or visible
- Child actors no longer obey layer ordering and are instead layered in the order in which they are declared inside. You can override this behaviour by adding Orphan="true" property to the actor definition
- Child actors will steal input events from the parent actor, if the parent actor is drawn before the child actors, unless the parent has Bubbling="true" set, in which case both the child actor and its parent will both receive the input event

4.7 Absolute v Percentage Positioning and Sizing

When you begin developing apps and games for mobile you will quickly discover that you need to deal with a large selection of different screen sizes, orientations and aspect ratios. The scenes virtual canvas can go a long towards helping to alleviate this problem. However (for apps in particular) its often more appropriate to render the user interface at the screens native resolution but use a series of clever layout panels to position and size content.

Actors can be positioned in a scene using two methods:

- Absolute Positioned - The position you specify is in absolute or actual scene coordinates
- Percentage Positioned - The position you specify is a percentage of the devices screen size

The default mode of actor positioning is absolute. To change an actors positioning mode to percentage based you need to add PercPos="true" to the actors definition. Lets take a look at an example:

```
    <!-- Create a label at 10, 20 of size 200 x 100 -->
    <Label Font="serif" Text="Hello World" Position="10, 20" Size="210, 100" />

    <!-- Create a label at 10%, 20% of size 200 x 100 -->
    <Label Font="serif" Text="Hello World" Position="10, 20" PercPos="true" Size="210,
100" />
```

As well as absolute and percentage based positioning the size of an actor can be absolute or percentage based:

- Absolute Size - The size that you specify is in absolute / actual width and height in scene coordinates
- Percentage Size - The size that you specify is a percentage of the actors parents size or if the actor has no parent a percentage of the devices screen size

The default mode of actor sizing is absolute. By passing a negative width or height you switch the actors sizing mechanism to percentage based. Lets take a look at an example:

```
    <!-- Create a label at 0, 0 of size 50% screen width and height -->
    <Label Font="serif" Text="Hello World" Position="0, 0" Size="-50, -50" />
```

Using percentage based positioning and sizing with layout panels enables production of device screen independent apps and games

Its important to note that actors that are positioned / sized using percentages will change position / size when the devices screen orientation changes.

4.8 Docking and Margins

Its often very useful to place an actor somewhere on screen without worrying about the exact position where it needs to be placed. To solve this problem XOML introduces the concept of docking. Docking allows actors to be placed at the edge of the screen or the edge of a canvas. Lets take a look at a XOML example that shows actor docking:

```
        <!-- Create a bunch of docked labels -->
        <Label Font="serif" Background="Button1Brush" Size="100, 100" Text="Left" 
Docking="left" />
        <Label Font="serif" Background="Button1Brush" Size="200, 200" Text="Right" 
Docking="right" />
        <Label Font="serif" Background="Button1Brush" Size="150, 150" Text="Top" 
Docking="top" />
        <Label Font="serif" Background="Button1Brush" Size="180, 180" Text="Docked" 
Docking="bottom" />
```

You can see a working example of actor docking by taking a look at the ActorDocking example.

Note that when the devices screen orientation changes docked actors will re-dock themselves to dock to the new screen edges.

When an actor is docked at a screen or canvas edge you do not always want the actor to be appear right up against the edge, sometimes it looks better to leave a little space. XOML provies the Margin property that allows you to specify some space to leave around the actor. You can set the space to leave around an actor by adding Margin=”left space, right space, top space, bottom space”, where left, right, top and bottom space is the amount of space to leave around the actor. Lets revisit our previous example and add a margin to the first actor:

```
        <Label Font="serif" Background="Button1Brush" Size="100, 100" Text="Left" 
Docking="left" Margin="20, 0, 0, 0" />
```

Although this actor is docked to the left hand side of the screen it is no longer clinging to the edge, instead it is pushed our by 20 units, leaving a nice gap.

Margins sizes can also be specified using percentage sizing, by making margin values negative you force them to be percentage based. For example:

```
        <Label Font="serif" Background="Button1Brush" Size="100, 100" Text="Left"
Docking="left" Margin="-5, 0, 0, 0" />
```

This label now has a gap down its left hand side that is 5% of the screens width

4.9 Actor Layers

Its customary in game and app development to have some kind of visual order in which objects appear. For example, a button should appear over the background that it sits on or the foreground in a game world should appear above the background. To accomplish this, XOML uses actor layering. Each scene has a number of layers that actors can be placed on (layers 0 to 15 by default, although this can be changed in the scene definition). Actors that are placed on higher number layers will appear above actors on lower number layers. To set an actors layer you add the Layer=”layer number” property to the actor definition. e.g:

```
        <Icon Name=”icon1” Background="Button1Brush" Size="250, 200" Text="Layer 7"
Layer="7" />
        <Icon Name=”icon2” Background="Button1Brush" Size="500, 200" Text="Layer 1"
Layer="1" />
```

In this example, icon1 will appear above icon2 because icon1 is on layer 7 whereas icon2 is on layer 1. If the actor has no layer defined then it will be placed onto layer 0, if an actor has no layer defined but is a child of another actor then it will be placed on the same layer as the parent actor.

4.10 Actor Anchor

XOML enables actors to be placed using either their top-left or centre point. This point is known as the visual anchor point. Many traditional game engines utilise top-left anchor placement, however, when developing apps across multiple device resolutions we have found that it is much more useful and more intuitive to place actors using a centralised visual anchor point, thus it is the default anchor mode for all actors. You can however change the anchor point by adding Anchor=”topleft” to the actor definition.

One thing to note, most container UI actors do not currently support to-left anchor placement for actors that are placed within them.

4.11 Actor Origin

Sometimes we want objects to move a little differently than the norm. All actors by default spin and scale around their centre when rotated or scaled. We can modify this behaviour by moving the origin of the actor. The origin of an actor is the point around which the actor will spin and scale and by moving this origin we can change the position around which it spins and scales.

The ActorChildren example shows an example of moving the origin to change how one actor spins around another. If you pay attention to the small green cube that is orbiting Child 4 you will notice that it orbits Child 4 much like a space craft would orbit a planet. Lets take a look at the XOML for this green cube actor:

```
        <Icon Origin="0, 80" Background="Button1Brush" BackgroundColour="80, 255, 80, 255"
Size="20, 20" AngularVelocity="7.5" />
```

In this example, we add the Origin=”0, 80” to the icon definition which pushes its centre of rotation 80 units downwards, this in turn causes the actor to have an orbital distance of 80 units from the centre of the parent actor.

4.12 Actor Animation

Because there is a lot of focus on actors in XOML, XOML provides a number of ways to animate them. Actors can be assigned an animation timeline which contains many animations that target its properties for example (animations and timelines are covered in the animation section). Actors can also be placed under control of the physics system (covered in the next section) Besides timelines and physics, actors can be animated using linear and angular velocity. If we take a look our basic actor properties again we notice two particularly interesting properties:

- Velocity (x, y) - Initial velocity of the actor
- AngularVelocity (number) - The rate at which the orientation of the actor changes in degrees per second

These two properties allow us to set an initial linear velocity (moves the actor in a direction) and angular velocity (spins the actor at a set speed).

If you one again take a look at the ActorChildren example, you will notice that each actor has an AngularVelocity property defined:

```
        <Label Position="0, 0" Font="serif" Background="Button1Brush" BackgroundColour="80,
80, 255, 255" Size="100, 100" Text="Parent" AngularVelocity="1" >
```

By setting this property we can et the actor off spinning as soon as it comes into the game world.

4.13 Dragging Actors

XOML provides a neat little feature for all actors called dragging. Any actor can be marked as dragabble by adding Draggable=”true” to the actor definition. Marking the actor as draggable allows the user to drag the actor around the screen using their finger.

4.14 Actor Physics

There's nothing like adding realistic physics to a game to bring it to life and increase its immersion factor. To that end XOML provides out of the box physics via Box2D. Any actor can be made to use physics by simply assigning a shape and a Box2D material to it in the actor definition. This includes all actors including user interface components! Lets have a quick recap of the available properties that affect the physics of our actor:

- Velocity (x, y) - Initial velocity of the actor
- VelocityDamping (x, y) - The amount to dampen velocity each frame, values of less than 1.0 will slow the actor down over time, values of greater than 1.0 will speed the actor up over time.
- AngularVelocity (number) - The rate at which the orientation of the actor changes in degrees per second
- AngularVelocityDamping (number) - The amount of rotational velocity damping to apply each frame
- WrapPosition (boolean) - If true then the actor will wrap at the edges of the canvas
- Box2dMaterial (material) - The physical material type used by the Box2D actor
- Shape (shape) - Box2D fixture shape for that represents this actor during collisions
- COM (x, y) - Centre of mass of Box2D body
- Sensor (boolean) - Can be used to set the actor as a sensor
- CollisionFlags (category, mask, group) - Box2D collision flags
- BodyAwake (boolean) – The awake status of the actors physical body. A body goes to a sleep state when it comes to rest

Actors support a quick and easy single shape, material, COM and sensor setting as well as multiple shapes, materials, COM's and sensor setting via the Fixtures inner tag. Note that if no shape is assigned to an actor but it has a Box2dMaterial assigned then a default shape will automatically be created for you using the size of the actor as the shape size.

Now would be a good time to open up the ActorPhysics (Test16) example that has been provided. Lets take a quick look at some of the important parts of the XOML for this example:

```
<!-- Create a scene with physics enabled -->
<Scene Name="Scene1" Current="true" Physics="true" WorldScale="1, 1" Gravity="0, 30"
```

```
Camera="Camera1">
```

Firstly we create a scene that can support physics by enabling physics, settings the worlds scale to 1, 1 (this means 1 scene unit is 1 physical unit) and then we set the gravity to act downwards.

```
    <!-- create Box2D materials -->
    <Box2dMaterial Name="Bouncey" Type="dynamic" Density="1.0" Friction="0.3" 
Restitution="0.6" />
    <Box2dMaterial Name="Heavy" Type="static" Density="2.0" Friction="0.8" Restitution="0.8" 
/>
```

Next we create two box2d materials. The first is quite bouncey and will be used by our bouncey box. The second material is quite solid and will represent out floor. It is also marked as a static material because we do not expect the floor to move. The first material is marked as dynamic becuase we expect our boncey box to move.

```
    <!-- Create Box2D shapes -->
    <Shape Name="Button" Type="box" Width="100" Height="100" />
    <Shape Name="Floor" Type="box" Width="1000" Height="100" />
```

Next we create two shapes (thes represent the physical shape of our objects). In this example we have a small box 100x100 units in size that represents our bouncey box and a much larger 1000x100 box shape that represents our solid floor.

```
    <!-- Create the floor -->
    <Label Position="0, 200" Font="serif" Background="Button1Brush" BackgroundColour="255, 
80, 80, 255" Size="1000, 100" Text="Floor" Shape="Floor" Box2dMaterial="Heavy" 
CollisionFlags="1, 1, 1" />
    <!-- Create an actor to drop onto the floor -->
    <Label Position="-50, -180" Font="serif" Background="Button1Brush" BackgroundColour="80, 
255, 255, 255" Size="100, 100" Text="Bouncey" Shape="Button" Box2dMaterial="Bouncey" 
CollisionFlags="1, 1, 1" />
```

Lastly we create a bunch of actors (we have only included the floor and one box here). The first actor (Floor) represents our floor and is assigned the Floor shape and the Heavy material. The second actor (Bouncey) is assigned the Button shape and Bouncey material.

You may by now have noticed an additional property called CollisionFlags. Collision flags are described as:

- Category - The category bits describe what type of collision object the actor is
- Mask - The mask bits describe what type of other collision objects this actor can collide with
- Group - The group index flag can be used to override category and mask, but we generally do not need to use it and usually set it to 0 for all actors.

The ActorCollisionFlags example has been provided to show how collision flags work.

The centre of mass (COM) of an actor in simple terms is the point at which the mass of the object is balanced. If the centre of mass of an object is directly at its centre then when hit towards its centre of mass by another object will generally cause it to spin around its centre. We can move the centre of mass of an actor to give the impression that the mass is centred elsewhere. For example, we may want to move the centre of mass towards the bottom of a box to make it look like it is weighted at the bottom. We can move an actors centre of mass by setting the actors COM attribute to the position of the centre of mass.

We can mark actors as being sensors instead of interactive physical objects. When an actor collides with a sensor the sensor does not affect the actor and the actor does not physically affect the sensor. In essence a sensor is there just to detect that something has collided with it. This type of actor is good for actors that represent switches and other types of none interactive actors. To mark an actor as a sensor set Sensor=”true” in the actor definition.

Often its useful to be able to assign a more complex shape to actors, for example, you may have a tree object that consists of a rectangle for the trunk area and an oval for the leafy area. Clearly we could not create such a shape using a single shape and to represent this type of shape to Box2D we would need to create and assign two separate shapes to the actor. We can assign multiple shapes to an actor by declaring a Fixtures inner tag and assigning the individual fixtures within it. Lets take a quick look at the Advanced Collision example (Test17):

```
<Icon Position="0,0" Background="Land" Box2dMaterial="FixedHeavy"
BackgroundColour="255,80,80,255" CollisionFlags="1,1,1">
	<Fixtures>
		<Fixture Shape="LandScapeBase" Box2dMaterial="FixedHeavy"
CollisionFlags="1,1,1" COM="0,0"/>
		<Fixture Shape="LandScapePeak" Box2dMaterial="FixedHeavy"
CollisionFlags="1,1,1" COM="0,0"/>
	</Fixtures>
</Icon>
```

In this example we create two separate fixtures each with their own shape, material, collision flags and centre of mass.

4.15 Actor Modifiers

Modifiers can be thought of as small functional building blocks that can be stacked in an actor or scene to extend the functionality of that actor or scene. For example, a typical modifier for a scene could be one that tracks the players scores / hi-scores, change day / night settings or detects special gestures. An actor modifier example could be a modifier that allows the actor move around using heading and speed or even a modifier with functionality specific to your game such as make a baddy that walks left and right on a platform. Lets take a look at how we add a modifier to an actor:

```
<ActorImage Name="Car" ........="" >
    <Modifiers>
        <Modifier Name="iw_notifycollision" Active="false" Param1="0" />
    </Modifiers>
</ActorImage>
```

In this example we add the iw_notifycollision modifier which allows the actor to respond to collision events between actors.

A modifier accepts the following properties:

- Name - Name of the modifier
- Active - Active state of the modifier
- Param1 to Param4 - Parameters that can be passed to the modifier when it is initialised

At the moment only a few different modifiers are available for actors but more will be added over time. Lets take a look at the modifiers that are currently available.

- Collision Notification Modifier - The iw_ notifycollision modifier when attached to an actor allows it to generate and respond to collision events between actors using OnCollisionStart() and OnCollisionEnd() event handlers (See ActorModifier example for an example showing how to use this modifier). Ths modifier accepts a number of parameters which include:
    - Param1 - An optional mask that can be used to mask collision with actors by their Type. The value supplied will mask actors by type and only allow collision events to be called for those actors that pass the bit mask. For example actor 1 could have a mask of 3 and actor 2 a mask of 1. if the mask is set to 1 then both actors can collide, but if the mask was set to 3 then they could not.

- Script Modifier - the iw_callscript modifier when attached to an actor will call a function in a script each time the actor is updated. This modifier is useful as an alternative to using OnTick event handlers. This modifier accepts the following parameters:
  - Param1 - Script function name
  - Param2 to Param3 - Parameters to be passed to the script function

Future out of the box functionality will be added to XOML using modifiers.

4.16 Actor Scripts

Whilst XOML is very powerful it does have some limits when it comes to defining very complex application / app logic, for example path finding algorithms would be quite cumbersome to create using XOML. Actors support the calling of script functions via actions in response to various events occurring. Also, as shown in the previous section, a modifier can be added to an actor that automatically calls script functions every time the actor is updated.

See the events and actions sections for more details

4.17 Connector Actors

A connector actor is an image actor that connects either two actors together or an actor to an anchor point. Connector actors are useful for creating all sorts of objects such as strings and ropes.

A connector actor is declared using the ActorConnector XOML tag. The ActorConnector example has been provided to show how they work. Lets take a quick look at some example XOML:

```
        <ActorConnector Name="Joiner1" Size="100, 20" Brush="Button1Brush" TargetA="Box1"
TargetB="Box2" />
```

The above XOML creates a connector actor that connects Box1 and Box2 actors together using a visual connector that is 20 units in width and covers 100% of the length of the connector.

In addition to basic actor properties connector actors have a number of new or changed properties, which include:

- Size (length, width) - Size defines the width of the connector as well as the length as a percentage of the distance between the two end points of the connector. For example, if the length is set to 100% then the actor will stretch from the centre of target A to the centre of target B. If the length is less than 100% then the actor will fall short of the centre points of targets A and B;
- TargetA (actor) - Defines the actor to fix the start point of the connector actor
- TargetB (actor) - Defines the actor to fix the end point of the connector actor
- OffsetA (x, y) - An amount to offset the connection point from Target A. If TargetA actor is not specified then this will be classed as a static world position
- OffsetB (x, y) - An amount to offset the connection point from Target B. If TargetB actor is not specified then this will be classed as a static world position

4.18 Particle System Actors

Many games and in fact some apps make use of a special effects. One method of generating special effects is using a particle system. A particle system is a system that generates a group of particles over time using a particle generator. Particle systems are great for producing effects such as fireworks, sparkles, fire and smoke trails etc..

A particle actor is declare using the ActorParticles tag. The ActorParticles example has been provided to show how they work. Lets take a quick look at some example XOML:

```
<ActorParticles Name="SnowyParticles" Image="Particle" Position="0, 0" Scale="1.0"
       Depth="1.0" Layer="1" PosMode="random" AngVelMode="random" VelMode="random"
       AngMode="random" ScaleMode="random" DepthMode="random"  DepthVelMode="random"
       PositionRange="1000, 1000" AngleRange="0, 360" AngVelRange="-25, 25"
       ScaleRange="0.25, 0.5" DepthRange="0.1, 2.0" VelRange="-4, 4, -14, 0"
       ScaleVelRange="0, -0.1" DepthVelRange="-0.03, 0.01">

       <Particle Count="1000" Position="0, 0" VelocityDamping="0.99, 0.99"
              SrcRect="0, 0, 128, 128" ColourVelocity="0, 0, 0, -2" Duration="3" Repeat="-1"
              SpawnDelay="0.01" Gravity="0, 15" />

</ActorParticles>
```

In this example we create a particle actor generator that generates 1000 random spinning / flying particles over time that fall to the ground under the affect of gravity.

In addition to the basic actor properties particle actors have a number of new properties, which include:

General:

- Size (x, y) - Size of the particle actor (used to clip the actor)
- AutoDelete (boolean) – When set to true (default) the actor will automatically delete itself once all particles have finished
- Image (string) – The name of the image that should be used to draw particles
- VelocityAll (x, y (write-only)) – Sets the velocity of all particles to x, y
- ParticleCount (number (read-only)) – Returns the number of particle slots available in the particle system. This value does not represent the actual number of active particles, if new particles were dynamically added. The underlying implementation uses a slot array which grows to accommodate new particles, however to minimise memory allocations it grows at a rate of 4 slots each time a new particle is added when there isn't enough space in the slot array

Regeneration Modes:

- PosMode (mode, write-only) - Determines how the position property of the particle will be regenerated when the particle comes to the end of its life.
- AngMode (mode, write-only) - Determines how the angle property of the particle will be regenerated when the particle comes to the end of its life.
- ScaleMode (mode, write-only) - Determines how the scale property of the particle will be regenerated when the particle comes to the end of its life.
- DepthMode (mode, write-only) - Determines how the depth property of the particle will be regenerated when the particle comes to the end of its life.
- VelMode (mode, write-only) - Determines how the velocity property of the particle will be regenerated when the particle comes to the end of its life.
- AngVelMode (mode, write-only) - Determines how the angular velocity property of the particle will be regenerated when the particle comes to the end of its life.
- ScaleVelMode (mode, write-only) - Determines how the scale velocity property of the particle will be regenerated when the particle comes to the end of its life.
- DepthVelMode (mode, write-only) - Determines how the depth velocity property of the particle will be regenerated when the particle comes to the end of its life.

The mode parameter can be either:

- random - The property will be randomly generated
- normal - The property will not reset back to its original value

Random Generation Ranges:

- PositionRange (x, y, write-only) - Sets the range for which positional coordinates can be randomly generated on the x and y axis. For example, setting a value of 50, 100 will generate random coordinates between -50 and 50 for the x axis as well as -100 to 100 for the y axis
- AngleRange (lower, upper, write-only) - Sets the range for which angle values can be generated from lower degrees to upper degrees. For example setting values of 10, 50 will generate random angles between 10 and 50 degrees.
- ScaleRange (lower, upper, write-only) - Sets the range for which scale values can be generated from lower to upper. For example setting values of 0.1, 2.0 will generate random scaling factors between 0.1 and 2.0
- DepthRange (lower, upper, write-only) - Sets the range for which depth values

can be generated from lower to upper depth. For example setting values of 0.1, 10.0 will generate random depths between 0.1 and 10.0

- VelRange (lower x, upper x, lower y, upper y, write-only) - Sets the range for which velocity values can be generated from lower x to upper x, lower y to upper y. For example setting values of -1, 1, -2, 2 will generate random velocities between -1 and 1 on the x axis and -2 and 2 on the y axis
- AngVelRange - (lower, upper, write-only) - Sets the range for which angular velocity values can be generated from lower degrees/sec to upper degrees/sec. For example setting values of -0.1, 0.1 will generate random angular velocities between -0.1 and 0.1
- ScaleVelRange (lower, upper, write-only) - Sets the range for which scale velocity values can be generated from lower to upper. For example setting values of 0.1, 0.2 will generate random scale velocities between 0.1 and 0.2
- DepthVelRange (lower, upper, write-only) - Sets the range for which depth velocity values can be generated from lower to upper. For example setting values of -0.01, 0.02 will generate random depth velocities between -0.01 and 0.02

Random generators are used when particular modes are set to random. For example, when PosMode=”random” PositionRange will be used to provide the range between which positional coordinates are generated.

Individual particles are added as children of the particle actor. In addition, a number of particles can be automatically generated for you. The inner Particle tag has the following properties:

- Attached (boolean) – When set to true particles will be attached to the emitter and will follow it, when set to false particles will remain where they are created (default is true)
- Count (number) - The number of particles to generate
- Position (x, y) - The position of the particle relative to the generator
- Angle (degrees) - The angle of the particle
- Scale (x, y) - The scale of the particle
- Depth (depth) - The depth of the particle, the greater the value of depth the farther away the actor appears
- Velocity (x, y) - Initial velocity of the particle
- Colour (r, g, b, a) - Initial colour and opacity of the particle
- VelocityDamping (x, y) - The rate at which to slow velocity down or speed it up. Values greater than 1 will increase velocity over time whilst values less than 1.0 will reduce the velocity over time

- AngularVelocity (number) - Initial angular velocity of the particle (degrees / sec)
- AngularVelocityDamping (number) - The rate at which to slow angular velocity down or speed it up. Values greater than 1 will increase angular velocity over time whilst values less than 1.0 will reduce the angular velocity over time
- DepthVelocity (number) - Initial depth velocity of the particle
- DepthVelocityDamping (number) - The rate at which to slow depth velocity down or speed it up. Values greater than 1 will increase depth velocity over time whilst values less than 1.0 will reduce the depth velocity over time
- ScaleVelocity (x, y) - Initial scale velocity of the particle
- ScaleVelocityDamping (x, y) - The rate at which to slow scale velocity down or speed it up. Values greater than 1 will increase scale velocity over time whilst values less than 1.0 will reduce the scale velocity over time
- ColourVelocity (r, g, b, a) - Initial colour velocity of the particle
- ColourVelocityDamping (r, g, b, a) - The rate at which to slow colour velocity down or speed it up. Values greater than 1 will increase colour velocity over time whilst values less than 1.0 will reduce the colour velocity over time
- SrcRect (x, y, width, height) - Sets a part of the image to draw instead of the whole image (used in sprite atlases where multiple images are contained within the same image)
- Repeat (number) - The number of times to repeat a particle (can be thought of its number of lives). Particles that run out of lives will be removed from the generator. Using a value of -1 will regenerate the particle whilst the generator is active
- SpawnDelay (seconds) - Sets an amount of time to wait before spawning this particle. If count is more than 1 then then the spawn delay of each consecutive particle will be compounded. For example if you generate 3 particles with a spawn delay of 1.0 then the first particle will be generated after 1 second, the second particle after 2 seconds and the 3rd particle after seconds.
- Duration (seconds) - This represents the life time of the particle. For example, if you set the duration to 5 seconds, it will be regenerated after 5 seconds (deleted if repeat is 1)
- Gravity (x, y) - The gravity property is used to add a gravitational affect to particles.

A number of other properties can be read / written via property getters and setters:

- OPosition(x, y) – Sets the original respawn position
- OVelocity(x, y) – Sets the original respawn velocity

- OScale (x, y) - Sets the original respawn scale
- OScaleVelocity (x, y) - Sets the original respawn scale velocity
- OAngle (number) - Sets the original respawn angle
- OAngularVelocity (number) - Sets the original respawn angular velocity
- OColour (r, g, b, a) - Sets the original respawn colour
- OColourVelocity (r, g, b, a) - Sets the original respawn colour velocity
- ODepth (number) - Sets the original respawn depth
- ODepthVelocity (number) - Sets the original depth velocity
- LifeTime (number) –The amount of time that the particle has been spawned
- Lives (number) – The number of lives that the particle has left

The Particles script library provides a number of useful functions for interacting with particle actors, including the ability to create and add new particles to a particle actor.

4.19 Actor Animation Properties

Actors can be animated using animations defined in XOML (we will look at animation in the animation chapter later). All writeable properties of an actor are animate-able.

Note that delta animations affect some properties differently and instead of the animation system adding onto their existing value, the delta value is added to the original value that was assigned to the actor. The following properties are affected:

- Position (x, y)
- PositionX (number)
- PositionY (number)
- Angle (number)
- Scale (x, y)
- ScaleX (number)
- ScaleY (number)
- Colour (r, g, b, a)
- Opacity (number)

4.20 Actor Modifiable Properties

Actors can be queried and modified after their creation using actions, commands and scripts. All writeable properties of an actor can be modified, with exception to those properties that can only be assigned during declaration.

4.21 Actor Geometries

Actors can be assigned distinctive geometries that enable them to be represented by more than simple rectangles on screen. Geometries can be simply polygons or collections of triangles / quads. The example below shows an actor that is assigned a polygon shape:

```
<Geometry Name="Geoms1" Vertices="-479,-385,-395,-370,-199,-232,-419,-172,-506,-286"
Type="Poly"/>
<Icon Name="Sprite1" Position="200,0" Background="bg1" Geometry="Geoms1" />
```

The actor now takes on the new polygon shape of the supplied geometry Geoms1

A tool has been supplied that enables you to convert SVG data to geometries

5.0 Image Resources

5.1 Introduction

An image resource is an in-memory version of a graphical file that is loaded and converted to a texture that the device can use to render the image. Actors use images extensively to represent them visually on screen. XOML images support the following image formats:

- PNG
- JPEG
- GIF
- BMP
- TGA

Images can contain opacity maps / transparency. All images also use the colour purple (full red, no green, full blue) as fully transparent which allows images with no opacity map to still use transparency.

To load an image you use the Image XOML tag. Lets take a look at an example of how to load a basic image:

```
<!-- Load an image -->
<Image Name="my_image" Location="some_image.png" />
```

The above XOML loads an image from the apps assets called some_image.png and names it my_image. To use this image you only need to refer to it my name. For example to attach the above image to a brush we would set the Image attribute of the brush like in the following example:

```
<Brush Name="my_brush" Image="my_image" Type="image" />
```

Images can also be located somewhere outside the app such as on a web server, e.g.:

```
<!-- Load an image from a web site -->
<Image Name="my_image" Location="http://www.mydomain.com/assets/some_image.png" />
```

The Image tag has a number of properties:

- Name (string) - Name of this image resource
- Tag (string) - Resource tag (used to group resources together)
- Location (filename) - File name of the image file including extension (can include web addresses)
- Preload (boolean) - If set to true then the image will be loaded immediately. By setting to false the image will be loaded when it is first used by an actor or a brush. This is useful if you want to defer image loading to reduce texture memory use (default is to pre-load)
- Blocking (boolean) - Web based images take time to download from the web so its useful to allow execution of the app to continue whilst it downloads. To prevent the image download from blocking the app set Blocking="false" (default is to not block)
- Condition (variable) - A condition variable that must evaluate to true for this resource to be loaded (this is an optional feature and can be used to conditionally load resources based on certain conditions such as screen size or device type etc..)
- Format (format) - Specifies the texture format that the image should be converted to when loaded. Sometimes it can be helpful to reduce images to lower colour formats to make them render quicker and to reduce how much texture memory space they use. The following texture formats are supported:
    - RGB_565
    - RGBA_4444
    - RGBA_5551
    - RGB_888
    - RGBA_6666
    - RGB_332
    - RGBA_8888
- Filter (boolean) - Image filtering can help to improve the quality of the apps graphics when rotating and scaling images (default is enabled)

Note that Location is a required property unless you are creating an image that can be dynamically changed by a system such as the VideoCam.

Images are added to the global resource system if they are declared outside a scene and will persist for the entire duration of the app. Images declared inside a scene will be added to the scenes resource system and destroyed when the scene is destroyed. When deciding when to load images you need to strike a balance between texture memory use and availability. To calculate how much texture memory an image is using you can use this formula:

memory used = width * height * bpp

bpp is the number of bytes per pixel. You can calculate this from the image format:

- RGB_565 - bpp = 2
- RGBA_4444 - bpp = 2
- RGBA_5551 - bpp = 2
- RGB_888 - bpp = 3
- RGBA_6666 - bpp = 3
- RGB_332 - bpp = 1
- RGBA_8888 - bpp = 4

Your typical 1024x1024 RGB_565 image will use 2MB of texture memory, whereas a 1024x1024 RGBA_8888 image will use 4MB. As you can see higher colour images use much more memory. Mobile devices are memory constricted devices so its worth considering converting loaded images into a lower colour format to save texture memory.

5.2 On-demand Images

Images do not have to be loaded when they are declared in XOML, in fact you can delay the loading of an image resource until it is required for rendering. By specifying Preload=”false” to the image definition you cause the image to be loaded when it is first required by something, for example a brush that is used by an actor. Lets take a look at an example to see how this works:

```
<!-- Load UI image -->
<Image Name="ui_image" Location="ui.png" Preload="false" />

<!-- Create UI brushe -->
<Brush Name="Button1Brush" Image="ui_image" SrcRect="320, 70, 200, 70" />

<!-- Create an icon button -->
<Icon Name="Button" Brush="Button1Brush" />
```

Firstly we create an image called ui_image but we tell the system not to preload the image. We then create a brush called Buton1Brush that utilises the brush, however the image is not loaded at this point because it does not yet require rendering. Lastly we create an Icon actor called button and assign the Butto1Brush to it. When this Icon is rendered it will load the ui_image and render it.

Whilst this method of late loading images can provide very useful when it comes to cutting down on how much texture memory is used it can cause delays in-game whilst the image is loaded. Late loading is best used with small image files.

Another feature of XOML images is the ability to use images that are located on a web server external from the app itself. This is a great technique you can use to reduce the size of you initial app as well as to update content after it has been released to the app stores without having to submit new builds of your app for approval. However, web images can take time to download which can cause short or even long pause sin your game. Its possible to tell the system that you would like to continue your app and download the image asynchronously. This style of resource loading is known as none blocking. Web images are automatically loaded as none blocking, but you can disable this default behaviour by adding Blocking=”true” to the image definition. Its worth noting that actors that are assigned an image that is being downloaded asynchronously will be hidden until the image is available.

5.3 Conditional Image Loading

This topic is a little ahead of itself but has been placed here for future reference. It's often useful to be able to load resources only under certain conditions. For example, you may have a large background image that looks great at iPad resolution but doesn't look too hot on an a small iPhone 3GS. If only there was a way to decide to load a smaller version instead for lower resolution displays.

XOML provides a mechanism for loading resources based on certain conditions called conditional loading. You could for example load an image based on the display size of the device that is running your app. Using a condition variable (explained in the variables chapter) we can tell the system to load specific images based on the devices screen rating or screen dimensions. The ConditionalImages example demonstrates the concept of loading different images based on screen size rating (screen size rating is a single number that rates the size of a screen). Lets take a look at the XOML for this example:

```
    <!-- Create condition variables that check the systems screen size rating  -->
    <Variable Name="LowResCheck" Type="condition" Value="system:3 LTE 2" />
    <Variable Name="MedResCheck" Type="condition" Value="system:3 GT 2 AND system:3 LTE
3" />
    <Variable Name="HighResCheck" Type="condition" Value="system:3 GT 3" />

    <!-- Load image based on screen size rating  -->
    <Image Name="background" Location="low.png" Condition="LowResCheck" />
    <Image Name="background" Location="medium.png" Condition="MedResCheck" />
    <Image Name="background" Location="high.png" Condition="HighResCheck" />

    <!-- Create bacjkground brush -->
    <Brush Name="background" Image="background" Type="image" SrcRect="0, 0, 512, 512" />

    <!-- Create scene with narrow extents -->
    <Scene Name="Scene1" Current="true">

        <!-- Create a label object -->
        <Icon Background="background" Size="-100, -100" />

    </Scene>
```

You do not need to worry about the finer details at this point as variables and conditional variables will be covered in the variables section.

In the above example we create 3 conditional variables that query the system variable array. The 4th (index 3 as indices start from 0 and not from 1) system variable contains a screen rating, which tells us using a single number the approximate size rating of the screen. Each of the conditional variables check for low, medium and high resolution ratings.

Next we declare 3 images each with their own condition to check. If the LowResCheck condition passes (we are running on a low resolution screen) then low.png will be loaded as the background. If the MedResCheck condition passes (we are running on a medium resolution display) then the medium;png image will be oaded as the background and so on.

The background brush and the icon declared inside the scene are oblivious as to which image is loaded as they are only interested in an image that is named “background”.

6.0 Brushes

6.1 Introduction

A brush is a resource that describes to an actor how and what it should render. You may be asking why when we already have images that we can just render. There are a number of reasons for using brushes including:

- Brushes come in different types. XOML supports different brush types including miage and 9-patch image brushes. Solid and gradient brushes will be added in the future
- Brushes are re-usable - A brush carries not only an image but other information such as the area of the image that it should render. For 9-patch images we also include scale area information. Because these properties are defined in the brush they do not need to be set-up on a per actor basis, instead they assigned in the brush only. If you need to change the area of the image that is rendered in the future then you only need change the brush
- Image Atlases - Its more efficient to combine many smaller images into one large image to a) load the images faster and to speed up the rendering of an actors that use those images. An image that is made up from many smaller images is known as an image atlas (also called sprite atlas or sprite sheet)

The following types of brushes are currently supported:

- Image - An image brush is a brush that contains an image and a SrcRect. A SrcRect is a rectangular area within the image that represents a sub image. This type of brush is useful for rendering images that are part of an image atlas. The image brush type also supports setting of UV coordinates directly.
- 9patch image - A 9patch image is an image that allows a certain portion of it to be rendered unscaled whilst the rest is scaled. This type of image is useful for displaying user interface components such as buttons where scaling of the border can look ugly and pixelated on higher resolution screens. The name 9patch comes from the fact that to render a 9patch image the image has to be rendered in 9 parts or patches.

Now we know a little about brushes lets take a look at brush properties:

- Name (string) - Name of this brush resource
- Tag (string) - Resource tag (used to group resources together)
- Type (type) - type of brush (image or 9patch)
- Condition (variable) - A condition variable that must evaluate to true for this resource to be loaded (this is an optional feature and can be used to conditionally load resources based on certain conditions such as screen size or device type etc..)
- Image (image) - Sets the image that is associated with the brush
- SrcRect (x, y, width, height) - A rectangular area that represents a sub image of a larger image. If not supplied then the whole image will be taken as the SrcRect
- UV (list of vec2) – Optional set of actual UV coordinates that are used instead of a SrcRect.
- ScaleArea (x, y, width, height) - In simple terms the ScaleArea represents a rectangular area of the sub image that can be scaled without worrying about pixelation. The remainder outside of the scaled region will not be scaled. This property is only required for 9patch brushes

Notes:

- Type and Image are required properties

6.2 Image Brushes

An image brush is a brush that represents a rectangular area of a larger image, also known as a sub-image. In app development it is customary to combine many smaller images into one large image. This format of image representation is called image atlases (also known as sprite atlases / sprite sheets). Using image atlases can greatly speed up rendering of objects, especially when combined with a scene batch rendering. One large image also generally loads faster than many smaller images.

To determine which portion of an image atlas to draw for a particulary object we use the SrcRect property of the brush. This property defines which rectangular potion of the image to use. Lets take a look at an example:

Lets say we have an image that is 1024x1024 pixels in size. We arrange a group of sub images each of size 128x128 pixels on that 1024x1024 image. This allows us to store 64 sub images on one large image. To access one of those images we set the ScrRect of the brush to the area where our sub image is situated. Lets look at smoe example XOML:

```
<Brush Name="AlienBrush" Image="ImageAtlas" SrcRect="0, 0, 128, 128" Type="image" />
```

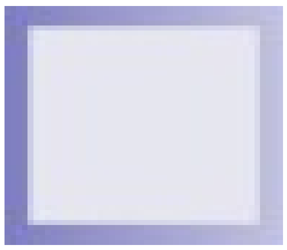
In this example we create a sub image located at the top left hand corner of the image that is 128x128 pixels in size.

The AppEasy Asset conversion tool has been provided that can create both images and brushes from Texture Packer data.

6.3 9-Patch Brushes

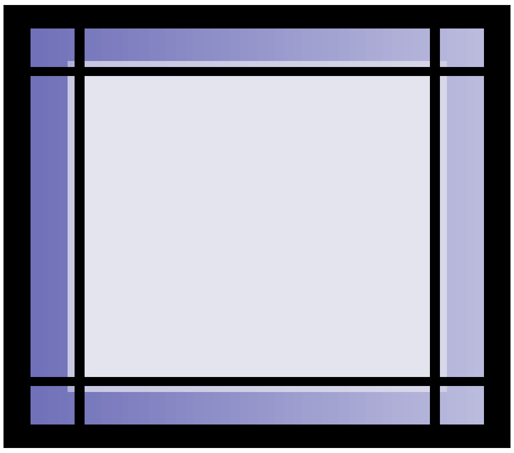
A 9patch image is an image that allows a specific portion of it to be rendered unscaled whilst the rest is scaled. This type of image is useful for displaying user interface components such as buttons where scaling of the border can look ugly and pixelated on higher resolution screens. The name 9patch comes from the fact that to render a 9patch image the image has to be rendered in 9 parts or patches. Its better to show how this works graphically.

Lets take a look at a check box buttno graphic that we would usually use to show a check box to the user:



This button graphic has a nice fancy border that we would like to display without it being scaled regardless of the size of the check box. The internal area of the button can however be freely scaled as scaling it wont really affect its quality on screen.

The 9patch image for the above image will be split into the following 9 areas depending upon what area value we specify:



A few points to note:

- Only the central light blue area will be scaled on the x and y axis.
- The 4 small corners will not be scaled at all
- The two horizontal areas will be scaled on the x axis if the button is wider or narrower than the graphics width
- The two vertical areas will be scaled on the y axis if the button is taller or shorter than the graphics height

The ScaleArea property of the brush is used to specify the left column, top row, right column and bottom row.

7.0 Fonts

7.1 Introduction

The XOML font system is based on true type fonts. XOML does not come with any fonts built in so you need to supply your own TTF font files for the fonts that you would like to use. Note that many fonts are copyrighted so you should not include them unless you have the permission to do so. There are also many free fonts out there which you can distribute with your app with no restrictions.

Lets take a look at a typical font declaration in XOML:

```
<!-- Create UI font -->
<Font Name="serif" Location="Serif.ttf" PointSize="8" Preload="true" />
```

This creates a font called serif from the true type font file Serif.ttf with a point size of 8. We can now use this font to render text as the following example shows:

```
<Label Font="serif" Text="Hello World" />
```

A true type font file contains information relating to how text should be rendered as well as which characters can be rendered. To create a font in XOML we use the Font tag which has the following properties:

- Name (string) - Name of this font resource
- Tag (string) - Resource tag (used to group resources together)
- Location (filename) - File name of the image file including extension (can include web addresses)
- Preload (boolean) - If set to true then the font will be loaded immediately. By setting to false the font will be loaded when it is first used by an actor. This is useful if you want to defer font loading to reduce texture memory use (default is to pre-load)
- Blocking (boolean) - Web based fonts take time to download from the web so its useful to allow execution of the app to continue whilst it downloads. To

prevent the font download from blocking the app set Blocking="false" (default is to not block)
- Condition (variable) - A condition variable that must evaluate to true for this resource to be loaded (this is an optional feature and can be used to conditionally load resources based on certain conditions such as screen size or device type etc..)
- PointSize (number) - Sets the size of the font. If AutoPointSize is set then this number becomes an offset from the point size calculated by the system.
- AutoPointSize (number) - If set then the system will automatically calculate the default point size based on the supplied number of lines that you would like to fit onto the display. For example AutoPointSize=”40” will calculate the size of the font to approximately fit 40 lines of text on the display.

Notes:
- Location and PointSize are required properties

Fonts are added to the global resource system if they are declared outside a scene and will persist for the entire duration of the app. Fonts declared inside a scene will be added to the scenes resource system and destroyed when the scene is destroyed. When deciding when to load fonts you need to strike a balance between texture memory use and availability.

True type fonts are converted to bitmap format and cached in a texture during rendering to speed up rendering. Some larger fonts / point sizes require a larger texture to cache the glyphs. If you notice that characters are missing when you render text then this means you need to increase the size of the texture that is being used to cache the glyphs. To do this you need to increase the size of “Cache Texture Max Size” in the AppEasy project manager.

7.2 Font Re-use

Usually when rendering text we would like to display it in different sizes. Loading the same font multiple times to generate different point sizes is slow and uses a lot of memory. To alleviate this problem XOML will automatically re-use fonts that have already been loaded. Lets look at an example:

```
<Font Name="serif10" Location="Serif.ttf" PointSize="10" Preload="true" />
<Font Name="serif16" Location="Serif.ttf" PointSize="16" Preload="true" />
```

In the above example we create two versions of the same font, one with a point size of 10 and another with a point size of 16. however XOML spots that both fonts share the same font file so only loads Serif.ttf the once.

8.0 Sound Effects & Music

8.1 Introduction

Ever tried watching a horror movie without the sound switched on? It just doesn't have the same effect. Games and some apps for that matter would be pretty boring without sound effects. Audio provides important feedback to the user and often can make or break a game.

XOML supports the loading and play back of sound effects using the Sound tag. The SoundEffects example shows how to load and play back a sound effect in response to the user tapping a button. Lets take a look at the XOML for this example to see how it works:

```
    <!-- Load sound effect -->
    <Sound Name="explosion" Location="explosion.wav" />

    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" >

        <!-- Create a label button that when tapped plays a sound effect -->
        <Label Font="serif" Background="Button1Brush" BackgroundColour="80, 80, 255, 255"
Size="100, 100" Text="Tap Me!" OnTapped="Explosion">
            <Actions Name="Explosion">
                <Action Method="PlaySound" Param1="explosion" />
            </Actions>
        </Label>

    </Scene>
```

Firstly we load the explosion.wav sound effect and create a sound from it called explosion. Next we create a label within a scene that handles the OnTapped event. When the user taps the button the Explosion actions list is called. The Explosions actions list contains an action called PlaySound which plays the specified sound, in this case our explosion sound.

XOML sounds support the following properties:

- Name (string) - Name of this sound resource
- Tag (string) - Resource tag (used to group resources together)
- Location (filename) - File name of the sound file including extension (can include web addresses)
- Preload (boolean) - If set to true then the sound will be loaded immediately. By setting to false the sound will be loaded when it is first used. This is useful if you want to defer sound loading to reduce memory use (default is to pre-load)
- Blocking (boolean) - Web based sounds take time to download from the web so its useful to allow execution of the app to continue whilst they download. To prevent the sound download from blocking the app set Blocking="false" (default is to not block)
- Condition (variable) - A condition variable that must evaluate to true for this resource to be loaded (this is an optional feature and can be used to conditionally load resources based on certain conditions such as screen size or device type etc..)

Location is a required property

Up to 16 mono sound effects can be played back simultaneously. WAV sound effect files can be stored in the following formats:

- PCM (8 and 16 bit) mono
- ADPCM (IMA only) mono

8.2 Music

Many games usually have some kind of music playing in the background to help set the mood for the player. Music files are usually much larger than the simple sound effects that we spoke about in the previous section and are usually played via streaming (this means played from wherever they are stored, instead of fully loading and playing them). XOML uses the devices in-built media player to play back music files so any other audio that is playing via the devices media player will be stopped. XOML does not have a tag for declaring music files instead they are played directly using their file name from an action, command or script.

The Music example has been provided to show you how to play and stop background music. Lets take a look at the XOML code for this example to see how it works:

```
    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" >

        <!-- Create a play button  -->
        <Label Position="-100, 0" Font="serif" Background="Button1Brush" 
BackgroundColour="80, 255, 80, 255" Size="100, 100" Text="Play" OnTapped="Play">
            <Actions Name="Play">
                <Action Method="PlayMusic" Param1="music.mp3" />
            </Actions>
        </Label>

        <!-- Create a stop button  -->
        <Label Position="100, 0" Font="serif" Background="Button1Brush" 
BackgroundColour="255, 80, 80, 255" Size="100, 100" Text="Stop" OnTapped="Stop">
            <Actions Name="Stop">
                <Action Method="StopMusic" />
            </Actions>
        </Label>

    </Scene>
```

In this example we create two label buttons, the first plays a music file and the second stops the music from playing. The first label calls the Play actions list when the user taps the button which in turns calls an action called PlayMusic which plays the music file music.mp3. The second label calls the Stop actions list when the user taps the Stop button which in turn calls the StopMusic action which stops the currently playing music

Music files should be in the MP3 format. Only locally stored music can be played at this time, support for web based music files will be added in the future.

9.0 Video

9.1 Introduction

Video can be incredibly useful both for apps and games. For example, you may have a game with high quality cut scenes that are used to introduce parts of the story. Another example could be an app that shows training videos to the user to help them learn more about the app. Video is played back using the devices media so any other music or video that is currently playing will be stopped when the video is played, this includes background music.

XOML supports the loading of video via the Video tag and the playback of video via the VideoOverlay. Note that video will be displayed above all other elements on screen. The Video example has been provided to show how to use both Video and VideoOverlay tags. Lets take a look at the XOML from this example:

```
    <!-- Create a video resource -->
    <Video Name="Video1" Location="video1.mp4" Codec="MPEG4" />

    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" >

        <!-- Create a video verlay to show the video -->
        <VideoOverlay Name="Vid1" Video="Video1" Size="-20, 100" AutoPlay="true"
Volume="0.3" Repeat="1" AspectLock="x" />

    </Scene>
```

In this example we firstly create a video resource called Video1 from the video1.mp4 video file and specify the code that should be used to play it back, in this case the file is MPEG4. Next we create a scene containing a VideOverlay called Vid1 which plays back the Video1 video resource.

The Video tag has the following properties:

- Name (string) - Name of this video resource

- Tag (string) - Resource tag (used to group resources together)
- Location (filename) - File name of the image file including extension (can include web addresses)
- Codec (codec) - Specifies the codec to use to play back the video file. The following codecs are supported (check for specific platform support):
    - MPEG4
    - 3GPP
    - 3GPP_VIDEO_H263
    - 3GPP_VIDEO_H264
    - 3GPP_AUDIO_AMR
    - 3GPP_AUDIO_AAC
    - MPEG4_VIDEO_MPEG4
    - MPEG4_VIDEO_H264
    - MPEG4_AUDIO_AAC
    - SWF
- Repeat (number) - The number of times to repeat the video, 0 forever
- Preload (boolean) - If set to true then the video will be loaded immediately. By setting to false the video will be loaded when it is first used. This is useful if you want to defer video loading to reduce texture memory use (default is to pre-load)
- Blocking (boolean) - Web based video can take time to download from the web so its useful to allow execution of the app to continue whilst it downloads. To prevent the video download from blocking the app set Blocking="false" (default is to not block)
- Condition (variable) - A condition variable that must evaluate to true for this resource to be loaded (this is an optional feature and can be used to conditionally load resources based on certain conditions such as screen size or device type etc..)
- OnStopped (actions list) – Defines an actions list that will be called when the video stops playing

Location and codec are required properties

Note that video can be played back for local video files that are part of your assets or from a web based video file.

Because different devices support different video formats you may find that you need to include a number of different versions of a video with your app to ensure that the video is played for the user. To facilitate this when you declare a Video resource, if the codec is not supported then the video is not created and loaded. This mechanism allows you to declare multiple videos with the same name, but only the first successfully supported video will be loaded, the rest will be ignored.

9.2 Video Camera

XOML provides a resource type that allows you to stream the live video camera of the device to a XOML image resource. This image can be used just ilke any other image and used to draw actors. The VideoCam XOML resource tag is used to instantiate a video camera resource. Note that an app can only have a single instance of VideoCam resource available in the app at once. The VideoCam XOML tag has the following properties:

- Name (string) – Name of the resource
- Tag (string) – Resource tag
- Target (string) – Target image where camera data will be written
- Quality (string) – The quality of the camera image that should be captured (low, medium or high). Note that higher quality video will use more processor resources
- Resolution (string) – The resolution of the camera image that should be captured (low, medium or high). Note that higher resolution video will use more processor resources
- Direction (string) – This determines which camera to use (front, rear, default is rear)
- Start (boolean) – Starts the camera immediately (default is false)
- OnError (event) – Defines an event that is called if the camera is unavailable or does not start

You can later use the ChangeVideoCam action or the media.changeVideoCam() script function to start / stop the video cam.

Lets take a look at a short example of how to map the video camera to an image:

```
<Image Name="Image1" />
<Brush Name="Brush1" Image="Image1" Type="image" />
<VideoCam Name="Cam1" Target="Image1" Quality="low" Resolution="low" Start="true" />
```

In the above example we create a blank image and a brush from that image. Lastly we create a VideoCam resource that targets the image. The video camera will now be started and data from it will be streamed to the Image1 image.

The video camera also supports snapshots. A snapshot is an image that is captured from the live camera stream which is then saved locally. You can capture snapshots from lua using media.snapshotVideoCam(). See Video Cam Snapshots example for reference.

10.0 Animation

10.1 Introduction

Animation is a big part of XOML as it adds so much rich and interesting interaction to otherwise mundane game objects and user interfaces. The animation approach used my XOML is called frame based which uses a series of key-frames to build complex animations. A key frame is the state of the property of an object at a specific time, for example the position of an object at 0 seconds. Lets look at an example of an object that wants to move around the screen using key frames. It needs to move between 4 different positions over the space of 5 seconds. Lets take a look at what the key frames will look like:

- Time 0 - Position is 0, 0 (object starts here)
- Time 1 - Position is 100, 0 (object moves to the right 100 units)
- Time 2 - Position is 100, 100 (object now moves down 100 units)
- Time 3 - Position is 0, 100 (object moves back to the left)
- Time 4 - Position is 0, 0 (object moves back up 100 units to where it originally started from)

If we just set these positions at time 0, 1, 2, 3, 4 our object would firstly appear at 0, 0 at Time 0, then suddenly appear at 100, 0 after 1 second, it would then suddenly appear at 100, 100 after 2 seconds and so on. The problem for us is that we would like the object to move slowly from Position 0, 0 to Position 100, 0 over the period of 1 second. To accomplish this we use a method called “interpolation” or “tweening” which smoothly calculates the frames in-between the key frames automatically for us.

Actors and scenes use animation extensively to animate their properties thus many types of different animation key frames are available. The following types of animation frames are available:

- Boolean - Data that can be either true or false
- Float - Data that uses decimal numbers
- Vec2 - Data that has two components such as position
- Vec3 - Data that has three components such as 3D vectors
- Vec4 - Data that has four components such as colour
- Rect - Data that has four components that form a rectangle such as image SrcRect's
- String - Data that uses string data such as camera names or text labels

In XOML animation is done in two parts:

- Animation definition - Here we define an animation along with all of its key frames
- Timeline definition - Here we create a timeline that contains references to each of the animations alnog with which properties of the object these4 animations target

When we attach an animation to either an actor or a scene we do not attach the actual animation but instead we attach the animation timeline.

The XOML animation system also supports easing between frames. Easing allows you to add acceleration / deceleration when moving from one key frame to the next. This allows us to add cool and interesting affect to our animations.

10.2 Creating an Animation

In XOML we create animations using the Animation tag which contains a group of key frames that are declared using the Frame tag. Lets take a quick look at an example of creating the animation for our key frame example shown earlier:

```
<!-- Create vec2 animation (we will use this to animate position of the label) -->
<Animation Name="PosAnim" Duration="4" Type="vec2">
    <Frame Time="0" Value="0, 0" />
    <Frame Time="1" Value="100, 0" />
    <Frame Time="2" Value="100, 100" />
    <Frame Time="3" Value="0, 100" />
    <Frame Time="4" Value="0, 0" />
</Animation>
```

In this example we create an animation called PosAnim that lasts for 4 seconds and animates vec2 (2 components x and y) data. We declare 5 animation key frames for this animation using the Frame tag. Each frame defines the value of the position at various points in time from 0 to 4 seconds.

Before we continue lets take a look at the properties of the Animation and Frame tags.

Animation properties:

- Name (string) - Name of this animation resource
- Tag (string) - Resource tag (used to group resources together)
- Type (type) - The type of animation frame data, can be one of the following:
    - bool
    - float
    - vec2
    - vec3
    - vec4
    - rect
    - string
- Duration (seconds) - The duration of the animation
- AutoDelete (boolean) – If set to true then this animation will be deleted when the timeline that is playing it finishes playing. Be careful not to delete animations that are shared across multiple timelines

Type and Duration are required properties

Animation Frame properties:

- Time (seconds) - The time at which this frame is available
- Value (any supported type) - The value of the frame at the above time. The value here depends on the type of animation that was defined. For example, if you declared the animation as a string type then this value should contain a string. If you declared the animation as a vec2 then the value shoudl contain pairs of values
- Easing (easing) - Type of easing to use, supported easing types include:
    - linear
    - quadin
    - quadout
    - cubicin
    - cubicout
    - quarticin
    - quarticout

Time and Value are required properties

Animations can also contain an additional inner tag called Atlas. This is a special tag that is used to auto generate sprite animation frames. We will cover this in a later section.

10.3 Creating a Timeline

Animations that are declared using the Animation tag can be thought of as simply data that does not do anything particular until it is applied to something else. XOML separates the definition of animation data and the application of animation data to enable re-use of animation data across different objects. For example, we could have an animation that targets the property of one object but also targets a different property of another object. XOML accomplishes this separation using Timelines.

A Timeline is a collection of animations that target specific properties of a scene or object (to see which properties of scenes and actors can be animated, see the scenes, actors and user interface sections). For example a timeless can contain multiple animations, one that targets an actors position and another that targets an actors angle. Now would be a good time to
introduce the BasicAnimation example. Lets take a quick look at the XOML for this example:

```
<!-- Create vec2 animation (we will use this to animate position of the label) -->
<Animation Name="PosAnim" Duration="4" Type="vec2">
    <Frame Time="0" Value="0, 0" />
    <Frame Time="1" Value="100, 0" />
    <Frame Time="2" Value="100, 100" />
    <Frame Time="3" Value="0, 100" />
    <Frame Time="4" Value="0, 0" />
</Animation>

<!-- Create rotation animation (we will use this to animate angle of the label) -->
<Animation Name="AngleAnim" Duration="4" Type="float">
    <Frame Time="0" Value="0" />
    <Frame Time="4" Value="360" />
</Animation>

<!-- Create a timline that can be used to animate n actor -->
<Timeline Name="Anim1" AutoPlay="true">
    <Animation Anim="PosAnim" Target="Position" />
    <Animation Anim="AngleAnim" Target="Angle" />
</Timeline>
```

```
    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" >

        <!-- Create a label with an animation -->
        <Label Font="serif" Background="Button1Brush" BackgroundColour="80, 80, 255, 255"
Size="100, 100" Text="Animation" Timeline="Anim1"  />

    </Scene>
```

Firstly we create two animations the first PosAnim as explained in the previous section will animation our label over 4 different positions over 4 seconds. The second of the animation creates an angle animation called AngleAnim which animations the angle from 0 to 360 degrees (one full revolution) over 4 seconds.

Next we create a single timeline called Anim1 which contains the PosAnuim and AngleAnim animations. Note that each of these animations declared the type of property that the animation will target. In this case we target the Position property of the actor with PosAnim and the Angle property of the actor with the AngleAnim.

Lastly we create a label inside a scene and attach the Anim1 animation timeline to the label.

Now we have a good idea how to use timelines lets take a look at their properties:

Timeline properties:

- Name (string) - Name of this animation timeline resource
- Tag (string) - Resource tag (used to group resources together)
- AutoPlay (boolean) - When set to true the timeline will begin animating as soon as it is attached to an actor or scene
- TimeScale (number) - This property can be used to change the speed at which the timelines animations are played back. For example, if we set this value to 2.0 then the animations in the previous example would play back twice as quickly. This feature allows us to speed up and slow down animations without having to create new instances of them
- Local (boolean) - By default timelines declared inside an actor will be local to the actor and not the scene. By setting Local=”false”the timeline will be placed into the containing scene instead. This is useful if you want to show a group of timeline across different actors but do not want to move the timeline definition

into the scene or global resource manager
- AutoDelete (boolean) – If set to true then this timeline will be deleted when it finishes playing. Be careful not to delete timelines that are shared across multiple objects

Timelines also contain child Animation tags which define which animations will appear in the timeline. Lets take a look at the properties of this inner tag:

- Anim (animation) - Names an animation to be included into the timeline
- Target - Sets the objects target property that should be updated by the animation. For example, Position would target an actor scene position updating its position
- Repeat (number) - The number of times that the animation should repeat before stopping (0 represents play forever, which is the default)
- StartAtTime (seconds) - Setting this parameter will delay the start of the animation
- Delta (boolean) - When set to true, instead of directly setting the target objects property it adds to it instead (called delta animations)
- Interpolate (boolean) - When set true (which id default) animation frames will be smoothly interpolated from one key frame to the next. When set to false animation frames will suddenly switch when their time marker is reached. For most animation you will want to use interpolation, however some kinds of animations are not suited to interpolation, for example, image sprite animations (we will cover this later when we talk about sprite atlases)
- OnStart (actions list) - Defines an actions list that will be called when this animation starts playing (only called for animations with a StartAtTime value that is not 0
- OnEnd (actions list) - Defines an actions list that will be called when this animation ends
- OnRepeat (actions list) - Defines an actions list that will be called when this animation repeats

Anim and Target are required properties

Note that a time line can be assigned to as many objects you like but the time line will be played back the same on all objects. This is very useful if you have a group of actors that need to run the same animations synchronously.

10.4 Bitmap Animations

Whilst animating objects using position, scale, colour etc.. is great and can produce some excellent animating objects, it sometimes just not enough to get the effect you want. For example, its very difficult to create soft body character animations without considerable work. Bitmap animations solve this problem by allowing you to create an animating object by simply displaying a group of images one after the other. These images are usually stacked into an image atlas and a none interpolating animation is created that displays the individual frames over time. The BitmapAnimation example has been provided to show how to do this. Lets take a look at the XOML involved in this example:

```
<!-- Load the frace animation image atlas -->
 <Image Name="FaceAnim" Location="faceanim.png" />

 <!-- Create a brush for the face anim image -->
 <Brush Name="FaceAnim" Image="FaceAnim" SrcRect="0, 0, 37, 40" Type="image" />

 <!-- Create rect animation for the bitmap animation -->
 <Animation Name="FaceAnim" Duration="1.6" Type="rect">
     <Frame Time="0"     Value="0, 0, 37, 40" />
     <Frame Time="0.2"   Value="0, 40, 37, 40" />
     <Frame Time="0.4"   Value="0, 80, 37, 40" />
     <Frame Time="0.6"   Value="0, 120, 37, 40" />
     <Frame Time="0.8"   Value="0, 160, 37, 40" />
     <Frame Time="1.0"   Value="0, 200, 37, 40" />
     <Frame Time="1.2"   Value="0, 240, 37, 40" />
     <Frame Time="1.4"   Value="0, 280, 37, 40" />
 </Animation>

 <!-- Create a timeline that animated the actors SrcRect -->
 <Timeline Name="FaceAnim" AutoPlay="true">
     <Animation Anim="FaceAnim" Target="SrcRect" />
 </Timeline>

 <!-- Create a scene -->
```

```
<Scene Name="Scene1" Current="true" >

    <!-- Create n Icon with a bitmap animation -->
    <Icon Background="FaceAnim" Size="148, 160" Timeline="FaceAnim" Draggable="true" />

</Scene>
```

Firstly we load the faceanim.png image then create an image brush from it. Next we create a rect animation (rect animations automatically do not interpolate their frames). We then create a timeline that targets the SrcRect property and finally create an Icon actor that is assigned the FaceAnim timeline.

We can simplify this XOML somewhat by using a speical inner tag of Animation called Atlas. This tag enables us to automatically generate SrcRect frames from an image atlas where all of the sub images are arranged at equal distances apart. Lets take a look at a quick example:

```
<!-- Create rect animation for the bitmap animation -->
<Animation Name="FaceAnim" Duration="1.6" Type="rect">
    <Atlas Count="8" Duration="0.2" Pitch="37, 40" Size="37, 40" Width="37" />
</Animation>
```

This XOML has cut down the total number of lines of XOML down to just one and we no longer have to specify each single frame.

The properties for the Atlas tag are as follows:

- Count (number) - Number of frames to generate
- Duration (seconds) - The amount of time to display each frame
- Pitch (x, y) - The amount to step across and and right to get to the next frame in pixels
- Size (x, y) - The width and height of each animation frame in pixels
- Width (number) - The width of the image atlas in pixels
- Position (x, y) - The position on the atlas where frames should start being taken from

11.0 Styles & Templates

11.1 Introduction

As you become accustomed to XOML and write more and more XOML you will begin to try and find ways to do more with less XOML. Styles and Templates were added to XOML for that very purpose. Styles offer a way to set groups of common property values by declaring them in a Style then assigning them to a scene or actor using the Style property. Templates offer a way of creating a complete piece of XOML that uses parameters then later instantiate them somewhere else using custom parameters. Both of these techniques can be used to vastly reduce the amount of XOML you create as well as create common shared styles that can easily be changed to completely restyle your app.

11.2 Styling

The Style tag enables you to create a collection of property setters that when attached to a scene or actor object sets the properties of the object for you. The Styles example has been provided to show styles in action. Lets take a look at the XOML from this example to see what it does:

```
<!-- Create style for our label to save lots of typing -->
<Style Name="LabelStyle1">
    <Set Property="Font" Value="serif" />
    <Set Property="Background" Value="Button1Brush" />
    <Set Property="BackgroundColour" Value="80, 80, 255, 255" />
    <Set Property="Size" Value="200, 200" />
</Style>
```

```
    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" >

        <!-- Create a bunch of styled labels -->
        <Label Style="LabelStyle1" Position="-100, -100" Text="Actor1" />
        <Label Style="LabelStyle1" Position="100, -100" Text="Actor2" />
        <Label Style="LabelStyle1" Position="100, 100" Text="Actor3" />
        <Label Style="LabelStyle1" Position="-100, 100" Text="Actor4" BackgroundColour="80,
255, 80, 255" />

    </Scene>
```

To begin with we create a style called LabelStyle1. We use a group of settisg to set the default value for the Font, Background, BackgroundColour and Size. We then create 4 labels inside a scene that each take their parameters from the style we created. Note how Actor4 has BackgroundColour defined even though it is defined ni the template. Using Styles is great but sometimes you need to modify the style slightly without having to create a completely new style. XOML accomplishes this using style property overrides. This is just a fancy phrase that says any properties supplied to the label will write over the parameters provided by the style. In this case we supplied a green background colour to Actor4 which wrote over the blue colour that was supplied by the style.

Now we've seen an example of Styles in use lets take a look at the Style tags properties:

- Name (string) - Name of this style resource
- Tag (string) - Resource tag (used to group resources together)

Styles contain inner tags called Set, these provide the linking between properties and values. The properties of a Set tag are:

- Property (property name) - The name of the property to target
- Value (string) - The value to set the property to

Property and Value are required properties

11.3 Templates

If Styles are cool then Templates could be thought of as awesome. A template allows you to create a huge piece of XOML then create copies of it elsewhere, modifying the XOML with parameters. Templates offer the ultimate way to re-use small and large sections of XOML alike. You can even instantiate a template during run time from an action, command or script. For example, you may have created a template for complex piece of XOML that represents an in game baddy. You could instantiate these baddies into the world based on certain events or happenings within your game. Instead of having to create the baddy from scratch, including all of its moving parts and animations you simply instantiate the baddies template passing in any parameters that you need to change such as spawn position, size, health etc..

The Templates example has been provided to show you how to use templates effectively. Lets take a quick look at the example XOML to see how it works:

```
<!-- Create a template for our car -->
<Template Name="CarTemplate">
    <Icon Name="$name$" Position="$pos$" Velocity="$speed$, 0" WrapPosition="true"
         BackgroundColour="$colour$" Background="Button1Brush" Size="200, 50">

        <Icon Background="Button1Brush" Size="50, 50" Position="-50, 25"
                AngularVelocity="$speed$" />
        <Icon Background="Button1Brush" Size="50, 50" Position="50, 25"
                AngularVelocity="$speed$" />
    </Icon>
</Template>

<!-- Create a scene -->
<Scene Name="Scene1" Current="true" Extents="-500, -500, 1000, 1000" >

    <!-- Create some cars -->
    <FromTemplate Template="CarTemplate" name="car1" pos="-100, -150" speed="10"
         colour="80, 80, 255, 255" />
    <FromTemplate Template="CarTemplate" name="car2" pos="-100, 0" speed="20"
         colour="80, 255, 80, 255" />
    <FromTemplate Template="CarTemplate" name="car2" pos="-100, 150" speed="5"
```

```
            colour="255, 255, 80, 255" />

    </Scene>
```

The above example creates 3 different coloured cars complete with wheels that race across the screen at different speeds.

We begin by creating a template called CarTemplate that contains the blueprint for our car. The blueprint consists of an Icon actor used as the car body with some odd looking property values as well as two additional Icon child actors that represent the cars wheels The add looking values include $name$, pos, $speed$ and $colour$. For example, the Name is set to $name$ instead of a proper name. Values that are enclosed inside double dollars ($$) are known as template parameters and must be in lower case. When you later instantiate the template you pass the named parameter to the template and the template values are replaced with whatever value you pass. For example, later on we instantiate the CarTemplate using the FromTemplate XOML tag, but we pass a bunch of parameters which include name, pos, speed and colour. These parameters allow us to change the XOML that is generated when we instantiate it. In this example we use these parameters to vary the start position of the cars, the body colour the speed at which the car travels and the name.

The Template tag has the following properties:

- Name (string) - Name of this template resource
- Tag (string) - Resource tag (used to group resources together)

The FromTemplate tag has a single default property called Template which specifies the Template to instantiate. Any additional properties that are passed to FromTemplate will be classed as template parameters.

11.4 Instantiating using Actions and Command

Templates can also be instantiated via a program command or an action. The Templates2 example that has been provided shows an example of both of these methods of instantiating a template. Lets take a look at the XOML for this example to see how it works:

```
<!-- Create a template for our car -->
<Template Name="CarTemplate">
    <Icon Name="$name$" Position="$pos$" Velocity="$speed$, 0"
         BackgroundColour="$colour$" Background="Button1Brush" Size="200, 50"
         WrapPosition="true">
        <Icon Background="Button1Brush" Size="50, 50" Position="-50, 25"
         AngularVelocity="$speed$" />
        <Icon Background="Button1Brush" Size="50, 50" Position="50, 25"
         AngularVelocity="$speed$" />
    </Icon>
</Template>

<!-- Create a scene -->
<Scene Name="Scene1" Current="true" Extents="-500, -500, 1000, 1000" >

    <!-- Create a program that instantiates a car from a template -->
    <Program Name="CreateCars" AutoRun="true">
        <Command Method="from_template" Param1="CarTemplate"
         Param2="name=Car1:pos=0,0:speed=1:colour=80,255,255,255" />
    </Program>

    <!-- Create a label button that when tapped will instantiate new cars -->
    <Label Font="serif" Text="New Car" Background="Button1Brush" Position="0, 200"
         OnTapped="CreateCar">
        <Actions Name="CreateCar">
            <Action Method="FromTemplate" Param1="CarTemplate"
                Param2="name=Car1:pos=0,-100:speed=3:colour=255,0,255,255" />
        </Actions>
    </Label>

</Scene>
```

Firstly we create a template for a car object. Next we define a single command program that calls the command from_template passing the CarTemplate as its first parameter and the parameters that should be passed to the template as Param2. Finally we add the CreateCar action to the label which gets called whenever the user taps the New Car button. This action calls the FromTemplate method passing the CarTmplate and the template parameters. Lets take a close look at the template parameters that are being passed in both cases (they are the same):

```
name=Car1:pos=0,-100:speed=3:colour=255,0,255,255
```

As you can see the parameters are paired up and separated by a colon character (:)

12.0 Events and actions

12.1 Introduction

A game or app would be pretty useless if it did nothing other than just sit and look pretty. Users generally expect lots of interaction with their apps. XOML uses the events and actions system to provide such interaction.

From a XOML point of view an event is something that has happened in the app that some scene or actor needs to know about. For example, when the user taps a button to see help he raises the tapped help button event. An action from XOML point of view is something that is done in response to an event. Continuing on with the same example, an action that could be carried out in response to the user pressing the help button is to load and show a help scene.

Scenes and different types of actors all handle a variety of different events (more on this later). Likewise scenes and different actors can carry our different actions. If we look back at our scene and actor property definitions we see many properties with names such as OnCreate, OnDestroy, OnTapped etc.. These properties represent an event handler. When we define one of these properties for an object we tell the system that we want to respond to that event occurring by calling an actions list.

An actions list is a list of actions that should be carried out in response to an event occurring. Actions lists can be declared globally, local to a scene or even local to an actor.

The BasicActions example shows an example how to use events with actions. Lets take a look at the XOML for this example to see how it works:

```
<!-- Create rotation animation (we will use this to animate angle of the label) -->
<Animation Name="AngleAnim" Duration="4" Type="float">
    <Frame Time="0" Value="0" />
    <Frame Time="4" Value="360" />
</Animation>

<!-- Create a timline that can be used to animate an actor -->
<Timeline Name="Anim1" AutoPlay="false">
    <Animation Anim="AngleAnim" Target="Angle" />
</Timeline>

<!-- Create a scene -->
<Scene Name="Scene1" Current="true" >



    <!-- Create a label with an animation -->
    <Label Font="serif" Background="Button1Brush" BackgroundColour="80, 80, 255, 255"
          Size="100, 100" Text="Animation" OnBeginTouch="Spin" OnEndTouch="StopSpin"
          Timeline="Anim1">
        <Actions Name="Spin">
            <Action Method="PlayTimeline" />
        </Actions>
        <Actions Name="StopSpin">
            <Action Method="PauseTimeline" />
        </Actions>
    </Label>

</Scene>
```

Firstly we create an animation and a timeline that will be used to spin our label. Next we create a label within a scene and assign action lists to the OnBeginTouch and OnEndTouch events. Next, we define two actions list, one called Spin and another called StopSpin. The first action list calls an action called PlayTimeline which tells the actor to start playing its current timeline. The second actions list calls an action called PauseTimeline which tells the actor to pause playback of its timeline. The OnBeginTouch event is raised when the user starts to press the actor, the OnEndTouch event is raised when the user stops touching the actor.

You use the Actions tag to declare a list of actions. Actions can be declared globally so that any scene or actor can call them, local to a scene so that the scene and any actors contained with the scene can call them or local to an actor in which case only that actor can call them. Lets take a look at the properties that are supported by an Actions list:

- Name (string) - Name of actions group
- Local (boolean) - When set to false, this actions group will be added to the scenes actions manager instead of the actors actions manager
- Condition (variable) - A condition variable that must evaluate to true for this action list to be executed (this is an optional feature and can be used to

conditionally execute actions lists based on the state of the app, such as the value of certain variables)
- Tag (string) - Resource tag (used to group resources together)

An actions list contains a list of action tags. An action tag has the following properties:

- Method (method name) - A method is the actual action that takes places such as PlayTimeline, SetProperty etc..
- Param1 to Param5 - Five parameters that can be passed to the action. with exception to parameters that are passed as optional container scenes. Note that the shorter P1 to P5 attributes can be substituted for Param1 to Param5. Variables can be passed to any parameter that is not expecting a variable name or optional container scene
- Condition (variable) - A condition variable that must evaluate to true for this action to executed (this is an optional feature and can be used to conditionally execute actions based on the state of the app, such as the value of certain variables)

Method is a required property. In some cases Param1 to Param5 may also be required properties (see individual actions)

Note that when passing a variable to an actions parameter the variable will be searched for in the container scene (or in many cases the optional container scene that was passed). If it is not found there then the global variables space will be searched. If the variable is found then the variables value will be substituted as the parameter. If the variable does not exist then XOML assumes that the variable is a value.

For clarity the container scene is the scene that contains the actor that called the action, if the action is called from an actor. If the action was called from a scene then that scene will be the container scene.

12.2 Supported Actions

XOML has an ever growing list of supported actions that can be called to carry out certain functions. In this section we will go through all of the actions that are currently available to use.

12.2.1 Scene Specific Actions

- ChangeScene - Changes the currently focused scene to the specified scene
    - Param1 - Target scene name
- SuspendScene - Suspends the specified scene, stopping all processing of its contents
    - Param1 - Target scene name (uses container scene if not specified)
- ResumeScene - Resumes the specified scene, resuming processing of all of its contents
    - Param1 - Target scene name (uses container scene if not specified)
- HideScene- Hides the specified scene
    - Param1 - Target scene name (uses container scene if not specified)
- ShowScene- Shows the specified scene
    - Param1 - Target scene name (uses container scene if not specified)
- KillScene - Destroys and removes the specified scene from the game, all contained resources amd actors will be also be destroyed
    - Param1 - Target scene name (uses container scene if not specified)
- KillAllScenes - Destroys and removes all scenes except the scene specified by Param1 to Param5
    - Param1 - Scene NOT to remove
    - Param2 - Scene NOT to remove
    - Param3 - Scene NOT to remove
    - Param4 - Scene NOT to remove
    - Param5 - Scene NOT to remove
- SetCurrentScene - Sets the currently active scene, bringing it to the front of the scene stack. If the previous scene can be suspended (AllowSuspend=”true”) then it will be suspended
    - Param1 - Target scene name

- BringSceneToFront - Brings the scene to the front of the scene stack
    - Param1 - Target scene name
- SetAllSceneTimelines - Sets the timelines of all active scenes
    - Param1 - Name of timeline

12.2.2 Actor Specific Actions

- HideActor - Hides the specified actor
    - Param1 - Target actor (uses container actor if not specified)
- ShowActor - Shows the specified actor
    - Param1 - Target actor (uses container actor if not specified)
- ActivateActor - Activates the specified actor
    - Param1 - Target actor (uses container actor if not specified)
- DeactivateActor - Deactivates the specified actor
    - Param1 - Target actor (uses container actor if not specified)
- KillActor - Kills and removes the specified actor
    - Param1 - Target actor (uses container actor if not specified)

12.2.3 Timeline Specific Actions

- PlayTimeline - Plays the specified timeline. If no timeline is supplied then the current actor / scenes timeline will be played / resumed (depends on where the action was defined)
    - Param1 - Name of timeline
    - Param2 - Scene or actor that contains the timeline (optional)
- StopTimeline - Stops the specified timeline. If no timeline is supplied then the current actor / scene timeline will be stopped. (depends on where the action was defined)
    - Param1 - Name of timeline
    - Param2 - Scene or actor that contains the timeline (optional)
- SetTimeline - Changes to the specified timeline. If no timeline is supplied then the current scene / actors timeline will changed and restarted
    - Param1 - Name of timeline
    - Param2 - Scene or actor that contains the timeline (optional)
- PauseTimeline - Pauses the specified timeline. If no timeline is supplied then the current scene / actors timeline will be paused. (depends on where the action was defined)
    - Param1 - Name of timeline

  - Param2 - Scene or actor that contains the timeline (optional)

12.2.4 Audio Specific Actions

- PlaySound - Starts playing a sound effect
  - Param1 - Sound effect name
- PlayMusic - Starts the playing specified music file
  - Param1 - Music file name
  - Param2 - Number of times to repeat, 0 for infinite
- StopMusic - Stops the music player playing
- AudioEnable – Enable4s / disable sound and music
  - Param1 – Enable / disable music
  - Param2 – Enable / disable sound

12.2.5 Variable Specific Actions

- SetVar - Sets the value of a variable. If variable is an array then the array is filled with the value
  - Param1 - Variable name
  - Param2 - Variable value
  - Param3 - Scene where variable lives (optional)
- SetVar (random version) - Generates a random number / character between min and max and places it into the target variable. If variable is an array then the array is filled
  - Param1 - Variable name
  - Param2 - “rand”, “randchar” - rand generates a random number, whilst randchar generates a random single character
  - Param3 - Scene where variable lives (optional)
  - Param4 - Minimum random number / character
  - Param5 - Maximum random number / character
- AddVar - Adds an amount onto a XOML variable capping the variable to an optionally specified limit
  - Param1 - Variable to update
  - Param2 - Amount to add
  - Param3 - Limit (optional)

- EnterValue - Brings up the devices on screen keyboard and allows the user to enter a value which is placed into the destination variable
    - Param1- Message to show user
    - Param2 - Variable to place entered text into
    - Param3 - Default text, if you want to replace the variables value as the default text that is shown to the user
- UpdateText - Updates the target text actor with the value of the specified variable
    - Param1 - Target text actor name
    - Param2 - Variable to write to the actor
    - Param3 - Scene where target actor lives (optional)

12.2.6 Resource Removal Actions

- RemoveResource - Removes the named resource from the global resource manager
    - Param1 - Resource to remove
- RemoveResources - Removes all resources that match the supplied tag from the global resource manager
    - Tag to match

12.2.7 Properties and Modifiers Actions

- AddModifier - Adds a modifier to a scene or actor
    - Param1 - Modifier name
    - Param2 - Modifiers first parameter
    - Param3 - Specify a specific actor to add the modifier to (optional)
    - Param4 - Specify which scene the actor lives in. If Param3 is not supplied then the modifier will be added to the scene specified by Param4 (optional)
- ChangeModifier - Changes an existing modifier in a scene or actor
    - Param1 - Modifier name
    - Param2 - Specified how the modifier is to be changed (activate, deactivate, toggle or remove)
    - Param3 - Specify a specific actor that contains the modifier to modify (optional)
    - Param4 - Specify which scene the actor lives in. If Param3 is not supplied then the modifier in the scene specified by Param4 will be changed (optional)
- SetProperty - Sets the property of an actor or scene directly
    - Param1 - Property name
    - Param2 - Property value
    - Param3 - Specific actor to set property of, if omitted then actor that the action is called from will be used (optional)
    - Param4 - Scene where the specific actor specified in Param3 lives, if Param3 not specified then property of this scene will be set (optional)
- AddProperty - Adds a value onto the property of an actor or scene

    - Param1 - Property name
    - Param2 - Property value, if the target value is a boolean then it will be toggled on / off
    - Param3 - Specific actor to add property of, if omitted then actor that the action is called from will be used (optional)
    - Param4 - Scene where the specific actor specified in Param 3 lives, if Param3 not specified then property of this scene will be set (optional)
- SetUserProp - Sets the user property of an actor or scene directly
    - Param1 – User property name
    - Param2 - User property value
    - Param3 - Specific actor to set user property of, if omitted then actor that the action is called from will be used (optional)
    - Param4 - Scene where the specific actor specified in Param3 lives, if Param3 not specified then user property of this scene will be set (optional)
- AddUserProp - Adds a value onto the user property of an actor or scene
    - Param1 – User property name
    - Param2 – User property value, if the target value is a boolean then it will be toggled on / off
    - Param3 - Specific actor to add user property of, if omitted then actor that the action is called from will be used (optional)
    - Param4 - Scene where the specific actor specified in Param 3 lives, if Param3 not specified then user property of this scene will be set (optional)
- UserPropToVar – Copies the user property of an actor or scene to a variable
    - Param1 – User property name
    - Param2 – Name of target variable
    - Param3 - Specific actor to use, if omitted then actor that the action is called from will be used (optional)
    - Param4 - Scene where the specific actor specified in Param3 lives, if Param3 not specified then this scene will be used (optional)
- SetKeyFocus - Sets the UI actor that has key focus (used by devices that have keys)
    - Param1 - Actor name

12.2.8 Programs and Commands Actions

- ChangeProgram - Changes a running program
    - Param1 - Name of program to change
    - Param2 - Command to pass, which can include start, stop, pause, next, priority, restart and goto
    - Param3 - Scene in which program lives
    - Param4 – Used by goto to specify command name to go to.

12.2.9 Loading / Saving / Instantiation Actions

- LoadFile - Tells a file resource to load itself
  - Param1 - Name of file resource
  - Param2 - If true then execution will pause until file is loaded
  - Param3 - New file name (optional). If new file name is supplied then the new file will be loaded into the file resource
  - Param4 = Scene in which file resource lives
- FromTemplate - Instantiates a template
  - Param1 - Template resource name
  - Param2 - Parameters to pass to the template (p1=val1:p2=val2:p3=val3 etc)
  - Param3 - Scene where template should be instantiated (optional)
- LoadXOML - Loads a XOML file into the scene or globally. If a scene is provided then the XOML will be loaded into that scene
  - Param1 - XOML file name to load
  - Param2 - Target scene to load XOML data into, if not supplied then XOML is loaded globally (default)

12.2.10 Actions and Scripts Actions

- CallActions - Calls the actions list specified by Param1
  - Param1 – actions list to call
  - Param2 – Parent actor name that contains the actions list (optional)
  - Param2 - Parent scene name that contains the actions list (optional)
- CallScript - Calls a script function in a script that has already been loaded into the scene.
  - Param1 - Script function name to call
  - Param2 - Scene or actor that should be passed as the first parameter to the script function. If not set the the actor or scene that the action is defined inside of will be passed.
  - Param3 to Param5 will be passed as the 2nd, 3rd and 4th parameters of the function call.

- CallGlobalScript - Calls a global script function that has already been loaded globally.
  - Param1 - Script function name to call

- Param2 - Scene or actor that should be passed as the first parameter to the script function. If not set the the actor or scene that the action is defined inside of will be passed.
- Param3 to Param5 will be passed as the 2nd, 3rd and 4th parameters of the function call.

- Inline – Executes snipits of script code directly from XOML
  - Param1 – Script snipit to execute
  - Param2 – Name of scene that contains the script engine that should execute the script. By not passing this parameter the scene that contains the action will be used. If the action is declared outside of a scene then the global script engine will be used.

12.2.11 Miscellaneous Actions

- DebugText - Outputs debug text to the console
  - Param1 - Text to output
  - Param2 - Variable to output
- Launch - Launches an external URL
  - Param1 - URL to launch. The system interprets what should be ran from the protocol name and file name extension. For example passing “mailto:” will send an email, whilst passing “http:” will launch a web page etc..
- setBGColour - Sets the apps current background colour
  - Param1 - Background colour in r, g, b, a format
- Exit - Requests that the app exits
- SetPaused – Pause / resume time
  - Param1 – true to pause time and false to resume time

Note that all parameters that are passed to actions can be values or variables, so be careful not to pass a value that is a variable name in the current scope

12.2.12 Remote Data

- RemoteReq – Calls a remote request
  - Param1 – Name of RemoteReq resource to call
  - Param2 – Data to send to the request (optional)

12.2.13 VideoCam

- ChangeVideoCam – Allows stopping and starting of video cam resource
  - Param1 – Name of VideoCam resource to call
  - Param2 – Command to use to change video cam (start or stop)

12.2.14 Market

- Purchase – Starts an in-app purchase
    - Param1 – Name of the product as defined in the XOML market

12.2.15 Timers

- ChangeTimer – Changes an existing timer
    - Param1 – Name of the timer to change
    - Param2 – Timer container (optional) – if not supplied then the containing scene or actor will be used. If the timer is not found then ethe global resource manager will be checked for the timer
    - Param3 – Command to pass to timer (start, stop, restart)

13.0 Variables

13.1 Introduction

Most apps and games usually have elements that vary. For example a game could keep track and display the players current score and number of lives left, or an app may want to remember which page of the app the user is currently viewing. XOML allows your app to remember information such this using variables. You can think of a variable as a storage box that holds some kind of data of a specific type. A variables type can be one of the following:

- bool - This variable can hold a value of true or false
- float -This variable can hold a value that is a decimal number
- int - This variable can hold a value that is a whole number
- vec2 - This variable can hold two values separated by commas
- vec3 - This variable can hold three values separated by commas
- vec4 - This variable can hold four values separated by commas
- string - This variable can hold a string of text
- xml - This variable can hold an XML data tree
- condition - This variable can hold a set of conditions
- arraybool - This variable can hold a collection of bool variables
- arrayint - This variable can hold a collection of int variables
- arrayfloat - This variable can hold a collection of float variables
- arraystring - This variable can hold a collection of string variables

XOML provides the Variable tag that is used to create variables. These variables can be modified using actions, commands and scripts as well as bound to actor / scene / UI properties. Here are a few examples of variables declared in XOML:

```
    <Variable Name="PlayerName" Type="string" Value="Player One" />
    <Variable Name="PlayerScore" Type="int" Value="0" />
    <Variable Name="PlayColour" Type="vec4" Value="255, 128, 64, 8" />
    <Variable Name="GridItems" Type="arraystring" Size="12" Value="Item 1, Item 2, Item 3, 
Item 4, Item 5, Item 6, Item 7, Item 8, Item 9, Item 10, Item 11, Item 12” />
```

In the above examples we firstly create a variable called PlayerName that is of type stringand assign it the text “Player One”. Next we create a variable called PlayerScore that is of type int and assign it the value of 0. Next we create a variable called PlayerColuor of type vec4 which holds 4 values (255, 128, 64, 0). Finally we create a variable called GridItems that is of type arraystring (An array is a collection of variables all of the same type) then assign each variable of the array with the values Item1 to Item 12.

The XOML Variable tag supports the following properties:

- Name (string) - Name of the variable
- Type (type) - Type of variable. Variables can be of type int, bool, float, vec2, vec3, vec4, string, xml and condition
- Value (string) - The initial value of the variable
- Persist (boolean) - If set to true then the variables value will be saved when the app exits and re-loaded when the app begins execution again
- Instant (boolean) – If set to true then the variable will instantly be saved whenever it changes. Setting instant to true wil also make the variable persistent
- Condition (variable) - A condition variable that must evaluate to true for this variable to be loaded (this is an optional feature and can be used to conditionally create variables based on certain the values of other variables)
- BindXML (xml expression) - Specifies variable:tag:attribute XML data to bind to an array
- Tag (string) - Resource tag (used to group resources together)

Type is a required property

The Variables example has been provided that shows how to use some types of variables. Lets take a look at how this XOML works:

```
    <!-- Create a variable that holds the players name -->
    <Variable Name="PlayerName" Type="string" Value="Player One" />

    <!-- Create a variable that holds the players sctore -->
    <Variable Name="PlayerScore" Type="int" Value="0" />

    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true">

        <!-- Create a button with the players name bound to the text property -->
        <Label Font="serif" Background="Button1Brush" Binding="[Text]PlayerName" />

        <!-- Create a button with the players score bound to the text property -->
        <Label Position="0, 100" Font="serif" BackgroundColour="80, 80, 255, 255"
Background="Button1Brush" Binding="[Text]PlayerScore" OnTapped="UpdateScore">
            <Actions Name="UpdateScore">
                <Action Method="AddVar" Param1="PlayerScore" Param2="1" />
            </Actions>
        </Label>

    </Scene>
```

In this example we create a variable called PlayerName that holds the string “Player One”. Next we create a variable called PlayerScore that holds the players score, we set its value to 0. We now create two labels the first contains a Binding expression that basically tells the label to set the text of the label to whatever is stored in the PlayerName variable. We then do the same with the PlayerScore binding it to the Text property of the players score label. To show the value of the variable changing we add an action to the 2nd label that increases the value of the PlayerScore by 1 each time the user taps the label button.

Do not worry if you do not understand the concept Of bindings just yet, we will be covering them in more detail very soon.

13.2 Variable Scope

Like all other XOML resources variables have scope. Scope refers to the ability to access a resource from specific places in your app. For example, if we have a variable called Variable1 that is defined in a Scene called Scene1 then generally you can only access that variable from within Scene1. So for example, if we have a 2nd Scene called Scene2, you cannot ordinarily access Variable 1 from Scene2 because it is outside Scene2's scope.

Variables defined outside a scene will be assigned to the global variables table, whilst variables defined inside a scene will be assigned to the scenes variable table. When a scene is destroyed all variables within that scene will be lost

13.3 Array Variables

Sometimes its just not good enough to have a variable hold a single value and often many apps have the need to hold many values in the same variable. For example, a game may want to store the list of top 20 highest scores that have been achieved or a a finance app may want to store a list of the latest currency exchange rates.

Arrays come to the rescue as they can hold as many values as you like. An array holds a collection of variables instead of a single variable.

Accessing an array is usually done via a numerical index starting from 0. With 0 representing the 1st element and 1 representing the 2nd element and so on. Usually to access a variable at a particular array index we use ArrayName:ArrayIndex format. For example:

HiScore:0 - The 1st high score
HiScore:1 - The 2nd high score
HiScore:99 - The 100th high score

13.4 Conditional Variables

Conditional variables are variables that describe a group of variables, values and operations on those variables. They are used to check the value of a single variable or collections of variables being set to specific values, enabling you to create complex logic based on variables values. You can think of conditional variables as a way of saying “if this and that is true then return true”. Lets take a look back one of our previous examples that used conditional variables, the ConditionalImages example, as this contains conditional variables that decide which image should be loaded based on the devices screen size rating:

```
    <!-- Create condition variables that check the systems screen size rating  -->
    <Variable Name="LowResCheck" Type="condition" Value="system:3 LTE 2" />
    <Variable Name="MedResCheck" Type="condition" Value="system:3 GT 2 AND system:3 LTE 
3" />
    <Variable Name="HighResCheck" Type="condition" Value="system:3 GT 3" />
```

We create 3 conditional variables that each check for that the screen size rating being set to specific values. Lets look at each one in turn:

LowResCheck - This variable is assigned the following expression - system:3 LTE 2. System:3 is a special variable that contains the screen size rating of the device that is running your app. Our expression checks to see if that value is “less than or equal to” 2. if it is then the conditions value will be true
MedResCheck - This variable is assigned the following expression - system:3 GT 2 AND system:3 LTE 3. This expression is read as “if screen rating is greater than 2 and screen rating is less than or equal to 3” then the condition becomes true
HighResCheck - This variable is assigned the following expression - system:3 GT 3. This expression is read as “if screen rating is greater than 3” then the condition becomes true

Conditions can be used for all sorts of purposes including deciding if certain resources are created and loaded or deciding if certain actions should occur.

The ConditionalActions example has been provided to show how we can use conditions to affect which actions within an actions list get ran. Lets take a quick look at the XOML for this example to see how using actions and conditions together can be used to create complex logic:

```
    <!-- Create a variable that holds the players sctore -->
    <Variable Name="PlayerScore" Type="int" Value="0" />

    <!-- Create a condition variable -->
    <Variable Name="PlayerScoreTooHigh" Type="condition" Value="PlayerScore GT 10"/>

    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true">

        <!-- Create a button with the players score bound to the text property -->
        <Label Font="serif" BackgroundColour="80, 80, 255, 255" Background="Button1Brush"
Binding="[Text]PlayerScore" OnTapped="UpdateScore">
            <Actions Name="UpdateScore">
                <Action Method="AddVar" Param1="PlayerScore" Param2="1" Condition="!
PlayerScoreTooHigh" />
                <Action Method="SetVar" Param1="PlayerScore" Param2="0"
Condition="PlayerScoreTooHigh" />
            </Actions>
        </Label>

    </Scene>
```

We begin by creating a variable called PlayerScore and assign it the value of 0. Next e create a condition variable called PlayerScoreTooHigh which checks to see if the PlayerScore is greater than the value of 10. Next we create a label inside a scene that displays the players score. It also handles the OnTapped event by calling the UpdateScore actions list. The interesting part of this example is where we have added conditions to the actions definition, e.g.:

```
<Action Method="AddVar" Param1="PlayerScore" Param2="1" Condition="!PlayerScoreTooHigh" />
<Action Method="SetVar" Param1="PlayerScore" Param2="0" Condition="PlayerScoreTooHigh" />
```

The first action will only be executed if PlayerScoreTooHigh is not true (pre-pending an exclamation mark (!) to the variable name will check for the condition being false instead of true). So in this case the first action will only be executed if PlayerScore is less than or equal to 10.

The second action will only be executed if PlayerScoreTooHigh is true. In this case the second action will only be executed if PlayerScore is greater than 10.

Now lets take a look at the rules that govern how we create a condition variable expressions. The conditional variable expression is broken up as follows:

<Variable> <Operator> <Value> <Join> <Variable> <Operator> <Value> <Join> etc..

Variable specifies the name of the variable for which you ant to check the value. The Value is the value that we want to check the variables value against. The Operator is one of a number of ways of telling the system how to compare the variables value against the supplied Value. The following operators are available:

- == - Checks if the variables value is equal to the supplied value (e.g. Variable=100)
- != - Checks if the variables value is not equal to the supplied value (e.g. Variable!=100)
- GT - Checks if the variables value is greater than the supplied value (e.g. Variable GT 100)
- GTE - Checks if the variables value is greater than or equal to the supplied value (e.g. Variable GTE 100)
- LT - Checks if the variables value is less than the supplied value (e.g. Variable LT 100)
- LTE - Checks if the variables value is less than or equal to the supplied value (e.g. Variable LTE 100)
- Bitwise AND (AND) - Performs a bitwse AND (mask) of the variables value and the supplied value (e.g. Variable AND 8)

And finally the Join parameter is also an operator that can be either AND or OR. Join specifies how the result of each variable comparison affects the next. If for example you join two checks with an AND then both must be true for the condition to be met. If on the other hand you join them with an OR then either can be true for the condition to be met.

There are a number of special case variable comparison operators that you should be aware of including:

- If the comparison variables type is string then comparison operators will act on the length of the string instead of the strings value
- If the comparison variables type is string then the AND operator will search for the supplied string within the variable
- If the comparison variables type is a vector then comparison will be with the length of the vector and not the values of the vectors parameters

Using conditions on arrays work somewhat differently to normal variable conditions:

String Arrays:

If the comparison variable type is an array of strings then the operators work as follows:

- == - Returns true if the string matches any of the strings in the array
- != - Returns true if the string does not match any of the strings in the array
- AND – Returns true if the string matches any of the strings or parts of the strings within the array

Here are a few examples string array conditions:

```
<Variable Name="String1" Type="arraystring" Size="5" Value="hello, ello, world, one, two" />
<Variable Name="Find" Type="condition" Value="String1 == hello" />
```

The condition "Find" searches the array "String1" for the word "hello", if any of the strings match then the condition will return true

```
<Variable Name="String1" Type="arraystring" Size="5" Value="hello, ello, world, one, two" />
<Variable Name="Find" Type="condition" Value="String1 AND ello" />
```

The condition "Find" searches the array "String1" for the word "ello", if any of the strings "contain" the word, including partial matches (e.g. Hello and ello both match) then the condition will return true.

Boolean Arrays:

If the comparison variable type is an array of boolean then the operators work as follows:

- == - Returns true if all of the variables in the array match the supplied boolean value
- != - Returns true if none of the variables in the array match the supplied boolean value
- AND – Returns true if any of the variables in the array match the supplied boolean value

Int / Float Arrays:

If the comparison variable type is an array of integers / floats then the operators work as follows:

- == - Returns true if any of the variables in the array match the supplied integer value
- != - Returns true if none of the variables in the array match the supplied integer value
- GT - Returns true if the supplied value is greater than any values in the array
- GTE - Returns true if the supplied value is greater than or equal to any values in the array
- LT - Returns true if the supplied value is less than any values in the array
- LTE - Returns true if the supplied value is less than or equal to any values in the array
- AND – Returns true if the supplied value masked (bitwise AND) with any of the values in the array yield the same value as the supplied value. AND is not used by float arrays and always returns false

Here are a few examples of int array conditions:

```
<Variable Name="Array1" Type="arrayint" Size="5" Value="5, 8, 11, 15, 200" />
<Variable Name="Find" Type="condition" Value="Array1 == 11" />
<Variable Name="Check1" Type="condition" Value="Array1 LTE 2" />
<Variable Name="Check2" Type="condition" Value="Array1 AND 7" />
```

Find returns true in this example because 11 is one of the numbers in the array
Check1 returns false because none of the numbers in the array are less than or equal to 2
Check2 returns true because 15 AND 7 == 7

13.5 Binding Variables

Its often useful to be able to create variables and when those variables change the changes automatically modify properties of various objects. XOML provides a mechanism to do this called data bindings. Data binding is the ability to map properties of objects to XOML variables. Changes to those variables will be immediately reflected in all properties of all objects that are bound. The Basic Data Bindings example has been provided that shows how to use data binding. Lets take a look at the XOML for this example to see how it works:

```
    <!-- Create a variable that holds the labels position -->
    <Variable Name="LabelPosition" Type="vec2" Value="0, 0" />

    <!-- Create a variable that holds the labels angle -->
    <Variable Name="LabelAngle" Type="float" Value="0" />

    <!-- Create a data bindings list -->
    <Bindings Name="LabelBindings">
        <Binding Property="Position" Variable="LabelPosition" />
        <Binding Property="Angle" Variable="LabelAngle" />
    </Bindings>

    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true">

        <!-- Create a button that is bound using the LabelBindings list -->
        <Label Font="serif" BackgroundColour="80, 80, 255, 255" Background="Button1Brush"
Bindings="LabelBindings" OnTapped="Update">
            <Actions Name="Update">
                <Action Method="AddVar" Param1="LabelPosition" Param2="20, 15" />
                <Action Method="AddVar" Param1="LabelAngle" Param2="5" />
            </Actions>
        </Label>

    </Scene>
```

In this example we create two variables that hold the labels position and angle. We then create a data bindings list using the Bindings tag. This Bindings list is assigned two bindings, the first binds the Position of whatever the list is bound to to the LabelPosition variable. The second binds the Angle of whatever the list is bound to to the LabelAngle variable. Finally we create a label with an actions list which is ran when the user taps the label. These actions update the LabelPosition and LabelAngle variables.

As you can see using bindings we can modify the variables insead of the actual objects properties and those changes will automatically be reflected in our object.

The Bindings tag as the following properties:

- Name (string) - Name of bindings list
- Tag (string) - Resource tag (used to group resources together)

The bindings tag also contains inner Binding tags that defines which properties of the object match to which variable. Lets take a look at the properties that the Binding tag supports:

- Property (property) - The name of the object property that should be bound
- Variable (variable) - The name of the variable that should be bound to the object

Property and Variable are required properties

Bindings lists are stored as resources. Bindings that are declared inside a scene will be added to the scenes resource manager whilst bindings that are created outside a scene will be added to the global resource manager.

13.6 Simple Bindings

Whilst bindings lists are very useful for binding many properties to variables across lots of different objects, they can sometime be surplus to requirements. Its possible to bind a single property to a single variable without the need for a full bindings list. This style of binding is called a “simple binding” and is done in the actual object definition.

If we take a look back the Variables example app we can see a place where we have already used simple bindings by looking at this line of XOML:

```
<Label Position="0, 100" Font="serif" BackgroundColour="80, 80, 255, 255"
      Background="Button1Brush" Binding="[Text]PlayerScore" OnTapped="UpdateScore">
```

In this example instead of using the Bindings property we use the Binding property instead. This binding tells the label to bind the PlayerScore variable to the Text property of the label. The format of a simple binding is as follows:

Binding=”[property]variable:index”

The property is the property name of the object, variable is the name of the variable and index is the array index if the variable is an array type, index can be a number of another variable that contains the index.

13.7 XML Variables

This is quite an advanced topic that you may want to return too when you have a little more experience with XOML. XML variables are a special kind of variable that store an XML tree. XML variables can be targeted by files with loaded data being converted to XML when written to the variable. In addition, array variables can be made to collect collections of named attributes from a named tag within the XML. The XMLVariable example has been provided as an example showing how to XML variables. Lets take a look at the XOML to see how it works:

```
<!-- Create a scene -->
<Scene Name="Scene1" Current="true">

    <!-- Create templates that is used by the grid -->
    <Template Name="GridItemTemp">
        <Label Name="GridItem$index$" Background="SmallPanelBrush"
         BackgroundColour="200, 200, 200, 255" SelectedColour="200, 255, 200, 255"
         Font="serif" GridPos="$gridpos$" Tappable="true" SelectType="normal"
         Selected="false" />
    </Template>

    <!-- Create data that will be bound to grid columns -->
    <Variable Name="TestXML" Type="xml" />
    <File Name="File1" Location="test1.xml" FileType="xml" Preload="true"
         Variable="TestXML" />
    <Variable Name="GridItems1" Type="arraystring" Size="1"
         BindXML="TestXML:Chapter:Name" />
    <Variable Name="GridItems2" Type="arraystring" Size="1"
         BindXML="TestXML:Chapter:Description" />
    <Variable Name="GridItems3" Type="arraystring" Size="1"
         BindXML="TestXML:Chapter:Pages" />

    <!-- Generate the grid -->
    <Grid Name="ItemsGrid" Size="-100, -100" Background="PanelBrush"
         BackgroundColour="255, 255, 255, 255" Tappable="true"
         ClipMargin="10, 10, 10, 10" ItemsTemplate="GridItemTemp" MultiSelect="false"
         Selection="grid_selection" SelectedIndex="5" UseParentOpacity="false">
        <RowDefinition Name="r0" AlignV="middle" Height="100" />
        <ColumnDefinition Name="c0" AlignH="centre" Width="300" ItemsData="GridItems1"
                ItemsTargetType="text" />
        <ColumnDefinition Name="c1" AlignH="centre" Width="300" ItemsData="GridItems2"
                ItemsTargetType="text" />
        <ColumnDefinition Name="c2" AlignH="centre" Width="300" ItemsData="GridItems3"
                ItemsTargetType="text" />
    </Grid>

</Scene>
```

Because this example is quite complex we will split it down into a number of sections.

Firstly this example is based on a an actor called a Grid which we will be looking at in great depth in the user interface section. For now a grid can be though of as a way to represent a collection of data using rows and columns. A grid requires templates to define what its grid cells (a cell is an entry in the grid at a specific column and row). We will begin by creating a template that will be used to decide what our grid cells look like and behave. Next we create an xml variable and load an xml file into it. Once the file has been loaded the xml variable will contain a tree of xml data. Next we pick out values from specific attributes and create 3 arrays from them (GridItems1, GridItems2 and GridItems3). Finally we create a grid and bind the GridItem variables to it

The important section of this XOML that demonstrates xml variables is listed below:

```
<!-- Create data that will be bound to grid columns -->
<Variable Name="TestXML" Type="xml" />
<File Name="File1" Location="test1.xml" FileType="xml" Preload="true"
     Variable="TestXML" />
<Variable Name="GridItems1" Type="arraystring" Size="1"
     BindXML="TestXML:Chapter:Name" />
<Variable Name="GridItems2" Type="arraystring" Size="1"
     BindXML="TestXML:Chapter:Description" />
<Variable Name="GridItems3" Type="arraystring" Size="1"
     BindXML="TestXML:Chapter:Pages" />
```

When we initially create the TestXML variable it has no value. It does not get a value until the file test1.xml is loaded. Once that file loads the xml is written to the TestXML variable and converted to an XML tree.

The three following variables use a special property of the variable called BindXML. This tells the variable that it should generate its values from an xml tree, picking out the values for the specified properties. The format of the BindXML value is:

<XML Variable Name>:<Tag Name>:<Property Name>

The Tag Name is the name of the tag that we want to extract information from. The Property Name is the property of the tag that we ant to get the actual value from. Lets take a look at the test1.xml file used in this example to see how this all fits together:

```
<?xml version="1.0"?>
<xml>
```

```
    <Contents>
        <Chapter Name="Chapter1" Description="This is chapter 1" Pages="10" />
        <Chapter Name="Chapter2" Description="This is chapter 2" Pages="12" />
        <Chapter Name="Chapter3" Description="This is chapter 3" Pages="11" />
        <Chapter Name="Chapter4" Description="This is chapter 4" Pages="5" />
        <Chapter Name="Chapter5" Description="This is chapter 5" Pages="7" />
        <Chapter Name="Chapter6" Description="This is chapter 6" Pages="9" />
        <Chapter Name="Chapter7" Description="This is chapter 7" Pages="2" />
        <Chapter Name="Chapter8" Description="This is chapter 8" Pages="4" />
        <Chapter Name="Chapter9" Description="This is chapter 9" Pages="6" />
        <Chapter Name="Chapter10" Description="This is chapter 10" Pages="16" />
        <Chapter Name="Chapter11" Description="This is chapter 11" Pages="16" />
        <Chapter Name="Chapter12" Description="This is chapter 12" Pages="16" />
        <Chapter Name="Chapter13" Description="This is chapter 13" Pages="16" />
        <Chapter Name="Chapter14" Description="This is chapter 14" Pages="16" />
    </Contents>
</xml>
```

Running the XMLVariable example you will see the correlation between the properties that are bound to the grid on the display to the properties seen in this xml file. You should notice how all of the Name properties are shown in the first column, all of the Description properties are shown in the second column and all of the Pages properties are shown in the 3rd column.

13.8 Updating Variables

Variables wouldn't be much use if we had no way to change their values. XOML provides a number of ways in which variables can be modified including:

- Initial value - This is where the initial value of the variable is set when it is created
- Actions -Actions can change the value of a variable using SetVar and AddVar
- Commands - Command can change the value of a variable using set_var and add_var
- Scripts can modify the value of a variable using getVariable() and setVariable()

Lets take a quick look at a small example of each method of setting or changing a variables value:

```
<!-- Setting a variables initial value -->
<Variable Name="Var1" Type="string" Value="Hello World" />

<!-- Setting a variables value using actions -->
<Actions Name="SetVar">
    <Action Method="SetVar" Param1="Var1" Param2="Set By Action" />
</Actions>

<!-- Updating a variables value using actions -->
<Actions Name="SetVar">
    <Action Method="AddVar" Param1="Var1" Param2="Added By Action" />
</Actions>

<!-- Setting a variables value using commands -->
<Program Name="Main" AutoRun="true">
    <Command Method="set_var" Param1="Var1" Param2="Set By Command" />
</Program>

<!-- Updating a variables value using commands -->
<Program Name="Main" AutoRun="true">
    <Command Method="add_var" Param1="Var1" Param2="Added By Command" />
</Program>

-- Set / add variable via script
function Script1()
    local var1 = variable.get("Var1")
    variable.set(var1, "Set By Script")
end
```

There are also a couple of other ways to update variables that will be covered when we come to them, such as the grid control can update a variable with its current selection index, entry from the on screen keyboard, two way bindings etc..

13.9 Command and Action Parameter Variables

Command and actions can both accept variables as parameters, e.g.:

```
<Variable Name="Var1" Type="string" Value="World" />
<Label Font="serif" Background="BG1" Size="100,100" Text="Hello" OnTapped="Tapped">
    <Actions Name="Tapped">
        <Action Method="SetProperty" P1="Text" P2="Var1"/>
    </Actions>
</Label>
```

In the above example we create a variable and a label. The label responds to the OnTapped event by calling the Tapped actions list. The SetProperty method transfers the value of the variable Var1 to the Text property of the label.

Command and action parameters can also accept specific elements of arrays that are indexed via direct indexes or index variables, e.g.:

```
<Variable Name="Var1" Type="arraystring" Value="Element1,Element2,Element3,Element4" />
<Variable Name="Index1" Type="int" Value="0"/>
<Label Font="serif" Background="BG1" Size="100,100" Text="Hello" OnTapped="Tapped">
    <Actions Name="Tapped">
        <Action Method="SetProperty" P1="Text" P2="Var1:Index1"/>
        <Action Method="AddVar" P1="Index1" P2="1"/>
    </Actions>
</Label>
```

The above example now uses an array of values. Each time the user taps the button the value at the current index of Var1 (Var1[Index1]) is assigned to the Text property of the label. The indexer variable is then incremented to the next index.

13.10 Persistent Variables

XOML variables can be made to persist across multiple sessins of an app by marking them as Persist="true" when you declare them. All Persistent variables will be saved on app exit and reloaded when the app is ran again. Note that no two persistent variables may share the same name regardless of scope. The PersistentVariable example has been provided to show how use persistent variables. Lets take a look at the XOML for this example to see how it works.

```
<!-- Create a variable that holds some text -->
<Variable Name="Var1" Type="string" Value="Hello" Persist="true"  />

<!-- Create a scene -->
<Scene Name="Scene1" Current="true">

    <!-- Create a label to show the entered text -->
    <Label Font="serif" Background="Button1Brush" Binding="[Text]Var1" />

    <!-- Create a TextBox that allows us to enter text -->
    <TextBox Name="TextBox1" Font="serif" Variable="Var1" Position="0, 100"
         Size="200, 80" Background="Button1Brush" Text="Tap to Enter Text" />

</Scene>
```

We begin by creating a variable Var1 that is marked as persistent. We then create a label that will show the current value of the variable followed by a TextBox which allows the user to enter data into the app. Note that the TextBox contains a property called Variable which represents the variable where the user input will be placed. If you run the app, enter a value and exit then run the app again you will see that the to label contains the text that you entered in your last session.

13.11 System Variables Array

The system variables array is a special array that contains information about the device that your app is running on. The array supports querying of the following information:

- 0 - Display width
- 1 - Display height
- 2 - Standard graphics display mode name, for example HVGA, QXGVA etc..
- 3 - Size hint - This is a hint that can be used to separate resources into groups that can be used across different sized screens. The calculation is based on (width + height) / 400
- 4 - Device type
    - Unsupported= -1
    - iPhone = 0
    - iPad = 1
    - Android = 2
    - Bada = 3
    - QNX = 4
    - Symbian = 5
    - WinMobile = 6
    - WebOS = 7
    - Windows = 8
    - OSX = 9
- 5 - Multi-touch - If set to 1 then device supports multi-touch
- 6 - Accelerometer - If set to 1 then device supports accelerometer
- 7 - Compass - If set to 1 then device supports compass
- 8 - Keys - If set to 1 then device supports keyboard
- 9 - HasPointer - If set to 1 then the device has touch screen / pointer support
- 10 – Last purchase ID – Contains the ID of the last item purchased via the Market, -1 for none
- 11 – Screen Orientation – The current screen orientation in degrees

Using the system array you can style your app to fit different device configurations.

13.12 System Touches Array

The system touches variables array is a special array that contains information about the position and state of up to 5 touches on the device. The array supports querying of the following information:

- 0 – Touch 1 position x
- 1 – Touch 1 position y
- 2 – Touch 1 status (1 for touching, 0 for not touching)
- 3 – Touch 2 position x
- 4 – Touch 2 position y
- 5 – Touch 2 status (1 for touching, 0 for not touching)
- 6 – Touch 3 position x
- 7 – Touch 3 position y
- 8 – Touch 3 status (1 for touching, 0 for not touching)
- 9 – Touch 4 position x
- 10 – Touch 4 position y
- 11 – Touch 4 status (1 for touching, 0 for not touching)
- 12 – Touch 5 position x
- 13 – Touch 5 position y
- 14 – Touch 5 status (1 for touching, 0 for not touching)

An example condition that checks for touch 1:

```
<Variable Name="Touched1" Type="condition" Value="touches:2 == 1" />
```

In this example Touched1 will evaluate to true if the user touches the screen

14.0 Programs and Commands

14.1 Introduction

Whilst the events / actions system is great for handling the event based parts of your app or game, XOML needed a new system that would allow some kind of game / app logic scripting enabling developers to define sets of complex functionality that can be ran at any point during the apps lifetime. Programs that are declared local to a scene will ordinarily only be accessible from the scene, whilst programs declared outside a scene will be global and can be accessed from all scenes

XOML's program system allows developers to create complex programs from a list of hierarchical commands. Commands are executed in the order in which they are declared until the end of the program is reached. Commands can also executed concurrently, allowing the program to run multiple commands together each frame. Lets take a look at the SimpleProgram example to see how programs work:

```
<!-- Create a variable that holds a message -->
<Variable Name="Message" Type="string" />

<!-- Create a scene -->
<Scene Name="Scene1" Current="true">

    <!-- Create a simple program that changes the message very 2 seconds -->
    <Program Name="Main" AutoRun="true">
        <Command Name="Start" Method="set_var" Param1="Message" Param2="Hello" />
        <Command Method="wait" Param1="2" />
        <Command Method="set_var" Param1="Message" Param2="Goodbye" />
        <Command Method="wait" Param1="2" />
        <Command Method="goto" Param1="Start" />
    </Program>

    <!-- Create a button with the message bound to the text property -->
    <Label Font="serif" Background="Button1Brush" Binding="[Text]Message" />

</Scene>
```

The example begins by creating a variable that will hold a message. Next we create a program inside our scene called Main and we tell it to begin running automatically. We add a number of commands to the program which do the following:

- Set the Message variable to "Hello"
- Wait for 2 seconds
- Set the Message variable to "Goodbye"
- Wait for 2 seconds

- Go back to the start and start again

Lastly, we create a label that will display our message.

The program tag has the following properties:

- Name (string) - Name of program
- AutoRun (boolean) - When set to true the program will automatically begin running
- Priority (boolean) - When true this program becomes the priority program
- Tag (string) - Resource tag (used to group resources together)

A Program also contains inner Command tags with the following properties:

- Name (string) - Name of command. Its sometimes useful to name certain commands so they can be jumped to to bypass or repeat parts of a program
- Method (command) - Name of command to execute
- Param1 - Param5 (string) - Five parameters that get passed to the command, The shorter P1 – P5 attributes can be substituted for Param1 to Param5. Variables can be passed to any parameter that is not expecting a variable name or optional container scene
- Parallel (boolean) - All commands declared within this command will be executed in parallel (concurrently)
- IfReturn (command name), Value (number) - These two attributes used together allow conditional execution of commands based on the return value of another command. IfReturn specifies the name of the command to check and Value is the value to which it should be compared (call_script and get_var commands both return a value)

Note that commands are executed at a rate of one command per game / app frame for normal commands. Commands that reside inside a command marked as “parallel” will all be exuecte4d on the same frame.

14.2 Commands

XOML supports a variety of commands that can be included in a program including:

- nop - Does nothing
- call - Calls another program, execution of program is paused until the called program returns
    - Param1 - Name of program to call
- goto - Changes execution to the named command
    - Param1 - Name of command to go to
- change – Changes the program
    - Param1 – Command (start, restart, stop, pause)
    - Param2 – Name of program (optional)
- stop - Stops the program
- return - Returns to a calling program
- priority - Changes the current priority program
    - Param1 - Name of program to set as priority
- run_actions - Executes an actions list on a target scene or actor
    - Param1 - Name of action list to execute
    - Param2 - Actor or scene to apply actions
    - Param3 - Scene in which actor lives (optional)
- set_property - Sets the specified property of the target actor / scene
    - Param1 – Name of property to set
    - Param2 - Property value
    - Param3 - Actor who's property to change
    - Param4 - Scene in which actor lives (optional) or scene who's property to change
- set_userprop - Sets the specified user property of the target actor / scene
    - Param1 – Name of user property to set
    - Param2 – User property value
    - Param3 - Actor who's user property to change
    - Param4 - Scene in which actor lives (optional) or scene who's user property to change
- set_var - Sets the named variables value. If variable is an array then the array is filled with the value
    - Param1 – Name of variable to set
    - Param2 - Variables value
    - Param3 - Scene in which variable lives (optional)
- set_var (random version) - Generates a random number / character between min and max and places it into the target variable. If variable is an array then the array is filled
    - Param1 - Variable name
    - Param2 - “rand”, “randchar” - rand generates a random number, whilst randchar generates a random single character
    - Param3 - Scene where variable lives (optional)
    - Param4 - Minimum random number / character
    - Param5 - Maximum random number / character
- get_var - Gets a variables value and sets it as the commands return value. The value is set as

the return value of this command
  - Param1 – Name of variable to get
  - Param3 - Scene in which variable lives (optional)
- add_var - Adds a value to the variables existing value
  - Param1 – Name of variable to modify
  - Param2 - Value to add
  - Param3 - Scene in which variable lives (optional)
- if_var – Checks a variable against a value using a specific operator
  - Param1 – Name of variable to check
  - Param2 – Operator (==, !=, gt, lt, gte, lte, and)
  - Param3 – Value to check against
  - Param4 - Scene in which variable lives (optional)
- wait_var_is_value - Pauses execution of the program until a specific variable is a set value
  - Param1 – Name of variable to check
  - Param2 - Value that variable should be to continue
- call_script - Calls a function located in a script that has been loaded into the scene. The value returned from the script is set as the return value for this command.
  - Param1 – Name of script function to call
  - Param2 – An optional scene that is passed as the first argument to the function. If not supplied then the scene that the action is declared inside of will be passed as the 1st parameter of the script function call
  - Param3, Param4, Parm5 - Parameters to pass as the 2nd, 3rd and 4th script function parameters
- call_global_script - Calls a function located in a script that has been loaded globally. The value returned from the script is set as the return value for this command. Note that the base game object will be passed to the script function as its first parameter, unless a valid scene is supplied for Param2
  - Param1 – Name of script function to call
  - Param2 – An optional scene that is passed as the first argument to the function
  - Param3, Param4, Parm5 - Parameters to pass as the 2nd, 3rd and 4th script function parameters
- inline – Executes snipits of script code directly from XOML
  - Param1 – Script snipit to execute
  - Param2 – Name of scene that contains the script engine that should execute the script. By not passing this parameter the scene that contains the program will be used. If the program is declared outside of a scene then the global script engine will be used.
- from_template - Instantiates a template
  - Param1 - Template name
  - Param2 - Parameters to pass to the template (p1=val1:p2=val2:p3=val3 etc)
  - Param3 - Scene where template should be instantiated (optional)
- load_xoml – Loads a XOML file
  - Param1 – Name of XOML file
  - Param2 - Name of scene to load XOML data into, if not supplied then XOML is loaded globally (default)

- music - Plays or stops a music file
    - Param1 - Command (play or stop)
    - Param2 - Music file name to play
    - Param3 - Repeat count (0 play forever)
- sound - Play sound effect
    - Param1 - Sound effect name to play
    - Param2 – volume (1.0 is default)
    - Param3 – pitch (1.0 is default)
    - Param4 – pan (1.0 is default)
    - Param5 – looped (default is false)
- wait - Pauses execution of this XOML program for the specified number of seconds (pauses the program execution but not the app)
    - Param1 – Number of seconds to wait
- remote_req – Calls a remote request
    - Param1 – Name of RemoteReq resource to call
    - Param2 – Data to send to the request (optional)

In most cases parameters can be passed by value or using variables

14.3 Return Values

Some program commands can return a value and all commands can decide if they should or should not be executed based on the return value of specific commands. This style of XOML enables you to add logic to your XOML app quite easily. The ComplexProgram example has been provided to show how to use return values. Lets take a look at the XOML to see how it works:

```
<!-- Create a scene -->
<Scene Name="Scene1" Current="true" Extents="-1000, -1000, 2000, 2000">

    <!-- Create Count variable (used by the loop) -->
    <Variable Name="Count" Type="int" Value="0" />
    <!-- Create LabelPos variable -->
    <Variable Name="LabelPos" Type="vec2" Value="0, 0" />

    <!-- Create a short program that demonstrates the use of IfReturn abd Value -->
    <Program Name="Main" AutoRun="true">
        <Command Name="Start" Method="set_var" Param1="LabelPos" Param2="0, 0" />
        <Command Method="set_var" Param1="Count" Param2="0" />
        <Command Name="Loop1" Method="add_var" Param1="LabelPos" Param2="10, 10" />
            <Command Method="add_var" Param1="Count" Param2="1" />
            <Command Method="wait" Param1="0.5" />
            <Command Name="LoopCheck" Method="if_var" Param1="Count" Param2="LT"
                Param3="10" />
            <Command Method="goto" Param1="Loop1" IfReturn="LoopCheck" Value="1" />
        <Command Method="goto" Param1="Start" />
    </Program>

    <!-- Create a label -->
    <Label Size="100, 100" Font="serif" Background="Button1Brush" Text="Loop"
```

```
          WrapPosition="true" Binding="[Position]LabelPos" />

    </Scene>
```

This app creates a label that moves across and down the screen every half of a second. The goto command is used in conjunction with IfReturn and Value to create a loop that is executed.

We firstly create two variables, the first holds our loop counter and the second holds a labels screen position. Next we create a program and add a number of commands to it. The first command is named Start and sets our labels position to the centre of the screen whilst our next command sets the loop count variable Count to 0. Next we create a command named Loop1 which adds 10, 10 onto the labels position. The next two commands add 1 onto the loop count variable and then waist for half a second. Next we check to see if the loop count variable Count is less than 10. if it is then the next goto command will be executed which sends the program back to the command named Loop1. If not then the program continues on to the last goto command which goes back to the start of the program.
The two important command in this example are:

```
<Command Name="LoopCheck" Method="if_var" Param1="Count" Param2="LT" Param3="10" />
<Command Method="goto" Param1="Loop1" IfReturn="LoopCheck" Value="1" />
```

The first command checks to see if Count is less than the value of 10, if it is then the commands return value will be set to 1 otherwise it will be set to 0.

The goto command next checks the LoopCheck command to see if its return value is 1. if it is then the goto command will be executed. If not then the program continue on to the next instruction.

15.0 Files

15.1 Introduction

Its often useful to be able to load files into an app and do something with the data that they contain. Files in XOML are just like any other resource and can be declared inside a scene in which case they become local to the scene or declared outside a scene in which case they become global. They can also be loaded from local storage or from a web server, Files can also convert their data after loading from an number of different formats. We will begin by taking a look at the properties that are available for the Files tag:

- Name (string) - Name of this file resource
- Tag (string) - Resource tag (used to group resources together) (optional)
- Location (filename) - Name of the file including extension (can include web addresses)
- Preload (boolean) - If set to true then the file will be loaded immediately. By setting to false the file will be loaded when it is first used or can be loaded by an action later (default is true).
- Blocking (boolean) - Web based files take time to download from the web so its useful to allow execution of the app to continue whilst it downloads. To prevent the file download from blocking the app set Blocking="false" (default is to block).
- Condition (variable) - A condition variable that must evaluate to true for this resource to be loaded (this is an optional feature and can be used to conditionally load resources based on certain conditions such as screen size or device type etc..) (optional)
- FileType (string) – Type of file (does not affect how the file is loaded) (optional)
- Variable (variable) – Name of the variable that the contents of the file should be written into once loaded (optional). If the variable is an array then the source file contents will be split at commas and written as separate elements to the array
- Script (script) - Name of the script that the contents of the file should be written into once loaded. This is used only to load a script resource with script (optional)
- Converter (type) – How to convert the loaded data to text (html, hex, urlencoded). This is useful if you have data in a common web format such as url-encoded. Passing urlencoded will cause the data to be converted from url-

encoded to plain text format (optional)

The Files example has been provided to show how to load a file and reload its contents. Lets take a look at the XOML for this example:

```
<!-- Create a variable to load a file into -->
<Variable Name="FileContents" Type="string" />

<!-- Declare a file -->
<File Name="File1" Location="file1.txt" FileType="txt" Variable="FileContents"
   Preload="true" />

<!-- Create a scene -->
<Scene Name="Scene1" Current="true" >

    <!-- Create a label to display our files contents -->
    <Label Font="serif" Size="200, 200" TextColour="255, 255, 128, 255"
         Background="Button1Brush" BackgroundColour="255, 80, 80, 255"
         Binding="[Text]FileContents" />

    <!-- Create a group of buttons to load 3 different files -->
    <Label Font="serif" Size="80, 50" Position="-100, 100" Text="Load File1"
         Background="Button1Brush" BackgroundColour="80, 80, 255, 255" OnTapped="Load">
        <Actions Name="Load">
            <Action Method="LoadFile" Param1="File1" Param2="true" Param3="file1.txt" />
        </Actions>
    </Label>
    <Label Font="serif" Size="80, 50" Position="0, 100" Text="Load File2"
         Background="Button1Brush" BackgroundColour="80, 80, 255, 255" OnTapped="Load">
        <Actions Name="Load">
            <Action Method="LoadFile" Param1="File1" Param2="true" Param3="file2.txt" />
        </Actions>
    </Label>
    <Label Font="serif" Size="80, 50" Position="100, 100" Text="Load File3"
         Background="Button1Brush" BackgroundColour="80, 80, 255, 255" OnTapped="Load">
        <Actions Name="Load">
            <Action Method="LoadFile" Param1="File1" Param2="true" Param3="file3.txt" />
        </Actions>
    </Label>

</Scene>
```

We begin this example by creating a variable called FileContents that we will later load our files contents into. Next we create a File that loads the contents of file1.txt and writes it to the FileContents variable. Next we create a label that will sho the files contents due to the binding to the FileContents variable. Lastly we create 3 label buttons that each call the LoadFile action when tapped. Each of the LoadFile actions load different files into the FileContents variable.

16.0 User Interfaces

16.1 Introduction

All apps and games each have some kind of way that allows the user to interface with them. The user interface could be anything from simple buttons and sliders to complex 4 finger controllers that allow the player to control the game in crazy ways.

XOML provides a large collection of controls that can be used to put together versatile and complex user interfaces very quickly that can change shape and resize themselves to different screen sizes and orientations.

The XOML user interface system offers the following features:

- Data binding (two way in some cases). Data binding allows you to map XOML variables / arrays to UI components and when those variables change the changes are reflected in the user interface components
- 14 different kinds of controls (buttons, icons, labels, text boxes, list boxes, grids, sliders, canvas, stack panels, wrap panels, image views, text views, web views and tab bars)
- Support for events and actions, such as handling button toggling, selection and value changes.
- Fully integrated into the XOML animation system
- Styling and templating
- 9-patch rendering which allows you to render clean looking buttons / borders without losing resolution
- Supports multi-touch, pan and pinch zoom (up to 5 simultaneous touches)
- Supports modifiers and other customisations, allowing you to augment the behaviour of UI components
- Supports proportional sizing
- Supports dynamic orientation / screen size changes

All user interface components are derived from image actors so they have access to all of the properties that are available to an actor / image actor.

All user UI components have a number of states, which include:

- Selected state – This state signifies that the component has been selected by the user

- Disabled state – This state signifies that the component has been disabled
- Normal state – This state signifies that the component is in its normal state and is not selected or disabled

UI components have 3 visual states that equate to the states listed above. Background brush, colour and in some cases text can be defined for each component, enabling automated selection, which means you do not have to take care of this.

We will now take a look at all of the different user interface components in turn.

16.2 Icon

You can think of an Icon as a UI (user interface) component that can act as a simple image control or a button. In fact, "all" UI components can be used as buttons. As the Icon UI component is derived from an image actor it supports all of the same features that an image actor does, so it can even be placed under control of the physics system if you wanted it to. As well as all of the usual image actor attributes, an Icon also supports the following attributes:

- Background (brush) – This is the normal background brush that is displayed when this UI component is not selected
- SelectedBackground (brush) – This is the background brush that is displayed when this UI component is selected
- DisabledBackground (brush) – This is the background brush that is displayed when this UI component is disabled
- BackgroundColour (r, g, b, a) – This is the normal colour of the UI component when it is not selected
- SelectedColour (r, g, b, a) – This is the colour of the UI component when it is selected
- DisabledColour (r, g, b, a) – This is the colour of the UI component when it is disabled
- Enabled (boolean) – Enabled state
- SelectType (type) – Method of selection (normal, sticky, toggle). Sticky selection will keep the component selected once it has been selected. Toggle selection will toggle the selected state of the component each time the user taps on it. Normal selection will immediately deselect the component after the user stops touching it.
- Selected (boolean) – Initial selected state
- Spring (boolean) – If this actor allows scrolling and the user scrolls out of bounds then the actor will spring back into place (much like the iOS UI system)
- OnToggledOn (actions, write-only) – Specifies an actions list to be called when this element is toggled on
- OnToggledOff (actions, write-only) – Specifies an actions group to be called when this element is toggled off
- OnBackKey (actions, write-only) - Specifies an actions group to be called when the user presses the back key whilst this actor has key focus
- OnHomeKey (actions, write-only) - Specifies an actions group to be called when the presses the home key whilst this actor has key focus
- ClipMargin (left, right, top, bottom) – If this element contains children that are

clipped then this margin will push the clipping inwards to create a border (left, right, top, bottom). Note that passing negative values will use proportional sizing
- ScrollRange (left, top, width, height) – Sets the range that the contents of this UI component can be scrolled
- ScrollPos (top, left) – Sets the initial scroll position of the contained content
- ShowTimeline (timeline) – Sets the animation timeline that will be played when this element is shown
- HideTimeline (timeline) – Sets the animation timeline that will be played when this element is hidden
- SizeToContent (bool) - When set to true resizes this control to fit the size of its contained children, value values are x, y, xy, none. Method x will size the width, y the height, xy both the width and the height
- KeyFocus (boolean) - If set to true the UI component will have key focus
- ToggledOn (boolean) – Changes toggled state
- ColourOffset (r, g, b, a) – Offsets current colour

Icons are useful as image buttons and general containers

Here's a quick XOML example:

```
<Icon Background="Button2Brush" Size="-90, 0" SelectedColour="128, 255, 200, 255" />
```

This XOML creates an Icon UI component that is 90% the screen width and the height of the background brush Button2Brush.

As well as all of the usual image actor properties, all of the above properties can be a target of animation and can be modified an queried.

16.3 Label

The Label UI component is an Icon that displays text, basically a traditional label with an image background. As well as all of the usual Icon properties, a Label also supports the following properties:

- Font (font) – Name of font used to display text
- Text (string) – The text to display
- SelectedText (string) – The text to display when selected
- DisabledText (string) – The text to display when control disabled
- TextColour (r, g, b, a) – Normal colour of text
- SelectedTextColour (r, g, b, a) – Colour of text when it is selected
- DisabledTextColour (r, g, b, a) – Colour of text when control is disabled
- AlignH (align, write-only) – Horizontal alignment of text (left, right, centre)
- AlignV (align, write-only) – Vertical alignment of text (top, bottom, middle)
- Wrap (boolean, write-only) – Text will be wrapped if true
- Rect (left, top, width, height) – Area that the text covers – Note that passing negative values for the width and height will cause the control to use proportional sizing
- TextSkew – (top, bottom, left, right) - Four parameter skewing, which allows the actor to be skewed in four different directions
- TextMargin (left, right, top, bottom) – The margin to leave around the text (Negative values will use proportional sizing)
- AutoHeight (boolean) – If set to true then this label will resize itself to match the height of the text that it contains
- TextUseParentOpacity (boolean) – If set to true then the labels text will scale its opacity by its background opacity
- ActorText (actor, read-only) – Returns the text actor that represents the text
- TextFilter (boolean) – When set to true rendered text will be filtered

Labels are useful as text buttons and general containers

Here's a quick XOML example:

```
<Label Name="Label1" Text="Hello" Size="300, 50" Font="font1" Background="Button2Brush"
      SelectedColour="255, 255, 255, 200" SelectedTextColour="255, 255, 128, 200"
      AlignH="centre" Tappable="true">
```

This XOML creates a label button called Label1 of size 300x50 units. The buttons text colour will be changed when the user taps on the button.

As well as all of the usual Icon actor properties, all of the above properties can be a target of animation and can be modified an queried.

16.4 TextBox

The TextBox UI component is a label derived control that allows the user to change the text contained within it. As well as all of the usual label attributes, a Text Box also supports the following attributes:

- Variable (variable) – Name of variable that will receive the input
- Prompt (string) – The prompt to display to the user when asked to enter text
- TypeHint (hint)– A hint to the system a to what type of data needs to be input (number, password, email, url)
- OnTextChanged (actions list, write-only) – Specifies an actions list that will be called when the text within the text box has been changed

TextBox's are useful for collecting text based data from the user.

Here's a quick XOML example:

```
<TextBox Name="EnterName" Size="260, 60" Text="I am Mr TextBox" Background="SmallPanelBrush"
      Font="trebuchet_12" AlignH="left" />
```

The UI_Textbox example that has been provided shows a data collection forum that uses 8 text boxes in a form to collect user data.

```
    <!-- Create variables to store the text we enter -->
    <Variable Name="FirstName" Type="string" />
    <Variable Name="Surname" Type="string" />
    <Variable Name="Age" Type="string" />
    <Variable Name="Address" Type="string" />
    <Variable Name="City" Type="string" />
    <Variable Name="ZipCode" Type="string" />
    <Variable Name="TelNo" Type="string" />
    <Variable Name="MobileNo" Type="string" />

    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" >

        <!-- Create a grid to arrange the controls -->
        <Grid Size="-100, -100">
            <RowDefinition Height="80" AlignV="middle" />
            <RowDefinition Height="80" AlignV="middle" />
            <RowDefinition Height="80" AlignV="middle" />
            <RowDefinition Height="140" AlignV="middle" />
            <RowDefinition Height="80" AlignV="middle" />
            <RowDefinition Height="80" AlignV="middle" />
            <RowDefinition Height="80" AlignV="middle" />
            <RowDefinition Height="80" AlignV="middle" />
            <ColumnDefinition Width="-30" />
            <ColumnDefinition Width="-70" />
            <Label Font="serif" Background="SmallPanelBrush" Size="-30, 76"
                    Text="First Name:" GridPos="0, 0" />
            <Label Font="serif" Background="SmallPanelBrush" Size="-30, 76"
                    Text="Surname:" GridPos="0, 1" />
            <Label Font="serif" Background="SmallPanelBrush" Size="-30, 76"
                    Text="Age:" GridPos="0, 2" />
            <Label Font="serif" Background="SmallPanelBrush" Size="-30, 136"
                    Text="Address:" GridPos="0, 3" />
            <Label Font="serif" Background="SmallPanelBrush" Size="-30, 76"
                    Text="City:" GridPos="0, 4" />
            <Label Font="serif" Background="SmallPanelBrush" Size="-30, 76"
                    Text="Zip Code:" GridPos="0, 5" />
            <Label Font="serif" Background="SmallPanelBrush" Size="-30, 76"
                    Text="Tel No:" GridPos="0, 6" />
            <Label Font="serif" Background="SmallPanelBrush" Size="-30, 76"
                    Text="Mobile No:" GridPos="0, 7" />
            <TextBox Font="serif" Background="SmallPanelBrush" Size="-65, 76"
                    Variable="FirstName" GridPos="1, 0" />
            <TextBox Font="serif" Background="SmallPanelBrush" Size="-65, 76"
                    Variable="Surname" GridPos="1, 1" />
            <TextBox Font="serif" Background="SmallPanelBrush" Size="-65, 76"
                    Variable="Age" GridPos="1, 2" />
            <TextBox Font="serif" Background="SmallPanelBrush" Size="-65, 136"
                    Variable="Address" GridPos="1, 3" />
            <TextBox Font="serif" Background="SmallPanelBrush" Size="-65, 76"
                    Variable="City" GridPos="1, 4" />
            <TextBox Font="serif" Background="SmallPanelBrush" Size="-65, 76"
                    Variable="ZipCode" GridPos="1, 5" />
            <TextBox Font="serif" Background="SmallPanelBrush" Size="-65, 76"
                    Variable="TelNo" GridPos="1, 6" />
            <TextBox Font="serif" Background="SmallPanelBrush" Size="-65, 76"
                    Variable="MobileNo" GridPos="1, 7" />
        </Grid>

    </Scene>
```

The XOML looks a bit long and intimidating but its actually quite simple and repetitive. Basically we use grid to arrange 8 labels and 8 text boxes into a tidy form. Note how each Textbox used a separate variable to place the data that is input.

16.5 Sliders

The slider UI component allows the user to select a value by sliding a thumb either up / down or left / right. Note that the slider only renders the thumb, to render a background to the slider it must be declared inside another UI component such as an Icon. As well as all of the usual Icon attributes, a Slider also supports the following attributes:

- Value (number) – Initial value of the slider
- ValueRange (lower, upper) – Lower an upper limits of the value
- SliderType (orientation) – Horizontal or vertical
- SliderSize (size) – Size of the slider
- OnValueChanged (actions, write-only) – Specifies an actions list that will be called when the slider changes value

Note that if you bind a variable to the Value property of a slider then the binding will be two way, which means changes to the slider will change the variable and changing the variable will change the sliders value.

Sliders are useful for allowing the user to select a value from a range of values.

The UI_Slider example has been provided to show how to use sliders. Lets take a look at the XOML for this project to see how it works:

```
<!-- Create variables to store the text we enter -->
<Variable Name="SliderVal" Type="float" Value="15" />

<!-- Create a scene -->
<Scene Name="Scene1" Current="true" >

    <!-- Create an Icon that contains a slider -->
    <Icon Background="SmallPanelBrush" Size="200, 50">
        <Slider Name="Slider1" Value="15" ValueRange="10, 20" SliderSize="-80"
         Background="Button1Brush" SelectedColour="255, 200, 200, 255" Size="48, 48"
         SliderType="horizontal" Binding="[Value]SliderVal" />
    </Icon>

    <!-- Create a label to show the sliders current value -->
    <Label Font="serif" Position="0, 80" Background="Button2Brush"
         Binding="[Text]SliderVal" />

</Scene>
```

Firstly we create a variable called SliderVal that can store the sliders value. We then create an Icon (used as the sliders thumb background) and then create the slider inside it. The sliders range is 10 to 20 with the initial value set to 15. The range which the slider can be moved is set to 80% of the parent icons size and the thumb size is set to 40x40. The slider type is set as horizontal and the value of the slider is bound to the SliderVal variable. Finally we create a label that displays the current value of the slider

16.6 Canvas

The Canvas control is a layout container element that allows other actors to be positioned anywhere on its canvas or docked to one of its edges. The Canvas control is good for representing content that has no particular formal layout. The contents of a canvas can be panned around and child content that overlaps the canvases boundary will be clipped.

The UI_Canvas example has been provided to show how to use sliders. Lets take a look at the XOML for this project to see how it works:

```
<!-- Create a scene -->
<Scene Name="Scene1" Current="true" >

    <!-- Create a canvas that contains a number of docked and none docked labels -->
    <Canvas Name="Canvas1" Position="0, 0" Size="-100, -100" Background="Button2Brush"
         ScrollRange="-500, -500, 1000, 1000" Bubbling="true">
        <Label Font="serif" Text="Dock Top" Docking="top" Background="Button1Brush"
                Size="100, 100" Margin="-20, -20, -20, -20" />
        <Label Font="serif" Text="Not docked" Background="Button1Brush" Size="100, 100"
                Margin="20, 20, 20, 20" AngularVelocity="10" />
        <Label Font="serif" Text="Not docked" Position="100, 100"
                Background="Button1Brush" Size="100, 100" Margin="20, 20, 20, 20"
                AngularVelocity="-10" />
        <Label Font="serif" Text="Dock Right" Docking="right" Background="Button1Brush"
                Size="100, 100" Margin="20, 20, 20, 20" />
        <Label Font="serif" Text="Dock Left" Docking="topleft" Background="Button1Brush"
                Size="100, 100" Margin="20, 20, 20, 20" />
        <Label Font="serif" Text="Dock BottomLeft" Docking="bottomleft"
                Background="Button1Brush" Size="100, 100" Margin="20, 20, 20, 20" />
        <Label Font="serif" Text="Dock TopRight" Docking="topright"
                Background="Button1Brush" Size="100, 100" Margin="20, 20, 20, 20" />
        <Label Font="serif" Text="Dock BottomRight" Docking="bottomright"
                Background="Button1Brush" Size="100, 100" Margin="20, 20, 20, 20" />
    </Canvas>

</Scene>
```

The above example creates a canvas that can be scrolled by adding the ScrollRange property which specifies the range the canvas is allows to scroll. The Bubbling property is added to allow the canvas to receive input events from its children. If this was not present then dragging over a label would not scroll the canvas as the drag event will be eaten up by the label. A number of Icons are added to the canvas, most of which dock to the edges of the canvas using a margin to push them away from the edges, Two other actors are added that are not docked. Note that the actors that are not docked will scroll with the canvas whereas the docked actors stay in place.

16.7 StackPanel

The StackPanel is another type of layout container that stacks its content either vertically or horizontally. StackPanel's are incredibly useful as they allow you to stack child actors (content) either on top of each other or side by side without having to worry about their positions and such. As well as all of the usual Icon attributes, a StackPanel also supports the following attributes:

- Orientation (orientation) – Horizontal or vertical stacking direction
- AlignH (align, write-only) – Decides horizontal alignment when stacking vertically (left, right, centre)
- AlignV (align, write-only) – Decides vertical alignment when stacking horizontally (top, bottom, middle)

The UI_StackPanel example has been provided to show how to use Stack Panels. Lets take a look at the XOML for this project to see how it works:

```
<!-- Create a scene -->
<Scene Name="Scene1" Current="true" >

    <!-- Create a vertical stack panel with a collection of buttons -->
    <StackPanel Background="PanelBrush" Orientation="vertical" Position="0, -200"
         SizeToContent="xy">
        <Label Font="serif" Text="Item 1" Background="Button1Brush"
                BackgroundColour="0, 120, 255, 255" Margin="10, 10, 10, 0" />
        <Label Font="serif" Text="Item 2" Background="Button1Brush"
                BackgroundColour="0, 120, 255, 255" />
        <Label Font="serif" Text="Item 3" Background="Button1Brush"
                BackgroundColour="0, 120, 255, 255" />
        <Label Font="serif" Text="Item 4" Background="Button1Brush"
                BackgroundColour="0, 120, 255, 255" />
        <Label Font="serif" Text="Item 5" Background="Button1Brush"
                BackgroundColour="0, 120, 255, 255" Margin="0, 0, 0, 10" />
    </StackPanel>

    <!-- Create an horizontal stack panel with a collection of buttons -->
    <StackPanel Background="PanelBrush" Orientation="horizontal"
         Position="0, 200" SizeToContent="xy">
        <Label Font="serif" Text="Item 1" Size="-32, 50" Background="Button1Brush"
                BackgroundColour="0, 120, 255, 255" Margin="10, 0, 10, 0" />
        <Label Font="serif" Text="Item 2" Size="-32, 50" Background="Button1Brush"
                BackgroundColour="0, 120, 255, 255" />
        <Label Font="serif" Text="Item 3" Size="-32, 50" Background="Button1Brush"
                BackgroundColour="0, 120, 255, 255" Margin="0, 10, 0, 10" />
    </StackPanel>
</Scene>
```

In this example we create two StackPanels. The first has a vertical orientation which stacks 5 buttons on top of each other. The second StackPanel has an horizontal orientation which stacks 3 buttons side by side. In this example we also apply the

SizeToContent property to the stack panels which causes the stack panel to shrink or grow its size to fit the area that the content uses. We supply a few Margin properties to put a little space around the content so the Stack Panels border can be seen.

You can add new actors to this panel by setting their LinkedTo property to the stack panel. You can also remove actors by setting their LinkedTo property to no actor.

16.8 WrapPanel

The WrapPanel is a special case implementation of a StackPanel. Content that overlaps the end of the visible area will be moved into the next available row or column depending on the stacking direction. For example, a horizontal WrapPanel that has 8 elements, but can only fit the first 5 into the available space, will place the additional elements beneath the first 5 elements, much like text goes across and down the page of a book. The WrapPanel will clip content that is outside its boundary and if it runs out of space then it will allow the content to be scrolled.

As well as all of the usual Icon attributes, a WrapPanel also supports the following attributes:

- Orientation (orientation) – Horizontal or vertical stacking direction

The UI_WrapPanel example has been provided to show how to use Wrap Panels. Lets take a look at the XOML for this project to see how it works:

```
<Style Name="Button1">
    <Set Property="Font" Value="serif" />
    <Set Property="Background" Value="Button1Brush" />
    <Set Property="BackgroundColour" Value="0, 120, 255, 255" />
    <Set Property="Margin" Value="10, 10, 10, 10" />
</Style>

<!-- Create a scene -->
<Scene Name="Scene1" Current="true" >

    <!-- Create a horizontal stack panel with a collection of buttons -->
    <WrapPanel Background="PanelBrush" Orientation="horizontal" Size="-100, -100" >
        <Label Text="Item 1" Size="-25, 50" Style="Button1" />
        <Label Text="Item 2" Size="-25, 50" Style="Button1" />
        <Label Text="Item 3" Size="-25, 50" Style="Button1" />
        <Label Text="Item 4" Size="-35, 50" Style="Button1" />
        <Label Text="Item 5" Size="-25, 50" Style="Button1" />
        <Label Text="Item 6" Size="-25, 50" Style="Button1" />
        <Label Text="Item 7" Size="-45, 50" Style="Button1" />
        <Label Text="Item 8" Size="-25, 50" Style="Button1" />
        <Label Text="Item 9" Size="-55, 50" Style="Button1" />
        <Label Text="Item 10" Size="-25, 50" Style="Button1" />
```

```
            <Label Text="Item 11" Size="-25, 50" Style="Button1" />
            <Label Text="Item 12" Size="-25, 50" Style="Button1" />
        </WrapPanel>

    </Scene>
```

We begin this example by creating a style to save having to define the same common parameters for each label. Next we create an horizontal Wrap Panel that contains 12 different width labels. Notice that when a label does not fit horizontally it is pushed down to the start of the next line. The Wrap Panel is great for organising data in a page format.

You can add new actors to this panel by setting their LinkedTo property to the wrap panel. You can also remove actors by setting their LinkedTo property to no actor.

16.9 ListBox

A ListBox is much like a StackPanel in that content can be stacked within it either horizontally or vertically. The main different between StackPanels and ListBoxes include:

- List Box will clip content that overlaps its boundaries
- List Box allows scrolling of its contents
- List Box allows selection of its contents using single and multi-select
- Data can be bound to a List Box
- List Box items can automatically be generated from a template

A List Box is generally used to display a list of data that allows selection of one or more items.

A List Box can be auto generated from a variable array, item template and target item type combination or populated in the usual way using child actors

As well as all of the usual Icon attributes, a ListBox also supports the following attributes:

- Orientation (orientation) – Horizontal or vertical stacking direction (vertical is default)
- AlignH (align, write-only) – Decides horizontal alignment when stacking vertically (left, right, centre)
- AlignV (align, write-only) – Decides vertical alignment when stacking horizontally (top, bottom, middle)
- ItemsTargetType (string, write-only) – The type of property that the bound data should target (for example Brush for as list of Icons and Text for a list of Labels)
- ItemsData (variable) – An array variable that will be used to determine the content of the generated list box items. Each element in the array will create a list box item
- ItemsTemplate (template) – A template that defines the type of list box items that are generated from the array
- MultiSelect (boolean) – If true then the user will be able to select multiple items
- OnSelectionChanged (actions list, write-only) - Specifies an actions list that will be called when the user changes a selection
- Selection (variable, read-only) – Sets a variable that will be bound to the

current selected index. This variable can be used to gain access to the currently selected item index
- SelectedIndex (number) – Sets the currently selected index (zero based indexing)
- ItemCount (number, read-only) – Returns the total number of items in the stack
- CaretColourOffset (r, g, b, a) – Colour that is added onto the item that is currently marked as the caret
- Caret Index (number) – Sets the index of the caret item

The simple example below shows how to create a basic list box with no bound data:

```
<!-- Create a scene -->
<Scene Name="Scene1" Current="true" >

    <!-- Create a basic list box -->
    <ListBox Name="Menu" Size="-100, -100" Background="PanelBrush" AlignH="centre"
         ClipMargin="10, 10, 10, 10" SelectedIndex="1">
        <Label Background="Button1Brush" Size="-80, -20"
         SelectedColour="128, 255, 128, 255" Text="Option 1" Font="serif" />
        <Label Background="Button2Brush" Size="-90, -20"
         SelectedColour="128, 255, 200, 255" Text="Option 2" Font="serif" />
        <Label Background="Button1Brush" Size="-90, -20"
         SelectedColour="128, 255, 200, 255" Text="Option 3" Font="serif" />
        <Label Background="Button2Brush" Size="-90, -20"
         SelectedColour="128, 128, 200, 255" Text="Option 4" Font="serif" />
        <Label Background="Button1Brush" Size="-90, -20"
         SelectedColour="128, 255, 200, 255" Text="Option 5" Font="serif" />
        <Label Background="Button2Brush" Size="-90, -20"
         SelectedColour="128, 255, 128, 255" Text="Option 6" Font="serif" />
        <Label Background="Button1Brush" Size="-90, -20"
         SelectedColour="128, 128, 200, 255" Text="Option 7" Font="serif" />
    </ListBox>

</Scene>
```

In the above example we create a vertical list box that contains seven label controls. We change the selected background colour to different colours to show a little variation. Note that the List Box sets the SelectedIndex to 1 (indices start from 0) which causes the second label to be selected by default.

The above example creates List Box from static labels that you knew about before hand. Its often useful to generate a List Box from a set of data which you do not know about up front.

The UI ListBox (auto generated) example that has been provided shows how to create a more complex List Box that shows how to auto generate the list box from a template and some data. Lets take a look at the XOML for this example to see how it works:

```
    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" >

        <!-- ListBoxItems - Defines a list of items that will be bound to the list box -->
        <Variable Name="ListBoxItems" Type="arraystring" Size="10" Value="Item 1, Item 2,
Item 3, Item 4, Item 5, Item 6, Item 7, Item 8, Item 9, Item 10" />

        <!-- Create a template that will be used to generate the list box actors -->
        <Template Name="ListBoxItemTemp">
            <Label Name="ListItemA$index$" Size="-90, 70" Background="SmallPanelBrush"
             BackgroundColour="210, 210, 210, 255" SelectedColour="80, 80, 255, 255"
             Font="serif" Margin="20, 20, 20, 20" SelectType="toggle" Selected="false" />
        </Template>

        <!-- Create the list box -->
        <ListBox Name="ListBox" Size="-100, -100" Background="PanelBrush" MultiSelect="true"
             ItemsData="ListBoxItems" ItemsTargetType="text"
             ItemsTemplate="ListBoxItemTemp" ClipMargin="10, 10, 10, 10" />

    </Scene>
```

Note that the example that accompanies AppEasy has changed somewhat in 1.4.6, but this example has been left here as it easier to follow.

We begin this example by creating an array called ListBoxItems that contains some data that we will show in our List Box. Next e create a template that represents the type of actor that will be used to display each of our List Box items. Within this template we define a named label, although this is not required, we just include this as an example to show how the List Box control uses an internal template parameter called $index$. This parameter represents the current index of the item being generated as the List Box internally generates its List Box items. Lastly we declare the actual List Box control passing it the following:

- ItemsData – This is the ListBoxItems array we previously created
- ItemsTargetType – This is the type of property of the label within the template that we want to update with data from the array. In this case we want the “text” property of the label to be updated from the array data
- ItemsTemplate – This is the template that should be used to generate each List Box item

You can add new actors to the list box by setting their LinkedTo property to the list box. You can also remove actors by setting their LinkedTo property to no actor.

16.10 Grid

The Grid layout container can be thought of as a two dimensional ListBox, Content is organised into rows and columns. A Grid is generally used to display a collection of related two dimensional data that allows selection of one or more items.

A Grid can be auto generated from a variable array, item template and target item type on a per grid or per column basis or populated in the usual way using child elements

As well as all of the usual Icon attributes, a Grid also supports the following attributes:

- AlignH (align) – Decides horizontal alignment of data in columns (left, right, centre)
- AlignV (align) – Decides vertical alignment of data in rows (top, bottom, middle)
- ItemsTargetType (string, write-only) – The type of property that the bound data should target (for example Brush for a list of Icons and Text for a list of Labels)
- ItemsData (variable) – An array variable that will be used to determine the content of the generated grid cell items. Each element in the array will create a grid cell item
- ItemsTemplate (template) – A template that defines the type of grid items that are generated from the array.
- MultiSelect (boolean) – If true then the user will be able to select multiple items
- OnSelectionChanged (actions, write-only) - Specifies an actions list that will be called when the user changes a selection
- Selection (variable, read-only) – Sets a variable that will be bound to the current selected index. This variable can be used to gain access to the currently selected item index
- SelectedIndex (number) – Sets the currently selected index (zero based indexing)
- ColumnCount (number, read-only) – Returns the number of columns in the grid
- RowCount (number, read-only) – Returns the number of rows in the grid
- ShowColumn (column, boolean) – Show or hide a row (0 based index)
- ShowRow (row, boolean) – Show or hide a row (0 based index)

When defining a grid in XOML or in code you firstly need to add rows and columns. In XOML this can be done by adding row and column definitions like this:

```
<Grid ........ >
    <RowDefinition Height="100" />
    <RowDefinition Height="100" />
     <ColumnDefinition Width="-33" />
     <ColumnDefinition Width="-33" />
```

In this example we tell the system that the Grid should have two rows and two columns. We define the row heights as 100 units and the column widths as 33% of the size of the container. The UI_Grid example has been provided to show how to create a normal grid from actors. Lets take a look at the XOML for this example to see how it works:

```
<Style Name="Button1">
     <Set Property="Font" Value="serif" />
     <Set Property="Background" Value="Button1Brush" />
     <Set Property="BackgroundColour" Value="0, 120, 255, 255" />
     <Set Property="Margin" Value="10, 10, 10, 10" />
     <Set Property="SelectType" Value="toggle" />
     <Set Property="Size" Value="-25, 50" />
 </Style>

 <!-- Create a scene -->
 <Scene Name="Scene1" Current="true">

     <!-- Create a grid -->
     <Grid Name="ItemsGrid" Size="-100, -100" Background="PanelBrush"
          BackgroundColour="255, 255, 255, 255" Tappable="true"
          ClipMargin="10, 10, 10, 10" SelectedIndex="5">

         <!-- Create grid rows -->
         <RowDefinition Height="100" />
         <RowDefinition Height="100" />
         <RowDefinition Height="100" />
         <RowDefinition Height="100" />
         <RowDefinition Height="100" />
         <RowDefinition Height="100" />
         <!-- Create grid columns -->
         <ColumnDefinition Width="-33" />
         <ColumnDefinition Width="-33" />
         <ColumnDefinition Width="-33" />

         <!-- Fill grid cells -->
         <Label Style="Button1" Text="Cell 0" GridPos="0, 0" />
         <Label Style="Button1" Text="Cell 1" GridPos="1, 0" />
         <Label Style="Button1" Text="Cell 2" GridPos="2, 0" />
         <Label Style="Button1" Text="Cell 3" GridPos="0, 1" />
         <Label Style="Button1" Text="Cell 4" GridPos="1, 1" />
         <Label Style="Button1" Text="Cell 5" GridPos="2, 1" />
         <Label Style="Button1" Text="Cell 6" GridPos="0, 2" />
         <Label Style="Button1" Text="Cell 7" GridPos="1, 2" />
         <Label Style="Button1" Text="Cell 8" GridPos="2, 2" />
         <Label Style="Button1" Text="Cell 9" GridPos="0, 3" />
         <Label Style="Button1" Text="Cell 10" GridPos="1, 3" />
```

```
            <Label Style="Button1" Text="Cell 11" GridPos="2, 3" />
            <Label Style="Button1" Text="Cell 12" GridPos="0, 4" />
            <Label Style="Button1" Text="Cell 13" GridPos="1, 4" />
            <Label Style="Button1" Text="Cell 14" GridPos="2, 4" />
            <Label Style="Button1" Text="Cell 15" GridPos="0, 5" />
            <Label Style="Button1" Text="Cell 16" GridPos="1, 5" />
            <Label Style="Button1" Text="Cell 17" GridPos="2, 5" />
        </Grid>

    </Scene>
```

In this Grid example we create a grid that has 6 rows and 3 columns. We define each rows as having 100 units height and each column as 33% of the grids size. We then add 18 labels to the grid specifying which column and row each label should be placed in using the GridPos=”column, row” property of the label. Note that it is possible to add more than one element to the same grid cell.

Note that if grid row and column definition sizes are missing then the available space will be split between those rows and columns that did not specify a height or width.

Row definitions have the following attributes:

- Name (string) – Name of the row
- Height (number) – Height of the row, if negative then proportional sizing will be used
- AlignV (align) – Determines how cell actors will be vertically aligned (top, bottom or middle)
- Visible (boolean) – Visible state of row

Column definitions have the following attributes:

- Name (string) – Name of the column
- Width (number) – Width of the column, if negative then proportional sizing will be used
- AlignH (align) – Determines how cell actors will be horizontally aligned (left, right or centre)
- Visible (boolean) – Visible state of column
- ItemsTargetType (string) – The type of property that the bound data should target (for example Brush for Icons and Text for a Labels)
- ItemsData (variable) – An array variable that will be used to determine the content of the generated column cell items. Each element in the array will create a grid cell item
- ItemsTemplate (template) – A template that defines the type of grid items that are generated from the arrays. If you do not specify the template and the grid

has an ItemsTemplate defined then generated grid columns will use the grids ItemsTemplate as a default

Notice that data can be bound to columns but not rows, this is because XOML does not currently support mixed arrays at this time.

Now lets take a look at a more complex example that involves binding a number of arrays to the separate columns of a grid. UI Grid (auto generated) has been provided as an example showing how to generate a grid from data instead of specifying the separate cell actors as we did the previous example. Lets take a look the XOML for this example:

```
    <!-- Create a grid label animation -->
    <Animation Name="GridButtonBreath" Duration="1.0" Type="vec2">
        <Frame Value="1.1, 1.1" Time="0" Easing="quadin" />
        <Frame Value="0.5, 0.5" Time="0.5" Easing="quadin" />
        <Frame Value="1.1, 1.1" Time="1.0" Easing="quadin" />
    </Animation>

    <!-- Create variable to accept the current grid cell selection -->
    <Variable Name="grid_selection" Type="string" Value="0"/>

    <!-- Create a template that will be used to generate the animating label grid cell
actors -->
    <Template Name="GridItemTemp">
        <Label Background="SmallPanelBrush" BackgroundColour="200, 200, 200, 255"
             SelectedColour="200, 255, 200, 255" Font="serif" GridPos="$gridpos$"
             OnTapped="GridOnTapped" SelectType="toggle" Selected="false">
            <Timeline Name="GridItemAnim" AutoPlay="true">
                <Animation Anim="GridButtonBreath" Repeat="0" Target="Scale" />
            </Timeline>
            <Actions Name="GridOnTapped">
                <Action Method="SetTimeline" Param1="GridItemAnim" />
            </Actions>
        </Label>
    </Template>

    <!-- Create a template that will be used to generate the toggle button grid cell actors
-->
    <Template Name="GridItemTemp2">
        <Label Background="Button1Brush" BackgroundColour="200, 200, 200, 255"
             SelectedColour="200, 255, 200, 255" Font="serif" GridPos="$gridpos$"
             SelectType="toggle" Selected="false" Binding="[Text]grid_selection" />
    </Template>

    <!-- Create data that will be bound to grid columns -->
    <Variable Name="GridItems1" Type="arraystring" Size="10" Value="Button1Brush,
Button2Brush, Button2Brush, Button1Brush, Button2Brush, Button1Brush, Button2Brush,
Button1Brush, Button2Brush, Button1Brush" />
    <Variable Name="GridItems2" Type="arraystring" Size="10" Value="B1, B2, B3, B4, B5, B6,
B7, B8, B9, B10" />
    <Variable Name="GridItems3" Type="arraystring" Size="10" Value="C1, C2, C3, C4, C5, C6,
C7, C8, C9, C10" />
    <Variable Name="GridItems4" Type="arraystring" Size="10" Value="D1, D2, D3, D4, D5, D6,
D7, D8, D9, D10" />

    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" >

        <!-- Generate the grid -->
        <Grid Name="ItemsGrid" Size="-100, -100" Background="PanelBrush"
              BackgroundColour="255, 255, 255, 255" Tappable="true"
              ClipMargin="10, 10, 10, 10" ItemsTemplate="GridItemTemp"
              MultiSelect="false" Selection="grid_selection" SelectedIndex="5"
              UseParentOpacity="false">
            <RowDefinition AlignV="middle" Height="100" />
            <RowDefinition AlignV="middle" Height="100" />
            <RowDefinition AlignV="middle" Height="100" />
            <RowDefinition AlignV="middle" Height="100" />
            <RowDefinition AlignV="middle" Height="100" />
            <RowDefinition AlignV="middle" Height="100" />
            <RowDefinition AlignV="middle" Height="100" />
            <ColumnDefinition AlignH="centre" Width="300" temsData="GridItems1"
                    ItemsTemplate="GridItemTemp2" ItemsTargetType="background" />
            <ColumnDefinition AlignH="centre" Width="300" ItemsData="GridItems2"
                    ItemsTargetType="text" />
            <ColumnDefinition AlignH="centre" Width="300"
                    ItemsData="GridItems3" ItemsTargetType="text" />
            <ColumnDefinition AlignH="centre" Width="300" ItemsData="GridItems4"
                    ItemsTargetType="text" />
        </Grid>

    </Scene>
```

Note that the example that accompanies AppEasy has changed somewhat in 1.4.6, but this example has been left here as it easier to follow.

In the above example we create a grid with 4 columns and 7 rows. We bind data to the first column that represents a bunch of buttons. We then bind animating text labels to the other 3 columns.

To accomplish this auto generation of the grid we create two templates that will be used to instantiate the grid cell UI components. The first template is(GridItemTemp) is used to create the animating labels and the second is used to create the buttons in the first column.

Next we create 4 arrays, each one is bound to a separate column when we generate the grid. The first array contains brush names whilst the remaining 3 contain simple text names.

Finally we generate the actual grid. Note that we bind the template for the animating buttons to the actual grid and not the columns. When defining templates for a grid, we can specify a grid wide template that will be used by default if columns do not have their own template. This allows you to customise which columns you like. If you take a look at the column definitions you will notice that only the first column has its own item template defined.

16.11 Image View

The ImageView UI component is an image navigation control that allows the user to pan and pinch zoom around a large image. Note that you can also add other UI and actors to the ImageView and they will also be panned and zoomed along with the image.

As well as all of the usual Icon attributes, an ImageView also supports the following attributes:

- Area (left, top, width, height) – The area that the image should fit into, can use proportional sizing
- ImageBrush (Brush) – The image to pan / zoom
- Zoom (number) – The initial zoom value
- MinZoom (number) – The minimum amount of zoom allowed
- MaxZoom (number) – The maximum amount of zoom allowed

The UI_ImageView example has been provided to show how to create an ImageView that contains an image a label that can be pinch zoomed and panned. Lets take a look at the XOML for this example to see how it works:

```
<!-- Create image and brush for the test image -->
<Image Name="test_image" Location="test_image.jpg" Preload="true" Format="RGBA_5551"
         Filter="true" />
<Brush Name="test_image" Image="test_image" SrcRect="0, 0, 1024, 768" />

<!-- Create a scene -->
<Scene Name="Scene1" Current="true" Batch="false">

    <!-- Create a variable to ohld the scale value -->
    <Variable Name="image_scale" Type="float" Value="1" />

    <!-- Create an Image View that allows pinch zoom / pan of an image with controls -->
    <ImageView Name="ImageView1" Background="Button2Brush" Size="-100, -100"
         ImageBrush="test_image" Area="800, 480" Zoom="1.0" MinZoom="0.5" MaxZoom="4">

        <Label Font="serif" Size="-30, -20" Background="Button1Brush"
         SelectedBackground="Button2Brush" SelectedColour="80, 80, 255, 255"
         Tappable="true" Binding="[Text]image_scale" />

    </ImageView>

</Scene>
```

In the above example we begin by creating an image and a brush that will be used as our pinch zoom / pan image. Next we create the ImageView that allows us to pinch zoom and pan the image and the label button that we have also created.

16.12 Text View

The TextView behaves in much the same way as the ImageView control, but instead of an Icon the base UI component is label that contains text. This control enables pinch zoom / pan navigation of large areas of text.

As well as all of the usual Icon attributes, a TextView also supports the following attributes:

- Zoom (number) – The initial zoom value
- MinZoom (number) – The minimum amount of zoom allowed
- MaxZoom (number) – The maximum amount of zoom allowed

The UI_TextView example has been provided to show how to create a TextView that can be pinch zoomed and panned. Lets take a look at the XOML for this example to see how it works:

```
<!-- Create a scene -->
<Scene Name="Scene1" Current="true" Batch="false" CanvasFit="none">

    <!-- Create a Text View that allows pinch zoom / pan of text -->
    <TextView Name="TextView1" Size="-100, -100" Rect="0, 0, -100, -200" Font="serif"
         Text="Text shortened" Background="Button2Brush" Zoom="1.0" MinZoom="0.5"
         MaxZoom="4" />

</Scene>
```

In the above example we display a large selection of text (shortened in this example for readability)that the user can pan and punch zoom. Size in this case defines the size of the texts background whilst the Rect defines the area that the text should fit into.

16.13 Web View

The WebView control is a control that allows you to display local or remote web pages either full screen or contained in a rectangular area of the screen. Note that the WebView will be drawn on top of all other controls so depth sorting is not possible.

As well as all of the usual Icon attributes, a WebView also supports the following attributes:

- URI (string, write-only) – Address of the web site or html file to view. Changing the URI property from actions, command and scripts will cause the web veiw to navigate to the new URI
- CurrentURI (string, read-only) – Address of the web page that is currently in view. This can potentially differ, especially if the requested page is redirected
- Modal (boolean, read-only) – If true then a full screen modal web view will be displayed
- Html (string, write-only) – Writes the html directly to the page and displays it (instead of navigating to the URI). You can also set this property to create dynamic web content within the web view.
- Javascript (string) – String of Javascript to pass to the page. When read from Lua will contain the Javascript that was passed back from the web view
- Transparent (boolean) – If set to true then the web pages background will be made transparent. This property cannot be read or written to
- OnError (actions list) – Defines an actions list that is called if the WebView is unavailable. Cannot be read or written to
- OnPageLoaded (actions list, write-only) – Defines an actions list that is called when a page is loaded. The WebView CurrentURI property holds the target web page
- OnPageLoading (actions list, write-only) – Defines an actions list that is called when a page starts loading. The WebView CurrentURI property holds the target web page
- OnPageError (actions list, write-only) – Defines an actions list that is called when a page load encounters an error. The WebView CurrentURI property holds the target web page
- OnJavascript (actions list, write-only) – Defines an actions list that is called when the web page calls back to XOML from JavaScript via s3e.exec(string_to_pass_back)

The UI_WebView example has been provided to show how to create a WebView. Lets take a look at the XOML for this example to see how it works:

```
    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" Batch="false" CanvasFit="none">

        <!-- Create a Text View that allows pinch zoom / pan of text -->
        <WebView Name="WebView1" Size="-100, -100" URI="http://www.pocketeers.co.uk"
Modal="false" Transparent="false" />

    </Scene>
```

In the above example we create a WebView that shows a web site and fills the whole screen.

An additional web view example called DynamicWebWiew is supplied that shows how to write html to the web view dynamically as well as navigate between web pages / sites

WebView coordinates and sizes are specified in raw device coordinates and not virtual canvas coordinates. Sizes and positions can be percentage based and the web view can be docked as well as utilise margins. The WebView is not currently supported on Windows desktop

16.14 Tab Bars

The TabBar UI control is a control that allows navigation of a number of different views (container UI components such as the Canvas). A bar is displayed either at the top, bottom, left or right of the main view that contains buttons which when tapped will change the current view. When the view is changed the outgoing and incoming view can be animated using a set of pre-set animations or using your own custom animations. Each view can have its own incoming and outgoing animations defined.

As well as all of the usual Canvas attributes, a TabBar also supports the following attributes:

- OnViewChanged (actions list, write only) – Defines an actions list that will be called when the user changes the view
- AutoHide (boolean) – If set to true then the outgoing view will automatically be hidden
- AddTab (actor-name) – Adds a new tab actor to the tab view
- RemoveTab (actor-name) – Removes an existing tab actor from the tab view

Tabs are derived from Label and have the following properties:

- View (actor) – The view that the tab is associated with

Defining a TabBar in XOML is much like defining a Grid. To create the tabs you define them inside a TabBar like shown below:

```
<TabBar Name="TabBar1" Background="PanelBrush" AutoHide="true">
    <Tabs Background="SmallPanelBrush" Size="-95, -10" Orientation="horizontal">
        <Tab Name="Tab1" Text="Tab 1" Font="trebuchet_12" Background="Button2Brush" />
        <Tab Name="Tab2" Text="Tab 2" Font="trebuchet_12" Background="Button2Brush" />
        <Tab Name="Tab3" Text="Tab 3" Font="trebuchet_12" Background="Button2Brush" />
        <Tab Name="Tab4" Text="Tab 4" Font="trebuchet_12" Background="Button2Brush" />
    </Tabs>
</TabBar>
```

The actual views that the tabs will activate are then defined after the tabs. Note that views are mapped to tabs in the order in which they are declared.

The UI_TabBar example has been provided to show how to create a TabBar that can switch between 4 different views. Lets take a look at the XOML for this example to see how it works:

```
    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" Batch="false" CanvasFit="none">

        <!-- Create a tab bar with 4 tabs that displays 4 different views -->
        <TabBar Name="TabBar1" Size="-100, -100" Background="PanelBrush" AutoHide="true"
             Draggable="true">
            <!-- Create horizontal tabs -->
            <Tabs Background="SmallPanelBrush" Size="-95, -10" Margin="0, 0, -2, -2"
             Orientation="horizontal">
               <Tab Name="Tab1" Text="Tab 1" Font="serif" Background="Button2Brush"
                   Size="-22, -90" Margin="-2, -2, 0, 0"
                   SelectedColour="200, 200, 200, 255" SelectType="sticky" Selected="true"
                   />
               <Tab Name="Tab2" Text="Tab 2" Font="serif" Background="Button2Brush"
                   Size="-22, -90" Margin="-2, -2, 0, 0"
                   SelectedColour="200, 200, 200, 255" SelectType="sticky" />
               <Tab Name="Tab3" Text="Tab 3" Font="serif" Background="Button2Brush"
                   Size="-22, -90" Margin="-2, -2, 0, 0"
                   SelectedColour="200, 200, 200, 255" SelectType="sticky" />
               <Tab Name="Tab4" Text="Tab 4" Font="serif" Background="Button2Brush"
                   Size="-22, -90" Margin="-2, -2, 0, 0"
                   SelectedColour="200, 200, 200, 255" SelectType="sticky" />
            </Tabs>

            <!-- Create 4 different views -->
            <Canvas Name="View1" Size="-95, -84" Background="PanelBrush"
             BackgroundColour="200, 255, 200, 255" ShowTimeline="PA_ScrollOnFromLeft"
             HideTimeline="PA_ScrollOffToRight">
               <Label Text="View 1" Font="serif" Rect="-50, -25, 100, 50"
                   Background="Button1Brush" SelectedColour="80, 80, 255, 255" />
            </Canvas>
            <Canvas Name="View2" Size="-95, -84" Background="PanelBrush"
             BackgroundColour="255, 80, 200, 255" ShowTimeline="PA_ScrollOnFromBottom"
             HideTimeline="PA_ScrollOffToTop">
               <Label Text="View 2" Font="serif" Rect="-50, -25, 100, 50"
                   Background="Button1Brush" SelectedColour="80, 80, 255, 255" />
            </Canvas>
            <Canvas Name="View3" Size="-95, -84" Background="PanelBrush"
             BackgroundColour="200, 255, 200, 255" ShowTimeline="PA_ScrollOnFromLeft"
             HideTimeline="PA_ScrollOffToRight">
               <Label Text="View 3" Font="serif" Rect="-50, -25, 100, 50"
                   Background="Button1Brush" SelectedColour="80, 80, 255, 255" />
            </Canvas>
            <Canvas Name="View4" Size="-95, -84" Background="PanelBrush"
             BackgroundColour="200, 255, 255, 255" ShowTimeline="PA_FadeOn"
             HideTimeline="PA_FadeOff">
               <ListBox Name="Menu" Size="-90, -90" Background="PanelBrush" AlignH="centre"
                   ClipMargin="10, 10, 10, 10" SelectedIndex="1">
                   <Label Background="Button2Brush" Size="-90, 0"
                   SelectedColour="128, 255, 200, 255" Text="Option 1" Font="serif" />
                   <Label Background="Button2Brush" Size="-90, 0"
                   SelectedColour="128, 255, 200, 255" Text="Option 2" Font="serif" />
                   <Label Background="Button2Brush" Size="-90, 0"
                   SelectedColour="128, 255, 200, 255" Text="Option 3" Font="serif" />
```

```
                    <Label Background="Button2Brush" Size="-90, 0"
                    SelectedColour="128, 255, 200, 255" Text="Option 4" Font="serif" />
                    <Label Background="Button2Brush" Size="-90, 0"
                    SelectedColour="128, 255, 200, 255" Text="Option 5" Font="serif" />
                    <Label Background="Button2Brush" Size="-90, 0"
                    SelectedColour="128, 255, 200, 255" Text="Option 6" Font="serif" />
                    <Label Background="Button2Brush" Size="-90, 0"
                    SelectedColour="128, 255, 200, 255" Text="Option 7" Font="serif" />
                    <Label Background="Button2Brush" Size="-90, 0"
                    SelectedColour="128, 255, 200, 255" Text="Option 8" Font="serif" />
                    <Label Background="Button2Brush" Size="-90, 0"
                    SelectedColour="128, 255, 200, 255" Text="Option 9" Font="serif" />
                    <Label Background="Button2Brush" Size="-90, 0"
                    SelectedColour="128, 255, 200, 255" Text="Option 10" Font="serif" />
                </ListBox>
            </Canvas>

        </TabBar>

    </Scene>
```

In the above example we firstly declare a tabs control that is oriented horizontally which contains our tab buttons. We then declare each of our tab buttons, making them of type sticky. A sticky button will retain its selected state once it has been selected. Note that the TabBar will take care of changing the selected state of the tab buttons.

Next we populate the tab bar with views View1 to View 4. You will notice that each of the views supports the ShowTimeline and HideTimeline events. The ImageView (View1) uses the PA_ScrollOnFromLeft and PA_ScrollOffToRight animations, which causes the view to scroll in from left tleft when being shown and scroll off to the right when being hidden. Note that we did not declare these animations ourselves as they are pre-defined animations that are set-up by XOML. However, you do not have to use these pre-defined animations but can instead place your own timelines in ShowTimeline and HideTimeline.

Note that the following pre-defined animations are currently available:

- PA_ScrollOnFromLeft
- PA_ScrollOffToLeft
- PA_ScrollOnFromRight
- PA_ScrollOffToRight
- PA_ScrollOnFromTop
- PA_ScrollOffToTop
- PA_ScrollOnFromBottom
- PA_ScrollOffToBottom
- PA_FadeOn
- PA_FadeOff

16.15 VideoOverlay

XOML supports the loading of video via the Video tag and the playback of video at any position and size via the VideoOverlay tag. Note that video will be displayed above all other elements on screen. The Video example has been provided to show how to use both Video and VideoOverlay tags. Lets take a look at the XOML from this example:

```
    <!-- Create a video resource -->
    <Video Name="Video1" Location="video1.mp4" Codec="MPEG4" />

    <!-- Create a scene -->
    <Scene Name="Scene1" Current="true" >

        <!-- Create a video verlay to show the video -->
        <VideoOverlay Name="Vid1" Video="Video1" Size="-20, 100" AutoPlay="true"
Volume="0.3" Repeat="1" AspectLock="x" />

    </Scene>
```

In this example we firstly create a video resource called Video1 from the video1.mp4 video file and specify the code that should be used to play it back, in this case the file is MPEG4. Next we create a scene containing a VideOverlay called Vid1 which plays back the Video1 video resource.

VideoOverlay supports the following properties:

- Video (string) – The name of the video resuorce
- AutoPlay (boolean) – if true then the video will automatically begin to play
- Repeat (number) – Number of times to repeat playback
- Volume (number) –The volume to play back the video (1.0 is default)
- Command (command, write-only) – Sends a command to the video (stop, play, pause, resume)
- IsPlaying (boolean, read-only) – Returns true if the video is currently playing
- Started (boolean, read-only) – Returns true if the video has started playing
- OnStart (actions list, write only) - Defines an actions list that will be called when video playback has started
- OnEnd (actions list, write only) - Defines an actions list that will be called when video playback ends
- OnPause (actions list, write only) - Defines an actions list that will be called

when the video is paused
- OnResume (actions list, write only) - Defines an actions list that will be called when video playback is resumed
- OnError (actions list, write only) - Defines an actions list that will be called when an error occurs playing the video

16.16 UIStyle

We have included a simple UIStyle.xml file along with ui.png image file a font that you can use for creating your own UI. The UIStyle XOML file contains some basic fonts and brushes that you can use to get UI up and running quickly. To style your own UI you should edit the ui image file and XOML style file to add your own UI. The original SVG and GIMP layout files have also been included in the UIStyle folder for reference. Feel free to use the current UI style or derivatives in your own products.

17.0 Physics

17.1 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. The physics system used by XOML is Box2D based developed by the clever guys over at http://box2d.org/. The latest in-depth Box2D documentation can be found at http://box2d.org/documentation/

Physics is not an easy subject, however XOML bends over backwards to help make the integration of physics into your apps and games as painless as possible.

XOML physics is split into 4 sections:

- The physics world – The worlds parameters such as gravity and size etc are defined by the Scene using the following cene properties:
  - Gravity (x, y) - Directional world gravity
  - WorldScale (x, y) - Box2D world scale
  - DoSleep (boolean) - If set to true then actors that utilise physics will be allowed to sleep when they are not moving / interacting
  - Physics (boolean) - Enables or disables physics processing in the scene
- Physics material and shape – All actors can be involved in a physics simulation as long as they have a physics material and shape defined in their properties:
  - Velocity (x, y) - Initial velocity of the actor
  - VelocityDamping (x, y) - The amount to dampen velocity each frame, values of less than 1.0 will slow the actor down over time, values of greater than 1.0 will speed the actor up over time.
  - AngularVelocity (number) - The rate at which the orientation of the actor changes in degrees per second
  - AngularVelocityDamping (number) - The amount of rotational velocity damping to apply each frame
  - WrapPosition (boolean) - If true then the actor will wrap at the edges of the canvas
  - Box2dMaterial (material) - Sets the physical material type used by the Box2D actor

    - Shape (shape) - Box2D fixture shape for that represents this actor during collisions
    - COM (x, y) - Centre of mass of Box2D body
    - Sensor (boolean) - Can be used to set the actor as a sensor
    - CollisionFlags (category, mask, group) - Box2D collision flags
- Joints - Its often useful to be able to connect physical objects together in some way to make them interact in realistic ways. The joints system enables you to connect together a multitude of actors in different configuratins to create all sorts of interesting objects
- Collision – Actors that have physical materials and shapes defined will automatically collide and niteract. However and more usefully you can respon to collisions between objects by adding the collision notification modifier (iw_ notifycollision). The details for this modifier as shown below:
    - Collision Notification Modifier - The iw_ notifycolision modifier when attached to an actor allows it to generate and respond to collision events between actors using OnCollisionStart() and OnCollisionEnd() event handlers (See ActorModifier example for an example showing how to use this modifier). This modifier accepts a number of parameters which include:
        - Param1 - An optional mask that can be used to mask collision with actors by their type. The value supplied will mask actors by type and only allow collision events to be called for those actors that pass the bit mask. For example actor 1 could have a mask of 3 and actor 2 a mask of 1. if the mask is set to 1 then both actors can collide, but if the mask was set to 3 then they could not.

Actors that are set up with a collision shape and physics material will be put under control of the scenes Box2D world controller.

Actors also support multiple fixtures using the Fixtures inner tag:

```
<Icon Position="0,0" Background="Land" Box2dMaterial="FixedHeavy">
      <Fixtures>
            <Fixture Shape="LandScapeBase" Box2dMaterial="FixedHeavy" 
CollisionFlags="1,1,1" COM="0,0"/>
            <Fixture Shape="LandScapePeak" Box2dMaterial="FixedHeavy" 
CollisionFlags="1,1,1" COM="0,0"/>
      </Fixtures>
</Icon>
```

In this example we create two separate fixtures each with their own shape, material, collision flags and centre of mass.

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.

I would like to refer you back to the ActorPhysics example that shows a basic example of actors acting under control of the physics system. To understand how this example works in depth please refer back to the Actor Physics section.

17.2 Box2dMaterial

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 Box2dMaterial tag and has the following properties:

- Name (string) – Name of the physics material
- Type (type) – Type of physics material (values can be static, dynamic and kinematic)
- Density (number) – The objects density (default is 1.0)
- Friction (number) – The coefficient of friction (default is 1.0)
- Restitution (number) - The coefficient of restitution / bounciness (default is 0.1)
- IsBullet (boolean) – If set to true, will force the object that this material is attached to to be treat as a high speed moving object (required more processing so use wisely) (default is false)
- FixedRotation (boolean) – If set to true then the object that this material attaches to will not be allowed to rotate (default is false)
- GravityScale (number) – This is the amount to scale the affect of gravity on objects that this material is attached to (default is 1.0)
- Tag (string) - Resource tag (used to group resources together)

Like any other resource a Box2dMaterial can be declared inside a scene in which case it will become local to the scene or declared outside a scene in which case it will become global and can be accessed by any actor from any scene

17.3 Shapes

XOML provides the ability to create shapes of a variety of types including:

- Box – A rectangular shape with a width and height
- Circle – A circular shape with a radius
- Polygon – An arbitrary shape that is built up out of 2D points

Shapes are generally used by the physics system to define the shape of actors that are under control of the physics system.

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" Points=”-100,-100,-100,-200,100,-100,100,100,-
100,100" />
```

Or the following form can be used:

```
    <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>
```

If a shape is not assigned to an actor but the actor has a physics material assigned to it then a default rectangular shape of the same size as the actor will be created and attached.

17.4 Joints

Joints are contraptions that enable you to connect together various actors that are under control of the physics system. Joints generally limit the movement of actors depending upon the type of joint and properties of the joint. XOML supports the following types of joints:

- Weld – A weld joint simply welds together two bodies
- Distance – A distance joint limits the distance of two bodies and attempts to keep them the same distance apart, damping can also be applied
- Revolute – A revolute joint forces two bodies to share a common anchor point. It has a single degree of freedom and the angle between the two bodies can be limited. In addition a motor can also be applied to the joint
- Prismatic – A prismatic joint limits movement between the two bodies by translation (rotation is prevented). The translational distance between the two joints can be limited. In addition a motor can also be applied to the joint
- Pulley Joint – A pulley joint can be used to create a pulley system between two bodies so that when one body rises the other will fall.
- Wheel Joint – A wheel joint restricts one body to the line on another body and can be used to create suspension springs. A motor can also be applied to the joint

Joints are declared inside the actor that represents Body A. Lets take a look at a short example:

```
<ActorImage Name="Crate1" ...... />
<ActorImage Name="Crate2" ...... >
    <Joints>
        <Joint type="distance" ActorB="Crate1" OffsetA="0, 50" OffsetB="0, -50"
            Frequency="10" Damping="0" SelfCollide="true" />
    </Joints>
</ActorImage>
```

This small piece of XOML creates a distance joint that connects Crate1 and Crate2 together the place which the joint is connected is offset by the amounts specified by OffsetA and OffsetB. This allows the joint to act on the edges of the crate instead of the centre.

Lets take a look at the properties that each type of joint supports:

Common to all joint types:

- Type (type) – Type of joint (distance, revolute, prismatic, pulley and wheel)
- OffsetA (x, y) – Anchor point on body A
- OffsetB (x, y) – Anchor point on body B
- ActorA (string) – The name of the primary actor that the joint is attached to. When joints are declared inside an actors hierarchy ActorA is assigned as the container actor
- ActorB (string) – The name of the other actor that the joint is attached to
- SelfCollide (boolean) – Determines if the two joined actors can collide with each other

Distance Joint:

- Length (number) – The max length between the two bodies – This is calculated if not supplied
- Frequency (number) – Oscillation frequency in Hz
- Damping (number) – Oscillation damping ratio

Revolute Joint:

- ReferenceAngle (number) – The initial angle between the two bodies – This is calculated if not supplied
- LimitJoint (boolean) – When set to true will limit joint rotation
- UpperLimit (number)– Upper angle limit in degrees
- LowerLimit (number)– Lower angle limit in degrees
- MotorEnabled (boolean) – When set to true the joint motor is enabled
- MotorSpeed (number) – Speed of the motor
- MaxMotorTorque (number) – Maximum torque that the motor will apply

Prismatic Joint:

- ReferenceAngle (number) – The initial angle between the two bodies – This is calculated if not supplied
- Axis (x, y) – Axis of movement
- LimitJoint (boolean) – When set to true will limit joint translation
- UpperLimit (number) – Upper translation limit
- LowerLimit (number) – Lower translation limit
- MotorEnabled (boolean) – When set to true the joint motor is enabled
- MotorSpeed (number) – Speed of the motor
- MaxMotorForce (number) – Maximum force that the motor will apply

Pulley Joint:

- GroundAnchorA (x, y) – Anchor point where pulley point for Body A is situated
- GroundAnchorB (x, y) – Anchor point where pulley point for Body B is situated

- LengthA (number) – Distance between BodyA and Ground Anchor A - This is calculated if not supplied
- LengthB (number) – Distance between BodyB and Ground Anchor B - This is calculated if not supplied
- Ratio (number) – The ratio of side A to side B

Wheel Joint:

- Axis (x, y) – Axis of movement
- MotorEnabled (boolean) – When set to true the joint motor is enabled
- MotorSpeed (number) – Speed of the motor
- MaxMotorTorque (number) – Maximum torque that the motor will apply
- Frequency (number) – Oscillation frequency in Hz (should be less than half the frame rate)
- Damping (number) – Oscillation damping ratio

Note that all coordinates are specified relative to the relevant body (local)

The Joints example has been provided to show how to use a variety of different joint types. We wont examine this example in close detail as the XOML is quite substantial.

17.5 Collision

XOML enables you to detect when one actor that is under control of the physics system collides with another. This functionality is not enabled by default so you must add the iw_notifycollision modifier to the actor in order for it to be able to handle the OnCollisionStart and OnCollisionEnd events as is shown in this example:

```
<!-- Create an actor that checks for collision -->
<Icon Name="Crate20" Type="5" Position="-400, 100" Background="Crate"
     Shape="CrateShape" Box2dMaterial="FixedHeavy" CollisionFlags="1, 1, 1"
     Collidable="true" OnTapped="PushMe2" Draggable="true"
     OnCollisionStart="CollisionStart" OnCollisionEnd="CollisionEnd" >
    <Modifiers>
        <Modifier Name="iw_notifycollision" Active="true" Param1="0" />
    </Modifiers>
    <Actions Name="CollisionStart">
        <Action Method="SetProperty" Param1="Colour" Param2="255, 255, 255, 80"/>
    </Actions>
    <Actions Name="CollisionEnd">
        <Action Method="SetProperty" Param1="Colour" Param2="255, 255, 255, 255"/>
    </Actions>
</Icon>
```

In the above example we create an actor that is placed under control of the physics system because it is assigned a shape and a physics material. Next we add the collision modifier which allows the actor to receive collision start and end events. We react to a collision by modifying the actors opacity to show that it has hit something then change it back to normal when the collision ends. This type of collision is quite simple because we do not know exactly what object you are colliding with. There are a number of ways in which you can take more control of collision between actors including:

- Collision Flags – You can limit what other actors your collision actor can collide with by using Collision Flags
- Actor Types – When you add the iw_notifycollision modifier to an actor you can specify a mask that can be used to mask collision with actors of specific types (types set by the Type property of the actor). Lets take a quick look at an example of this type of masking:

```
<!-- Create the floor (default actor type is 0) -->
<Label Position="0, 200" ........ />

<!-- Create an actor that checks for collision -->
<Icon Name="Crate1" Type="1" ......... >
    <Modifiers>
        <Modifier Name="iw_notifycollision" Active="true" Param1="1" />
    </Modifiers>
    <Actions Name="CollisionStart">
        <Action Method="SetProperty" Param1="Colour" Param2="255, 255, 255, 80"/>
```

```
            </Actions>
            <Actions Name="CollisionEnd">
                <Action Method="SetProperty" Param1="Colour" Param2="255, 255, 255, 255"/>
            </Actions>
        </Icon>

        <!-- Create another actor that checks for collision -->
        <Icon Name="Crate2" Type="1" ........ >
            <Modifiers>
                <Modifier Name="iw_notifycollision" Active="true" Param1="1" />
            </Modifiers>
            <Actions Name="CollisionStart">
                <Action Method="SetProperty" Param1="Colour" Param2="255, 255, 255, 80"/>
            </Actions>
            <Actions Name="CollisionEnd">
                <Action Method="SetProperty" Param1="Colour" Param2="255, 255, 255, 255"/>
            </Actions>
        </Icon>
```

In this example we create a floor actor and two crates that handle collision notification. Note that both crates share the same Type of 1 whereas the floors Type is set to 0 (by default). When the iw_notifycollision modifier is added to each create actor the parameter passed along is the value of 1, which allows the two creates to receive collision notification events when they collide.

- Programs and actions - Using a program or actions its possible to latch a number of collision events in an array then compare them
- Scripts – Its possible to query and act upon collisions from script

The SimpleCollision example has been provided that demonstrates how to mask collisions using the ActorType method.

Using XOML it is not currently possible to query and interact with the list od collidables that an actor starts or ends collision with. However, script provides access to this information. Below is a short Lua example showing how to interact with the collision contacts lists:

```
function CollisionStarted(_object)

       -- Get list of actors that just hit us
       local collisions_started = actor.getStartContacts(_actor)
       if (collisions_started == nil) then
              return 1;
       end

       -- Change colour of all actors that just hit us
       for key,collider in ipairs(collisions_started) do
              actor.set(collider, "Colour", "255, 255, 0, 255")
       end

       return 1;
end
```

17.6 Physics Timestep

The physics engine is usually updated at a different rate to the frame rate of the game to ensure the integrity of the simulation. We use the name “time step” to refer to how fast the engine is updated and we measure it in seconds. Each scene can have its own physics time step defined which determines how fast the physics system will update. This can be set using the scenes PhysicsTimestep property. The value passed to this property represents the amount of time that has passed since the last physics update took place. So a value of 1/30 (0.03333) would be ideal if your app runs at 30 frames per second. Passing a value of 0 will switch the physics engine update into variable time step mode which estimates the speed to update physics based on how long it took to run the last game frame. The default value for physics time step is 1/30.

You can also modify the value of the physics time step in real-time by changing the scenes time step property or by targeting the PhysicsTimestep property from an animation. This allows you to create special effects with time, such as slow time down or speed it up.

18.0 Scripts

18.1 Introduction

Whilst XOML is a powerful mark-up language it does have some limitations when it comes to defining complex logic for more complex apps and games. To alleviate this problem support for scripting languages has been added. At the time of writing only Lua is currently supported, although support for other languages such as Python, Angelscript and Javascript are also planned for future releases.

Lets take a look at a simple Lua script example:

```
function Scene_OnTick(_object)

	local pos = actor.get(_object, "PositionX")
	pos = pos + 1;
	if (pos > 200) then
		pos = -200;
	end
	actor.set(_object, "PositionX", pos)

end
```

This short example is a function that is called each time the scene updates itself (approximately 30 to 60 times per second). The example scrolls the scene slowly to the left.

18.2 Scene Scripts

In order to be used a script must first be declared using the Script tag. If you define a script inside a Scene then it will become local to the scene and cleaned up when the scene is cleaned up. If declared outside a scene then the script will become global and will remain in memory at all times.

The Script tag has a number of properties including:

- Name (string) - Name of this script resource
- Tag (string) - Resource tag (used to group resources together)
- Type (type) - Type of script (Currently only Lua is supported)
- Condition (variable) - A condition variable that must evaluate to true for this resource to be loaded (this is an optional feature and can be used to conditionally load resources based on certain conditions such as screen size or device type etc..)

Note that you do not declare the actual script when you declare the script tag, instead you later load the script file and write it to the script as the following example shows:

```
<!-- Define and load a Lua script -->
<Script Name="Script1" Type="lua" />
<File Name="File1" Location="Spawn.lua" FileType="lua" Script="Script1" />
```

In this example we declare a script called Script1 then we load Spawn.lua as a file and write it to the Script1 script using Script=”Script1” in the file definition. This style of script loading allows you to load multiple script files into the same script.

Note that for a scene to be able to run a script you need to tell the scene what type of script it will run by adding the ScriptEngine=”lua” property to the scene definition. The type of script engine to use for global scripts will be decided from the type of the first script that you load globally.

Scripts can be ran via actions using the CallScript action and via commands using call_script command. Global scripts can be ran via the CallGlobalScript action and call_global_script command. In addition short pieces of script can be ran inline using the “Inline” action and the “inline” command.

The SimpleScript example has been provided to show you how to load and run scripts in a scene. Lets take a look at the XOML for this example to see how it works:

```
<!-- Create a scene with physics enabled -->
<Scene Name="Scene1" Current="true" Physics="true" Gravity="0, 5" Camera="Camera1"
         DoSleep="false" ScriptEngine="lua" OnTick="TickActions">

    <!-- Define and load a Lua script -->
    <Script Name="Script1" Type="lua" />
    <File Name="File1" Location="Spawn.lua" FileType="lua" Preload="true"
         Script="Script1" />

    <!-- Create scene OnTick handler that calls a script function each scene update -->
    <Actions Name="TickActions">
        <Action Method="CallScript" Param1="Scene_OnTick" />
    </Actions>

    <!-- Create touch pan camera -->
    <Camera Name="Camera1" TouchPanX="true" TouchPanY="true" IgnoreActors="false" />

    <!-- Create Box2D materials -->
    <Box2dMaterial Name="Heavy" Type="dynamic" Density="2.0" Friction="0.8"
         Restitution="0.8" />
    <Box2dMaterial Name="FixedHeavy" Type="static" Density="1.0" Friction="0.8"
         Restitution="0.8" FixedRotation="true" GravityScale="0.5" />

    <!-- Create Box2D shapes -->
    <Shape Name="CrateShape" Type="box" Width="64" Height="64" />
    <Shape Name="Floor" Type="box" Width="2000" Height="100" />
    <Shape Name="LeftWall" Type="box" Width="100" Height="600" />
    <Shape Name="RightWall" Type="box" Width="100" Height="600" />

    <!-- Create the floor and walls -->
    <Label Position="0, 200" Font="serif" Background="Button1Brush"
         BackgroundColour="255, 80, 80, 255" Size="2000, 100" Text="Floor"
         Shape="Floor" Box2dMaterial="FixedHeavy" CollisionFlags="1, 1, 1" />
    <Icon Position="-1000, -100" Background="Button1Brush"
         BackgroundColour="255, 80, 80, 255" Size="100, 600" Shape="LeftWall"
         Box2dMaterial="FixedHeavy" CollisionFlags="1, 1, 1" />
    <Icon Position="1000, -100" Background="Button1Brush"
         ackgroundColour="255, 80, 80, 255" Size="100, 600" Shape="RightWall"
         Box2dMaterial="FixedHeavy" CollisionFlags="1, 1, 1" />

    <!-- Create an actor that checks for collision -->
    <Template Name="CrateTemplate">
        <Icon Type="1" Position="$pos$" Background="Crate" Shape="CrateShape"
         Box2dMaterial="Heavy" CollisionFlags="1, 1, 1" Draggable="true"
         OnCollisionStart="CollisionStart" OnCollisionEnd="CollisionEnd" >
            <Modifiers>
                <Modifier Name="iw_notifycollision" Active="true" Param1="1" />
            </Modifiers>
            <Actions Name="CollisionStart">
                <Action Method="AddVar" Param1="num_collisions" Param2="1"/>
            </Actions>
        </Icon>
    </Template>

    <!-- Create a variable to track number of collisions -->
    <Variable Name="num_collisions" Type="int" Value="0" />

    <!-- Create a label to display number of collisions -->
    <Label Font="serif" Background="Button1Brush" Docking="topleft"
         Binding="[Text]num_collisions" />
    </Scene>
```

This example uses a Lua script to spawn new crates under the control of physics into the world every two seconds until 100 crates are reached. We could have quite easily accomplished this with pure XOML but in this case we have used Lua to demonstrate using XOML and scripting together.

We've covered many of the elements shown in the above example in previous chapters so we will only draw out the important bits that relate to scripting.

Firstly we define the scripting language used by the scene as lua and set an OnTick event handler to call the TickActions list each time the scene is updated:

```
    <Scene Name="Scene1" Current="true" Physics="true" Gravity="0, 5" Camera="Camera1"
             DoSleep="false" ScriptEngine="lua" OnTick="TickActions">
```

Next we create a lua script called Script1 then load the Spawn,lua script file into it:

```
        <!-- Define and load a Lua script -->
        <Script Name="Script1" Type="lua" />
        <File Name="File1" Location="Spawn.lua" FileType="lua" Preload="true"
             Script="Script1" />
```

Finally we create an actions list that calls the script function called “Scene_OnTick”

```
        <!-- Create scene OnTick handler that calls a script function each scene update -->
        <Actions Name="TickActions">
            <Action Method="CallScript" Param1="Scene_OnTick" />
        </Actions>
```

Lets take a look at the Spawn.lua script to see what’s going on:

```
local total_objects = 0;
local last_time = 0;
local crate_template = nil;

function Scene_OnTick(_object)

       -- Dont allow more than 100 objects to be spawned
       if (total_objects > 100) then
              return;
       end

       local this_time =  os.time()

       -- Spawn a new object every few seconds
       if ((this_time - last_time) > 1) then

              print("Spawning new crate")
```

```
            -- Find template
            if (crate_template == nil) then
                  crate_template = template.find("CrateTemplate", _object)
                  print("Cached template")
            end

            -- Create template parameters table
            local params = {};
            params["pos"] = "0, -400";

            -- Instantiate the template
            template.from(crate_template, _object, params)

            total_objects = total_objects + 1;

            last_time = this_time;


      end

end
```

The code is quite simple. It begins by declaring the function Scene_OnTick that accepts an _object parameter as its first parameter. This _object will be whatever the type is that called this function (the first parameter when passed from command and actions). Usually this parameter is the scene / actor that called the action / command or the global game object for global scripts.

Next we check to see if we have spawned more than 100 objects, if so we return and spawn no more. Next we get the current time in seconds and check to see if at least 2 seconds have passed. If so then we find the CrateTemplate template in the scene (_object that was passed into this function is Scene1). Note that we only get the template once, it is cached for future calls to save the overhead of finding it every time the OnTick function is called.

Next we build a list of template parameters (just the one in this case pos=”0, -400” to tell the template the position where it should spawn the crate) and call template.from() to instantiate the new crate object from its template.

18.3 Calling Scripts from Actions

You will most likely be calling scripts from actions, especially if your app or game is mainly event driven. The following actions enable you to call scripts from actions:

- CallScript - Calls a script function in a script that has already been loaded into the scene.
  - Param1 - Script function name to call
  - Param2 - Scene or actor that should be passed as the first parameter to the script function. If not set the the actor or scene that the action is defined inside of will be passed.
  - Param3 to Param5 will be passed as the 2nd, 3rd and 4th parameters of the function call.
- CallGlobalScript - Calls a global script function that has already been loaded globally.
  - Param1 - Script function name to call
  - Param2 - Scene that should be passed as the first parameter to the script function. If not set the the main game object will be passed.
  - Param3 to Param5 will be passed as the 2nd, 3rd and 4th parameters of the function call.
- Inline – Executes snipits of script code directly from XOML
  - Param1 – Script snipit to execute
  - Param2 – Name of scene that contains the script engine that should execute the script. By not passing this parameter the scene that contains the action will be used. If the action is declared outside of a scene then the global script engine will be used.

Lets take a look at a few quick examples:

```
<!-- Create a scene with physics enabled -->
<Scene Name="Scene1" ........... ScriptEngine="lua" OnTick="TickActions">

    <!-- Define and load a Lua script -->
    <Script Name="Script1" Type="lua" />
    <File Name="File1" Location="Spawn.lua" FileType="lua" Preload="true"
          Script="Script1" />

    <Actions Name="TickActions">
        <Action Method="CallScript" Param1="Scene_OnTick" />
    </Actions>
```

In the above example we declare the script local to the scene, so we have to call it via CallScript

```
    <!-- Define and load a Lua script -->
    <Script Name="Script1" Type="lua" />
    <File Name="File1" Location="Spawn.lua" FileType="lua" Preload="true"
             Script="Script1" />

    <!-- Create a scene with physics enabled -->
    <Scene Name="Scene1" ........... ScriptEngine="lua" OnTick="TickActions">

        <Actions Name="TickActions">
            <!--Action Method="CallScript" Param1="Scene_OnTick" /-->
            <Action Method="CallGlobalScript" Param1="Scene_OnTick" />
        </Actions>
```

In the above example, notice how the script is declared outside of the scene (globally), therefore it must be called with CallGlobalScript.

In the case of calling scene scripts the first parameter that is passed to the Lua script function is the scene or actor that called the action (unless overridden). When calling global scripts the first parameter that is passed to the Lua script function is the game object (unless overridden). Commands and actions can both pass in additional parameters to the script function, allowing 3 additional custom parameters to be passed.

Now lets take quick look at an example showing Inline script execution:

```
        <Actions Name="Tapped" Condition="outcome">
            <Action Method="Inline" P1="variable.set(variable.get('test1'), '10')"
P2="Scene1" />
        </Actions>
```

This action executes the following script:

```
variable.set(variable.get('test1'), '10')
```

The above snippet changes the XOML variable test1 to the value of 10.

18.4 Calling Scripts from Commands

Its often useful to be able to call a script to perform some calculations or affect other objects in a complex manner from a program. To facilitate this two command have been created that enable you to call scripts and retrieve any value that they return.

The following commands can be called from a program:

- call_script - Calls a function located in a script that has been loaded into the scene. The value returned from the script is set as the return value for this command.
    - Param1 – Name of script function to call
    - Param2 – An optional scene that is passed as the first argument to the function. If not supplied then the scene that the action is declared inside of will be passed as the 1st parameter of the script function call
    - Param3, Param4, Parm5 - Parameters to pass as the 2nd, 3rd and 4th script function parameters
- call_global_script - Calls a function located in a script that has been loaded globally. The value returned from the script is set as the return value for this command. Note that the base game object will be passed to the script function as its first parameter, unless a valid scene is supplied for Param2
    - Param1 – Name of script function to call
    - Param2 – An optional scene that is passed as the first argument to the function
    - Param3, Param4, Parm5 - Parameters to pass as the 2nd, 3rd and 4th script function parameters
- inline – Executes snipits of script code directly from XOML
    - Param1 – Script snipit to execute
    - Param2 – Name of scene that contains the script engine that should execute the script. By not passing this parameter the scene that contains the program will be used. If the program is declared outside of a scene then the global script engine will be used.

19.0 Geometries

19.1 Introduction

Geometries are resources that contain collections of vertices, UV texture coordinates, RGBA colours and face vertex index lists that can be used to create 2D geometry that can be attached to Actors to modify their visual shape. Currently 3 types of geometry are supported by XOML:

- Poly – A convex polygon with any number of vertices
- TriList – A list of triangles
- QuadList – A list of quadrangles

Note that concave polygons are not supported however you can create the same concave shape using a triangle list. A geometry resource is declared using the Geometry XOML tag, e.g:

```
<!--Create a single triangle-->
<Geometry Name="Geoms1" Vertices="-200,-200,200,200,-200,200" UV="0,0,1,1,0,1"
Type="TriList" />

<!--Create two triangles-->
<Geometry Name="Geoms2" Vertices="-200,-200,200,200,-200,200,-400,-400,-200,-400,-200,0"
Type="TriList" />

<!--Create a house shape -->
<Geometry Name="Geoms3" Vertices="-200,-200,0,-300,200,-200,200,200,-200,200"
Indices="0,1,2,3,4"
Colours="255,0,0,255,0,255,0,255,0,0,255,255,255,255,255,255,255,255,255,255" Type="Poly"/>
```

The above XOML shows 3 different examples of creating geometries.

The Geometry XOML tag has the following properties:

- Name (string) – The name of the geometry
- Tag (string) – Resource tag name
- Vertices (vec2 list) – A list of pairs of x,y coordinates that describe the shape of the geometry in 2D
- UV (vec2 list) – A list of pairs of x,y texture coordinates that describe how the assigned image is mapped to the polygon(s) vertices. If this not supplied then it is automatically calculated
- Colours (vec4 list) – A list of RGBA colours that are assigned to each vertex. If this is not supplied then all vertices will be assigned RGBA of

  255,255,255,255
- Indices (list of numbers)– The face index list which determines the order in which vertices form face. If this is not supplied then this will be automatically be generated
- Type (geometry-type) – The type of geometry:
  - Poly – A polygon
  - TriList – A list of triangles
  - QuadList – A list of quadrangles
- PercVerts (boolean) – If set to true then the geometries vertices will be treated as proportional percentage based. The geometry when attached to an actor will be resized to fit the width and height of the host actor.

Assigning a geometry to an Actor will force the actor to render the geometry in place of its default rectangular shape, e.g:

```
<Icon Name="Sprite1" Position="200,0" Background="bg1" Geometry="Geoms1" />
```

Note that hit detection and overlap detection is supported for all types of geometry, including disconnected geometry.

20.0 Lua API

20.1 Introduction

The aim of XOML is to keep all things small and simple. With that in mind the Lua API has been made simple to use and compact. Lets take a look at the complete API to see what is available:

The prototype for all script functions that are called from XOML is as follows:

```
function function-name(object, [params])

      return number
end
```

object – This is the object that called the script. For example, if a scene or actor action called this function then the objects scene or actor is passed as the object, although this can be overridden by some actions and commands.
number – A function called from XOML must always return a number
[params] – Up to 3 optional parameters can be passed from commands or actions

Here is an example:

```
    <Scene Name="Scene1" .... OnTick="TickActions">
        <Actions Name="TickActions">
            <Action Method="CallScript" Param1="Scene_OnTick" />
        </Actions>
```

In this example we create a scene that calls the TickActions actions list. The TickActions list calls the following script function Scene_OnTick:

```
function Scene_OnTick(_object)

      -- Function code goes here

      return 1;
end
```

As you can see from the above code, our Scene_OnTick function follows the prototype definition. The "_object" that is passed into this function is the scene Scene1.

The Lua API is sectioned into libraries of functions. Each library deals with a different aspect of XOML. The following libraries are currently available:

- actions – Deals with XOML actions
- actor – Deals with actors
- brush – Deals with brushes (coming soon)
- camera - Deals with virtual scene cameras
- display – Deals with the display
- facebook – Deals with facebook interaction (coming soon)
- font – Deals with fonts (coming soon)
- geometry – Deals with modification of geometries (coming soon)
- http – Deals with HTTP communications using remote requests
- image – Deals with images
- input – Deals with device input
- market – Deals with in-app purchasing
- media – Deals with audio and video playback
- particles – Deals with particle system actors
- physics – Deals with physics
- program – Deals with XOML programs
- resource – Deals with generic resources
- scene – Deals with scenes
- shape – Deals with shapes (coming soon)
- sqlite3 – Deal with working with SQL databases
- sys – Deals with general system interaction
- template – Deals with templates
- timeline – Deals with timelines
- timer – Deals with timers
- userprops – Deals with user properties
- variable – Deals with XOML variables
- xml – Deals with reading, writing xml data

20.1 Action Library

Actions **actions.find(**actions-name (string), container (scene or actor, optional)) – Finds the specified action “actions-name” in the global actions manager or the actions manager of the supplied scene or actor “container”. Note that if the resource is not found in the supplied scene or actor then the global actions manager will be searched

```
      local my_scene = scene.find(“Scene1")
      local actions1 = actions.find("TapActions1", my_scene)
```

bool **actions.call**(actions (object), target (scene or actor)) – Calls the specified actions list and applies the actions to the supplied target scene or actor. If target not supplied then containing scene / actor will be used

```
      local actions = actions.find("TestActions")
      actions.call(actions, _object)
```

20.2 Actor Library

bool **actor.set**(actor (object), property (string), value (any), user-property (bool, optional)) -
Sets an actors property “property” to the value of “value”, if user-property is set to true then the user properties list will be searched for the property instead of the usual property list, e.g.:

```
      actor.set(_object, "PositionX", pos)
      actor.set(_object, "Timeline", my_timeline_object)
      actor.set(_object, "UserProperty1", user_value1, true)
```

bool **actor.add**(actor (object), property (string), value (any), user-property (bool, optional)) -
Adds a value “value” onto the property “property” of the actor, if user-property is set to true then the user properties list will be searched for the property instead of the usual property list, e.g:

```
      actor.add(_object, "PositionX", pos)
      actor.add(_object, "UserProperty1", user_value1, true)
```

value **actor.get(**actor (object), property (string), user-property (bool, optional)) - Gets the value of the actors property “property”, if user-property is set to true then the user properties list will be searched for the property instead of the usual property list, e.g.:

```
      local pos = actor.get(_object, "PositionX")
      local user_prop1 = actor.get(_object, "UserProperty1", true)
```

actor **actor.find**(actor-name (string), container-scene (object)) – Finds the named actor within the named scene. If the container scene is not supplied then the scene in which the script engine resides will be used, e.g.:

```
      local actor1 = actor.find("Crate1", scene)
      local actor2 = actor.find("Crate2")
```

actor **actor.findOfType**(type (number), container-scene (object)) – Returns a table of actors that are of the specified type within the named scene. If the container scene is not supplied then the scene in which the script engine resides will be used, e.g.:

```
      local actor1 = actor.findOfType(1, scene)
      local actor2 = actor.findOfType(2)
```

actor **actor.findTagged**(tag (string), container-scene (object)) – Returns a table of actors that are tagged with the specific tag. If the container scene is not supplied then the scene in which the script engine resides will be used, e.g.:

```
      local actor1 = actor.findTagged("game", scene)
```

table **actor.findAll**(container-scene (object)) – Finds and returns all actors within the named scene as a table. If the container scene is not supplied then the scene in which the script engine resides will be used, e.g.:

```
      local all_actors = actor.findAll(scene)
```

actor **actor.create(**actor-type (string) container (scene or actor), ...)
actor-type can be:

- Icon background (brush), width (number), height (number)
- ActorText font (font), text (string)
- Label font (font), text (string), background (brush), width (number), height (number), e.g.:

```
      -- Create an icon actor
      local actor = actor.create("icon", _object, brush, 200, 200)
      actor.set(actor, "Velocity", "1, 0")

      -- Create an ActorText actor
      local font = resource.find("serif", "font")
      local text_actor = actor.create("actortext", _object, font, "Hello World")
      actor.set(text_actor, "Velocity", "1, 0")

      -- Create a label actor
```

```
	local label_actor = actor.create("label", _object, font, "Hello World", brush,
150, 150)
	actor.set(label_actor, "Velocity", "1, 0")
	actor.set(label_actor, "BackgroundColour", "80, 80, 255, 255")
```

```
bool		actor.destroy(actor (object)) - Destroys the specified actor. Returns true
if the actor was destroyed, false if not
```

```
bool		actor.destroyTagged(tag (string), scene (object)) – Destroys all actors
within the specified scene that are tagged with the specified tag
```

```
bool		actor.clipped(actor (object)) - Returns true if the actor is on screen,
false if not
```

```
bool		actor.overlaps(actor1 (object), actor2 (object), quick (boolean)) –
Returns true if the two actors overlap. If quick is true then a quick method of
detection is used that uses the CollisionSize parameter of each actor to test for
overlap. If quick is set to false then the actors sprites are tested for overlap (takes
into account rotation and scaling)
```

```
float		actor.distance(actor1 (object), actor2 (object)) – Returns the distance
between two actors in world coordinates
```

```
float		actor.anglediff(actor1 (object), actor2 (object)) – Returns the angle
between two actors in degrees
```

```
table		actor.getStartContacts(actor (object), collision-mask (number, optional))
– Returns a table of actors that collided with the specified actor in the last frame.
The optional collision-mask can be used to mask only actors whos Type property match
the collision-mask
```

```
table		actor.getEndContacts(actor (object), collision-mask (number, optional)) –
Returns a table of actors that stopped colliding with the specified actor in the last
frame. The optional collision-mask can be used to mask only actors whos Type property
match the collision-mask
```

```
table		actor.children(actor (object)) – Returns a table of actors that are
children of the specified actor
```

```
bool         actor.changeTimeline(actor (object), command (play, stop, pause, restart))
– Changes the existing timeline that is attached to an actor
```

```
number       actor.checkState(actor (object)) – Returns the status of the actor (0 –
actor not found, 1 – actor is destroyed, 2 – actor is not destroyed)
```

```
bool         actor.bringToFront(actor (object)) – Removes the actor from the scene and
adds it at the end of the scenes actor list. This causes the actor to be placed above
any other actors that are on the same layer.
```

```
actor        actor.closest(actor (object), type (number)) – Searches the scene to find
the actor of the specified type that is the closest in distance to the supplied actor.
```

```
actor        actor.furthest(actor (object), type (number)) – Searches the scene to find
the actor of the specified type that is the furthest in distance to the supplied actor.
```

20.3 Brush Library

Coming soon.....

20.4 Camera Library

```
bool         camera.set(camera (object), property (string), value (any)) -
Sets an actors property “property” to the value of “value”, e.g.:
```

```
      camera.set(_object, "Position", pos)
```

```
bool         camera.add(camera (object), property (string), value (string)) -
Adds a value “value” onto the property “property” of the camera, e.g:
```

```
      camera.add(_object, "Position", pos)
```

```
value        camera.get(camera (object), property (string)) - Gets the value of the
cameras property “property”, e.g.:
```

```
      local pos = camera.get(_object, "Position")
```

```
actor        camera.find(camera-name (string), container-scene (object, optional)) –
Finds the named camera within the specified scene. If the container scene is not
supplied then the scene in which the script engine resides will be used, e.g.:
```

```
      local camera1 = camera.find(“Camer1", scene)
```

20.5 Display Library

```
number       display.width() - Gets the display width
```

```
number       display.height() - Gets the display height
```

```
number       display.orientation() - Gets the display orientation in degrees
```

20.6 Facebook Library

bool facebook.available() - Returns true if Facebook is available

bool facebook.login(permissions (table), app-id (string, optional)) – Opens the Facebook login dialog to allow the user to log into Facebook. Parameters:
- permissions – Log in permissions (See https://developers.facebook.com/docs/reference/login#permissions for reference)
- app-id - Your Facebook App ID. Note that you should also supply your Facebook App ID in the AppEasy project manager for the iOS platform

Example login:

```
function FacebookLoginCallback(request,event)
	if (event=="loggedin") then
		-- Logged in
	end
end

facebook.setCallback(FacebookLoginCallback)
facebook.login({"read_stream","publish_stream"},"my_app_id")
```

bool facebook.logout() - Logs the user out of Facebook, e.g:

```
facebook.logout()
```

string facebook.token() - Returns the Facebook token, e.g:

```
-- Reload an image using the path to the users profile picture
local my_image=resource.find("MyImage","Image",_scene)
image.reload(my_image,"https://graph.facebook.com/" .. about_me.id .. "/picture?access_token=" .. facebook.token(),false)
```

bool facebook.reauthor(permissions (table)) - Reauthorise permissions (See https://developers.facebook.com/docs/reference/login#permissions for referecnce), e.g:

```
facebook.reauthor({"read_stream","publish_stream"})
```

object facebook.request(method-name (string), http-method (string), parameters (table, optional)) – Makes a Facebook request with method method-name using http method specified by http-method and parameters specified by parameters. Returns the request

```
-- Post status update to users Facebook wall
```

```
local data={}
data["message"]="Test Message"
facebook.request("facebook.stream.publish","POST",data)
```

object facebook.graph(graph-path (string), http-method (string), parameters (table, optional)) – Makes a Facebook graph request with graph path graph-path using http method specified by http-method and parameters specified by parameters. Returns the request – See https://developers.facebook.com/docs/opengraph#howitworks for reference.

```
function FacebookGetAboutMeCallback(request,event)
	if (event=="response") then
		local response=facebook.response(request)
		if (response~=nil) then
			-- Decode the JSON response from Facebook
			local about_me=json.decode(response)
			-- Find the app scene,this is where the controls live that we want
to update
			local _scene=scene.find("scene1")
			-- Get name and assign to about me name label
			actor.set(actor.find("MyName",_scene),"Text","Name:" ..
about_me.name)
			-- Get user id and assign to about me name label
			actor.set(actor.find("MyID",_scene),"Text","ID:"..about_me.id)
			-- Show the about me panel
			actor.set(actor.find("AboutMePanel",_scene),"Visible",true)
			--To get the users profile picture we need to pass the direct graph
url request into an image
			local my_image=resource.find("MyImage","Image",_scene)
			image.reload(my_image,"https://graph.facebook.com/" ..
about_me.id .. "/picture?access_token="..facebook.token(),false)
		end
	end
end

--Get about me info from Facebook
facebook.setCallback(FacebookGetAboutMeCallback)
facebook.graph("me","GET")
```

bool facebook.setCallback(request (object)) - Sets a callback that will be called when a Facebook event is raised. The callback should use the following prototype:

```
function callback(request, event)
- request – The request that raised the event
- event – The event
```

Possible even types include:
- loggedin – A log in event
- logginerror – An error occurred during a log in
- response – A Facebook response occurred
- responseerror – An error occurred during a Facebook response occurred

```
number       facebook.errorCode(request (object)) - Returns the last an error code
```

```
string       facebook.error(request (object)) - Return last error string
```

```
string       facebook.response(request (object)) – Returns the response to the request.
See facebook.graph() example
```

```
void         facebook.waitCallback() - Waits for the Facebook callback to fire before
continuing
```

20.7 Font Library

Coming soon.....

20.8 Geometry Library

Coming soon.....

20.9 HTTP Library

```
object      http.find(remote-request-name (string), scene (object, optional)) – Finds
a remote request
```

```
object      http.set(remote-request (object), property-name (string), value (any)) –
Sets the property of the remote-request object to the supplied values. Supported
properties include:

- Name - Name of request
- Tag - Resource tag name of request
- URL – URL of request
- Data – Data to send
- OnResponse – Actions list that will be called when response occurs
- OnError – Actions list that will be called when an error occurs
- Post – If true then data will be sent via post instead of get
- Variable (object) – Variable that will receive the response data
```

```
any         http.get(remote-request (object), property-name (string)) – Gets the
specified property of the remote-request object
```

```
any         http.send(remote-request (object)) – Sends the remote specified request
```

```
any         http.responseCode() – Returns the last remote requests response code
```

```
any         http.responseHeader(header-name (string)) – Returns the specified header
```

20.10 Image Library

```
vec         image.getSize() – Returns the width and height of the image
```

```
vec         image.reload(image (object), filename (string), blocking (boolean,
optional)) – Reloads the image with the new parameters
```

20.11 Input Library

```
bool         input.multitouch() - Returns true if multi touch input is supported
bool         input.compass() - Returns true if the compass is supported
bool         input.accelerometer() - Returns true if the accelerometer is supported
bool         input.keyboard() - Returns true if the keyboard is supported
bool         input.backPressed() - Returns true if the back buttno is pressed
bool         input.menuPressed() - Returns true if the home button is pressed
```

```
bool         input.startCompass() - Starts the compass, return true if successfully
started
void         input.stopCompass() - Stops the compass
vec          input.getCompass() - Returns compass heading_x, heading_y, heading_z and
direction as a vector
```

```
      input.startCompass()
      local vec = input.getCompass()
      print(vec)
```

```
void         input.startAccelerometer() - Starts the accelerometer, return true if
successfully started
void         input.stopAccelerometer() - Stops the accelerometer
vec          input.getAccelerometer()  - Returns accelerometer position_x, position_y,
position_z as a vector
vec          input.getAccelerometerOffset()  - Returns accelerometer position_x,
position_y, position_z as an offset from the reference point
void         input.setAccelerometerRef() - Sets the current accelerometer value as a
reference point
```

```
      input.startAccelerometer()
      local vec = input.getAccelerometer()
      print(vec)
```

```
string       input.textInput(message (string), default_text (string))  - Shows the on
screen keyboard allowing the user to input information. Message is the prompt message
that is displayed to the user and desfult_text will be the default text displayed in
the text entry box
```

```
      local name = input.textInput(“Enter name”, “noname”)
```

```
vec          input.touchInfo(touch-index (number)) – Returns information about the
specified touch at index touch-index (0 to 9). Returns a vector (position_x (x),
position_y (y), touch-state (z), touch-id (w))
```

```
      local touch_info = input.touchInfo(0)
```

```
	if (touch_info.z ~= 0) then
		local scene_pos = scene.toScene(_object, touch_info.x, touch_info.y, true)
		print(scene_pos.x .. "," .. scene_pos.y)
	end
```

number	**input.getKeyPressed**() – Returns the last pressed key, see Keys and KeyNames tables in Test65.lua for list of key code mappings and key names.

```
	local key_code = input.heyKeyPressed()
```

boolean	**input.isKeyUp**(key-code (number)) – Returns specified keys up state
boolean	**input.isKeyDown**(key-code (number)) – Returns specified keys down state

```
	local label = actor.find("Label1")
	if (input.isKeyDown(Keys.KeyLeft)) then
		actor.add(label,"PositionX",-2)
	end
	if (input.isKeyDown(Keys.KeyRight)) then
		actor.add(label,"PositionX",2)
	end
```

20.12 Market Library

product **market.find(**product-name (string)) – Searches the market for the named product returning the product object or nil if not found

```
local product = market.find("Coins10")
```

table **market.products()** – Returns a table of all market product names

boolean **market.purchased(**product (object)) – Returns the purchased status of the supplied product

boolean **market.consumable(**product (object)) – Returns the consumable status of the supplied product

product **market.purchase(**product (object), purchase-callback (function, optional)) – Starts purchase of the supplied product. The optional purchase-callback function will be called when the purchase is completed or an error occurs

```
function PurchaseCallback(status, product_id)
      print("Purchased - " .. product_id .. " - " .. status)
end

local product = market.find("Levels1to10")
market.purchase(product, PurchaseCallback)
```

bool **market.restore()** - Attempts to restore all previously purchased products

product **market.currentProduct()** – Returns the last product for which a purchase, refund or restore was attempted

bool **market.busy()** – Returns true if the Market is busy or false if not

number **market.price(**product (object)) – Returns price of the specified product

string **market.name(**product (object)) – Returns name of the specified product

```
number market.status(product (object)) – Returns the status of the last purchase
request
```

```
bool market.available() – Returns true if the market is available
```

20.13 Media Library

sound-instance **media.playSound(**sound (object), looped (boolean), volume (number), pitch (number), pan (number)) – Plays the specified sound. Looped, volume, pitch and pan can be optionally specified. Volume range is 0 to 1.0, Pitch range is 1 to no maximum, Pan range is -1.0 to 1.0. Returns a sound-instance which can be used to modify the playing sound and or check if it is still playing

```
local sound = resource.find("Explosion", "sound")
local sound_instance = media.playSound(sound, false, 1, 1, 1)
```

bool **media.stopSound**(sound-instance (object)) – Stops a sound from playing

bool **media.pauseSound**(sound-instance (object)) – Pauses playback of the sound

bool **media.resumeSound**(sound-instance (object)) – Resumes playback of a paused sound

bool **media.changeSound**(sound-instance (object), parameter (string), value (number)) – Changes a playing sound-instance. Parameter can be volume, pitch or pan. Value is the new value to assign to the specified parameter.

```
media.changeSound(sound_instance, "pitch", sound_pitch)
```

bool **media.soundPlaying(**sound-instance (object)) – Checks to see if the specified sound-instance is still playing.

```
if (sound_instance ~= nil) then
      if (media.soundPlaying(sound_instance)) then
      end
end
```

bool **media.playMusic(**file-name (string), repeat-count (number, optional)) – Plays music located in the local file file-name. Repeat-count is the number of times to repeat the music. Passing 0 or excluding repeat-count will result in music playing forever

bool **media.musicPlaying**() – Returns true if music is playing

bool **media.stopMusic**() – Stops the currently playing music

bool **media.pauseMusic**() – Pauses the currently playing music

bool **media.resumeMusic**() – Resumes the previous playing music

```
      media.playMusic("music.mp3", 0)
      media.stopMusic()
      media.pauseMusic()
      if (media.musicPlaying()) then
            media.resumeMusic()
      end
```

```
bool        media.setMusicVolume(volume (number)) – Resumes the previous playing music
```

```
bool        media.enableMusic(enable (bool)) – Enables / disable music playback
```

```
bool        media.enableSound(enable (bool)) – Enables / disable sound playback
```

```
bool        media.audioCodecSupported(codec-name (string))  Checks the system to see
if a particular music playback format is supported. The following codecs are supported:
      - midi
      - mp3
      - aac
      - aacplus
      - qcp
      - pcm
      - spf
      - amr
      - mp4
```

```
      if (media.audioCodecSupported("mp3")) then
            media.playMusic("music.mp3", 0)
      end
```

```
bool        media.videoCodecSupported(codec (string)) - the system to see if a
particular video playback format is supported. The following codecs are supported:
      - mpeg4
      - 3gpp
      - 3gpp_video_h263
      - 3gpp_video_h264
      - 3gpp_audio_amr
      - 3gpp_audio_aac
      - mpeg4_video_mpeg4
      - mpeg4_video_h264
      - mpeg4_audio_aac
      - swf
```

```
bool        media.changeVideoCam(video-cam (object) command (string (start, stop)) –
Changes a the specified video-cam object. The following commands can be passed to the
object:
```

```
- start – Starts video cam play back
- stop – Stops video cam play back
```

```
      local videocam = resource.find(“Cam1”, “videocam”)
      media.changeVideoCam(videocam, “start”)
```

```
bool         media.snapshotVideoCam(filename (string), quality (number, optional) –
Takes a snap shot of the current web cam view and saves it as JPEG to the specified
file at the optional quality setting (default is 90). Note that filename should not
contain an extension as one is automatically added
```

20.14 Particles Library

particle **particles.create**(particle-actor (object), count (number), x (number), y (number), srcrect (vec4, optional)) – Adds count number of particles to the specified particle actor at the specified x, y coordinates. If srcrect is specified then each particle actor will be assigned the specified srcrect, if not then each added particle will be assigned the same scrrect as the first particle that exists in the particle system. Returns the first created particle or nil if an error occurred.

```
Local test1 = actor.find("Test1", scene.find("GameScene"))
Local p = particles.create(test1, 1, 100, 100, vec.new(0,139,102,102))
```

particle **particles.find**(particle-actor (object), index (number)) – Returns the particle at the specified index or nil if not found.

```
local p = particles.find(test1, 0)
particles.set(p, "Scale", "2.0,2.0")
```

bool **particles.set**(particle (object), property-name (string), value (string, number, boolean or vec)) – Sets the specified property of the specified particle to the specified value.

bool **particles.add**(particle (object), property-name (string), value (string, number, boolean or vec)) – Adds the specified value onto the specified property of the specified particle.

value **particles.get**(particle (object), property-name (string)) – Returns the specified property of the specified particle.

bool **particles.removeAll**(particles-actor (object)) – Destroys all particles in the specified particle actor.

20.15 Physics Library

```
joint       physics.findJoint(actor (object), joint-name (string)) – Finds the named
joint of the specified actor
```

```
joint       physics.destroyJoint(actor (object), joint (object)) – Finds the named
joint of the specified actor
```

```
joint       physics.setJointBodyA(joint (object), actor (object)) – Sets the joints
BodyA actor
```

```
joint       physics.setJointBodyB(joint (object), actor (object)) – Sets the joints
BodyB actor
```

20.16 Program Library

program   **program.find(**program-name (string), container-scene (object, optional)) - Finds the specified program “program-name” in the global program manager or the program manager of the supplied scene “container-scene”. If the container scene is not supplied then the scene in which the script engine resides will be used. Note that if the program is not found in the supplied scene then the global programs manager will be searched

```
    local my_scene = scene.find(“Scene1")
    local my_program = program.find("Main", my_scene)
```

bool   **program.destroy**(program (object)) – Destroys the specified program

bool   **program.start**(program (object)) – Starts the specified program

bool   **program.restart**(program (object)) – Restarts the specified program

bool   **program.pause**(program (object)) – Pauses the specified program

bool   **program.nextCommand**(program (object)) – Moves the specified program to its next command

bool   **program.setPriority**(program (object)) – Makes the specified program this the priority program

bool   **program.goto**(program (object), command-name (string, optional)) – Changes specified programs execution to the specified named command “command-name”

bool   **program.running(**program (object)) – Checks to see if a program is currently running (running means not stopped)

bool   **program.paused**(program (object)) - Checks to see if a program is currently paused

20.17 Resource Library

Object **resource.find**(resource-name (string), resource_type (string), container-scene (object, optional)) – Finds a resource “resource-name” of a specific type “resource-type” in the global resource manager or the resource manager of the supplied scene “scene”. If the container scene is not supplied then the scene in which the script engine resides will be used. Note that if the resource is not found in the supplied scene then global resources will be searched, e.g.:

```
local my_scene = scene.find(“Scene1")
local alien_image = resource.find("alien_image", "image", my_scene)
```

table **resource.findOfType**(resource-type (string), container-scene (object, optional)) – Finds all resources of the specified type in the global resource manager or the resource manager of the supplied scene “scene” and returns them as a table. If the container scene is not supplied then the scene in which the script engine resides will be used.

bool **resource.destroy**(resource (object)) – Removes and destroys the specified resource. This does not include variables, actions or timelines.

```
Resource.remove(alien_image)
```

bool **resource.destroyTagged**(tag (string), scene (object, optional)) – Removes and destroys tagged resources from global resources collection or the supplied scenes resource collection. This does not include variables, actions or timelines.

```
Resource.removeTagged("Group1", scene)
```

object **resource.create**(type-name (string), parameters (table), parent (object)) – Creates an instance of the specified type of XOML object using the supplied parameters. If parent is set then it will be passed as the parent of the created object. Returns the created resource. Can be used to create any XOML object.

```
 -- Createa XOMLl abel from a table
local label = resource.create("Label", {
    Name="Label1",
    Font="serif",
    Position="0,-100",
    Text="Hello World",
    BackgroundColour="80,80,80,255",
    Background="Button1Brush",
    Size="-100,50",
    AutoHeight="true"}, _scene)
actor.set(label,"Text","Hey man")
```

```
bool          resource.createFromString(xoml-string (string), parent (object)) –
Instantiates a string of XOML.
```

```
   -- Create XOML label from string
   local xoml = [[
        <Label Position="0,100" Font="serif" Text="Test creating XOML resources from
Lua" BackgroundColour="80,80,80,255" Background="Button1Brush" Size="-100,-10"
AutoHeight="true" IgnoreCamera="true"/> ]]
   resource.createFromString(xoml, _scene)
```

20.18 Scene Library

```
bool         scene.set(scene (object), property (string), value (any)) -
Sets a scenes property “property” to the value of “value”, e.g.:
```

```
      scene.set(_object, "PositionX", pos)
```

```
bool         scene.add(scene (object), property (string), value (any)) -
Adds a value “value” onto the property “property” of the scene, e.g:
```

```
      scene.add(_object, "PositionX", pos)
```

```
value        scene.get(scene (object), property (string)) - Gets the value of the
scenes property “property”, e.g.:
```

```
      local pos = scene.get(_object, "PositionX")
```

```
scene        scene.find(scene-name (string)) – Finds the named scene
```

```
      local my_scene = scene.find(“Scene1")
```

```
table        scene.findOfType(scene-type (number)) – Finds and returns a table of all
scenes of the specified type
```

```
scene        scene.create(scene-name (string), width (number), height (number),
canvas_fit (string), origin (string), physics (boolean), batching (boolean), script-
engine (string)) – Creates a named scene, parameters are:

- scene-name – Name of the scene-name
- width – Width of the scene
- height – Height of the scene
- canvas_fit – Virtual canvas fit method (none, width, height, both, best)
- origin – Virtual canvas origin (centre, top, left, topleft)
- physics - true to enable physics processing
- batching – true to enable batch rendering
- script-engine – Name of script engine to use (e.g. lua)
```

```
      local my_scene = scene.create(“Scene1", 800, 600, “best”, “centre”, false, false,
“lua”)
```

```
bool         scene.destroy(scene (object)) – Destroys the specified scene
```

```
scene        scene.setCurrent(scene (object)) – Sets the currently active scene and
returns the previous active scene.
```

```
scene        scene.getCurrent() – Gets the currently active scene
```

```
vec          scene.toScreen(scene (object), x (number), y (number)) – Converts the
supplied scene coordinates to raw device coordinates
```

```
vec          scene.toScene(scene (object), x (number), y (number), include_camera
(boolean)) – Converts the supplied raw device coordinates to scene coordinates. If
include_camera is true then the camera transform is taken into account
```

```
vec          scene.updatePhysics(scene (object), elapsed-time (number)) – Updates the
scenes physics engine, useful for stabilising physics pre game
```

```
vec          scene.cleanup(scene (object)) – Cleans up all deleted actors from the
scene. This is useful if you want to immediately remove any actors that were deleted
```

20.19 Shape Library

Coming soon.....

20.20 SQLite3 Library

For information pertaining to this library please see
http://lua.sqlite.org/index.cgi/doc/tip/doc/lsqlite3.wiki

20.21 Sys Library

```
bool          sys.isType(object, type-name (string)) – Checks to see if the supplied
object is of a specific type.
```

```
      if (sys.isType(_object, "actor")) then
            _scene = actor.get(_object, "scene")
      end
```

```
bool          sys.isTypeOf(object, type-name (string)) – Checks to see if the supplied
object is derived from a specific base type. For example the base type of all UI
elements is actor.
```

```
      if (sys.isTypeOf(_object, "actor")) then
            print(“This object is derived from an actor”)
      end
```

```
bool          sys.launchURL(URL (string)) – Launches an external URL
```

```
      LaunchURL("http://www.appeasymobile.com")
```

```
bool          sys.exit() – Exits the app
```

```
device_name  sys.getDeviceType() – Returns the type of device that the app is running
on. Possible values include:
- iphone – Apple iPhone
- ipad – Apple iPad
- android – Google Android
- windows – Windows simulator
- unsupported – An unsupported device
```

```
string        sys.getLocale() – Returns device language in ISO 639 and ISO 3166 format
(en_GB / en_US for example)
```

```
number        sys.getTimeMs() – Returns the current time in milliseconds
```

```
number        sys.getTimeUTC() – Returns the current time in UTC
number       sys.getFreeMem() – Returns the amount of free memory available to the
application in bytes
```

```
bool         sys.vibrate(duration (number), priority (number 0 to 255, optional)
) – Starts device vibrating for duration seconds. Priority is the priority level that
should be given to the vibration
```

```
bool         sys.stopVibrate() – Stops vibration
```

```
bool         sys.changePowerSaving(enable (boolean)) – Enable or disable device power
saving mode
```

```
bool         sys.yield(yield-time (number)) – Yields to the OS in seconds
```

```
bool         sys.loadXoml(filename (string), scene (object, optional)) – Loads the
specified XOML file. If scene is supplied then the XOML file will be loaded into the
scene
```

```
number       sys.getTotalFrames() – Returns number of frames that have been processed
by the app since boot
```

```
number       sys.getFrameRate() – Returns the rate at which the apps main loop is
running at
```

```
bool         sys.pauseTime(paused-state (bool)) – Pauses / un-pauses time. Passing
false will reset the time delay from the last frame, useful when performing long
operations that take a long time such as loading and prevents sudden jumps at the start
of a level
```

```
bool         sys.isTimePaused() – Returns the paused state of time
```

```
bool         sys.getOSVersion() – Returns current OS version as a string
bool         sys.setBackgroundColour(red, green, blue) – Sets the apps main background
colour
```

20.22 Template Library

```
template     template.find(template-name (string), container-scene (object, optional))
– Finds the named template, optional container-scene can be used to search specific
scenes
```

```
local crate_template = template.find(“CrateTemplate", main_scene)
```

```
bool         template.destroy(template (object)) – Destroys the specified template
```

```
bool         template.from(template (object), container-scene (object), template-
parameters (table)) – Instantiates the specified template, e.g.:
```

```
-- Find template
local template = template.find("button_temp", _object)

-- Create template parameters table
local params = {};
params["name"] = "Button1";
params["pos"] = "150,150";
params["brush"] = "Button2Brush";
params["text"] = "Hello2";

-- Instantiate the template
template.from(template, _object, params)
```

20.23 Timeline Library

timeline **timeline.find**(timeline-name (string), container (scene or actor, optional)) - Finds the specified timeline “timeline-name” in the global timelines manager or the timelines manager of the supplied scene or actor “container”. Note that if the timeline is not found in the supplied scene or actor then the global timelines manager will be searched

```
    local my_scene = scene.find(“Scene1")
    local my_timeline = timeline.find("IntroAnimation", my_scene)
```

bool **timeline.stop**(timeline (object)) – Stops the specified timeline

bool **timeline.play**(timeline (object)) – Plays the specified timeline

bool **timeline.pause**(timeline (object)) – Pauses the specified timeline

bool **timeline.restart**(timeline (object)) – Restarts the specified timeline

bool **timeline.playing(**timeline (object)) – Checks to see if a timeline is currently playing

20.24 Timer Library

```
timer        timer.find(timer-name (string), container (scene or actor, optional)) -
Finds the specified timer “timer-name” in the global timers manager or the timers
manager of the supplied scene or actor “container”. Note that if the timer is not found
in the supplied scene or actor then the global timers manager will be searched
```

```
      local my_scene = scene.find(“Scene1")
      local my_timer = timer.find("CannonTimer", my_scene)
```

```
bool         timer.stop( timer (object)) – Stops the specified  timer
```

```
bool         timer.start( timer (object)) – Plays the specified  timer
```

```
bool         timer.restart( timer (object)) – Restarts the specified  timer
```

```
bool         timer.running(timer (object)) – Checks to see if a timer is currently
running
```

20.25 User Properties Library

```
user-property userprops.find(user-property-list (object), property-name (string)) –
Searches the supplied user property list for the named property
```

```
bool         userprops.set(user-property (object), value (any)) – Sets the specified
user property to the specified value
```

```
bool         userprops.add(user-property (object), value (any)) – Adds the specified
value onto the specified user property
```

```
value        userprops.get(user-property (object)) – Gets the value of the specified
user property
```

20.26 Variable Library

Variable **variable.find**(variable-name (string), container-scene (object, optional)) – Finds the specified variable "variable-name" in the global variable manager or the variables manager of the supplied scene "scene". If the container scene is not supplied then the scene in which the script engine resides will be used. Note that if the resource is not found in the supplied scene then the global variables manager will be searched

bool **variable.set**(variable (variable), value (any), index (number)) – Sets the specified variable "variable" to the specified value "value". If an index is added and the variable is an array then the element at array index "index" will be written

```
local var1 = variable.find("Message")
variable.set(var1, "Woohoo!")
```

bool **variable.add**(variable (variable), value (any), index (number)) – Adds the specified value "value" onto the value of the specified variable "variable". If an index is added and the variable is an array then the element at array index "index" will be modified

```
local var1 = variable.find("Message")
variable.add(var1, "Woohoo!")
```

value **variable.get**(variable (object), index (number, optional)) – Return the value of the specified variable, an optional index can be supplied for to return the nth item of an array

```
local var1 = variable.find("Message")
local value = variable.get(var1)
```

bool **variable.array**(variable (object)) – Return true if the specified variable is an array

number **variable.size**(variable (object)) – Return the size of the array variable (the number of elements that it contains)

```
local var1 = variable.find("Message")
local size = variable.size(var1)
```

```
number       variable.count(variable (object)) – Return the number of valid elements in
the array variable
```

```
bool         variable.append(variable (variable), value (any), count (number,
optional)) – Appends the specified value “value” onto an array enlarging its sizes to
accommodate. If count is supplied then the value will be added count number of times.
```

```
      local var1 = variable.find("MyArray")
      variable.add(var1, "Woohoo!", 10)
```

```
bool         variable.save(variable (object), index (number, optional)) – Saves the
specified persistent variable to permanent storage
```

```
table        variable.asTable(array-variable (object)) – Returns the array variable as
a table
```

20.27 XML Library

bool **xml.destroyParser**(parser (object)) – Destroys the specified xml parser object

node **xml.createNode**(node-name (string)) – Creates an Xml node with the specified name

```
	local actors_node = xml.createNode("Actors")
```

bool **xml.destroyNode**(node (object)) – Destroys the specified xml node removing it from its parent node

```
	xml.destroyNode(actors_node)
```

bool **xml.addNode**(parent-node (object), child-node (object)) – Adds the specified child node to the specified parent nodes child list

```
	for i = 0,10 do
		local actor_node = xml.createNode("Actor")
		xml.addNode(actors_node, actor_node)
	end
```

bool **xml.removeNode**(node (object)) – Removes the specified node from its parent node (does not destroy the node)

bool **xml.setNodeName**(node (object), node-name (string)) – Sets the specified nods name

string **xml.getNodeName**(node (object)) – Gets the specified nodes name

bool **xml.setNodeValue**(node (object), node-value (string)) – Sets the specified nodes value

string **xml.getNodeValue**(node (object)) - Gets the specified nodes value

bool **xml.setAttribs**(node (object), attributes (table)) – Sets node attributes of the specified node. Table keys are used to specify the attribute name whilst table values are used to specify attribute values

```
      local actor_node = xml.createNode("Actor")
      local attribs = {}
      attribs.Name = "Actor1"
      attribs.Position = "0,0"
      attribs.Angle = "0"
      attribs.Type = "4"
      xml.setAttribs(actor_node, attribs)
```

table **xml.getAttribs**(node (object)) – Gets the attributes of the specified node as a table of key value pairs

```
      local attribs = xml.getAttribs(actor_node)
      print("attribs.Name = " .. attribs.Name)
```

node **xml.root**(parser (object)) – Gets the root node of the specified xml parser object

node **xml.findFirstNamedNode**(parent-node (object), node-name (string)) – Finds the first node with the specified name within the specified node

```
      local hello_node = xml.findFirstNamedNode(actors_node, "Hello")
      print("hello_node - " .. xml.getNodeName(hello_node))
```

table **xml.children**(parent-node (object)) – Gets all child nodes that of the specified parent node as table of key / value pairs

```
      local children = xml.children(actors_node)
      for k,node in ipairs(children) do
            print("Nodename - " .. xml.getNodeName(node))
      end
```

bool **xml.save**(node (object), filename (string)) – Saves the specified node and all child nodes as an xml file with the specified filename

```
      -- Create some test xml
      local actors_node=xml.createNode("Actors")
      local attribs = {}
      attribs.Name="Actors"
      attribs.Position="0,0"
      attribs.Angle="0"
      xml.setAttribs(actors_node, attribs)

      for i = 0,10 do
            local actor_node = xml.createNode("Actor")
            local attribs = {}
            attribs.Name = "Actor" .. tostring(i)
            attribs.Position = "0,0"
            attribs.Angle = "0"
            attribs.Type = "4"
            xml.setAttribs(actor_node, attribs)
            xml.addNode(actors_node, actor_node)
      end

      -- Save the xml to a file
      xml.save(actors_node,"myfile.xml")

      -- Clean-up nodes
      xml.destroyNode(actors_node)
```

parser **xml.load**(filename (string)) – Loads the specified xml file and parses it

```
      -- Load xml file
      local parser = xml.load("myfile.xml")
      local root = xml.root(parser)
      -- Get Actors node
      local actors = xml.findFirstNamedNode(root, "Actors")
      -- Iterate over all Actor child nodes
      local children = xml.children(actors)
      for k,node in ipairs(children) do
            print(">>>>" .. xml.getNodeName(node))
            -- Iterate over all node attributes
            local attribs = xml.getAttribs(node)
            if (attribs ~= nil) then
                  for attrib,value in pairs(attribs) do
                        print("----" .. attrib .. "-" .. value)
                  end
            end
      end
```

20.28 Example Code

Here’s a little example Lua code so you can see what to expect:

```
function Scene_OnTick(_object)

        local this_time =  os.time()

        if (sound_instance ~= nil) then
                if (media.soundPlaying(sound_instance)) then
                        media.changeSound(sound_instance, "pitch", sound_pitch)
                        sound_pitch = sound_pitch + 0.02;
                else
                        sound_pitch = 0.01;
                end
        end

        -- Don t allow more than 10 objects to be spawned
        if (total_objects > 100) then
                return;
        end

        -- Spawn a new object every few seconds
        if ((this_time - last_time) > 1) then

                print("Spawning new crate")

                -- Find template
                if (crate_template == nil) then
                        crate_template = template.find("CrateTemplate", _object)
                        print("Cached template")
                end

                -- Create template parameters
                local params = {};
                params["pos"] = "0, -400";

                -- Instantiate the template
                template.from(crate_template, _object, params)

                total_objects = total_objects + 1;

                last_time = this_time;


        end

        return 1
end
```

```
function CreateActors(_object)

	local brush = resource.find("Button1Brush", "brush")

	if (isTypeOf(_object, "actor")) then
		_object = actor.get(_object, "scene")
	end

	local my_actions = actions.find("TestActions")
	actions.call(my_actions, _object)

	local my_program = program.find("Program1")
	program.start(my_program)

	local my_actor = actor.create("icon", _object, brush, 200, 200)
	actor.set(my_actor, "Velocity", "1, 0")

	local font = resource.find("serif", "font")
	local text_actor = actor.create("actortext", _object, font, "Hello World")
	actor.set(text_actor, "Velocity", "1, 0")
	actor.set(text_actor, "Name", "RemoveMe")

	local label_actor = actor.create("label", _object, font, "Hello World", brush,
150, 150)
	actor.set(label_actor, "Velocity", "1, 0")
	actor.set(label_actor, "BackgroundColour", "80, 80, 255, 255")

	local sound = resource.find("Explosion", "sound")
	sound_instance = media.playSound(sound)

	return 1
end
```

20.29 Errors and Warnings

If you pass incorrect parameters to Lua functions you will generally receive an error or a warning that tells you what you did wrong. A couple of example errors are shown below:

```
Warning: LUA_PlayMusic, invalid value for repeat-count in Param1
Warning: LUA_SetProperty not enough parameters, expected scene or actor (object), property
(string), value (string)
```

The message also provides information on how to use the function properly.

You may also receive additional errors from Lua itself, which generally point towards a syntax error in your code. Here is an example:

```
LUA ERROR: ---------------------------------------------------------------------------
LUA ERROR: [string "scene0"]:89: unexpected symbol near '1'
LUA ERROR: ---------------------------------------------------------------------------
```

This is telling is that it found an error at line 89 near the symbol '1', in our case it was the following typo:

```
      local scene 1= CreateScene("Scene2")
```

That should have read:

```
      local scene = CreateScene("Scene2")
```

When a Lua error occurs the lua script engine is closed down and no more scripts will run.

21.0 Adding Ads

21.1 Introduction

AppEasy provides a mechanism for monetising your apps and games using HTML ads. Using an HTML view you can display both ad URL's and HTML content, including Javascript.

We decided to use HTML ads as they offer a wide range of features compared to basic text or banner ads. For example using HTML ads you have access to the following types of ads:

- Animating banner and text ads of any size
- Video ads
- Interstitial ads
- Interactive ads
- Forms based ads
- Offer walls

Having such a wide range of ad options gives you a much better chance of earning a high eCPM and increasing CTR, maximising the profit yuo can earn from your app.

To add ads to your app you simply create a WebView and navigate the web view to a particular URL or send HTML / Javascript to the WebView. Lets take a quick look at both methods:

Here is an example that shows an offer wall in your app wall simply navigating to a web page:

```
<WebView Name="Ads" Position="0, 0" Docking="top" Size="-100, -90"
URI="http://ad.leadboltmobile.net/show_app_wall?section_id=89739817373" />
```

And here is an example that shows a banner ad in your app by assigning html / javascript to the web view:

```
<WebView Name="Ads" Transparent="true" Docking="top" Size="320,50" Html="&lt;script
type='text/javascript' src='http://ad.leadboltmobile.net/show_app_ad.js?
section_id=1739018273'&gt; &lt;/script&gt;" />
```

The actual HTML we pass in the above example is:

```
<script type='text/javascript' src='http://ad.leadboltmobile.net/show_app_ad.js?section_id=1739018273'> </script>
```

This basically calls a script that is located on the ad service providers web server which collects an ad for your app to display.

Note that when you supply the ad script to the web view you cannot pass open and close tag markers < and > signs. Instead you substitute &lt; for < and &gt; for >.

21.2 Integrating Leadbolt Ads

Leadbolt is an ad provider that provides HTML based ads amongst other types of ads to display and monetise in mobile apps and on web sites. You can find out more about them at http://www.leadbolt.com/ .

To receive Leadbolt ads you will firstly need to create a developer account and wait for verification. Once enabled you should go to the Apps section of the dashboard. The first thing you need to do is create an app by clicking the “Create new App \ Mobile Web” button, fill out the info for your app then click the “create” button.

Next you need to create ad units for your app; an ad unit is basically a type of ad for a specific app. To add an ad unit click the “Add ad” button. Choose from either “App Banner” or “App Wall”. App Wall is great for displaying when your app first boots or during a pause or break in game play as it fills most of the screen. Small banner ads are best displayed constantly / often within the game (usually at the top of the screen), whilst large banner ads are useful for pauses and breaks in game play. You should create ad units for all types of ad if possible as you will want to display larger ads on larger screen devices.

Now return to the Apps section of the dashboard. You will notice that all of your ad units are listed. To the right of each ad unit there are a couple of buttons (Get Code and Edit). Click the Get Code button to retrieve the code for your ad. For App Wall ad units this will be a simple URL that looks something like this:

http://ad.leadboltmobile.net/show_app_wall?section_id=98274893948

For banner ads the code will be a small piece of html / Javascript that looks something like this:

 <script type="text/javascript" src="http://ad.leadboltmobile.net/show_app_ad.js?section_id=8274892389"></script>

Now lets jump over to XOML and take a look at how we integrate both of these types of ads.

21.2.1 Integrating App Walls

Integrating the App Wall ad unit is done by simply navigating the web view to the URL supplied previously by Leadbolt. Our XOML will look like this:

```
<WebView Name="Ads" Docking="top" Size="-100, -90"
URI="http://ad.leadboltmobile.net/show_app_wall?section_id=98274893948" />
```

Note that we have told the WebView to use 100% of the screen width and 90% of the screen height. We did this because we want to add a skip button to allow the user to skip the ad. The complete XOML to create a scene for an AppWall is shown below:

```
<?xml version="1.0"?>
<xml>

       <Scene Current="true" OnCreate="Setup" OnKeyBack="Exit" >
        <Actions Name="Exit">
            <Action Method="KillScene" />
            <Action Method="LoadXOML" P1="Menu.xml" />
        </Actions>

        <Image Name="appeasy-button" Location="appeasy-button.png" Preload="true"/>
        <Brush Name="ButtonBrush" Image="appeasy-button" Type="9patch" ScaleArea="7, 8, 186,
54" />

        <Icon Brush="ButtonBrush" BackgroundColour="128, 195, 255, 255" SelectedColour="80,
80, 128, 255" Docking="bottom" Margin="0, 0, 0, -1" Size="-20, -9" AspectLock="y"
OnTapped="Exit" />

        <WebView Name="Ads" Position="0, 0" Docking="top" Size="-100, -90"
URI="http://ad.leadboltmobile.net/show_app_wall?section_id=98274893948" />

    </Scene>

</xml>
```

The above XOML creates a scene that displays an App Wall and a “Skip” button. When the skip button is pressed the ad scene is closed down and the Menu scene is loaded.

To quickly add an App Wall to your app simply clear out your Start.xml (move that XOML to another file, lets call it MainMenu.xml) copy and paste the above XOML into your Start.xml. Replace the URI with your own Leadbolt code then change Menu.xml in the LoadXOML action to the name of the XOML file you want to be loaded when the user exits the ad screen (MainMenu.xml in this example). This will make the App Wall the first scene that is shown to the user. Once the skip button is pressed your game scene will be loaded and ran.

21.2.2 Integrating Banner Ads

Integrating the banner ad unit is a little more involved compared to implementing an ad URL. Lets take a look at the XOML of a typical integration:

```
<WebView Name="Ads" Transparent="true" Docking="top" Size="320,50" Html="&lt;script
type='text/javascript' src='http://ad.leadboltmobile.net/show_app_ad.js?
section_id=8274892389'&gt;&lt;/script&gt;" />
```

This section of XOML shows a small WebView that is docked to the top of the screen which shows banner ads of 320x50 pixels or smaller in size.

A few things you need to decide when integrating banner ads:

- Position – Where will you locate your ad. Wherever you decide to locate your ad, docking is probably the best way to ensure that the banner ad remains in place when screen orientations change
- Size – Ensure that the web view is large enough to fit the ad in or you may end up with browser scroll bars
- Transparency – Ensure that you have the Transparent property set or a white background will be shown around the ad

The last part of the integration involves converting the tag markers in the banner ad code provided by Leadbolt, e.g.:

<script type="text/javascript" src="http://ad.leadboltmobile.net/show_app_ad.js?section_id=8274892389"></script>

becomes:

&lt;script type="text/javascript" src="http://ad.leadboltmobile.net/show_app_ad.js?section_id=8274892389"&gt;&lt;/script&gt;

This is copied into the Html="" section of the WebView.

22.0 In-app Purchasing

22.1 Introduction

Its a tough job to get noticed in the app stores these days with approaching half a million apps available in the larger stores, its an even tougher job persuading the public to part with their money for your app. Many developers are turning to developing freemium titles as a means to increase visibility (app users are much more likely to download a free app) and earn money by offering the app in a limited form then allowing users to purchase additional content and game features.

XOML provides a way to manage and sell an inventory of consumable and none consumable products as well as purchase those products remembering products which have been purchased using the XOML Market tag.

This tag has the following properties:

- Name (string) – Name of the market resource
- AndroidPublicKey (string) – Your Google Play pubilc key if you plan to use Android in-app purchasing (Android only)
- Simulate (string) – This is a string that tells the simulator to simulate specific in-app purchase responses to aid testing using the simulator. Possible simulation modes include:
    - complete – Purchase completes without errors (default)
    - error – Purchase doesnt complete and has errors
    - refund – The purchase was refund (Android only)
    - billingdisabled – In-app purchasing is disabled on the device (Android only)
- OnComplete (actions-list) – Defines an actions list that will be called when a purchase is successful
- OnError (actions-list) – Defines an actions list that will be called when a purchase error occurs
- OnUnavailable (actions-list) – Defines an actions list that will be called if in-app purchasing is unavailable on the platform
- OnBillingDisabled (actions-list) – Defines an actions list that will be called if in-app purchasing is disabled on the device
- OnRefund (actions-list) – Defines an actions list that will be called if the purchase was a refund
- Tag (string) - Resource tag (used to group resources together)

The Market tag enables you to declaratively define products by adding inner Product tags to the Market declaration, e.g:

```
<Market Name="Market1" AndroidPublicKey="my_android_public_key">
    <Product ............ />
    <Product ............ />
    <Product ............ />
    <Product ............ />
    <Product ............ />
    <Product ............ />
    <Product ............ />
</Market>
```

The XOML market works asynchronously, so when a purchase is started a response will not be immediately available. Usually when a purchase request is made the user is taken to the purchasing section of the app store where they will confirm their purchase. You do not need to handle confirmation or cancelling of purchases

22.2 Adding Products

To add products to your app you simply declare them using the Product tag inside the Market tags. Lets take a look at a quick example:

```
        <Market Name="Market1" .............. >
            <Product Name="Coins1" iOSId="com.pocketeers.coins1" AndroidId="coins1" 
Consumable="true" Price="0.99" />
            <Product Name="Coins2" iOSId="com.pocketeers.coins2" AndroidId="coins2" 
Consumable="true" Price="1.99"/>
            <Product Name="Coins3" iOSId="com.pocketeers.coins3" AndroidId="coins3" 
Consumable="true" Price="2.99"/>
            <Product Name="Coins4" iOSId="com.pocketeers.coins4" AndroidId="coins4" 
Consumable="true" Price="3.99"/>
            <Product Name="Levels1to10" iOSId="com.pocketeers.levels1to10" 
AndroidId="levels1to10" Consumable="false" Price="0.99" />
            <Product Name="Levels11to20" iOSId="com.pocketeers.levels11to20" 
AndroidId="levels11to20" Consumable="false" Price="0.99" />
        </Market>
```

In this example, we create 6 products, 4 consumable products and 2 none consumable products

The Product tag has the following attributes:

- Name (string) – The name ofthe product
- iOSId (string) – The ID that you have assigned to this product in your iTunes connect account
- AndroidId (string) – The ID that you have assigned to this product in your Google Play account
- Consumable (boolean) – If true then this item cab be bought as many times as the user likes. If false then the product will not be purchasable again once purchased
- Price (number) – The price of the product (has no function, it does not affect the price of your product defined in the app store)

XOML will save the purchase status of all products that you list in the market. In addition, when a refund takes place the purchased status of the refunded product will be set to false.

22.3 Purchasing with Actions

Actions provide a convenient easy to use method of starting a purchase.

```
<Actions Name="MakePurchase">
    <Action Method="Purchase" P1="Coins1" />
</Actions>
```

The Purchase action takes a single parameter which is the name of the product as listed in the market.

In-app purchasing results do not come back immediately, they happen at some point in the future after the purchase or restore request has been made. When the result comes back an event will be raised.

Its worth noting that this is the most basic form of enabling purchasing of in-app products. We recommend that you handle all of the possible Market in-app purchase events to ensure a smooth in-app purchasing experience. These include:

- OnComplete
- OnError
- OnUnavailable
- OnBillingDisabled
- OnRefund

22.4 Purchasing via Script

XOML also enables purchasing directly from script giving you a little more control over how to handle purchase responses. The market library provides a number of functions that deal with in-app purchasing:

- market.find() - Finds a product by name
- market.products() - Returns a table of product names
- market.purchased() - Returns purchased state of the product
- market.consumable() - Returns consumable state of the product
- market.setCallback() - Sets a callback that is called when market status changes
- market.purchase() - Purchases a product
- market.currentProduct() - Returns the last product for which a purchase, refund or restore was attempted
- market.restpre() - Attempts to restore all previously purchased products
- market.busy() - Returns the markets busy state
- market.price() - Returns the price of the specified product
- market.name() - Returns the name of the specified product
- market.status() - Returns the status of the last request

Below is a short example showing how to make a purchase using Lua:

```lua
function MarketCallback(status, product_id)
        print("MarketCallback:")
        print("Status:" .. status)
        print("Product:" .. product_id)
end

function MakePurchase(_object)
        local product = market.find("coins1")
        market.setCallback(MarketCallback)
        market.purchase(product)
end
```

In this example note how we use a callback function. This callback function is called when the system has finished the purchase. Note that status can be one of the following:

- purchased – The purchase was successful
- error – Error during purchase
- disabled – In-app purchasing is disabled on the device

- refund - The purchase was a refund
- unavailable – In-app purchasing is unavailable on the device

22.5 iOS In-App Purchase Testing

Testing in-app purchasing is not an easy task, but to ease the pain a basic walk-through for iOS has been laid out in this section.

1. Create your app in iTunes Connect (do not put into waiting for upload state)
2. Click the "Manage In-App Purchases" button
3. Create your in-app purchases but leave them in the "Waiting for screenshot" state
4. If you dont have one already then create a test user account
5. Log into the test user account on your iOS device
6. Test your in-app purchases
7. When finished testing ensure that you upload your in-app purchase screenshots before you upload your binary for approval

22.6 Android In-App Purchase Testing

Testing in-app purchasing is not an easy task, but to ease the pain a basic walk-through for Android has been laid out in this section.

1. Create and save your app in the Android market control panel (do NOT publish it)
2. Click in "In-app Products" link below your app name in the Android market control panel
3. Create and "publish" your in-app purchases
4. If you dont have one already then create a 2nd gmail user account
5. Edit your Android market profile and add the 2nd user gmail to the "Test Users" field then save
6. Log into the 2nd user gmail account on your test device and make it the primary user (note that older devices will not allow you to change the primary user, so you have to factory reset the device and then add the 2nd user from scratch)
7. Test your in app purchases
8. When finished testing upload your app to the Android market

23.0 Timers

23.1 Introduction

Most games and apps alike usually have some mechanism to cause events to occur regulalrly or now and again. For example, if the player hasn't touched the screen for 2 minutes then the player character could dance, or maybe you want a turret to fire a bullet every 5 seconds until it is destroyed. Timers offer an incredibly easy way to implement the timing of such functionality. Lets take our cannon example. If we created a timer that fires an event every 5 seconds, this event could call an action that causes the cannon to create a new bullet. Lets take a look at a short piece of XOML that could accomplish this:

```
<Actions Name="CannonFireBullet">
    <Action Method="CallScript" P1="TellCannonFireBullet"/>
</Actions>
<Timer Name="CannonFireTimer" Duration="5" Repeat="10" OnRepeat="CannonFireBullet"/>
```

In the above example we create an actions list that contains an action that calls a script called "TellCannonFireBullet", this script would create a new bullet and fire it. Next we create a timer called "CannonFireTimer" that fires the OnRepeat event every 5 seconds over the period of 50 seconds (because repeat is set to 10). The OnRepeat event calls the "CannonFireBullet" actions list which in turn calls the "TellCannonFireBullet" script.

Timers can be changed via the ChangeTimer action or via script using the timer library.

23.2 Properties

To create a timer you declare it in XOML or from script using resource.create(). The following properties are supported by Timers:

General Properties:

- Name (string) - Name of the actor, used to refer to the actor from scripts and such. Note that no two actors in the same scene should share the same name.
- Tag (string) - Group tag
- AutoStart (boolean) – If set to true then the timer will be started as soon as it is declared. If set to false then it must be manually started from script ot using an action. Default is to auto-start
- AutoDelete (boolean) – If set to true then the timer will automatically delete

itself from the resources once it times out. Default is to not delete itself

- Duration (number) – The duration of the timer in seconds
- Repeat (number) – Total number of times to repeat the timer (0 represents repeat forever). Default is set to 1
- OnTimeout (actions-list) – Defines an actions list that will be called when the timer times out. The timeout event will not be raised if the timer has repeats left
- OnRepeat (actions-list) – Defines an actions list that will be called when the timer repeats. The repeat event will not be raised if the timer is about to timeout and has no repeats left
- Local (boolean) – If set to false then the timer will be declared within the scene resource manager instead of the actor that contains it. Default is true

24.0 Asset Conversion Tool

The new AppEasy Asset Conversion Tool (ACT) is a small app that enables the conversion of certain types of format to XOML for easy integration into AppEasy. ACT currently supports conversion from the following formats:

- Texture Packer (Generic XML) – Texture Packer is a fantastic tool that enables you to create sprite atlases / sheets – See http://www.codeandweb.com/texturepacker for reference
- SVG – SVG is a format that contains information about vector based scenes

When you launch ACT it will display a simple window with buttons at the top and a large text box. The text box will show imported files as XOML (note that the window is cleared after each import).

24.1 Converting Texture Packer Data

To import Texture Packer data click the “Import Texture Packer” button then select the Texture Packer exported file. XOML in the form shown below will be written to the output window:

```
<Image Name="generic_xml" Location="generic_xml.png" />
<Brush  Name="Bamboo" Image="generic_xml" SrcRect="2,44,126,193" Type="Image" />
<Brush  Name="Credit_Card" Image="generic_xml" SrcRect="2,385,34,53"
Type="Image" />
<Brush  Name="Dragon" Image="generic_xml" SrcRect="2,286,77,97" Type="Image" />
<Brush  Name="Flag" Image="generic_xml" SrcRect="173,2,80,73" Type="Image" />
```

The importer will convert texture atlases and sprites to images and brushes. You can copy and post ths XOML directly into your own XOML files (dont forget to also copy the images across to your assets folder)

24.2 Converting SVG Data

To import SVG data click the “Import SVG” button then select a SVG or Inkscape SVG file. XOML in the form shown below will be written to the output window:

```
<Scene Name="svg2" Extents="-512,-384,1024,768">
<Geometry Name="box1" Vertices="-68.57,-72.86,68.57,-72.86,68.57,72.86,-
68.57,72.86 Type="Poly" />
<Icon Name="rect3758" Position="389.12,252.86" Size="128.57,134" Angle="-32.21"
Colour="23,255,64,192" />
<Icon Name="Crate1" Background="crate" Position="-398.2,281.15" Size="157,142"
Angle="48.27" OnTapped="Tapped" OnBeginTouch="TouchBegan"
OnEndTouch="TouchEnded" OnCreate="Created" />
<Geometry Name="rect3000" Vertices="30.48,-56.73,5.11,25.02,64.31,60.26,91.08,-
20.09,-85.1,-38.42,-73.82,50.39,-14.62,25.02,-17.44,-45.47 Type="QuadList" />
<Geometry Name="rect3761" Vertices="-79.37,-96.03,4.95,-80.62,200.58,56.78,-
19.5,116.77,-106.65,3.09 Type="Poly" />
</Scene>
```

Note that the Precision parameter specifies how many digits to round coordinates up to.

The SVG converter will convert the following SVG objects:

- svg is converted to Scene and will convert attributes as shown below:
    - id – This becomes scene name
    - width, height – These become the scenes CanvasSize
- group is not used except for the layer attribute which is applied to all actors within that group and the transform
- Rects will be exported as physics shapes and will convert attributes as shown below:
    - id – This becomes physics shapes name
    - width, height – These become the physics shapes width and height
    - label (Inkscape SVG format only) – If label is set to "circle" then the shape will exported as a circle physics shape with the radius set to half the width of the rect
- Images will be converted to Icon actors which supports the following attributes:
    - id – This becomes the actors name
    - x, y – These become Position
    - width, height – These become Size
    - transform – These become Position, Scale and Angle (does not support transform hierarchies)
    - style.colour – This becomes Colour
    - onclick – Becomes OnTapped event handler

  - onmousedown – Becomes OnBeginTouch event handler
  - onmouseup – Becomes OnEndTouch event handler
  - onload – Becomes OnCreate event handler
  - description – This is output directly within the tag, which allows you to add your own custom attributes
- Paths will be converted to geometries which supports the following attributes:
  - id - This becomes the geometries name
  - label (Inkscape SVG format only) – If label is set to "shape" then the geometry will exported as a physics shape instead
  - Path commands are converted to coordinates. If "Absolute Paths" is checked then paths will be exported with absolute coordinates, otherwise relative coordinates will be used

These features enable you to use SVG editors such as Inkscape as game layout editors to lay out levels and design complex geometry.

There are a few tips you should follow when using the likes of Inkscape to create XOML data:

- Create separate shape and geometry SVG files or place them on separate layers
- When creating geometry to be exported as specific types such as quad or tri list, create the geometry using individual quads or tris then use Path->Combine to combine them into a single geometry

Coming Soon

Add more ad provider integrations
Other
  Sending Mail
  Facebook posting

Glossary

App - An application that the user can interact with
App Frame - See Game Frame
Action - An action is some process or collection of processes that need to be carried out in response to an event occurring
Actor – An actor is the name given to any app or game object that lives ina scene. An actor generally has a specific purpose such as acting as a text label or an image etc..
Aspect Ratio - Refers to the ratio of a screens height to its width. For example if the devices screen size is 800x400 pixels then the aspect ratio is 2:1 (read as two to one) or 2.0
Asset - Refers to any component of an application or game, this could be anything from image and audio files to XOML files and Lua scripts
Binding – A binding is the connection of a variable to the property of an actor or scene. When the variable changes the property of the actor scene will also change
Collision – A collision is what happens when two or more actors that are under control of the physics system touch. Collisions are usually split into collision started and collision ended events
Command – A command is a single unit of functionality such as setting a variables value or waiting a specified amount of time
Frame - Refers to a particular value of an animation at a specific point in time.
Game Frame - Most games and apps are updated 30 to 60 times per second. A single update is called a game or app frame
Event - An event is something that happens within an app that you or the user need to know about and act upon
Font - A true type font that specifies how text should be look on screen
IDE - Integrated development environment - This refers to a software package that is used to develop code for native app development.
Image - A bitmap or file containing a bitmap in a specific format
Instantiate - In a XOML sense this is the process of creating something usually an actor or scene so that it appears in the XOML app. Instantiation usually refers to creating an object or collection of objects from a XOML template
Key Frame - See Frame
Lua – Lua is a small and efficient programming language that XOML uses to allow complex app and game development. Lua is an interpreted language and no compilation is required
Mask – A mask is the term used for the data use in bitwise operations such as AND, OR, XOR etc. XOML uses masks to allow grouping of objects, particularly to determine which groups of objects are allowed to collide with other groups of objects
Modifier – A modifier is a unit of functionality that can be added to basic actors and scenes to enhance its functionality and modify its default behaviour

Physics - Refers to the simulation of real world physical behaviour by actors
Program – A program is a list of commands that are executed sequentially
Resource - A resource is a file or data that can be used by the app to provide a better user experience by way of visual, audible and logical enhancements
Scene – A scene is a container that holds game and app actor objects
Script – A script is a program using a specific programming language such as Lua
Script Function – A script function is a single function within a script that can be called by XOML. Script functions usually carry out some kind of complex logic that would be quite difficult to achive using pure XOML
SDK - Software Developer Kit - This refers to a collection of code libraries, tools and documentation using specific programming languages that aid developers to create apps and games
Sound Effect - A sound effect refers to a sound file that is played back in response to certain events happening in the app or game
UI - Refers generally to user interface, which is a visual interface that allows the user to interact with an app or game
URL - Uniform resource locator used to refer to a resource on the internet
Variable – A variable is a place where some type of data is stored. Variables in XOML have specific types
XOML - XOML is an XML based mark-up language that is used by the AppEasy development system to mark-up games rapidly