FALL FURY

A DirectX/C++/XAML Tutorial

Development:	Den Delimarsky
PM & Testing:	Clint Rutkas, Dan Fernandez
Code Review:	Brian Peek
Design:	Arturo Toledo, Rick Barraza
Audio:	David Walliman

CODING4FUN.com

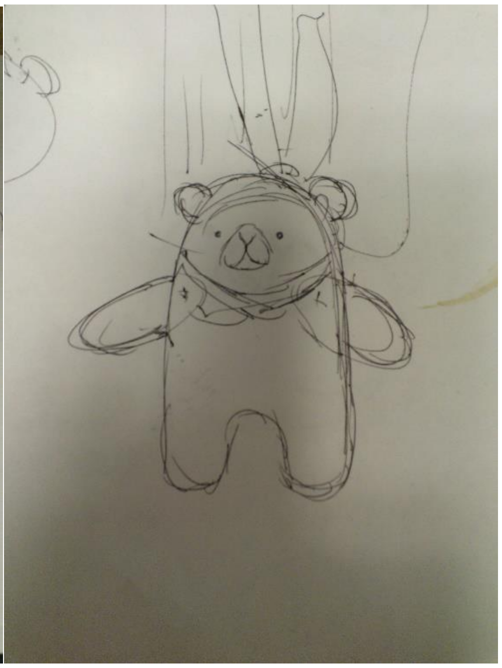
Chapter 1 – Introduction

In late May, I arrived in Redmond to work as an intern on the Channel9 team. I had the freedom to choose what I was going to work on, so I decided to challenge myself and work outside my comfort zone, utilizing less C# and managed code and more C++ and DirectX. To do so, I decided to highlight the capabilities of Windows 8—that's how FallFury was born.

FallFury is a 2D platformer in which the player controls a falling bear, trying to avoid obstacles, dodge missiles, and destroy monsters as the bear falls. The project incorporates several of the new Windows 8 APIs, including the accelerometer and touch as well as integrations with core OS capabilities such as settings and share charms. Additionally, the project leverages the most exacting addition to the Visual Studio development environment—hybrid application development with XAML, C++, and DirectX.

Design & Idea

From the outset, Rick Barraza and I decided that since our target audience was composed of both kids and adults, the main character had to be familiar to both groups. Teddy bears turned out to be the best choice. Rick spent a day creating tens of potential bear drawings—out of which I had to choose one:



As the bear's fall progresses, the character encounters a variety of obstacles dependent on the level type and complexity. Those obstacles should of course be avoided, so as the user tilts the device, the character in the game moves in the associated direction.

The design of the project took a week, and during this time the following items were determined and conceptualized. Considerations for both tablet and desktop environments directed our decisions:

- The main character layout.
- The way the game progresses as the character falls down.
- How the user interacts with the game in a wide variety of possible scenarios.
- What the game screens look like.
- What the menu system interaction looks like.
- How the game looks in different screen modes and on different device types.
- Some of the bonuses that the main character can pick up during free fall.
- What happens when the user progresses through the game and iterates through levels.
- What is shared and how this is accomplished.

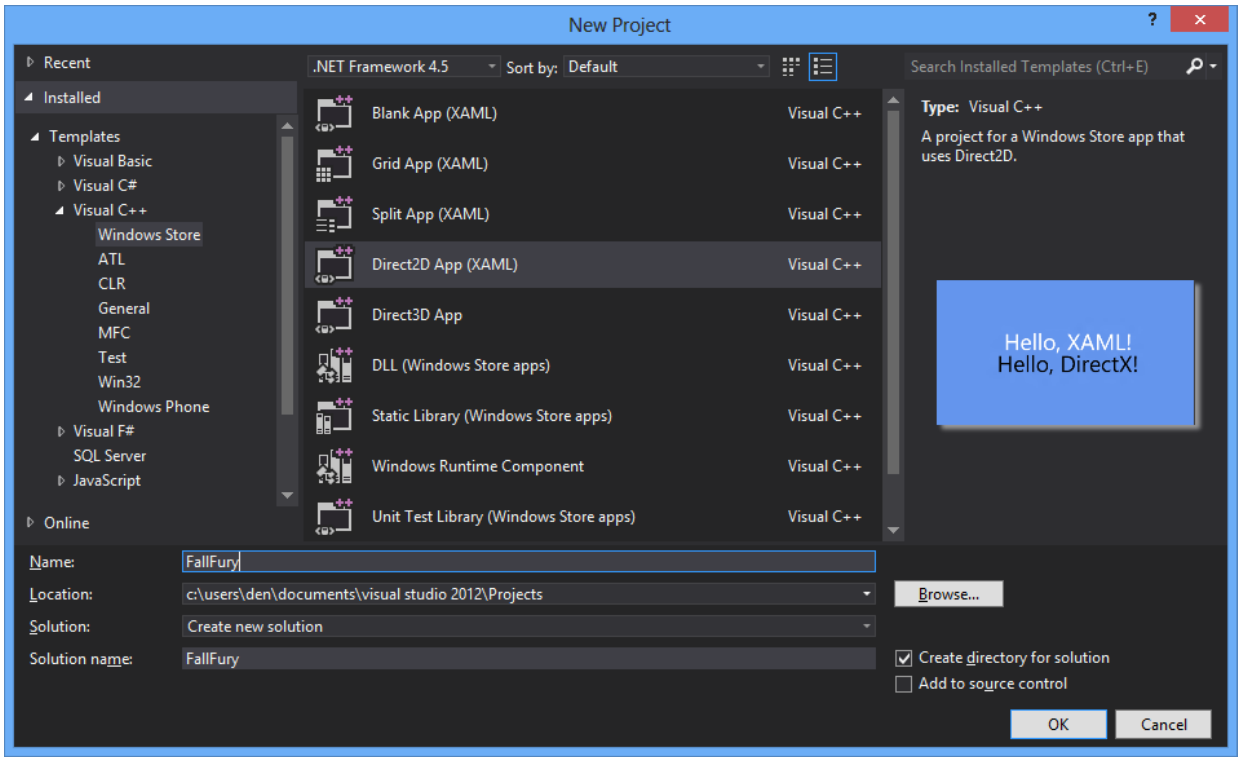
As the game ideas were outlined, Arturo Toledo, the designer behind ux.artu.tv, was brought on to create the game assets. Then the core design decisions were made and we jumped into the development process.

Beginning the development

You will need to download and install Microsoft Visual Studio 2012 Express for Windows 8 in order to be able to follow the steps that I am describing in this series. The development has to be done on Windows 8, because the end result is a Windows Store application that relies on Windows 8 APIs. Though specific hardware is not required, both ARM and x86 devices will work well, so whether you have a Microsoft Surface RT or Samsung Series 7 slate, you will be able to test the code when you have the opportunity.

To get started, open Visual Studio 2012 and select the C++ project types. You will notice that there are several options you can choose from. You want to create a Windows Store application, so choose the appropriate category. Windows Store applications run in a sandbox, outside the boundaries of the standard .NET runtime.

Next, select the **Direct2D (XAML)** app type from the project list. This is a new application type introduced in Visual Studio 2012 that allows developers to combine native DirectX graphics with XAML overlays. Do not be confused by the fact that there is a Direct2D in the name—you can still invoke DirectX capabilities supported in the WinRT sandbox:


New Project
Recent
Installed
Templates
Visual Basic
Visual C#
Visual C++
Windows Store
ATL
CLR
General
MFC
Test
Win32
Windows Phone
Visual F#
SQL Server
JavaScript
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Blank App (XAML) Visual C++
Grid App (XAML) Visual C++
Split App (XAML) Visual C++
Direct2D App (XAML) Visual C++
Direct3D App Visual C++
DLL (Windows Store apps) Visual C++
Static Library (Windows Store apps) Visual C++
Windows Runtime Component Visual C++
Unit Test Library (Windows Store apps) Visual C++
Hello, XAML!
Hello, DirectX!
Browse...
OK
Cancel

Type: Visual C++

A project for a Windows Store app that uses Direct2D.

Name: FallFury

Location: c:\users\den\documents\visual studio 2012\Projects

Solution: Create new solution

Solution name: FallFury

☑ Create directory for solution

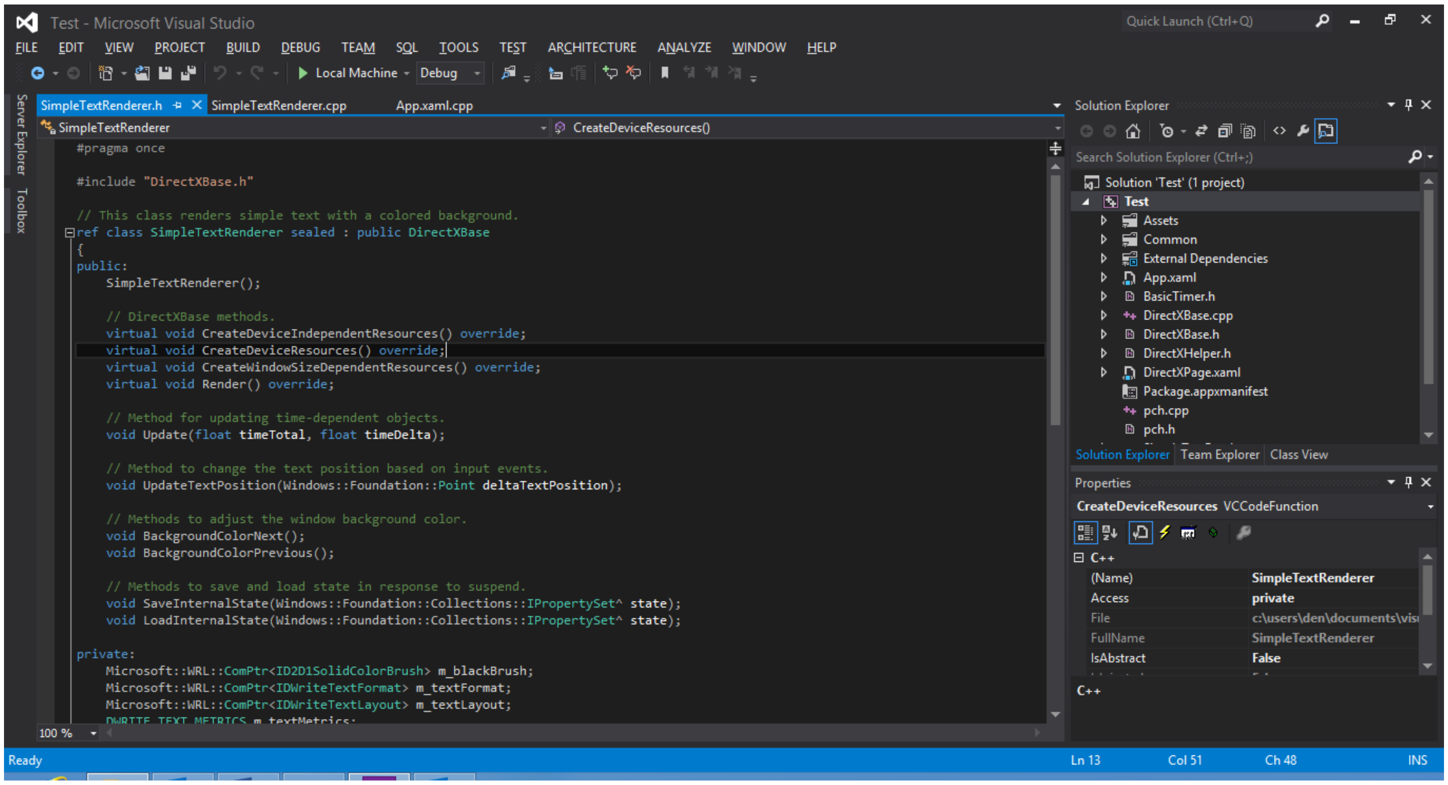
☐ Add to source control

At this point you might be wondering, why choose a hybrid application instead of a fully-native project? The reason behind this decision is that is allows the developer to focus more on fine-tuning the gameplay instead of creating the UI core in pure DirectX. Because XAML is a part of the game, we can create dynamic UI layouts for the HUD, settings charm and menus without touching the graphical backbone. It is possible to do the same directly through DirectX and was a perfect approach for FallFury, considering the time constraints and the fact that I needed minimal UI overlays. Most of the graphics were already processed through the DirectX pipeline, so I did not have to invest significant resources and time into designing low-level structures for the interactivity layer.

FallFury uses XAML for the following:

- **Menus** – depending on the screen, the user is able to trigger a number of actions. For example, when the game starts, the user might select the New Game option or decide to take a look at the About screen. In Paused mode, the menu is used to resume the game, adjust settings, or possibly skip a level.
- **Game HUD** (score indicator, pick-up indicator, health indicator) – during the gameplay, the user is interested in keeping track of where the character is and what is the state of it. The game HUD is shown in active game mode.
- **Settings charm extensions** – the way the Settings charm works, the OS provides the core harness to hook to the Settings popup. Once shown, it is up to the developer to provide a multitude of options that customize the application behavior; any additional popups shown on selection should be designed individually in XAML.
- **User notification** – when something happens that can potentially affect the gameplay, the user should be alerted. The core framework provides the capabilities to use a MessageDialog, but in some cases it might not be enough. For example, if new levels are available for download, the user might want to check those out in a custom popup including full previews rather than just text.

When you create the project, a default infrastructure that prints text on the screen—both through Direct2D and XAML—is available. Direct2D is a subset of DirectX APIs and facilitates hardware-accelerated 2D graphics processing. It is used to create basic geometry elements and text. Since I am here working mostly with XAML, I am not going to cover Direct2D in-depth in this article:



```
#pragma once

#include "DirectXBase.h"

// This class renders simple text with a colored background.
ref class SimpleTextRenderer sealed : public DirectXBase
{
public:
    SimpleTextRenderer();

    // DirectXBase methods.
    virtual void CreateDeviceIndependentResources() override;
    virtual void CreateDeviceResources() override;
    virtual void CreateWindowSizeDependentResources() override;
    virtual void Render() override;

    // Method for updating time-dependent objects.
    void Update(float timeTotal, float timeDelta);

    // Method to change the text position based on input events.
    void UpdateTextPosition(Windows::Foundation::Point deltaTextPosition);

    // Methods to adjust the window background color.
    void BackgroundColorNext();
    void BackgroundColorPrevious();

    // Methods to save and load state in response to suspend.
    void SaveInternalState(Windows::Foundation::Collections::IPropertySet^ state);
    void LoadInternalState(Windows::Foundation::Collections::IPropertySet^ state);

private:
    Microsoft::WRL::ComPtr<ID2D1SolidColorBrush> m_blackBrush;
    Microsoft::WRL::ComPtr<IDWriteTextFormat> m_textFormat;
    Microsoft::WRL::ComPtr<IDWriteTextLayout> m_textLayout;
    DWRITE_TEXT_METRICS m_textMetrics;
```

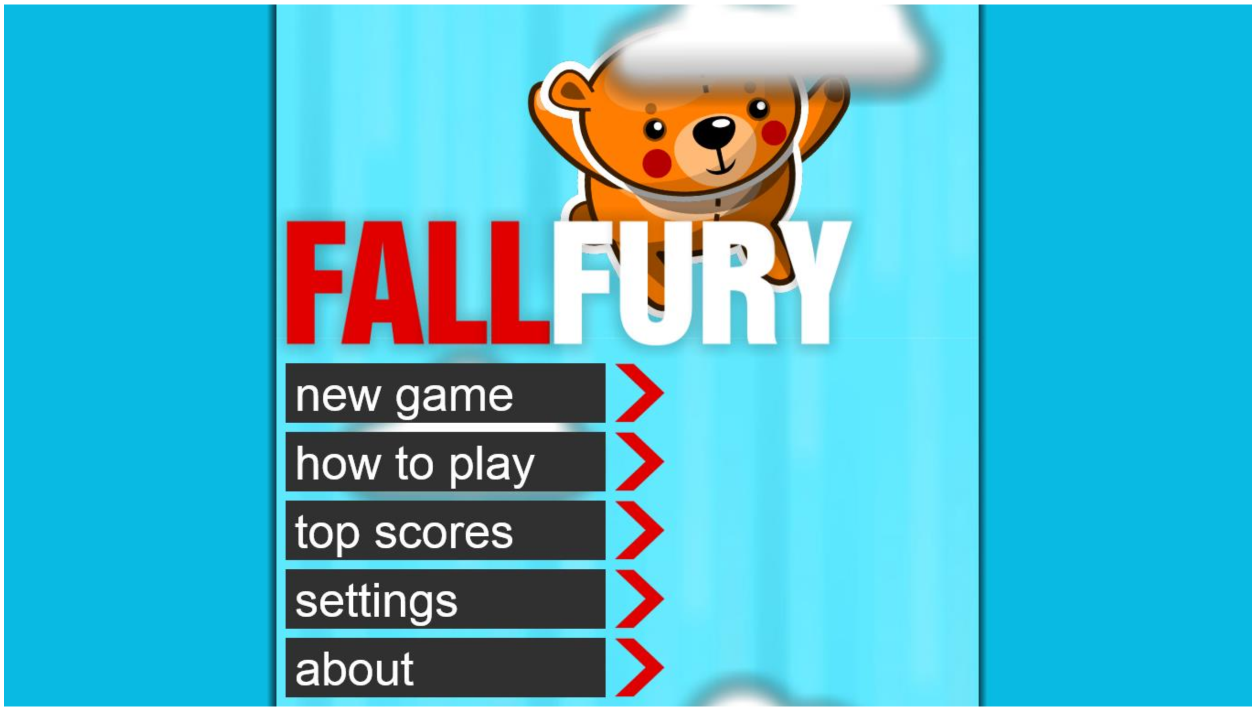
(Name)	SimpleTextRenderer
Access	private
File	c:\users\den\documents\vis
FullName	SimpleTextRenderer
IsAbstract	False

I will go into more details regarding the XAML and DirectX interaction model later in the series, but for now, take a look at which parts of the project you have available. First and foremost, you probably notice the combination of both C++ source and header files and **DirectXPage.xaml**. Starting with Visual Studio 2012, you are now able to create XAML applications in C++. So even if you are an experienced C++ developer who never worked with the Extensible Application Markup Language, you can create the product core in your familiar environment and either import existing XAML structures or delegate the XAML writing to a designer.

If you open the XAML page, you will notice one significant change that you haven't experienced in standard XAML applications, such as WPF or Silverlight for Windows Phone:

```
<SwapChainBackgroundPanel x:Name="SwapChainPanel" PointerMoved="OnPointerMoved"
                          PointerReleased="OnPointerReleased">
    <TextBlock x:Name="SimpleTextBlock" HorizontalAlignment="Center" FontSize="42" Height="72"
               Text="Hello, XAML!" Margin="0,0,0,50"/>
</SwapChainBackgroundPanel>
```

A SwapChainBackgroundPanel is a component that lets the developer overlay XAML on top of the core DirectX-based experience. Look, for example, at the FallFury main menu as the game runs in landscape mode:



The menu buttons, the label and the side curtains are designed and rendered entirely in XAML. The clouds in the background, as well as the teddy bear, are rendered directly through the DirectX stack. The end-user does not notice any difference in the way these elements interact or are displayed. From a development perspective, however, there are several conditions that must be met.

There can only be one instance of SwapChainBackgroundPanel per app. Therefore, you can have only one overlaid XAML controls set. This does not mean that you can't have multiple controls, but it implies that to do so you have to implement a control management flow that handles content adaptation. For example, if I invoke the pause state in the game, I don't show the HUD but rather the screen-specific controls that let me resume or abandon the game and the PAUSE label. This switch needs to be handled on both the DirectX and XAML because a state change affects what is shown on the screen and what behaviors are tracked. As you will see through this series, this is not too hard to implement with a helper class that will store the global game state, that can be accessed from anywhere in the game.

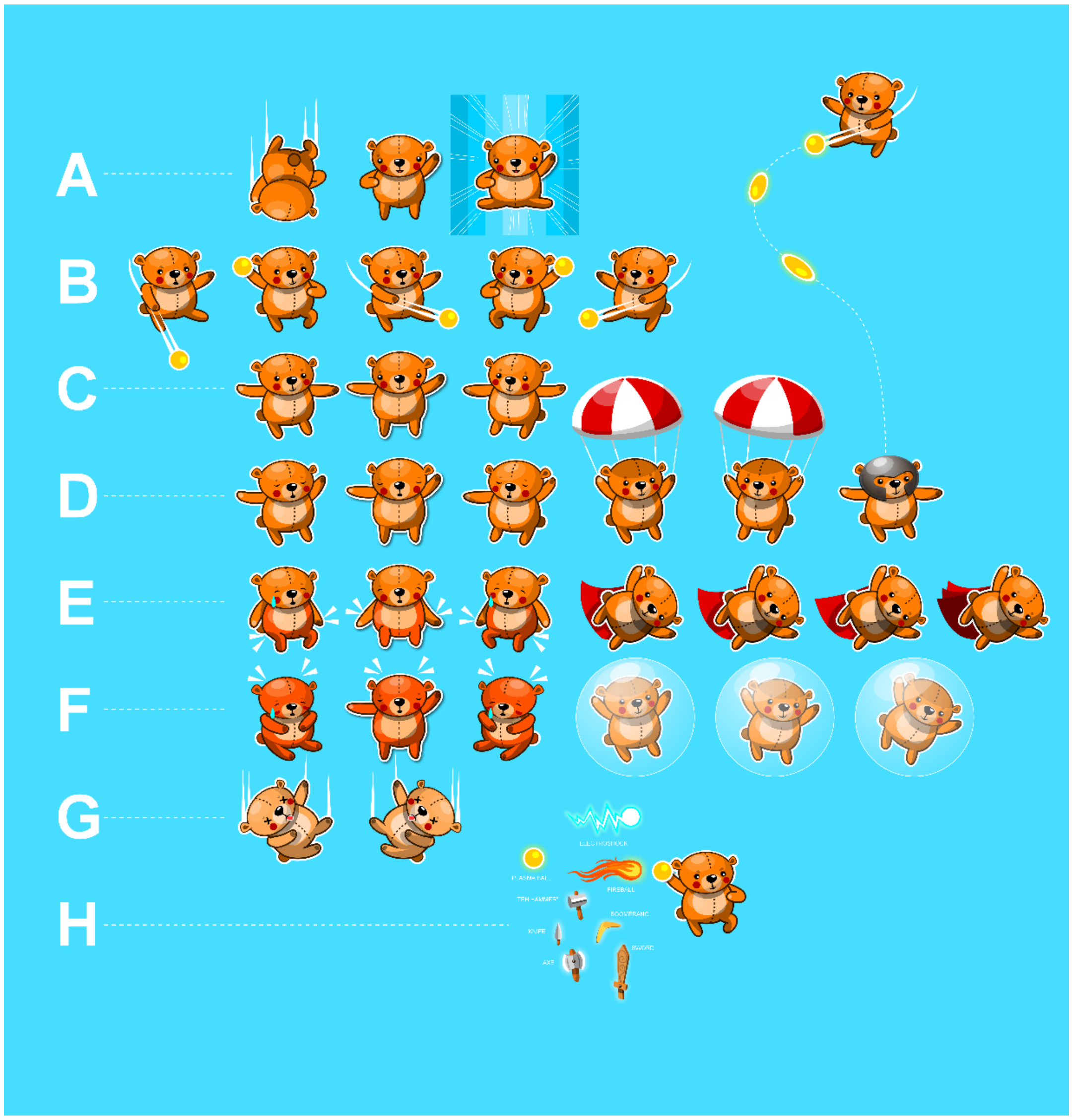
When using SwapChainBackgroundPanel, remember that XAML is in all cases overlaid on top of the DirectX renders. So, no matter what controls you are using, those will always be placed on top of what DirectX shows to the user. For more details about how DirectX and XAML interoperate, check out this MSDN article.

Assets for the game

As with any other game, there is not only code involved in production—there are also sound and graphical assets that create a unique experience for the user. FallFury includes a wide variety of graphical assets designed by Toledo Design as well as audio created by David Walliman.

It is important that all graphical resource requirements are established at the very beginning of development. As I mentioned in the Design section of this article, I had to put together a list of all the game screens, power-ups, obstacles, backgrounds, character states and possible particles that were generated from a texture. That way, when the designer started creating the assets, all components blended together well and their styles were compatible with the vision of the game.

While working on the assets, Arturo Toledo created multiple variations of the same set up that showed how assets integrate in different game conditions:



As you follow this series, you will not have to create your own game assets—we at Coding4Fun decided to provide all graphical and audio assets, which you can download at http://fallfury.codeplex.com. We not only provided you with the final PNG and DDS files, but also with the raw assets that can be used in Microsoft Expression Design and Adobe Illustrator. Let's take a look at what is in the package.

You will notice that the project Assets folder is split in several subfolders. All these assets are used mostly in the XAML layer or in game conditions where texture/sound processing is not necessary. There is also an additional asset folder I will discuss later.

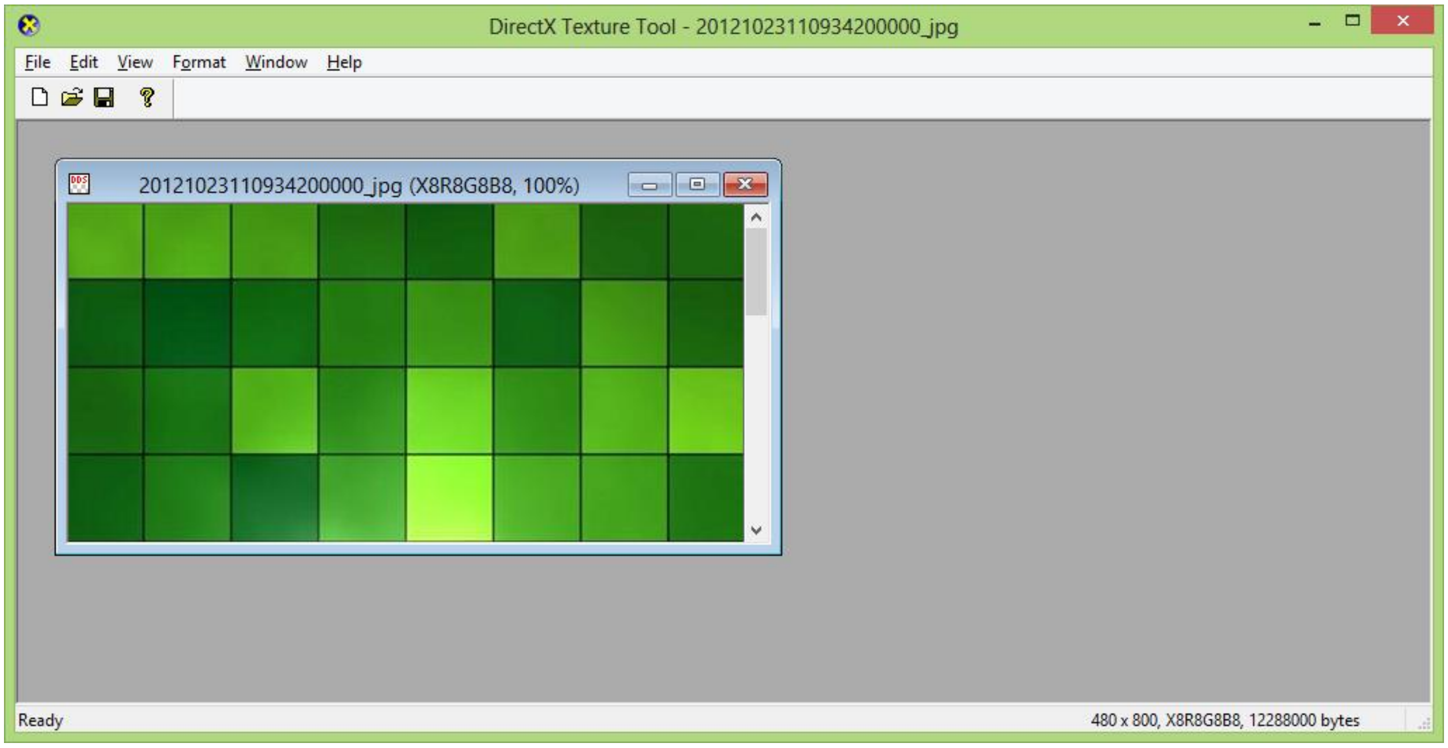
- Backgrounds –the level backgrounds, such as the blue, purple or red sky, as well as the overlays, such as clouds that move simultaneously with the backgrounds. Each background/overlay combination is assigned to a specific level type.
- Bear – some of the bear elements that are displayed at different times in the game, such as when the game is over or when the player wins the entire level set.
- HUD – basic elements that are displayed during the game.

- Icons – the application icons, in a variety of sizes, required for a Windows Store application.
- Medals – winning players are awarded a medal. It can be golden, silver, or bronze.
- Misc – you get some branding elements as well as some graphics that are used in combination with other game items, such as particles.
- Monsters – monsters are used in the "How To" part of the game.
- Music – the long tracks used in different levels and screens.
- Sounds – short sound clips used when different game behaviors in the game are triggered. For example, when the bear gets hit, he lets out a brief cry.

All graphical assets mentioned here are PNG resources. The way I structured the game, some elements are larger than the others and I needed to take some measures to cut down on the package size. PNGs are fairly well-sized without much quality loss compared to raw images. At the beginning of the development process, all graphics were stored in DDS files.

DirectDraw Surface, or DDS, is a file format developed by Microsoft that helps developers optimize some of the graphics by avoiding re-compression performance loss. One of the benefits of using DDS files is the fact that whenever they are processed by the GPU, the amount of memory taken by them is the same as the file size of the DDS file itself. Usually, for 2D games the compression-based performance loss is not necessarily noticeable. For PNG files used on the DirectX stack, resources are allocated to decompress the texture. Not so for DDS files.

Depending on the case, DDS files can be generated on the fly. FallFury preserves relatively small textures in DDS format and larger ones, such as backgrounds, in PNG. It's the best of both worlds. DDS files can be generated by a tool bundled with the DirectX SDK called **dxtex** (usually located in *C:\Program Files (x86)\Microsoft DirectX SDK (June 2010)\Utilities\bin\x64*):


DirectX Texture Tool - 20121023110934200000_jpg
File Edit View Format Window Help
20121023110934200000_jpg (X8R8G8B8, 100%)
Ready
480 x 800, X8R8G8B8, 12288000 bytes

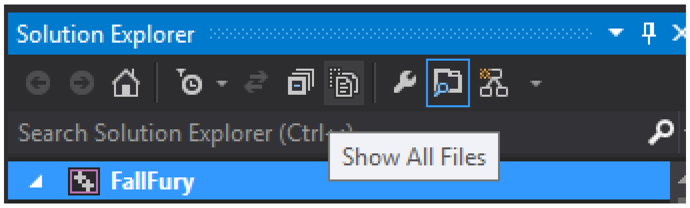
There is also **texconv** that allows you to create DDS components from the command line, but we'll explore that later in the series.

All DDS assets are located in the DDS folder in the solution. All that's there are assets related to levels, such as obstacles, and assets related to character behavior, such as bear states and used weapons and powerups.

Unlike graphic assets that can be mocked from the very start, audio assets are a bit harder to come up with, mainly because at that point you need to be sure what the game will be like at the end. Music and effects should go together with the overall game feel. There are two types of audio components in FallFury—the music and the action-based sound effects. Music is played constantly, whether in the game or on a game screen such as the main menu, unless disabled by the user.

The way the audio engine works in FallFury, files are handled differently depending on where in the game they are used. All music is stored in MP3 files and the short sound effects are stored in uncompressed WAV files.

As a part of the new project that you are creating, replicate the folder structure for the Assets and DDS folders and add them to the solution. The way C++ project references work, you might want to switch to the "Show All Files" mode in the Solution Explorer:



Then, simply copy the folders to the solution folder itself. That way you will have all these resources as a part of the project itself and not just links to external files.

Source Control

It's a really good idea to use source control. There are multiple options available at no cost, such as CodePlex, Assembla, and Team Foundation Service. You need source control for multiple reasons. The most important reason of them all, however, is that code will break. There were multiple situations where I changed parts of the project and all of a sudden some components stopped working. With the project hooked to a source control system, all I needed was to do a quick rollback to the previous check-in and I was good to go.

A good practice I learned from Clint Rutkas is performing atomic check-ins. When something goes wrong, it is much easier to go back to the check-in where only 20 or 30 code lines were modified from what is currently in the stack, compared to going back to the solution where you will be missing two or more entire source files.

Modular Design & Prototyping

As you are following this project creation from scratch, notice how the entire code base is modular and interchangeable. If I decide to create a new game screen, I can do so easily by inheriting from an existing base class that provides the basic harness. If I want to replace a character model, I can do so by modifying a single class without breaking the entire interaction model.

Prototyping is also a big part of FallFury and it is key that no time is wasted working on features that will have to be entirely replaced or re-written. A good example to this scenario happened just as I started writing the project code. I noticed that the Settings charm required some XAML work for secondary popups that extended from the

OS-invoked layer. As I was working on a pure DirectX application, I had to create the XAML in code-behind and that was one of the experiences that could've been avoided in a hybrid application. During the prototyping stage it is easy to spot potential integration problems and later change parts of the project to work better together. It is much more complicated to replace project components when the core is wired-in than when you have small parts that independently show how a part of the game works.

Hardware

As you are about to start writing large amounts of code, you are probably wondering whether actual hardware is needed to test the application. The way FallFury was designed, you will be able to run it on any Windows 8 compatible machine, whether a desktop computer or a tablet. If the game is being run on the desktop, I assume that you probably do not have access to an accelerometer or a touch display. If you are running the game on the tablet, you probably do not have a physical keyboard constantly attached to it. These facts ultimately affect how you experience the game itself. From a development perspective, it is important to have multiple types of target hardware available, because the game might behave differently depending on the device configuration. But it is not required to follow this article series. As long as you have a Windows 8 machine, you are good to go.

Chapter 2 – Shaders

FallFury not only relies on standard C++/C# and XAML code, but also on shaders. This article is intended for developers who are not aware of what shaders are and want to know how to use them in their projects. I will talk about creating shaders, as well as the shaders used in my project.

Code on GPU

In simple terms, shaders are small programs that are executed on the Graphical Processing Unit (GPU) instead of the Central Processing Unit (CPU). In recent years, we've seen a major spike in the capabilities of graphic devices, allowing hardware manufacturers to design an execution layer tied to the GPU, therefore being able to target device-specific manipulations to a highly optimized unit. Shaders are not used for simple calculations, but rather for image processing. For example, a shader can be used to adjust image lighting or colors.

Modern GPUs give access to the rendering pipeline that allows developers to execute arbitrary image processing code. This is a step forward from a fixed-function pipeline that was present in older GPUs, where image processing tasks were integrated into the hardware unit and were only able to perform a limited set of actions, such as transforms. Unlike the rendering pipeline, the fixed-function pipeline was not programmable, therefore the developers were often tied to the hardware they used for offering specific game effects, in some cases having to resort to software-based adjustments.

Types of Shaders

There are different types of image manipulations that can be performed on a given input, and so there are different types of shaders designed to handle that. Currently, we can highlight three main shader types:

- Vertex shaders – because what the user sees on the screen is not really three-dimensional, but rather a three-dimensional simulation in a 2D space, vertex shaders translate the coordinates of a vector in 3D space in relation to the 2D frame. Vertex shaders are executed one time per vector passed to the GPU. Typically, vectors carry data related to their position and the coordinate of the bound texture as well as color. A vertex shader is able to manipulate all these properties, but there is never a situation where a new vertex is created as a result of the execution.
- Pixel shaders – these programs are executed on the GPU in relation to every passed pixel, working on a much lower level. For example, if you want specific pixels adjusted for lighting or 3D bump mapping, a pixel shader can provide the desired effect for a surface. Rarely, there are situations where only a few

pixels should be re-adjusted at once. Pixel shaders are often run with an input of millions of pixels, resulting in complex effects.

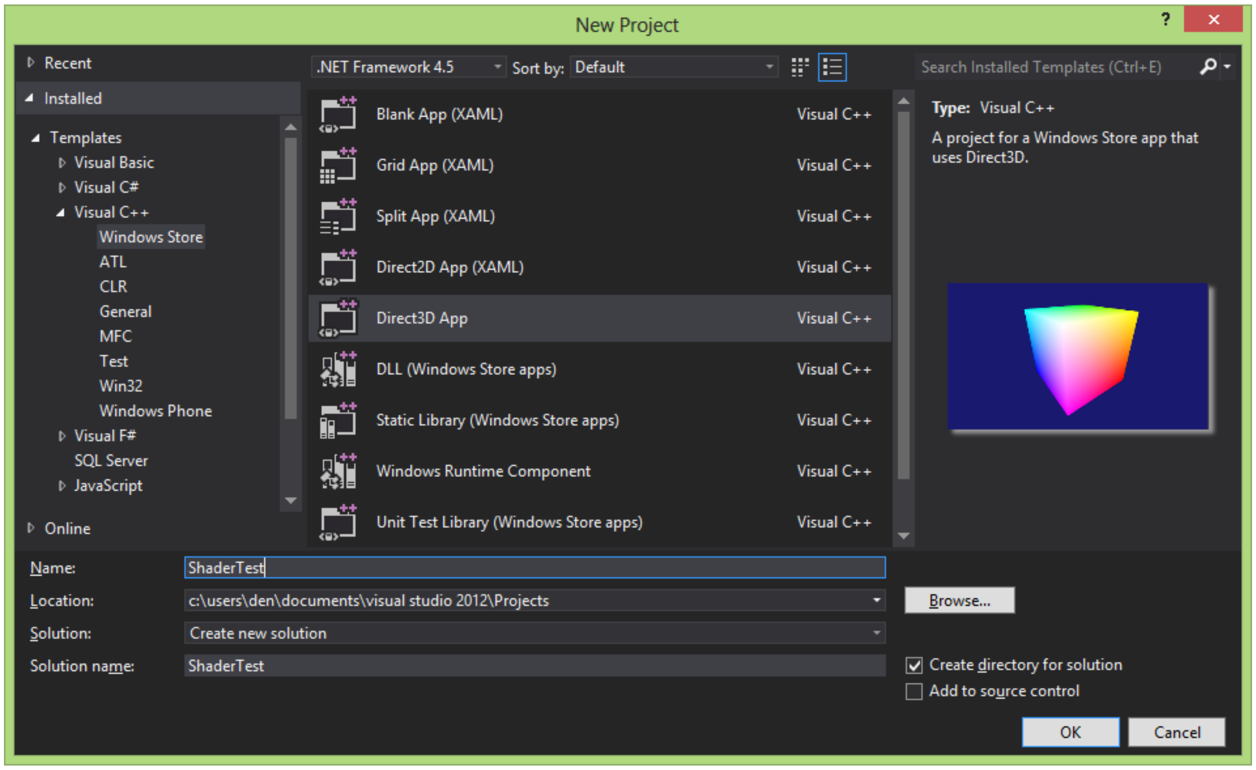
- Geometry shaders – these shaders are the next progression from vertex shaders, introduced with DirectX 10. The developer is able to pass specific primitives as input and either have the output represent the modified version of what was passed to the program or have new primitives, such as triangles, be generated as a result. Geometry shaders are always executed on post-vertex processing in the rendering pipeline. When vertex shader execution is completed, geometry shaders step in, if present. Geometry shaders can be used to refine the level of detail of a specific object. For example, when an object is closer or further away from the viewport camera, the mesh would have to be refined to minimize the rendering load.

FallFury uses all these shader types as a part of the game.

Enough Talk, Let's Code

Now that you are aware of what shaders are, let's write some sample code and test it. It is worth mentioning that shaders are not written in a standard high-level language, but rather in a language defined by the environment the shader is used in. For Direct3D, this is the High-Level Shader Language, or HLSL. It is somewhat similar to C, but with some specific nuances.

Start by creating a test Direct3D Windows Store application:


New Project

Recent
Installed
Templates
Visual Basic
Visual C#
Visual C++
Windows Store
ATL
CLR
General
MFC
Test
Win32
Windows Phone
Visual F#
SQL Server
JavaScript
Online

.NET Framework 4.5 Sort by: Default

Search Installed Templates (Ctrl+E)

Blank App (XAML)	Visual C++
Grid App (XAML)	Visual C++
Split App (XAML)	Visual C++
Direct2D App (XAML)	Visual C++
Direct3D App	Visual C++
DLL (Windows Store apps)	Visual C++
Static Library (Windows Store apps)	Visual C++
Windows Runtime Component	Visual C++
Unit Test Library (Windows Store apps)	Visual C++

Type: Visual C++

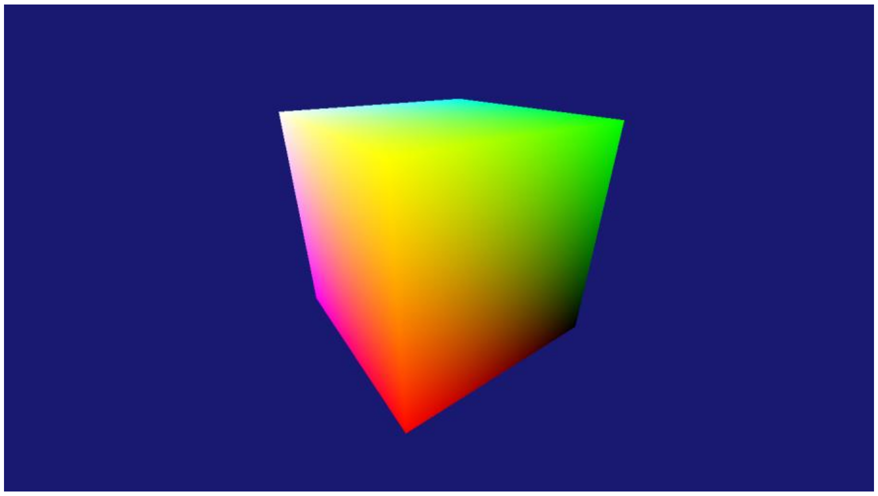
A project for a Windows Store app that uses Direct3D.

Name: ShaderTest
Location: c:\users\den\documents\visual studio 2012\Projects
Solution: Create new solution
Solution name: ShaderTest

Browse...
[✓] Create directory for solution
[ ] Add to source control

OK Cancel

If you create and run this sample application, you will see that the output of it is a simple 3D spinning cube:



You probably also noticed that there are two shaders that are a component part of the newly created solution: **SimplePixelShader.hlsl** and **SimpleVertexShader.hlsl**. Take a look inside the pixel shader:

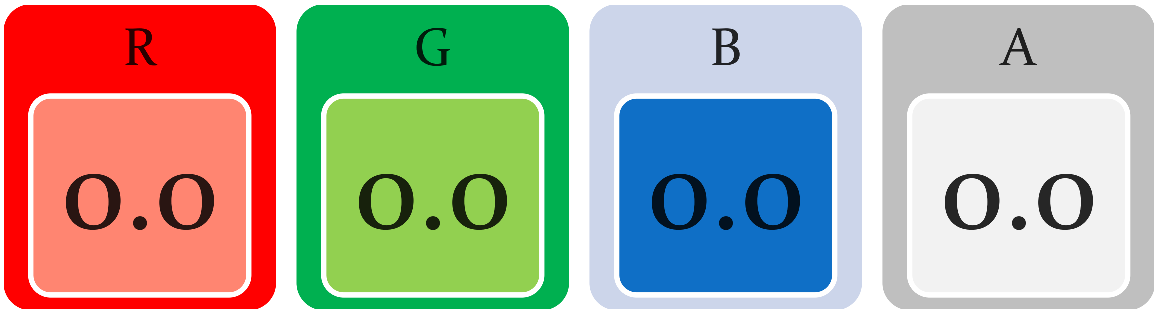
```
struct PixelShaderInput
{
    float4 pos : SV_POSITION;
    float3 color : COLOR0;
};

float4 main(PixelShaderInput input) : SV_TARGET
{
    return float4(input.color,1.0f);
}
```

First of all, there is a PixelShaderInput struct. It represents a single pixel – it has a float4 field that represents the position of the pixel and a float3 field that represents the RGB pixel color. The field itself is also marked by a predefined type, such as SV_POSITION or COLOR. This is a string that determines the use of the field. You can read more about the shader semantics here.

The strings that are prefixed with SV_ are representing system-value semantics. Those have special meaning in the pipeline during the processing stages. In the pixel shader above, SV_POSITION will always mean the pixel position.

Look at what is being returned from the pixel shader—instead of the standard float3 color indicator, you are now returning a float4, which couples the existing value, input.color, with a 1.0f float value that represents the alpha-channel value. Remember, that inside shaders the color is clamped between 0 and 1 instead of the standard 255-value limit.

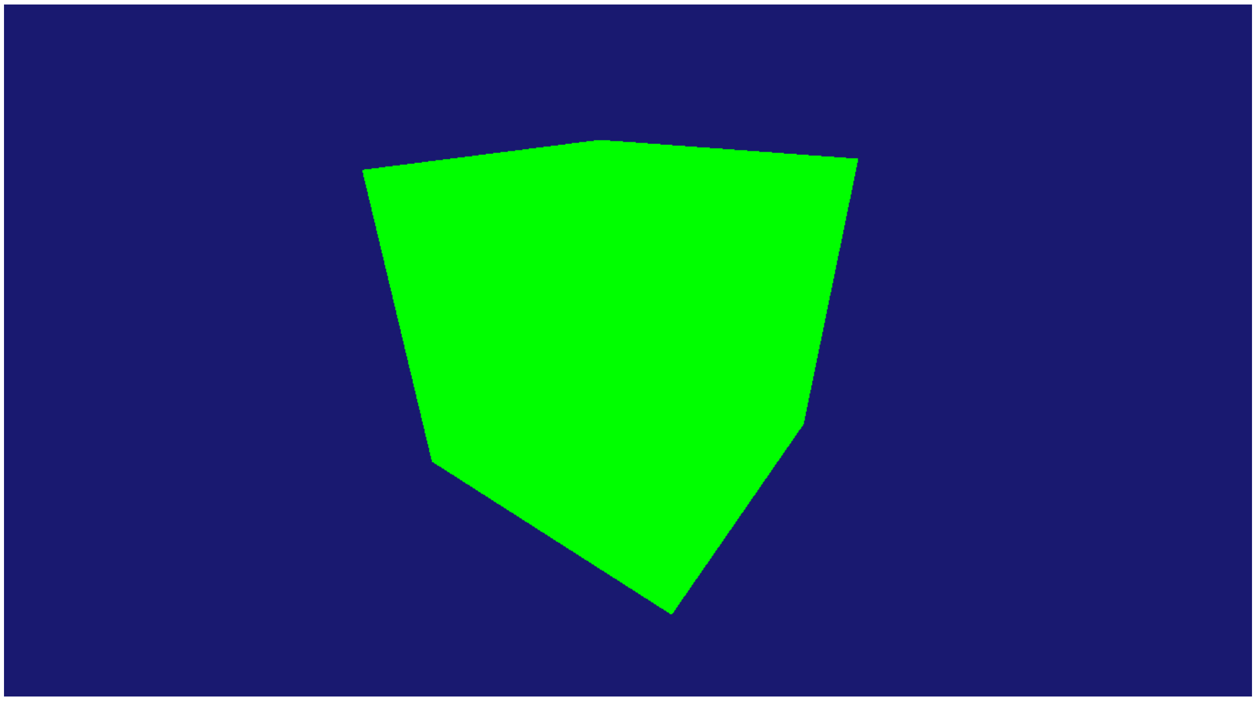
Since the cube rendering mechanism is already in place, let's experiment with the pixel shader a bit. You've already got the float4 color representation:



You can set any of these values to 1.0 to return the solid color, so let's do that. Let's render the cube green. Modify the return statement to be this:

```
return float4(0.0, 1.0, 0.0, 1.0);
```

At this point, if you will run the program you will get a rendering very similar to this:



This is a good start.. Now, let's now take a look at the vertex shader in the project. By default, you get this:

```
cbuffer ModelViewProjectionConstantBuffer : register(b0)
{
    matrix model;
    matrix view;
    matrix projection;
};

struct VertexShaderInput
{
    float3 pos : POSITION;
    float3 color : COLOR0;
};

struct VertexShaderOutput
{
    float4 pos : SV_POSITION;
    float3 color : COLOR0;
};

VertexShaderOutput main(VertexShaderInput input)
{
    VertexShaderOutput output;
    float4 pos = float4(input.pos, 1.0f);

    pos = mul(pos, model);
    pos = mul(pos, view);
    pos = mul(pos, projection);
    output.pos = pos;

    // Pass through the color without modification.
    output.color = input.color;

    return output;
}
```

There are a couple of differences here compared to the above mentioned pixel shader:

- **cbuffer ModelViewProjectionConstantBuffer** – represents a constant buffer that contains three matrices, to which vertices are related—the model, view and projection matrices. Constant buffers are interesting structures that have been optimized to allow block-wise updates, where multiple constants are grouped in one boxed unit and can be updated simultaneously instead of having a one-by-one iterative update cycle.
- **struct VertexShaderInput** – vertices are passed one-by-one to the shader, and this specific structure represents the input. Notice that it carries the 3D representation of the vertex position as well as its RGB color value.
- **struct VertexShaderOutput** – the final processed output, which now carries the position in relation to all three matrices mentioned above as well as the processed color. Notice that the fourth value is the camera distance in the demo.
- **VertexShaderOutput main(VertexShaderInput input)** – this is the main function that performs the vertex processing. Initially, it performs the position adjustment relative to the camera and then uses mul, a built-in function that multiplies matrices, to represent the current vertices in the projected space.

So let's play around with what we have in stock. Take a look at the main function and how initially, the float3-based position is transformed to contain the camera distance:

```
float4 pos = float4(input.pos, 1.0f);
```

This position is normalized in relation to the current viewport. Afterwards, the vertex is transformed in the current 3D space (relative to three matrices – model, view, and projection):

```
pos = mul(pos, model);
pos = mul(pos, view);
pos = mul(pos, projection);
output.pos = pos;
```

As with the pixel shader, these are some very basic manipulations. Let's flip the cube upside-down. To do this, we need to apply a rotation transformation. Here is an interesting piece of advice—when working with manipulations in shaders, make sure that you know basic matrix operations.

For example, to perform a simple 2D rotation, you need to use the rotation matrix:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

But since we're in 3D space, we not only have the X and Y coordinates, which relate to the matrix above, but also the Z coordinate. Therefore, a different method should be applied. There are three fundamental matrices that can be used to rotate an object around the three possible axes:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

For now, we want to rotate the cube along the X-axis. To do this, inside the main function, we need to declare a constant that represents the rotation angle, in radians:

```
const float angle = 1.3962634;
```

Looking at the matrix above, we need to find out the sine and cosine of the given angle. When working with vertex shaders, HLSL offers a built-in function, called sincos that is able to get both values from an input value:

```
float cosLength, sinLength;
sincos(angle, sinLength, cosLength);
```

Now you need to declare the rotation matrix. Again, HLSL comes with some built-in capabilities to declare matrices and set their values:

```
float3x3 xAxisRotation = { 1.0, 0.0, 0.0,                 // Row 1
                           0, cosLength, -sinLength,    // Row 2
                           0, sinLength, cosLength};    // Row 3
```

The standard format for matrix declaration in this case follows the following pattern:

```
Matrix<type,size> localName
```

For matrix multiplication, we can once again leverage the mul function:

```
float3 temporaryPosition;
temporaryPosition = mul(input.pos,xAxisRotation);
```

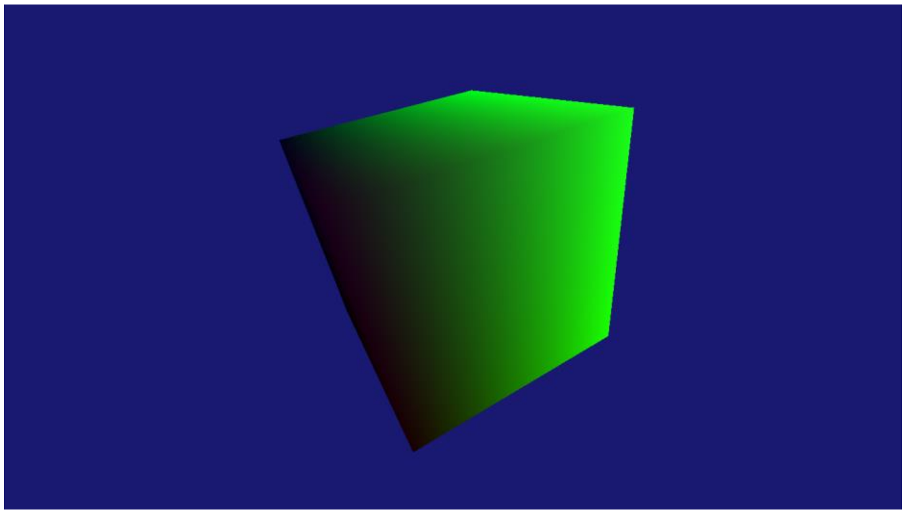
Now we can normalize the position to a float4 value that also includes the camera distance:

```
float4 pos = float4(temporaryPosition, 1.0f);
```

The rest of the transformation procedures can be taken directly from the original shader, multiplying the float4 position by the model, view, and projection matrices. Your entire main function should now look like this:

```
VertexShaderOutput main(VertexShaderInput input)
{
    VertexShaderOutput output;

    const float angle = 1.3962634;
    float cosLength, sinLength;
    sincos(angle, sinLength, cosLength);

    float3x3 xAxisRotation = { 1.0, 0.0, 0.0,                // Row 1
                               0, cosLength, -sinLength,   // Row 2
                               0, sinLength, cosLength};   // Row 3

    float3 temporaryPosition;
    temporaryPosition = mul(input.pos,xAxisRotation);

    float4 pos = float4(temporaryPosition, 1.0f);

    // Transform the vertex position into projected space.
    pos = mul(pos, model);
    pos = mul(pos, view);
    pos = mul(pos, projection);
    output.pos = pos;

    // Pass through the color without modification.
    output.color =  input.color;

    return output;
}
```

The current angle is set to 80 degrees, or 1.3962634 radians. If you run the rendering test application, the result you will get will be similar to this:



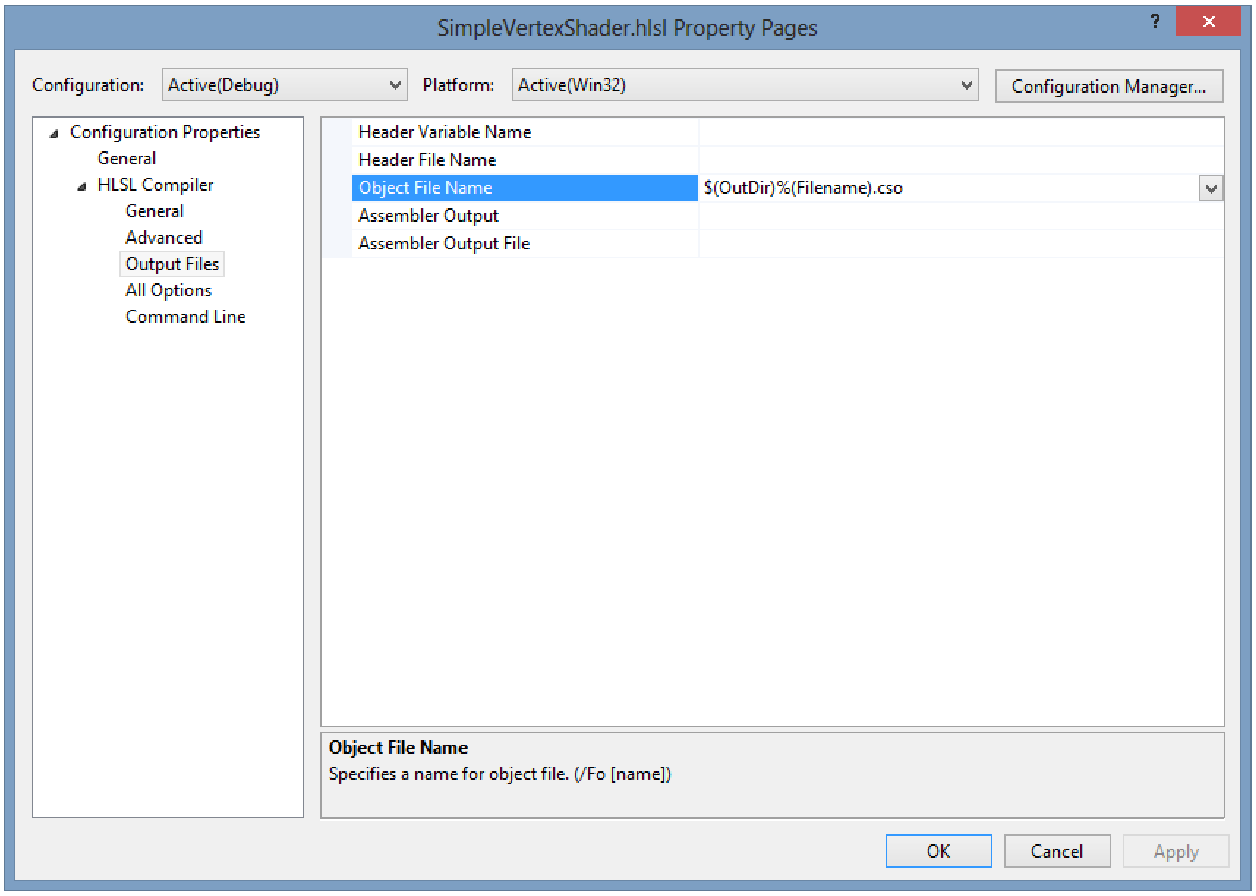
Now, let's look at how shaders are handled on the DirectX side.

Shaders in DirectX

Open **CubeRenderer.cpp**. This is the source file where internal shaders are read and passed to the device to be executed. Notice one interesting aspect of the process – the shader file contents are being read first and then passed to **m_d3dDevice->CreateVertexShader**. m_d3dDevice is a pointer to a virtual adapter that can be used to create device-specific resources. CreateVertexShader creates a vertex shader from an already compiled shader. You can notice it from the fact that the data is read not from the initial **.hlsl** file, but rather from the **.cso (Compiled Shader Object)**:

```
auto loadVSTask = DX::ReadDataAsync("SimpleVertexShader.cso");
```

As with any program, before it is passed onto the execution layer, it has to be compiled first. Visual Studio 2012 comes with a bundled HLSL compiler, **fxc**—the Effect Compiler Tool. The default behavior for a shader is to be compiled with the .cso file extension in the same output directory as the project, but this can be changed by setting the **Object File Name** in the shader properties:



Header Variable Name
Header File Name
Object File Name: $(OutDir)%(Filename).cso
Assembler Output
Assembler Output File

Object File Name
Specifies a name for object file. (/Fo [name])

You can read more about the shader compilation process here.

Looking back at the sample vertex shader that was created as a part of the project, you will notice that there is a specific input layout set for the incoming object:

```
struct VertexShaderInput
{
    float3 pos : POSITION;
    float3 color : COLOR0;
};
```

The virtual adapter is not aware of the structure layout. Therefore, the developer needs to explicitly create an internal D3D11_INPUT_ELEMENT_DESC array that will contain the type of information passed to the shader:

```
const D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,  D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "COLOR",    0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```

Once the description is complete, the input needs to be assembled for processing. That's where CreateInputLayout comes into play:

```
DX::ThrowIfFailed(
    m_d3dDevice->CreateInputLayout(
        vertexDesc,
        ARRAYSIZE(vertexDesc),
        fileData->Data,
        fileData->Length,
        &m_inputLayout
        )
    );
```

You are basically passing a shader signature to the input assembler that will perform all the consequent processing. A similar process is applied for the existing pixel shader, with the main difference being the fact that instead of CreateVertexShader, CreatePixelShader is called:

Passing Custom Parameters to Shaders

Going back to our vertex shader where we performed the rotation relative to the X-axis, you probably noticed that the angle is hard-coded and is applied to each vector in the same way. This is rarely the case, as the angle would normally come from inside the game itself, often responding to internal behavior, such as character movement or action.

To pass a parameter to the shader, you need to first of all redefine the input structure. Let's add an angle carrier to the **VertexShaderInput** struct in **SimpleVertexShader.hlsl**:

```
struct VertexShaderInput
{
    float3 pos : POSITION;
    float3 color : COLOR0;
    float angle : TRANSFORM0;
};
```

This alone won't do anything. Go the entry point function (**main**) and make sure that the angle constant is set to **input.angle**:

```
const float angle = input.angle;
```

Now the shader is ready, but you also need to let your game know that the vertex shader has a modified input layout. To do this, go to **CubeRenderer.cpp** and find the D3D11_INPUT_ELEMENT_DESC array that defines the input layout for incoming vertex shaders. Simply add an identifying item to the existing array:

```
const D3D11_INPUT_ELEMENT_DESC vertexDesc[] =
{
    { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,  D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "COLOR",    0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { "TRANSFORM", 0, DXGI_FORMAT_R32_FLOAT,      0, 24, D3D11_INPUT_PER_VERTEX_DATA, 0 }
};
```

The first parameter is the pre-defined semantic used in the shader. The second parameter defines the input index. After that, use DXGI_FORMAT_R32_FLOAT to indicate that the value passed will be a standard float. Pay close attention when you specify the input offset. Notice that the two FLOAT3 values, POSITION and COLOR, take 12 bytes each —a float takes 32 bits (4 bytes), and you have a triplet, therefore the offset for TRANSFORM should be 24 bytes (two times 12 bytes from previous indicators).

Now you need to modify the internal vector descriptors that are being passed to the rendering pipe. In the sample project those are created with the help of the VertexPositionColor class, located in the **CubeRenderer.h**. Add a simple float field to the existing struct, so it looks like this:

```
struct VertexPositionColor
{
    DirectX::XMFLOAT3 pos;
    DirectX::XMFLOAT3 color;
    float angle;
};
```

In **CubeRenderer.cpp**, find the **cubeVerticles** array. For each created **VertexPositionColor** you are now able to add a third value representing the rotation:

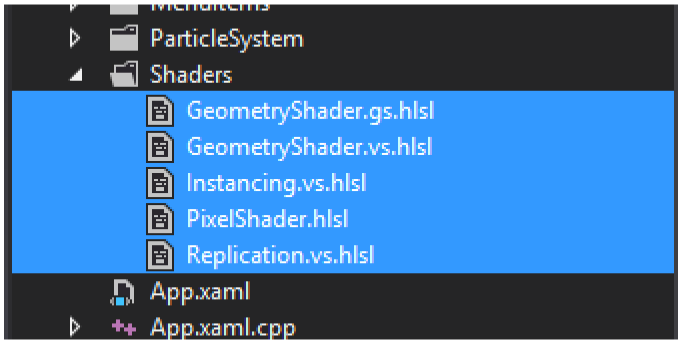
```
auto createCubeTask = (createPSTask && createVSTask).then([this] () {
    VertexPositionColor cubeVertices[] =
    {
        {XMFLOAT3(-0.5f, -0.5f, -0.5f), XMFLOAT3(0.0f, 0.0f, 0.0f), 1.38f},
        {XMFLOAT3(-0.5f, -0.5f,  0.5f), XMFLOAT3(0.0f, 0.0f, 1.0f), 1.38f},
        {XMFLOAT3(-0.5f,  0.5f, -0.5f), XMFLOAT3(0.0f, 1.0f, 0.0f), 1.38f},
        {XMFLOAT3(-0.5f,  0.5f,  0.5f), XMFLOAT3(0.0f, 1.0f, 1.0f), 1.38f},
        {XMFLOAT3( 0.5f, -0.5f, -0.5f), XMFLOAT3(1.0f, 0.0f, 0.0f), 1.38f},
        {XMFLOAT3( 0.5f, -0.5f,  0.5f), XMFLOAT3(1.0f, 0.0f, 1.0f), 1.38f},
        {XMFLOAT3( 0.5f,  0.5f, -0.5f), XMFLOAT3(1.0f, 1.0f, 0.0f), 1.38f},
        {XMFLOAT3( 0.5f,  0.5f,  0.5f), XMFLOAT3(1.0f, 1.0f, 1.0f), 1.38f},
    };
```

Now you will be able to pass parameters to the vertex shader without having to manually modify the shader itself.

Shaders in FallFury

There are several shaders used in FallFury, which must be selected depending on the Direct3D feature level available on the machine. A feature level determines what a video adapter can do in terms of rendering—even though Direct3D is a unified framework, it still heavily relies on how individual graphic cards can perform.

There are 5 shaders used internally for texture rendering. Specifically, there is a pixel shader, a replication vertex shader, an instancing vertex shader and geometry shaders. This could be familiar if you worked with Microsoft DirectX sprite samples before:



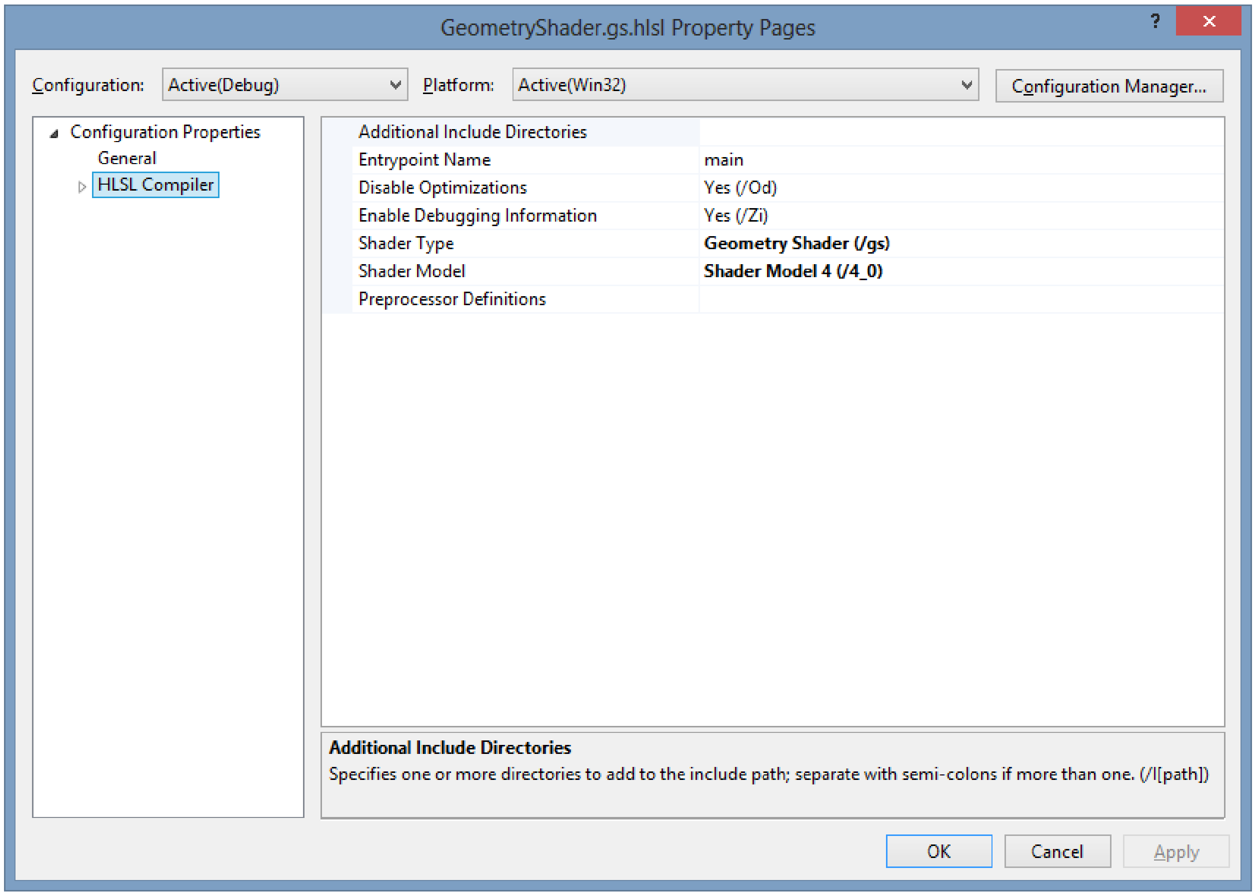
Given the platform limitations, geometry shaders can only be used on devices supporting Direct3D Feature Level 10. When FallFury detects that the feature level is lower than that, it has to fall back on either the instancing or the replication shaders. Some devices, such as the Microsoft Surface, are at feature level 9.1. Therefore, the replication render technique must be enforced.

Following the Microsoft DirectX Sprite Sample, it is fairly easy to simply select the correct render technique once the feature level is detected with GetFeatureLevel:

```
auto featureLevel = m_d3dDevice->GetFeatureLevel();

if (featureLevel >= D3D_FEATURE_LEVEL_10_0)
{
    m_technique = RenderTechnique::GeometryShader;
}
else if (featureLevel >= D3D_FEATURE_LEVEL_9_3)
{
    m_technique = RenderTechnique::Instancing;
}
else
{
    m_technique = RenderTechnique::Replication;

    if (capacity > static_cast<int>(Parameters::MaximumCapacityCompatible))
    {
        // The index buffer format for feature-level 9.1 devices may only be 16 bits.
        // With 4 vertices per sprite, this allows a maximum of (1 << 16) / 4 sprites.
        throw ref new Platform::InvalidArgumentException();
    }
}
```

Depending on the selected render technique, which is determined on application startup, the proper input layout is selected for each shader and is used to control the rendering pipe.

Different graphic adapters also support different shader models. You need to account for those, and before running the application, the HLSL compiler will take on the task of determining whether a proper shader component is supported by the given model. In the shader properties, make sure that you specify the used shader model, as well as the type of the shader:


GeometryShader.gs.hlsl Property Pages

Configuration: Active(Debug) Platform: Active(Win32) Configuration Manager...

Configuration Properties
General
HLSL Compiler

Additional Include Directories	
Entrypoint Name	main
Disable Optimizations	Yes (/Od)
Enable Debugging Information	Yes (/Zi)
Shader Type	**Geometry Shader (/gs)**
Shader Model	**Shader Model 4 (/4_0)**
Preprocessor Definitions	

Additional Include Directories
Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

OK Cancel Apply

Without this, you are bound to experience compile time errors that will prevent you from launching the application or deploying it to different devices.

Conclusion

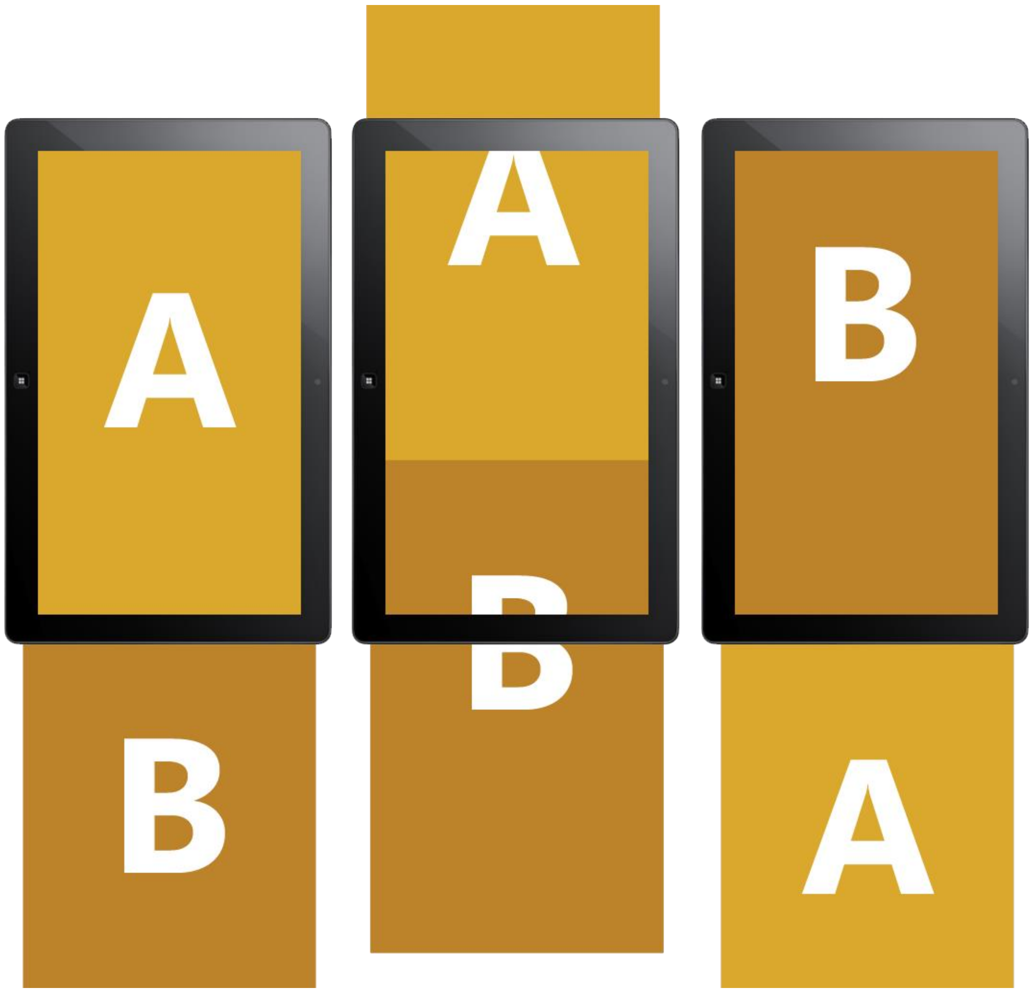
Shaders are not an easy subject. I highly recommend reading The Cg Tutorial, published online for free by nVidia. Even though it describes the Cg shader language, it is virtually identical to HLSL and you will not have any problems using the practices you learned there in building HLSL shaders.

Chapter 3 – Basic Rendering and Movement

Continuing the FallFury series, in this article I talk about basic game element rendering and character movement. As you already know, the FallFury user experience stack is split across two layers—native DirectX and XAML. Here, I will only be talking about the native DirectX component.

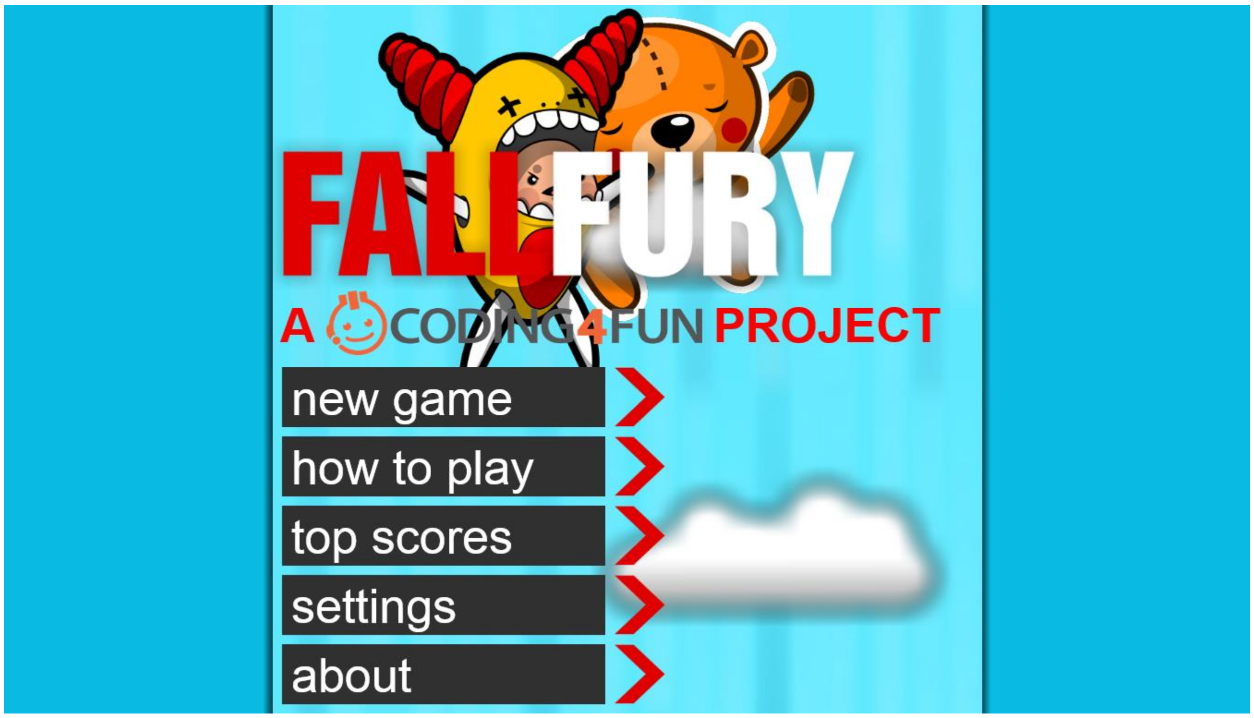
The Background

Each game screen in FallFury has a moving background that creates the illusion of a fall. The way the screen is designed, it simulates vertical parallax scrolling, as the background moves faster than the overlaid objects. There is a simple way to make background movement possible without actually having to replicate parts of the texture and render them all over again. Take a look at this image, showcasing the process:



First of all, two textures are loaded that, connected at the bottom, create the illusion of a single texture. In the beginning, texture A takes the entire screen and texture B is positioned directly underneath it, with a non-existing gap between them. To initiate the scrolling, texture A is being displaced vertically by an arbitrary number of pixels, and texture B follows it at the same pace. As texture B reaches the zero point (top of the viewport), texture A is no longer visible, therefore it is displaced vertically to be below texture B. This cycle can be repeated as many times as necessary during gameplay as well as while the user is in the menu.

Let's take a look at the code that makes it possible, starting with the main menu screen:



First of all, you need to be aware that every game screen is automatically inheriting the properties and capabilities of base class **GameScreenBase**. This is the class the offers the foundation both for basic texture loading and movement:

```
protected private:
    // Core background textures
    Microsoft::WRL::ComPtr<ID3D11Texture2D> m_backgroundBlockA;
    Microsoft::WRL::ComPtr<ID3D11Texture2D> m_backgroundBlockB;

    void MoveBackground(float velocity);
```

It also offers the core texture containers for the overlaid elements that are present in most screens, such as clouds:

```
// Overlayed clouds
Microsoft::WRL::ComPtr<ID3D11Texture2D>                                      m_overlayA;
Microsoft::WRL::ComPtr<ID3D11Texture2D>                                      m_overlayB;
```

The overlay movement is inherently dependent on the base background displacement and can be adjusted relative to the initial velocity. Let's take a look at **GameScreenBase.cpp**, specifically at **MoveBackground**:

```
void GameScreenBase::MoveBackground(float velocity)
{
    if (m_backgroundPositionA <= -BACKGROUND_MIDPOINT)
        m_backgroundPositionA = m_backgroundPositionB + (BACKGROUND_MIDPOINT * 2);

    if (m_backgroundPositionB <= -BACKGROUND_MIDPOINT)
        m_backgroundPositionB = m_backgroundPositionA + (BACKGROUND_MIDPOINT * 2);

    m_backgroundPositionA -= velocity;
    m_backgroundPositionB -= velocity;
}
```

The BACKGROUND_MIDPOINT value is relative to the height of the background texture. As we are working with a variable screen size, given that tablets and desktops do not have the same resolution, the movement has to be adjusted accordingly. One way, however, to ensure the proper texture positioning would be to place it in relation to its previous instance. Hence, this snippet in **UpdateWindowSize**:

```
BACKGROUND_MIDPOINT = 1366.0f / 2.0f;

m_backgroundPositionA = BACKGROUND_MIDPOINT;
m_backgroundPositionB = m_backgroundPositionA * 3;
```

Because overlays are not necessarily a part of every screen, I am not including the **MoveOverlay** method in the base class. Let's now take a look at **MenuScreen.cpp**. Take a look at the Load method and you will see several lines that prepare the background and overlays:

```
m_loader = ref new BasicLoader(Manager->m_d3dDevice.Get(), Manager->m_wicFactory.Get());

m_loader->LoadTexture("Assets\\Backgrounds\\generic_blue_a.png", &m_backgroundBlockA,
                      nullptr);
m_loader->LoadTexture("Assets\\Backgrounds\\generic_blue_b.png", &m_backgroundBlockB,
                      nullptr);
m_loader->LoadTexture("Assets\\Backgrounds\\cloud_overlay_a.png", &m_overlayA, nullptr);
m_loader->LoadTexture("Assets\\Backgrounds\\cloud_overlay_b.png", &m_overlayB, nullptr);

CurrentSpriteBatch->AddTexture(m_backgroundBlockA.Get());
CurrentSpriteBatch->AddTexture(m_backgroundBlockB.Get());
CurrentSpriteBatch->AddTexture(m_overlayA.Get());
CurrentSpriteBatch->AddTexture(m_overlayB.Get());
```

The **BasicLoader** class was imported from the Direct3D sprite sample. A call to LoadTexture will read the data from a DDS or PNG file and output the data in an ID3D11Texture2D object. Once loaded, the texture is added to the **SpriteBatch** instance associated with the screen, also declared as a part of **GameScreenBase**.

Depending on the screen, this procedure might have to be done for multiple textures, as you will see later in this article. Each page also has two timed loops—**RenderScreen** and **Update**. RenderScreen is responsible for taking everything from the SpriteBatch instance and showing it to the user. If you've used the XNA SpriteBatch before, you are aware that you need to start the drawing cycle by calling **Begin** and end it by calling **End**. The same applies here:

```
CurrentSpriteBatch->Begin();

CurrentSpriteBatch->Draw(
    m_backgroundBlockA.Get(),
    float2(Manager->m_windowBounds.Width / 2, m_backgroundPositionA),
    PositionUnits::DIPs,
    m_screenSize,
    SizeUnits::Pixels);

CurrentSpriteBatch->Draw(
    m_backgroundBlockB.Get(),
    float2(Manager->m_windowBounds.Width / 2 ,m_backgroundPositionB),
    PositionUnits::DIPs,
    m_screenSize,
    SizeUnits::Pixels);

if (m_showBear->IsLoaded)
    m_showBear->Render();

if (m_showMonster->IsLoaded)
    m_showMonster->Render();

CurrentSpriteBatch->Draw(
    m_overlayA.Get(),
    float2(Manager->m_windowBounds.Width/2, m_backgroundPositionA),
    PositionUnits::DIPs,
    m_screenSize,
    SizeUnits::Pixels);

CurrentSpriteBatch->Draw(
    m_overlayB.Get(),
    float2(Manager->m_windowBounds.Width / 2 ,m_backgroundPositionB),
    PositionUnits::DIPs,
    m_screenSize,
    SizeUnits::Pixels);

CurrentSpriteBatch->End();
```

The current overloaded **Draw** call gets the following items:

- The texture object
- The position where the object has to be drawn on the screen
- The way the object is positioned (can also be a normalized value or a pixel value)
- The size of the texture (stretching may occur)
- The type of the size value (in this case, I am using pixels, but can also be normalized)

The order in which the **Draw** calls are arranged determines the order of objects drawn on the screen. The calls on top will place texture objects at the bottom of the rendering stack, and the calls at the end will place the objects at the top of the stack.

The Gameplay Screen Background & Overlays

Let's move on to the arguably most critical screen in the project,, **GamePlayScreen.cpp**. There are a few nuances that differentiate it from any other screen—in particular, the background loading routine.

For every game, there is a different level type that is being used. With each level type, there is a different combination of a background and the overlay. Currently, there are four supported level types:

- Space
- Nightmare
- Magic Bean
- Dream

When the proper level is detected and the metadata is loaded from the associated XML file (more about this process later on in the series), the level textures are loaded into memory:

```
switch (m_currentLevelType)
{
    case LevelType::SPACE:
    {
        m_loader->LoadTexture("Assets\\Backgrounds\\generic_dark_blue_a.png",
                              &m_backgroundBlockA, nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\generic_dark_blue_b.png",
                              &m_backgroundBlockB, nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\star_overlay_a.png", &m_overlayA, nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\star_overlay_b.png", &m_overlayB, nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\galaxy_overlay_a.png", &m_overlayGalaxyA,
                              nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\galaxy_overlay_b.png", &m_overlayGalaxyB,
                              nullptr);

        CurrentSpriteBatch->AddTexture(m_overlayGalaxyA.Get());
        CurrentSpriteBatch->AddTexture(m_overlayGalaxyB.Get());
        break;
    }
    case LevelType::NIGHTMARE:
    {
        m_loader->LoadTexture("Assets\\Backgrounds\\generic_red_a.png", &m_backgroundBlockA,
                              nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\generic_red_b.png", &m_backgroundBlockB,
                              nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\cloud_overlay_a.png", &m_overlayA,
                              nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\cloud_overlay_b.png", &m_overlayB,
                              nullptr);
        break;
    }
    case LevelType::MAGIC_BEANS:
    {
        m_loader->LoadTexture("Assets\\Backgrounds\\generic_blue_a.png", &m_backgroundBlockA,
                              nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\generic_blue_b.png", &m_backgroundBlockB,
                              nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\cloud_overlay_a.png", &m_overlayA,
                              nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\cloud_overlay_b.png", &m_overlayB,
                              nullptr);

        break;
    }
    case LevelType::DREAM:
    {
        m_loader->LoadTexture("DDS\\Levels\\Dream\\TEST_backgroundDream_01.dds",
                              &m_backgroundBlockA, nullptr);
        m_loader->LoadTexture("DDS\\Levels\\Dream\\TEST_backgroundDream_02.dds",
                              &m_backgroundBlockB, nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\cloud_overlay_a.png", &m_overlayA,
                              nullptr);
        m_loader->LoadTexture("Assets\\Backgrounds\\cloud_overlay_b.png", &m_overlayB,
                              nullptr);
        break;
    }
}
```

This way, no unnecessary textures are loaded. A screen-based level flag allows me to control the incoming assets. At this point, the level environment does not change dynamically, so it is safe to assume that the textures should be loaded on a per-level basis.

The RenderScreen method is called in the same manner as in the menu screen, with the moving background and overlays located at the bottom of the rendering stack:

```
CurrentSpriteBatch->Draw(
    m_backgroundBlockA.Get(),
    float2(Manager->m_windowBounds.Width / 2, m_backgroundPositionA),
    PositionUnits::DIPs,
    m_screenSize,
    SizeUnits::Pixels);

CurrentSpriteBatch->Draw(
    m_backgroundBlockB.Get(),
    float2(Manager->m_windowBounds.Width / 2, m_backgroundPositionB),
    PositionUnits::DIPs,
    m_screenSize,
    SizeUnits::Pixels);

CurrentSpriteBatch->Draw(
    m_overlayA.Get(),
    float2(Manager->m_windowBounds.Width / 2, m_overlayPositionA),
    PositionUnits::DIPs,
    float2(768.0f, 1366.0f),
    SizeUnits::Pixels);

CurrentSpriteBatch->Draw(
    m_overlayB.Get(),
    float2(Manager->m_windowBounds.Width / 2, m_overlayPositionB),
    PositionUnits::DIPs,
    float2(768.0f, 1366.0f),
    SizeUnits::Pixels);
```

Character Movement

As you are now aware of the basic screen structure and how the basic rendering process is built, let's take a look at how the main game character moves on the screen. Falling down, the teddy bear also needs to move left and right to ensure that he is able to pick up power-ups and buttons as well as avoid obstacles and enemy ammo.

There are several important considerations here. The most important one is to not assume that the user will have a specific input device. Potential movement controllers include a keyboard, the mouse, the touch screen and the accelerometer. In the most common scenarios, the desktop machines will not have an accelerometer, and the tablet computers will not have a constantly attached keyboard and a mouse. In FallFury, I decided to leverage all potential input engines and let the user choose the best option for himself.

When the current GamePlayScreen instance loads, I am attempting to get access to the system accelerometer device:

```
m_systemAccelerometer = Windows::Devices::Sensors::Accelerometer::GetDefault();
```

m_systemAccelerometer is of type Windows::Devices::Sensors::Accelerometer and is declared in **GamePlayScreen.h**. If an accelerometer is detected, I need to bind it to a ReadingChanged event handler that will give me the current G-force transformed into X, Y, and Z displacement:

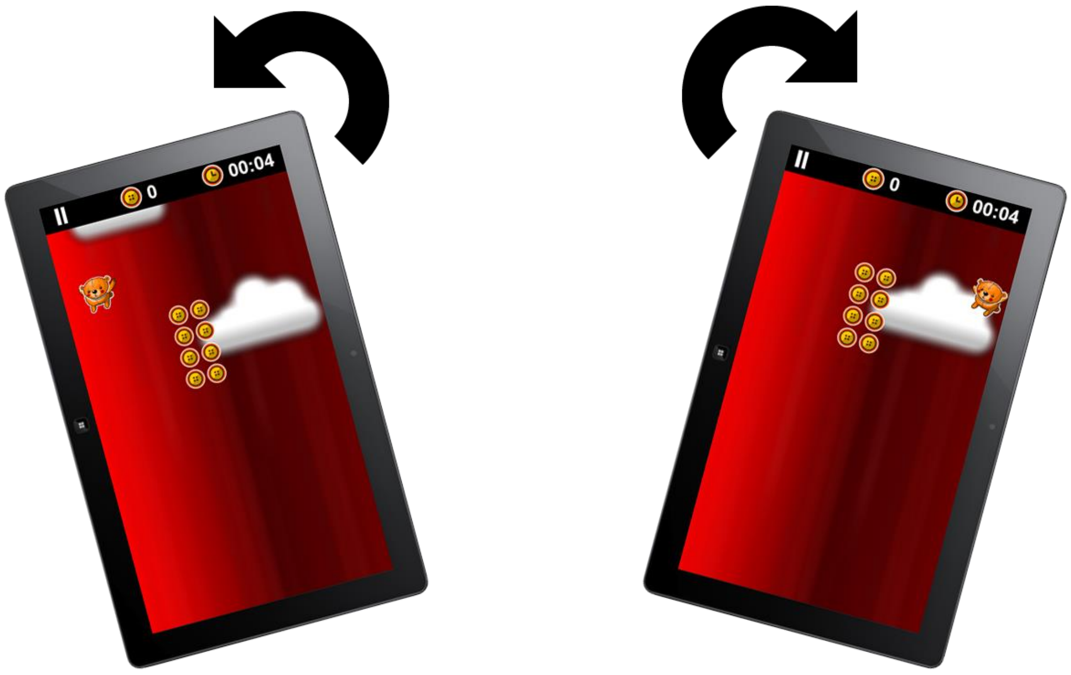
```
if (m_systemAccelerometer != nullptr)
{
    m_systemAccelerometer->ReadingChanged +=
              ref new TypedEventHandler<Accelerometer^, AccelerometerReadingChangedEventArgs^>
             (this, &GamePlayScreen::AccelerometerReadingChanged);
}
```

ReadingChanged itself does not participate in updating the position for the character, but rather passes the current values to **m_xAcceleration**, which is later used in the **Update** method:

```
void GamePlayScreen::AccelerometerReadingChanged(_In_ Accelerometer^ accelerometer, _In_
AccelerometerReadingChangedEventArgs^ args)
{
    if (StaticDataHelper::IsAccelerometerEnabled)
    {
        auto currentOrientation = DisplayProperties::CurrentOrientation;
        float accelValue;

        if (currentOrientation == DisplayOrientations::Portrait)
            accelValue = args->Reading->AccelerationY;
        else if (currentOrientation == DisplayOrientations::PortraitFlipped)
            accelValue = -args->Reading->AccelerationY;
        else if (currentOrientation == DisplayOrientations::Landscape)
            accelValue = args->Reading->AccelerationX;
        else if (currentOrientation == DisplayOrientations::LandscapeFlipped)
            accelValue = -args->Reading->AccelerationX;
        else
            accelValue = 0.0f;

        if (StaticDataHelper::IsAccelerometerInverted)
            m_xAcceleration = -accelValue;
        else
            m_xAcceleration = accelValue;
    }
}
```

The reason for this lies in the fact that **ReadingChanged** is triggered at a much lower rate than the **Update** loop. If the character position would be adjusted through the core accelerometer event handler, the result would be choppy ("step-by-step") movement instead of a smooth transition.

Notice, also, that depending on the screen orientation, I need to get the acceleration either on the X- or Y-axes. Since an accelerometer is detected, I am assuming that the device that's being used is a tablet. Therefore, it can have auto-rotate enabled, which means that the reference axis (horizontal) might change depending on how the user holds the device. DisplayProperties::CurrentOrientation can give me the current orientation, whether portrait (Y acceleration value) or landscape (X acceleration value):

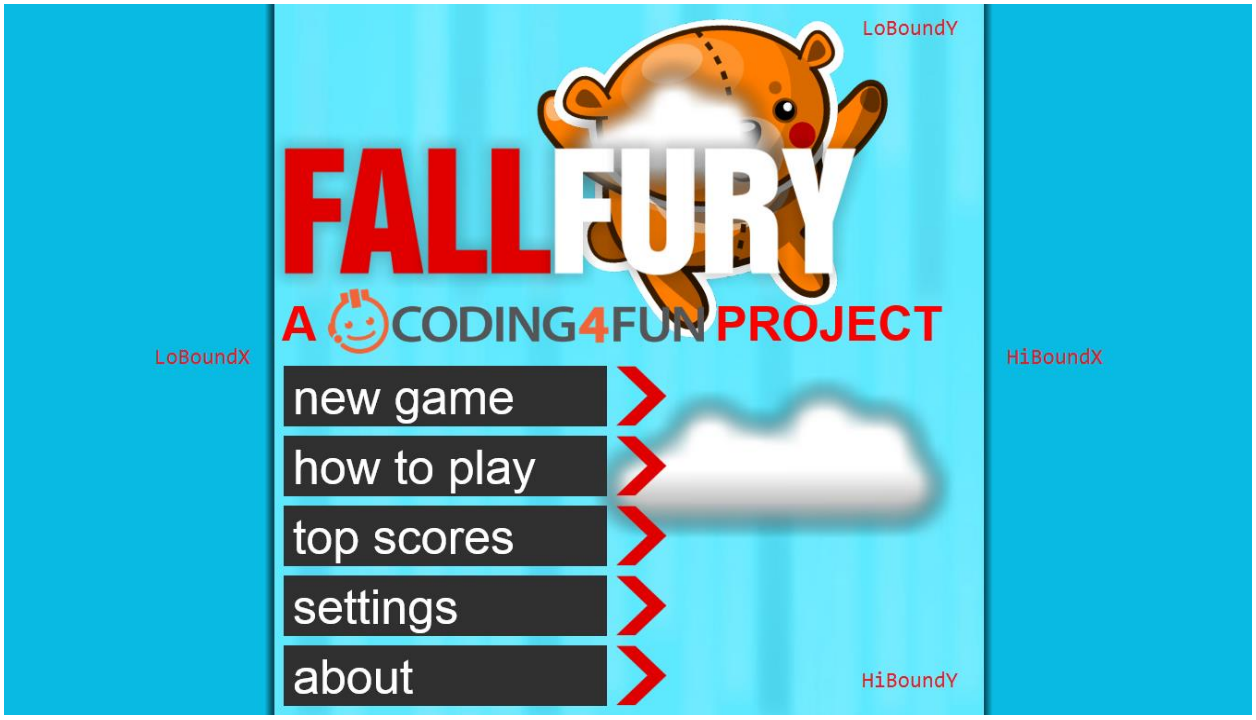


As with many other titles, the user may invert the movement based on the accelerometer reading. For example, if the device is tilted to the right, the character will move to the left, and vice-versa. The effect is easily achieved by negating the current reading, regardless of the axis it is relying on.

Obviously, as the character moves, there should be boundaries that restrict the movement within the context of the current game screen. To get the correct screen bounds, **GameScreenBase**, the foundation class for every screen, sets four properties: **LoBoundX**, **HiBoundX**, **LoBoundY** and **HiBoundY**:

```
LoBoundX = (rWidth - 768.0f) / 2.0f;
HiBoundX = LoBoundX + 768.0f;

LoBoundY = 0;
HiBoundY = LoBoundY + rHeight;
```

LoBoundX is the leftmost limit for the horizontal playable area and **HiBoundX** is the rightmost limit for the same horizontal area. **LoBoundY** and **HiBoundY** are responsible for carrying the limits for the vertical space. All four boundary values are relative to the screen size:



To consistently get the correct X boundary no matter the screen size, the playable area is set to a fixed width of 768 pixels. The rest of the screen is accordingly cancelled out and the remaining area split in two. The original X point (zero) is added to the curtain size and that way there is the leftmost X boundary. The Y boundaries are simply obtained from the screen height, as there are no gameplay experience restrictions in that domain.

The bear position is updated in the Update loop in the game screen. There are several parameters that need to be adjusted, such as the vertical velocity, the bear rotation on tilt and the horizontal displacement.

When the level just starts, the bear falls faster until he reaches the standard static Y point. Because of screen size differences, this should not be done by comparing the pixel distance but rather by utilizing a ratio built from the current bear Y position and the high Y boundary:

```
if ((GameBear->Position.y / HiBoundY) < 0.19f)
{
    GameBear->Position.y += GameBear->Velocity.y * (3.2f);
}
```

Assuming that the game is not paused, and thus the background is moving, the bear Y position should be adjusted relative to the current velocity set for the background scrolling. Because of how parallax scrolling works, the bear velocity has to be higher than the initial screen movement. Therefore, it is multiplied by a fixed value independent of the level played:

```
if (!m_isBackgroundMoving)
{
    if (GameBear->Position.y > m_screenSize.y)
    {
        Manager->CurrentGameState = GameState::GS_GAME_OVER;
    }
    else
    {
        GameBear->Position.y += GameBear->Velocity.y * 1.5f;
    }
}
```

In the statement above, there is also an extra condition that verifies whether the bear is below the low Y boundary. If it is, then the game is over. This is imposed by a game animation that takes the bear off the Y limits, meaning that the bear is dead.

Bear rotation is based on the current X acceleration, but the bear shouldn't rotate 360 degrees. To limit the rotation, there is a rotation threshold set (value taken in radians) that is being checked before rotating the character:

```
float compositeRotation = GameBear->Rotation - (float)m_xAcceleration / 10.0f;
if (compositeRotation < m_rotationThreshold && compositeRotation > -m_rotationThreshold)
    GameBear->Rotation = compositeRotation;
```

The composite value lets me verify the perspective rotation without actually assigning it to the bear. If it is within the imposed threshold, only then will the bear rotation be adjusted.

Using the composite value approach is also efficient when performing the horizontal displacement adjustment. The initial value is composed of the current position and the X-based acceleration multiplied by a dynamic multiplier value to create the inertia effect:

```
float compositePosition = GameBear->Position.x + ((float)m_xAcceleration *
m_accelerationMultiplier);

if (m_xAcceleration < 0)
    compositePosition -= 100.0f;
else
    compositePosition += 100.0f;

if (Manager->IsWithinScreenBoundaries(float2(compositePosition, GameBear->Position.y)))
{
    GameBear->Position.x += (float)m_xAcceleration * m_accelerationMultiplier;
}
else
{
    if (GameBear->Position.x > HiBoundX)
    {
        GameBear->Position.x = HiBoundX - 180.0f;
    }
    else if (GameBear->Position.x < LoBoundX)
    {
        GameBear->Position.x = LoBoundX + 180.0f;
    }
}
```

If the bear is attempting to hit a "wall" (screen limit), its position is set back to the one barely prior to the limit. By doing this, I am avoiding locking the character on the side of the screen.

There is something in this snippet above that you might not be aware of—a reference to **Manager->IsWithinScreenBoundaries**. This method belongs to the ScreenManager (ScreenManager.cpp) class—a utility class that ensures the proper screen is displayed depending on the current game mode, and also allows control of mouse actions and boundary checks:

```
bool ScreenManager::IsWithinScreenBoundaries(float2 position)
{
    if (position.x < CurrentGameScreen->LoBoundX
        || position.x > CurrentGameScreen->HiBoundX
        || position.y < CurrentGameScreen->LoBoundY
        || position.y > CurrentGameScreen->HiBoundY)
        return false;
    else
        return true;
}
```

You saw how the movement is accomplished with the help of the accelerometer. Let's take a look at how a keyboard can be used to do the same thing. There is a **HandleKeyInput** method, exposed through the **GamePlayScreen** class. As a matter of fact, **HandleKeyInput** is wired into the **GameScreenBase** class, therefore if a specific combination needs to be handled outside the context of the game play screen, it can be:

```
void GamePlayScreen::HandleKeyInput(Windows::System::VirtualKey key)
{
    if (key == Windows::System::VirtualKey::Right)
    {
        if (GameBear->IsWithinScreenBoundaries(GameBear->Size.x, 0.0f, GetScreenBounds()))
        {
            GameBear->Direction = TurningState::RIGHT;

            m_xAcceleration = 0.8f;

            if (GameBear->Rotation >= -m_rotationThreshold)
                GameBear->Rotation -= 0.02f;
        }
    }
    else if (key == Windows::System::VirtualKey::Left)
    {
        if (GameBear->IsWithinScreenBoundaries(-GameBear->Size.x, 0.0f, GetScreenBounds()))
        {
            GameBear->Direction = TurningState::LEFT;

            m_xAcceleration = -0.8f;

            if (GameBear->Rotation <= m_rotationThreshold)
                GameBear->Rotation += 0.02f;
        }
    }
}
```

As the accelerometer is no longer influencing the rotation or displacement, both indicators have to be manipulated through key presses. There are still standard thresholds in place to limit the potential incorrect movement, but the idea remains the same. This method is being called from the XAML page, which is overlaid on top of the DirectX renders:

```
<SwapChainBackgroundPanel
    x:Class="Coding4Fun.FallFury.DirectXPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Coding4Fun.FallFury" x:Name="XAMLPage"
    xmlns:controls="using:Coding4Fun.FallFury.Controls"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" Loaded="OnLoaded"
    KeyDown="OnKeyDown" LayoutUpdated="XAMLPage_LayoutUpdated">

void DirectXPage::OnKeyDown(Platform::Object^ sender,
Windows::UI::Xaml::Input::KeyRoutedEventArgs^ e)
{
    m_renderer->CurrentGameScreen->HandleKeyInput(e->Key);
}
```

When the keyboard is not available, movement can be controlled with the help of a mouse. Again, with standard event handlers, I am simply using **OnPointerMoved**:

```
void GamePlayScreen::OnPointerMoved(Windows::UI::Core::PointerEventArgs^ args)
{
    if (StaticDataHelper::IsMouseEnabled)
    {
        if (GameBear != nullptr)
        {
            m_touchCounter++;
            if (GameBear->IsWithinScreenBoundaries(GameBear->Size.x, 0.0f, GetScreenBounds()))
            {
                m_xAcceleration = (args->CurrentPoint->RawPosition.X - GameBear->Position.x) /
                                  m_screenSize.x;
            }
        }
    }
}
```

By using this method, the bear will accelerate to the point where the mouse cursor is located, having a higher velocity the further it is located from the cursor. Once again, this creates the inertia visualization.

Conclusion

Concluding Part 3 of the series, remember that because Windows 8 is not a tablet-only OS, some users might have different input devices. Try to accommodate as many of those as possible.

The next article in this series will focus on the XAML overlay used in FallFury.

Chapter 4 – XAML Interop

As mentioned earlier in the series, FallFury does not solely rely on DirectX to display content to the user. As a Windows Store game, FallFury leverages the new Direct2D (XAML) project template, available in Visual Studio 2012.

The Concept of a Swap Chain

Before I go into detail about the DirectX and XAML interop in FallFury, I want to cover one important aspect of DirectX development that you need to familiarize yourself with: the swap chain.

When your graphics adapter draws on the visual surface, you, as the user, see only minor potential redraws. Internally, however, the device switches buffers that reflect the displayed content, with each buffer representing a frame that has to be drawn. You can deduce from this that any swap chain has at least two buffers that it can switch between.

For example, if I want to display my character as being displaced by a specific amount of pixels, the buffer will at the outset present to me the character in its initial position, while the second buffer will be constructed in the background with the proper position adjustments. The first frame, made from the content from the first buffer, will be discarded, and then the second frame will be displayed, and so on. This process occurs at a very high speed that depends on the processing capabilities of the graphics adapter, so the user does not notice the swapping itself.

The most common swap chain is composed of two buffers—the screenbuffer and the secondary framebuffer.

SwapChainBackgroundPanel

DirectX interoperability with XAML simplifies a lot of routine tasks that would otherwise be handled with manual rendering procedures, such as a menu system or a simple game HUD. That being said, the way the XAML workflow is organized in a Direct2D project is quite different compared to a standard XAML Windows Store or Windows Phone application.

The core difference is that there is no navigational system per-se and the fundamental entity in a Direct2D (XAML) project that manages the XAML content is a SwapChainBackgroundPanel control. This control allows the developer to overlay XAML elements on top of the DirectX renders. It replaces the normal page-based layout with one in which it is sole container for every XAML control that has to be used in the application. This necessitates that you will have to organize the secondary elements in such a way that the correct set is displayed for the current application state.

For example, if the user is in the main menu, menu options as well as the game logo should be shown. When the user switches to the game mode, the HUD should appear and the menu should become hidden. Although both the menu and the HUD are a part of the same SwapChainBackgroundPanel, I will have to manually manage state and visibility changes.

Using the SwapChainBackgroundPanel also means that you will have to enforce specific graphic configuration rules in your application. One of them applies to setting up the swap chain. When you set up the scaling, it must be set to DXGI_SCALING_STRETCH:

```
DXGI_SWAP_CHAIN_DESC1 swapChainDesc = {0};
swapChainDesc.Width = static_cast<UINT>(m_renderTargetSize.Width);
swapChainDesc.Height = static_cast<UINT>(m_renderTargetSize.Height);
swapChainDesc.Format = DXGI_FORMAT_B8G8R8A8_UNORM;
swapChainDesc.Stereo = false;
swapChainDesc.SampleDesc.Count = 1;
swapChainDesc.SampleDesc.Quality = 0;
swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
swapChainDesc.BufferCount = 2;
swapChainDesc.Scaling = DXGI_SCALING_STRETCH;
swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL;
swapChainDesc.Flags = 0;
```

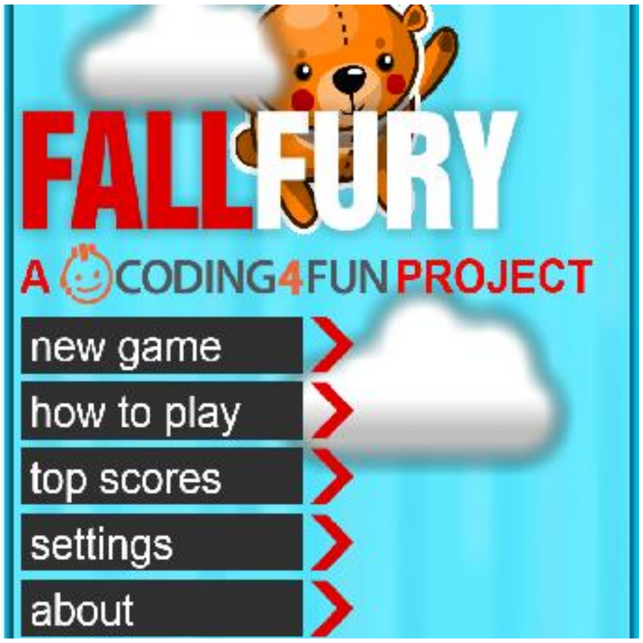
The swap chain itself should be configured for composition, mixing the native DirectX buffer with the overlaid XAML. This is done by calling CreateSwapChainForComposition:

```
ThrowIfFailed(
    dxgiFactory->CreateSwapChainForComposition(
        m_d3dDevice.Get(),
        &swapChainDesc,
        nullptr,
        &m_swapChain
        )
    );

ComPtr<ISwapChainBackgroundPanelNative> panelNative;
ThrowIfFailed(
    reinterpret_cast<IUnknown*>(m_panel)->QueryInterface(IID_PPV_ARGS(&panelNative))
    );

ThrowIfFailed(
    panelNative->SetSwapChain(m_swapChain.Get())
    );
```

There really isn't much additional configuration work beyond this point. Remember, that the XAML content will **always be overlaid on top of the DirectX content**; therefore, plan the game components accordingly.

The XAML Menu System

The menu is at the core of the FallFury experience. When designing it, I was inspired by how Dance Central built the user interaction mechanism, and I tried to implement a similar approach in which the user would have to slide the button to the right instead of simply tapping on it:



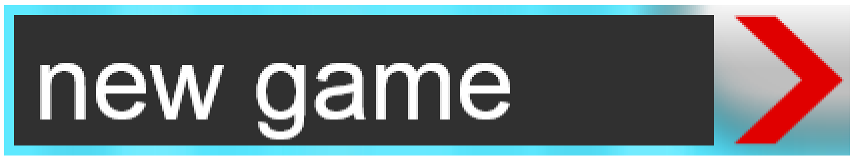
Each menu item is built around a custom **MenuItem** control (**MenuItem.xaml**):

```
<UserControl
    x:Class="Coding4Fun.FallFury.MenuItem"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Coding4Fun.FallFury"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    x:Name="menuItem"
    Margin="10, 0, 0, 10">

    <Grid
       Height="65"
       ManipulationMode="TranslateX"
       ManipulationCompleted="Grid_ManipulationCompleted"
       ManipulationDelta="Grid_ManipulationDelta"
       x:Name="ControlContainer"
       PointerPressed="Grid_PointerPressed"
       PointerReleased="Grid_PointerReleased">

        <Grid.Resources>
            <Storyboard x:Name="ArrowAnimator">
                <DoubleAnimation Storyboard.TargetName="ImageTranslateTransform"
                                 Storyboard.TargetProperty="X"
                                 From="0"
                                 To="20"
                                 Duration="0:0:0.4"
                                 RepeatBehavior="Forever"
                                 AutoReverse="True">
                </DoubleAnimation>
            </Storyboard>

        </Grid.Resources>

        <Grid.RowDefinitions>
            <RowDefinition Height="10" />
            <RowDefinition />
           <RowDefinition />
           <RowDefinition Height="10" />
        </Grid.RowDefinitions>
       <Grid.ColumnDefinitions>
           <ColumnDefinition Width="10" />
           <ColumnDefinition />
       </Grid.ColumnDefinitions>
       <StackPanel Orientation="Horizontal" Grid.RowSpan="4" Grid.ColumnSpan="4">
           <!-- width is sent in code behind, have to get this dynamic ...  -->
           <Grid Width="350" x:Name="coverRectangle">
               <Rectangle Fill="#303030" />
               <Rectangle
                      Fill="Red"
                      x:Name="coverActiveRectangle" />
           </Grid>
            <Image x:Name="MenuImage" Source="ms-appx:///MenuItems/single_arrow.png"
                   Margin="10,0,0,0" Stretch="Uniform">
                <Image.RenderTransform>
                    <TranslateTransform x:Name="ImageTranslateTransform"></TranslateTransform>
                </Image.RenderTransform>
            </Image>
        </StackPanel>
       <!---->

       <TextBlock
            Text="{Binding ElementName=menuItem, RelativeSource={RelativeSource Self},
                  Path=Label}"
           Grid.RowSpan="4"
```

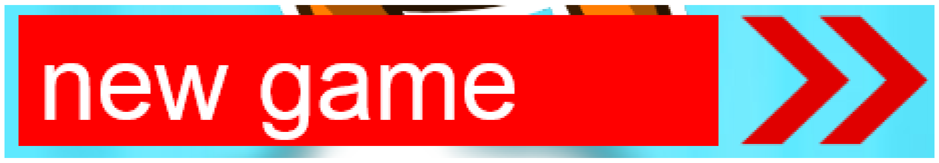
```
            Grid.ColumnSpan="4"
            Style="{StaticResource MenuItemText}">

        </TextBlock>

        <MediaElement x:Name="coreMenuMedia" Source="ms-appx:///Assets/Sounds/MenuTap.wav"
                      AutoPlay="False"></MediaElement>
        <MediaElement x:Name="slideMenuMedia" Source="ms-appx:///Assets/Sounds/MenuSlide.wav"
                      AutoPlay="False"></MediaElement>
    </Grid>
</UserControl>
```

There is a core storyboard that is used to perform an image bounce. Initial tests showed that that most users tend to follow the standard "click-and-go" pattern and even though the implemented button was a slider, they tried to click it at least once, expecting the game to take them to the next screen. To avoid this, I introduced a visual arrow indicator that bounces left and right to indicate the direction of the necessary slide.

When the sliding occurs, the overlay grid width is being adjusted relative to the manipulation. At the same time, the button changes from passive state (dark background):



to active state (red background):



Internally, the double-arrow image replaces the single-arrow one and is translated across the X-axis to create a bouncing effect:

```
void MenuItem::Grid_PointerPressed(Platform::Object^ sender,
Windows::UI::Xaml::Input::PointerRoutedEventArgs^ e)
{
    BitmapImage^ image = ref new BitmapImage(
                         ref new Uri("ms-appx:///MenuItems/double_arrow.png"));
    MenuImage->Source = image;

    coverActiveRectangle->Visibility = Windows::UI::Xaml::Visibility::Visible;
    ((Storyboard^)ControlContainer->Resources->Lookup("ArrowAnimator"))->Begin();

    coreMenuMedia->Play();
}
```

As the user drags the finger on the button, the red overlay size should be adjusted accordingly. The size adjustment can be easily tracked in the ManipulationDelta event handler for the focused grid:

```
void MenuItem::Grid_ManipulationDelta(Platform::Object^ sender,
Windows::UI::Xaml::Input::ManipulationDeltaRoutedEventArgs^ e)
{
    double diff = e->Cumulative.Translation.X;

    if (diff > maxDeltaSize)
    {
        diff = maxDeltaSize;
    }
    else if (diff < 0)
    {
        diff = 0;
    }

    coverRectangle->Width = initialBarWidth + diff;
}
```

When the user action on the button is completed, the ManipulationCompleted event handler is triggered on the same overlay grid. If the relative drag hits the critical threshold, the action linked to the button should be invoked:

```
void MenuItem::Grid_ManipulationCompleted(Platform::Object^ sender,
Windows::UI::Xaml::Input::ManipulationCompletedRoutedEventArgs^ e)
{
    float diff = e->Cumulative.Translation.X;

    if (diff > maxDeltaSize - 50) // slight buffer
    {
        slideMenuMedia->Play();
        OnButtonSelected(this, Label);
    }

    ResetElements();
}
```

There are also local sound effects that are played when the slider-button is tapped and moved to the end. Both are handled by separate MediaElement controls that avoid an internal sound file switch, calling instead the Play method as necessary. The **OnButtonSelected** event handler can be dynamically hooked in the application backend.

Each button also has a visible content area that can display a text label. It can be set through a DependencyProperty:

```
DependencyProperty^ MenuItem::_LabelProperty =
    DependencyProperty::Register("Label",
    Platform::String::typeid,
    MenuItem::typeid,
    nullptr);

 In its current configuration, the menu label can be also set in XAML:

<local:MenuItem x:Name="btnNewGame" Label="new game">
    <local:MenuItem.RenderTransform>
        <TranslateTransform></TranslateTransform>
    </local:MenuItem.RenderTransform>
</local:MenuItem>
```

When the button loses the focus, the state resets to the passive one, stopping the animation and resetting the arrow image to the single one:

```
void MenuItem::ResetElements()
{
    BitmapImage^ image = ref new BitmapImage(
                      ref new Uri("ms-appx:///MenuItems/single_arrow.png"));
    MenuImage->Source = image;

    ((Storyboard^)ControlContainer->Resources->Lookup("ArrowAnimator"))->Stop();

    coverRectangle->Width = initialBarWidth;
    coverActiveRectangle->Visibility = Windows::UI::Xaml::Visibility::Collapsed;
}
```

That's about as complex as the menu item control will get. The menu container itself can be any Grid or StackPanel control. The way the menu items are used across the game states, there is no need to have a separate unified container.

The Main XAML Container & State Changes

Going back to **DirectXPage.xaml**, there are several parts of the XAML layout that should be highlighted. First and foremost, the game curtains—the static parts of the screen that are being displayed instead of blacking out parts of the viewport that are not being used when the game runs in landscape mode. Because the game includes a rectangular frame instead of stretching the entire playable area to the size of the screen, there is an unknown amount of unallocated visual space on both the right and left sides of the frame itself:

```
<Grid HorizontalAlignment="Left" x:Name="containerA">
    <Rectangle Fill="#09bbe3" />

    <Rectangle Width="10" HorizontalAlignment="Left">
        <Rectangle.Fill>
            <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
                <GradientStop Color="#a000" Offset="0.0"></GradientStop>
                <GradientStop Color="#0000" Offset="1.0"></GradientStop>
            </LinearGradientBrush>
        </Rectangle.Fill>
    </Rectangle>
</Grid>

<Grid HorizontalAlignment="Right" x:Name="containerB">
    <Rectangle Fill="#09bbe3" />

    <Rectangle Width="10" HorizontalAlignment="Right">
        <Rectangle.Fill>
            <LinearGradientBrush StartPoint="1,0" EndPoint="0,0">
                <GradientStop Color="#a000" Offset="0.0"></GradientStop>
                <GradientStop Color="#0000" Offset="1.0"></GradientStop>
            </LinearGradientBrush>
        </Rectangle.Fill>
    </Rectangle>
</Grid>
```

Although these two grids have set alignments, the actual location on the screen will be set in code-behind because the size will also be re-calculated. Also, I need to make sure that the application is in the full display mode—if it is snapped, there is no need to display the curtains because the visible area will be reduced to an overlay grid.

The curtain resize and visibility are determined in the UpdateWindowSize method:

```
void DirectXPage::UpdateWindowSize()
{
    bool visibility = true;
    if (ApplicationView::Value == ApplicationViewState::Snapped)
        visibility = false;

    float margin = (m_renderer->m_renderTargetSize.Width - 768.0f) / 2.0f;
    if (margin < 2.0)
        visibility = false;

    if (visibility)
    {
        containerA->Width =  margin;
        containerA->HorizontalAlignment = Windows::UI::Xaml::HorizontalAlignment::Right;
        containerB->Width =  margin;
        containerB->HorizontalAlignment = Windows::UI::Xaml::HorizontalAlignment::Left;
        containerA->Visibility = Windows::UI::Xaml::Visibility::Visible;
        containerB->Visibility = Windows::UI::Xaml::Visibility::Visible;
    }
    else
    {
        containerA->Visibility = Windows::UI::Xaml::Visibility::Collapsed;
        containerB->Visibility = Windows::UI::Xaml::Visibility::Collapsed;
    }
}
```

The snippet above ensures that the curtains will only be displayed when there is extra, unused space in addition to the 768 pixels taken by the playable area.

Let’s take a look at state-specific content that is being displayed whenever the game switches states. Because of the nature of DirectX interaction, there is no way for me to hook to specific event handlers from the native loop. Therefore, I need to constantly check that the content displayed is associated with the current state.

This can be done with the help of the **SwitchGameState** method:

```
void DirectXPage::SwitchGameState()
{
    switch (m_renderer->CurrentGameState)
        {
    case GameState::GS_FULL_WIN:
            {
                grdCompleteWin->Visibility = Windows::UI::Xaml::Visibility::Visible;
                break;
            }
    case GameState::GS_PLAYING:
            {
                Hud->Visibility = Windows::UI::Xaml::Visibility::Visible;
                break;
            }
    case GameState::GS_MAIN_MENU:
            {
                stkMainMenu->Visibility = Windows::UI::Xaml::Visibility::Visible;
                break;
            }
    case GameState::GS_GAME_OVER:
            {
                UpdateResults();
                ResultPanel->Visibility = Windows::UI::Xaml::Visibility::Visible;
                grdGameOver->Visibility = Windows::UI::Xaml::Visibility::Visible;
                break;
            }
    case GameState::GS_SUBMIT_SCORE:
            {
                grdSubmitScore->Visibility = Windows::UI::Xaml::Visibility::Visible;
                txtSubmitScoreView->Text = StaticDataHelper::Score.ToString();
                break;
            }
    case GameState::GS_TOP_SCORES:
            {
                grdTopScores->Visibility = Windows::UI::Xaml::Visibility::Visible;
                break;
            }
    case GameState::GS_WIN:
            {
                UpdateResults();
                grdWinner->Visibility = Windows::UI::Xaml::Visibility::Visible;
                ResultPanel->Visibility = Windows::UI::Xaml::Visibility::Visible;
                break;
            }
    case GameState::GS_ABOUT_SCREEN:
            {
                grdAbout->Visibility =  Windows::UI::Xaml::Visibility::Visible;
                break;
            }
    case GameState::GS_LEVEL_SELECT_SINGLE:
            {
                animationBeginTime = 0;

                stkLevelSelector->Visibility = Windows::UI::Xaml::Visibility::Visible;
                break;
            }
    case GameState::GS_HOW_TO:
            {
                grdHowTo->Visibility = Windows::UI::Xaml::Visibility::Visible;
                Storyboard^ howToInitialBoard = (Storyboard^)grdMain->Resources-
>Lookup("StoryboardChainA");
                howToInitialBoard->Begin();
                break;
            }
        default:
            break;
        }

        Storyboard^ loc = (Storyboard^)grdMain->Resources->Lookup("FadingOut");
        loc->Begin();
}
```

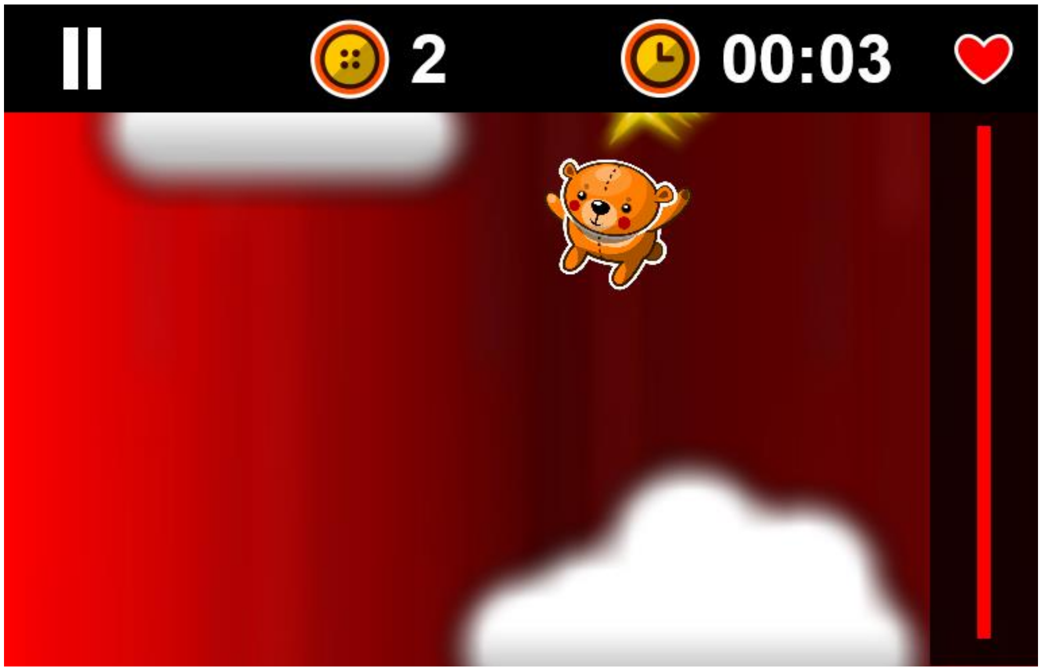
One thing you've probably noticed about the snippet above is the fact that there is no indicator showing that controls are being hidden when the state changes. Just calling **SwitchGameState** in the Update loop would cause multiple controls to be displayed at once. However, there is also the **HideEverything** method that goes through the visual tree and sets the **Visibility** to **Collapsed** for everything:

```
void DirectXPage::HideEverything()
{
    for (uint i = 0; i < grdMain->Children->Size; i++)
    {
        grdMain->Children->GetAt(i)->Visibility = Windows::UI::Xaml::Visibility::Collapsed;
    }
}
```

HUD Interaction

The Heads-Up Display (HUD) is used to alert the player about the current state of the game and the game character. In FallFury, it is used to display three indicators: how many buttons the character collected, how much time it took the character to get to the current level part, and current character health. It is also the container for the Pause button.

Here is the visual representation:



Here is the underlying XAML:

```
<Grid x:Name="Hud" VerticalAlignment="Top" Visibility="Collapsed">
    <Grid.RowDefinitions>
        <RowDefinition Height="80" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition Width="80" />
    </Grid.ColumnDefinitions>

    <Rectangle Fill="Black" Grid.ColumnSpan="4" />

    <Button
        Grid.Column="0"
        x:Name="btnPause"
        Click="btnPause_Click"
        Style="{StaticResource PauseButton}">
        <Image Source="ms-appx:///Assets/HUD/pauseButton.png" Stretch="None" /
    </Button>

    <StackPanel Grid.Column="1" Orientation="Horizontal">
        <Image Source="ms-appx:///Assets/HUD/buttonHud.png" Stretch="None" />

        <TextBlock
            x:Name="txtButtons"
            Text="0"
            Style="{StaticResource hudResult}"/>
    </StackPanel>

    <StackPanel Grid.Column="2" Orientation="Horizontal">
        <Image Source="ms-appx:///Assets/HUD/clockHud.png" Stretch="None" />

        <TextBlock
            x:Name="txtTimer"
            Text="00:00"
            Style="{StaticResource hudResult}"/>
    </StackPanel>

    <Image
        Grid.Column="3"
        Source="ms-appx:///Assets/HUD/heartHud.png"
        Stretch="None" />

    <Grid
        Grid.Column="3"
        Grid.Row="1">
        <Rectangle Fill="Black" Opacity=".8"/>
        <controls:HealthBar x:Name="healthBar"></controls:HealthBar>
    </Grid>
</Grid>
```

As I mentioned before, the DirectX layer cannot directly interact with the XAML layer. Therefore, there needs to be an intermediary binding class. As the game progress changes, the **UpdateHUD** method is called, taking a reference to the current DirectX screen and reading the game data from the character and the **StaticDataHelper** class, which is the container for the time elapsed:

```
void DirectXPage::UpdateHud(GamePlayScreen^ playScreen)
{
    healthBar->Update(playScreen->GameBear->CurrentHealth, playScreen->GameBear->MaxHealth);

    txtButtons->Text = StaticDataHelper::ButtonsCollected.ToString();

    // Find a better built-in string formatting code
    if (m_renderer->CurrentGameState == GameState::GS_PLAYING && !(StaticDataHelper::IsPaused)
        && playScreen->IsLevelLoaded)
    {
        StaticDataHelper::SecondsTotal += m_timer->Delta;

        txtTimer->Text = StaticDataHelper::GetTimeString((int)StaticDataHelper::SecondsTotal);
    }
}
```

As seen above, part of the HUD is taken by a health indicator control—**HealthBar**. It is a simple composite element made out of two overlaying rectangles, one of which is resized as the bear health changes:

```
<UserControl
    x:Class="Coding4Fun.FallFury.Controls.HealthBar"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:FallFury"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="410"
    d:DesignWidth="20" Height="410" Width="20">

    <Grid>
        <StackPanel VerticalAlignment="Top" Margin="0, 5, 0,20">
            <Rectangle Height="400" Style="{StaticResource HealthRectangle}" />
        </StackPanel>

        <StackPanel VerticalAlignment="Top" Margin="0, 5, 0, 10">
            <Rectangle x:Name="OverlayStacker" Height="0" Style="{StaticResource
                        HealthOverlayRectangle}" />
        </StackPanel>
    </Grid>
</UserControl>
```

Being of a fixed size, it is fairly easy to calculate the health-to-damage ratio and display that as the size of the overlaid rectangle:

```
DependencyProperty^ HealthBar::_MaxHealthProperty =
    DependencyProperty::Register("MaxHealth",
    double::typeid,
    HealthBar::typeid,
    nullptr);

DependencyProperty^ HealthBar::_CurrentHealthProperty =
    DependencyProperty::Register("CurrentHealth",
    double::typeid,
    HealthBar::typeid,
    nullptr);

void HealthBar::Update(double currentHealth, double maxHealth)
{
    CurrentHealth = currentHealth;
    MaxHealth = maxHealth;

    if (CurrentHealth >= 0)
    {
        OverlayStacker->Height = 400.0 - ((400.0 * CurrentHealth) / MaxHealth);
    }
}
```

Conclusion

Using XAML as a part of a DirectX application does not require the developer to do a massive overhaul of the infrastructure. That said, be mindful when deciding whether to use a hybrid XAML application, as it is much easier to integrate a SwapChainBackgroundPanel with the underlying DirectX configuration from the ground up instead of trying to do so when the DirectX component is completed.

Chapter 5 – Creating Levels

One of the goals set when developing FallFury was making the game extensible in regards to playable levels. Furthermore, I thought that it would make sense from the development and testing perspective to have levels as a completely separate entity that can be modified as necessary. For example, when a new power-up was introduced, I wanted to add an extra line in a level file and test it out. This was ultimately achieved by creating an XML-based level engine and in this article I will describe the level structure and design process.

The First Steps

When I started working on the level engine concept, I began designing the potential XML file structure for the level and ended up with the following requirements:

- **Level Type Identifier** – As different levels have different backgrounds and sound themes, there should be a way to mark a level type. There are currently four level types: dream, nightmare, space, and magic bean.
- **The Starting Character Descriptor** – When the game starts, the teddy bear has some initial, basic properties, such as maximum health level, horizontal position, and velocity.
- **A Collection of Obstacles** – For each level, obstacles are positioned differently, and it's important to specify that. For some obstacles it might be desirable to disable the damage infliction component, while for others it might be good to maximize the damage caused by colliding with them. Also, there are multiple textures associated with different obstacle types, so I wanted to specify the obstacles to render regardless of the selected level type.
- **A Collection of Monsters** – Obstacles are not the only component that can damage the bear during gameplay. There are also monsters that can pop up and shoot at the main character. Similar to the bear, monsters represent a living entity and have some specific properties, such as the initial health, damage, starting position, velocity, and type.
- **Buttons** – These are the bonus point boosters in FallFury. The player collects as many of those as possible, and each of them should be individually positioned to form either a trail or a shape.
- **Power-Ups** – With the basic set of abilities, the bear is able to get some bonuses such as a cape that will speed-up his descent or a bubble that will protect him from incoming shells.

The first build of the level engine integrated into FallFuryused percentage-based relative values to position elements on the screen. Although this seemed like a good idea at the time, it became problematic because

- It required the level to be a fixed size, which restricted element addition and level extension.

- It caused problems with obstacles that needed to be scaled and therefore had a non-standard size.
- Small modifications were harder to make because minimal adjustments would throw off the relative position.

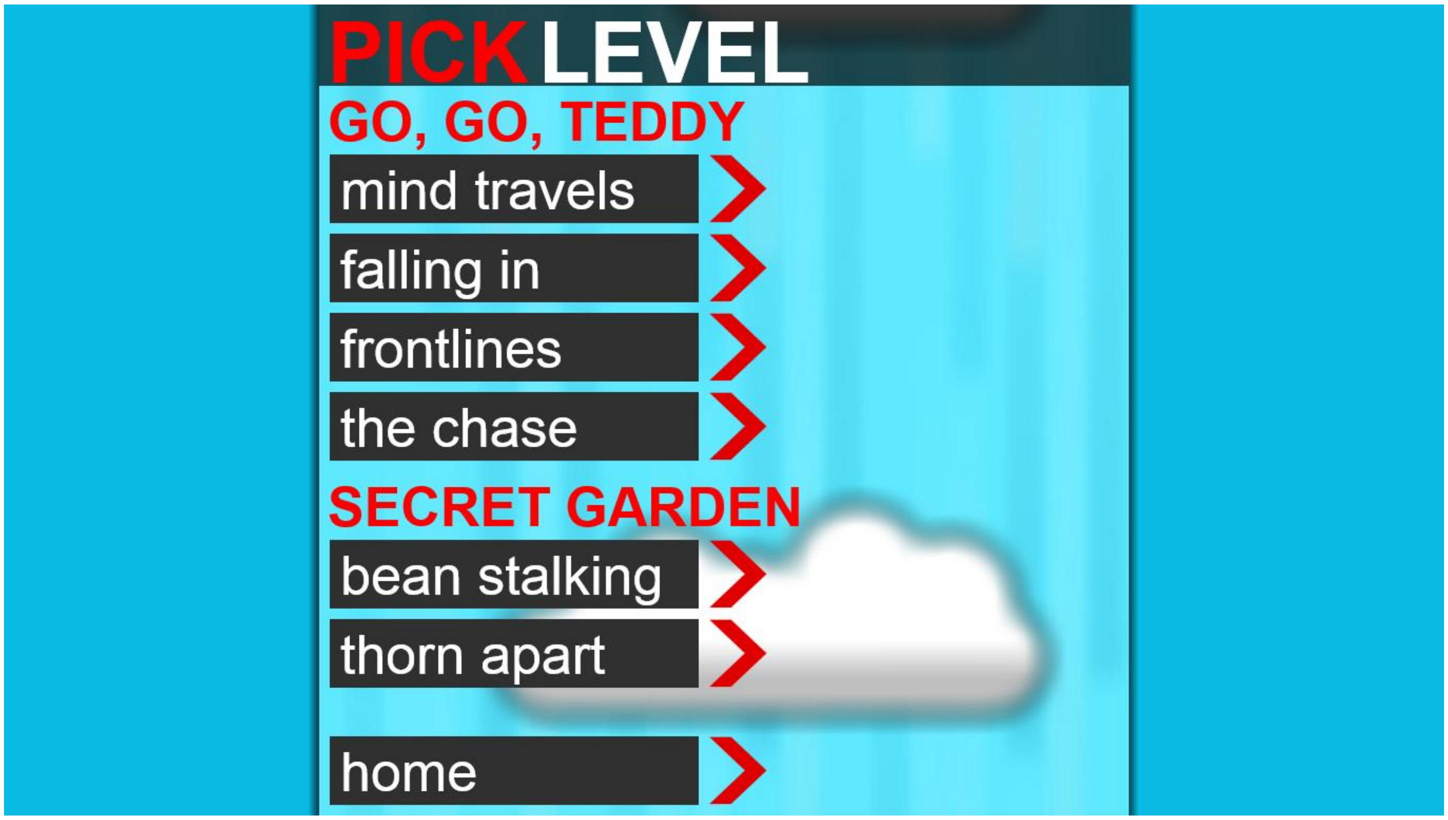
So, I switched to a pixel-based conditioning in which each position is relative to zero. With this in place, levels can be infinitely long (within the context of the machine's rendering and memory capabilities)and extra elements can be more seamlessly added.

Additionally, levels need to be packaged together in individual sets, normally grouped by themes, without restriction. This is achieved with the help of an extra XML file,called **core.xml**, which keeps track of level tiers, and acts as a container that allows the developer to name and easily enable or disable specific levels .

The structure for the **core.xml** file looks like this:

```
<tiers>
  <tier name="GO, GO, TEDDY">
    <level name="mind travels" file="GoGoTeddy\mind.xml"></level>
    <level name="falling in" file="Nightmare\full_pilot.xml"></level>
    <level name="frontlines" file="Nightmare\the_beginning.xml"></level>
    <level name="the chase" file="Nightmare\chasing_monsters.xml"></level>
  </tier>
  <tier name="SECRET GARDEN">
    <level name="bean stalking" file="Garden\bean_stalking.xml"/>
    <level name="thorn apart" file="Garden\thorn_apart.xml"/>
  </tier>
  <!--<tier name="Obstacle ***TEST***">
    <level name="Nightmare 0" file="test\Obstacle\nightmare\0.xml" />
    <level name="Nightmare 1" file="test\Obstacle\nightmare\1.xml" />
    <level name="Bean 0" file="test\Obstacle\bean\0.xml" />
    <level name="Bean 1" file="test\Obstacle\bean\1.xml" />
    <level name="Dream 0" file="test\Obstacle\dream\0.xml" />
    <level name="Dream 1" file="test\Obstacle\dream\1.xml" />
  </tier>-->
  <!--<tier name="Death ***TEST***">
    <level name="Monster 0" file="test\death\0.xml" />
  </tier>-->
  <!--<tier name="Monster ***TEST***">
    <level name="0" file="test\Monsters\0.xml" />
    <level name="1" file="test\Monsters\1.xml" />
    <level name="2" file="test\Monsters\2.xml" />
    <level name="3" file="test\Monsters\3.xml" />
    <level name="4" file="test\Monsters\4.xml" />
    <level name="5" file="test\Monsters\5.xml" />
    <level name="6" file="test\Monsters\6.xml" />
    <level name="7" file="test\Monsters\7.xml" />
    <level name="8" file="test\Monsters\8.xml" />
    <level name="9" file="test\Monsters\9.xml" />
    <level name="10" file="test\Monsters\10.xml" />
  </tier>-->
  <!--<tier name="MEDALS *****TEST******">
    <level name="gold" file="test\Medals\gold.xml"></level>
    <level name="silver" file="test\Medals\silver.xml"></level>
    <level name="bronze" file="test\Medals\bronze.xml"></level>
  </tier>
  <tier name="Buttons ***TEST***">
    <level name="1" file="test\Buttons\single.xml" />
    <level name="Lots" file="test\Buttons\multiple.xml" />
  </tier>
  <tier name="Obstacles ***TEST***">
    <level name="Cape" file="test\PowerUps\0.xml" />
  </tier>-->
</tiers>
```

Tiers that are commented out are ignored and the included levels aren't on the game list. Also, the paths indicated for each **file** attribute—for each individual tier—are relative to the game folder itself. There is no limit on the number of subfolders that can be included in the path. The above structure will render this level set:



The Level XML

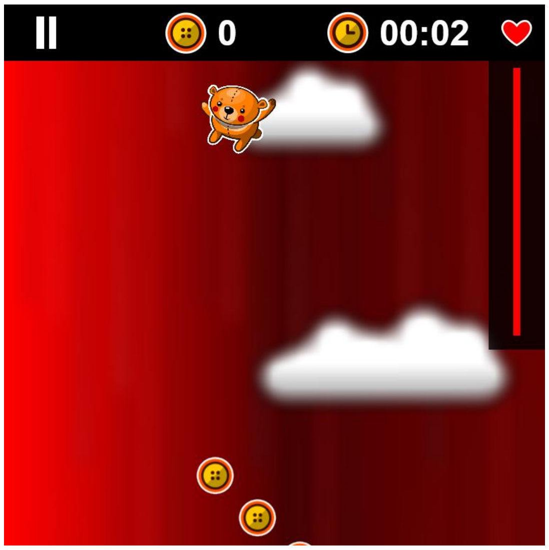
Let’s now take a look at the layout of the level descriptor XML file:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<level type="0">
  <meta score="0" buttonPrice="10"></meta>
  <bear maxHealth="100" startPosition="300" velocity="8.0" damage="11" criticalDamage="20"
       defaultAmmo="100" />
  <obstacles>
    <obstacle type="1" x="119" y="2300" inflictsDamage="true" healthDamage="5" rotation="3.14"
              scale="1" />
    <obstacle type="3" x="534.5" y="3000" inflictsDamage="true" healthDamage="5" rotation="0"
              scale="1" />
    <obstacle type="3" x="534.5" y="3300" inflictsDamage="true" healthDamage="5" rotation="0"
              scale="1" />
    <obstacle type="1" x="119" y="4200" inflictsDamage="true" healthDamage="5" rotation="3.14"
              scale="1" />
    <obstacle type="2" x="546" y="5000" inflictsDamage="true" healthDamage="5" rotation="0"
              scale="1" />
    <obstacle type="1" x="119" y="5800" inflictsDamage="true" healthDamage="5" rotation="3.14"
              scale="1" />
    <obstacle type="2" x="546" y="6600" inflictsDamage="true" healthDamage="5" rotation="0"
              scale="1" />
  </obstacles>
  <monsters>
    <monster lifetime="3000" scale=".2" velocityX="2" velocityY="2" type="0" x="460" y="19900"
             maxHealth="80" bonus="100" lives="0" damage="10" criticalDamage="8"
             defaultAmmo="50" />
    <monster lifetime="3000" scale=".2" velocityX="2" velocityY="2" type="1" x="460" y="25000"
             maxHealth="80" bonus="100" lives="0" damage="10" criticalDamage="8"
             defaultAmmo="50" />
    <monster lifetime="3000" scale=".2" velocityX="2" velocityY="2" type="2" x="460" y="34500"
             maxHealth="80" bonus="100" lives="0" damage="10" criticalDamage="8"
             defaultAmmo="50" />
    <monster lifetime="6000" scale=".4" velocityX="2" velocityY="2" type="3" x="460" y="41400"
             maxHealth="180" bonus="100" lives="0" damage="17" criticalDamage="8"
             defaultAmmo="50" />
  </monsters>
  <buttons>
    <button x="300" y="800" />
    <button x="360" y="800" />
    <button x="300" y="860" />
    <button x="360" y="860" />
    <button x="300" y="920" />
    <button x="360" y="920" />
    <button x="300" y="980" />
    <button x="360" y="980" />
  </buttons>
  <powerups>
    <powerup category="1" type="4" x="140" y="9200" effect="3" lifespan="4"></powerup>
    <powerup category="1" type="3" x="480" y="19800" effect="10" lifespan="6"></powerup>
    <powerup category="1" type="0" x="480" y="27500" effect="10" lifespan="6"></powerup>
    <powerup category="1" type="1" x="100" y="34000" effect="10" lifespan="6"></powerup>
    <powerup category="1" type="0" x="100" y="41200" effect="10" lifespan="6"></powerup>
  </powerups>
</level>
```

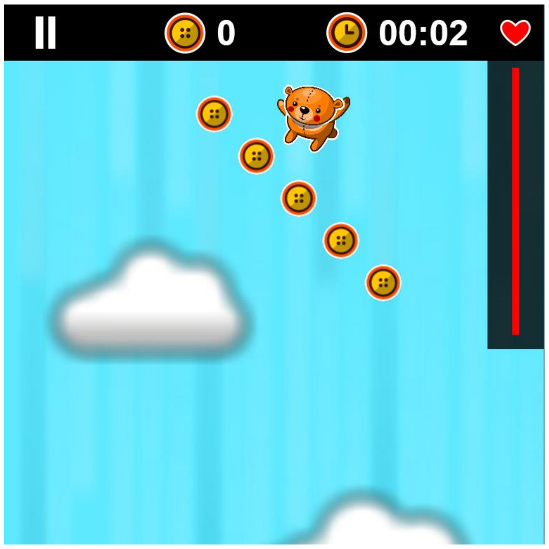
The opening **level** tag carries a type attribute. This is level theme flag. It can be set to one of the three four values:

- **0** – The Nightmare Theme
- **1** – The Magic Bean Theme
- **2** – The Dream Theme
- **3** – The Space Theme

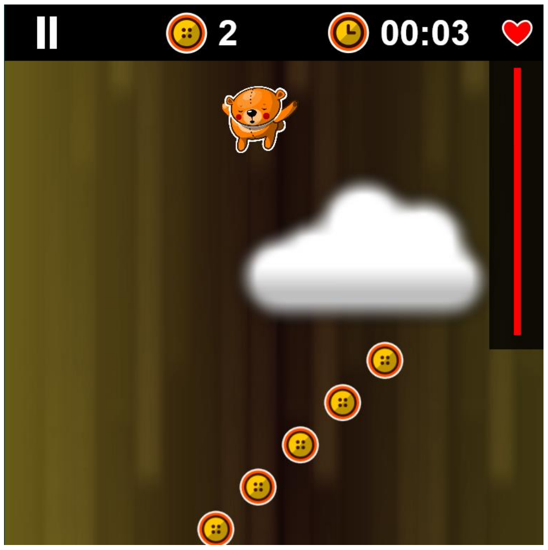
You can see the design differences in the images below:



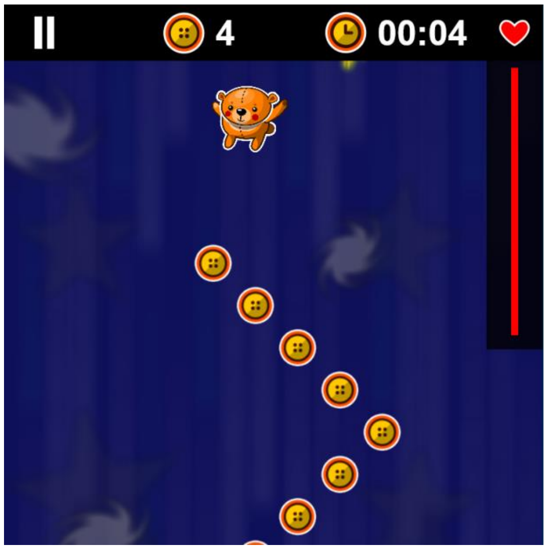
Nightmare



Magic Bean



Dream



Space

Remember, that obstacles and the level theme itself do not influence much other than the background and soundboard.

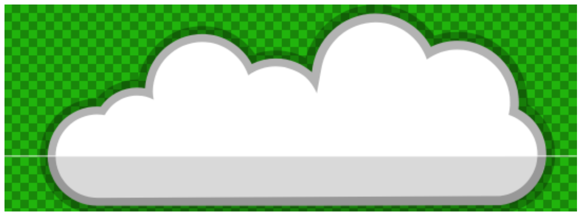
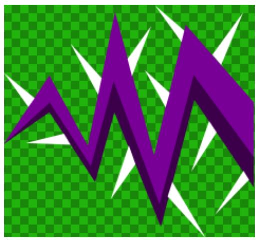
Once the type is specified, the **meta** tag brings up the **score** and **buttonPrice** attributes. If you for some reason want to create a level including an initial score, you can specify it here. And because buttons are fixed bonus assets that are all created equal, each of them carries a given bonus point weight. The score based on the collected buttons is calculated at the end of the game and relies on the value specified in the **meta** tag.

Obstacles

Next comes the obstacle collection, which is represented by the **obstacles** tag. This tag is required even if there are no obstacles on a given level. Simply use <**obstacles** /> as necessary. Each child node represents an instance of an obstacle that can be choosen from the following enum:

```
enum class ObstacleType
{
    OT_CLOUD = 0,
    OT_SPIKE_NIGHTMARE_LARGE = 1,
    OT_SPIKE_NIGHTMARE_MEDIUM = 2,
    OT_SPIKE_NIGHTMARE_SMALL = 3,
    OT_BEAN_A = 4,
    OT_BEAN_B = 5,
    OT_BEAN_C = 6,
    OT_BEAN_D = 7,
    OT_BEAN_E = 8,
    OT_SPACE_ROCKET = 9,
    OT_SPACE_COMET_A = 10,
    OT_SPACE_COMET_B = 11,
    OT_SPACE_SATELLITE = 12,
    OT_SPACE_UFO = 13,
    OT_SPACE_BALL = 14
};
```

Here is the complete table showing the appearance of each of them:

OT_CLOUD	
OT_SPIKE_NIGHTMARE_LARGE	

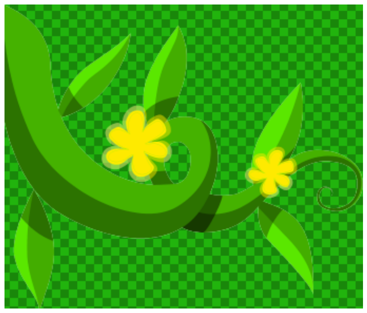
OT_SPIKE_NIGHTMARE_MEDIUM

OT_SPIKE_NIGHTMARE_SMALL

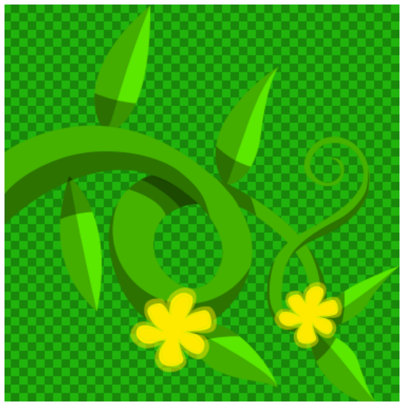
OT_BEAN_A

OT_BEAN_B

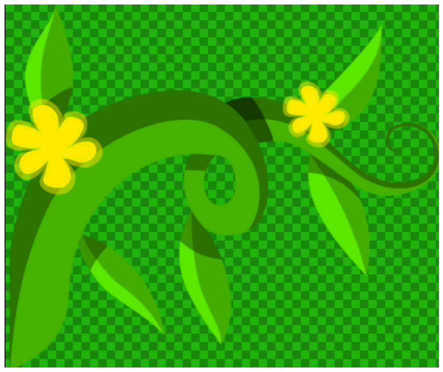
OT_BEAN_C



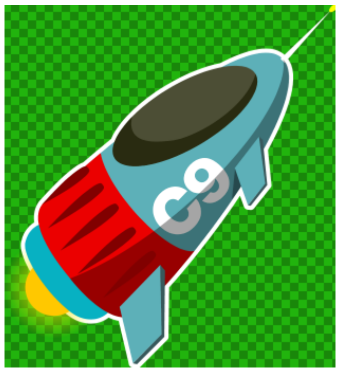
OT_BEAN_D



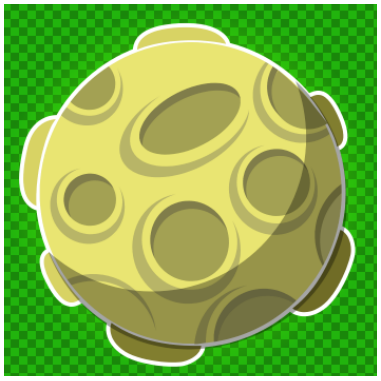
OT_BEAN_E



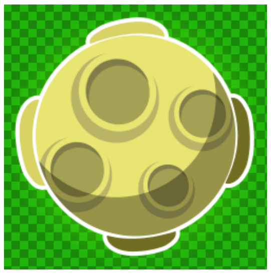
OT_SPACE_ROCKET



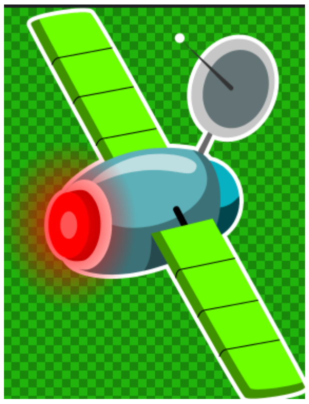
OT_SPACE_COMET_A



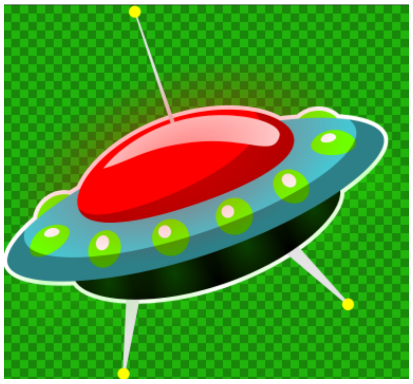
OT_SPACE_COMET_B



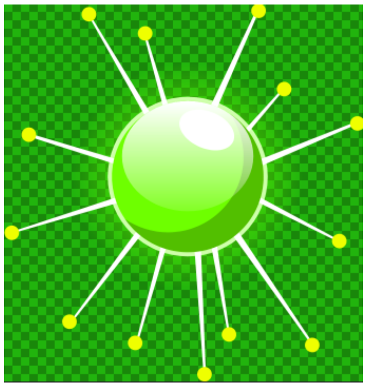
OT_SPACE_SATELLITE



OT_SPACE_UFO



OT_SPACE_BALL



No matter how the obstacles are positioned, they will either be located in the visible area or displaced outside the viewport and not displayed. The ultimate position is taken from the obstacle size and is relative to the center of the texture. For example, if an obstacle texture is 400- pixels wide, the X-relative position should be set to 200. If the position differs from the starting one, however, the obstacle texture is cut to include the area that fits in the 768 pixel wide playable zone visible. There are no restrictions regarding the Y position.

To help in level creation, some obstacles have pre-defined left and right margins. For example:

- **OT_BEAN_A:** 269.5 (Left), 498.5 (right, with 3.14 rotation)
- **OT_BEAN_B:** 128.5 (left), 638.5 (right, with 3.14 rotation)
- **OT_BEAN_C:** 172 (left), 596 (right, with 3.14 rotation)
- **OT_BEAN_D:** 188.5 (left), 579.5 (right, with 3.14 rotation)
- **OT_BEAN_E:** 206.5 (left), 561.5 (right, with 3.14 rotation)

The **inflictsDamage** attribute determines whether the obstacle harms the main character. If it is set to **false**, the character will still make the sound of colliding with it but will not lose any health points. The primary use for this attribute is level testing.If it is set to **true**, the character will loose the amount of health points indicated by the **healthDamage** attribute.

The **rotation** and **scale** attributes can be used to flip and resize the texture as needed. Rotation is measured in radians, and the scale is a normalized value in which 1.0 represents 100% of the scale.

Monsters

As with obstacles, the **monsters** node should never be omitted from the file and should at least contain a placeholder: <**monsters** />. Unlike obstacles, however, monsters are dynamic and do not have fixed positions. Moreover, monsters have limited active time during gameplay. The first attribute,**lifetime**, determines the length of the fall during which the monster will be visible in the viewport. With the **y** attribute as the Y-based position at which the monster appears, at the **y+lifetime** position the monster simply flies away if not killed.

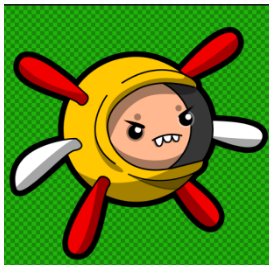
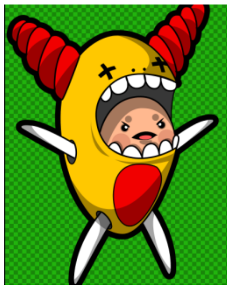
The **scale** attribute carries the same purpose as the one for the obstacle—normalized texture size relative to the size of the original image file. As such, the level designer does not have to worry about linking the width and height of the monster when resizing and can instead use a percentage-like value to scale the monster up or down, simultaneously modifying both the width and the height with zero stretching.

Nextup are **velocityX** and **velocityY**. These two attributes are used to set the motion velocity when the monster is already visible. Instead of being a static shooting entity, the enemy moves on a randomized zig-zag path at the bottom of the screen. The horizontal and vertical displacement—in pixels, per update cycle—is individually set through the values carried by the above-mentioned attributes. If necessary, this functionality can be disabled in the code-behind by assigning a fixed value for the vertical and horizontal movement for all monsters that are being loaded on a given level.

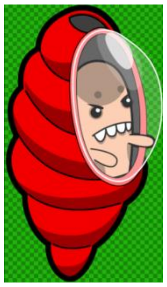
The monster type is an **integer** value that is translated in a value from the following enum (located in **MonsterType.h**):

```
enum class MonsterType
{
    MT_NIGHTMARE_A = 0,
    MT_NIGHTMARE_B = 1,
    MT_NIGHTMARE_C = 2,
    MT_MAGICBEAN_A = 3,
    MT_MAGICBEAN_B = 4,
    MT_MAGICBEAN_C = 5,
    MT_CANDYLAND_A = 6,
    MT_CANDYLAND_B = 7,
    MT_CANDYLAND_C = 8,
    MT_CANDYLAND_D = 9,
    MT_CANDYLAND_E = 10
};
```

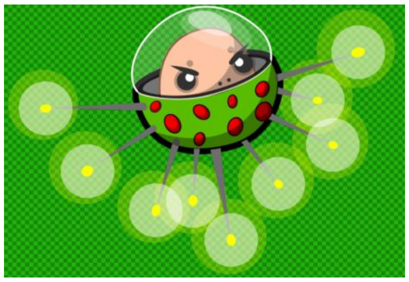
Here is a table that shows the texture associated with each identifier:

MT_NIGHTMARE_A	
MT_NIGHTMARE_B	

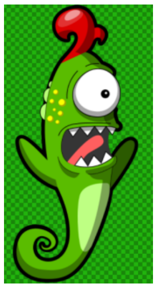
MT_NIGHTMARE_C



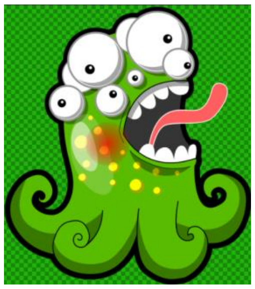
MT_MAGICBEAN_A



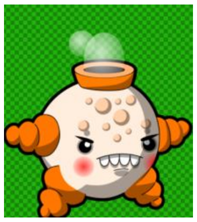
MT_MAGICBEAN_B



MT_MAGICBEAN_C



MT_CANDYLAND_A



MT_CANDYLAND_B

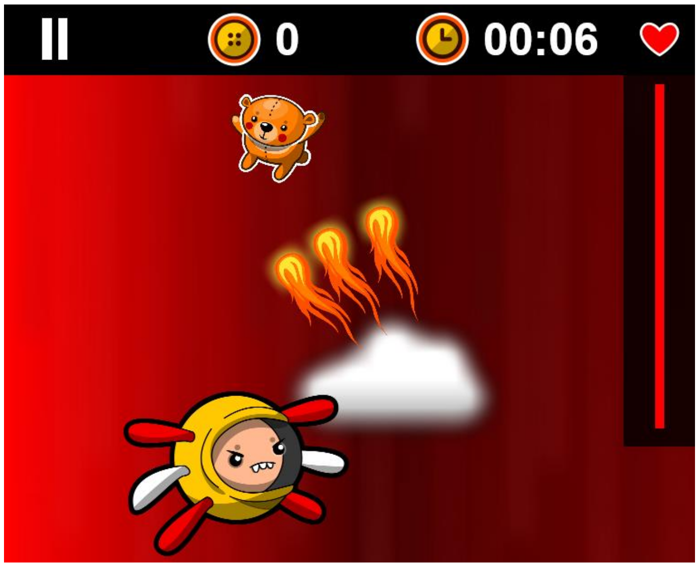
MT_CANDYLAND_C

MT_CANDYLAND_D

MT_CANDYLAND_E

Each monster has three separate textures associated with it. The three textures are cycled inside the update loop for each monster entity when the monster becomes visible, creating the movement effect:

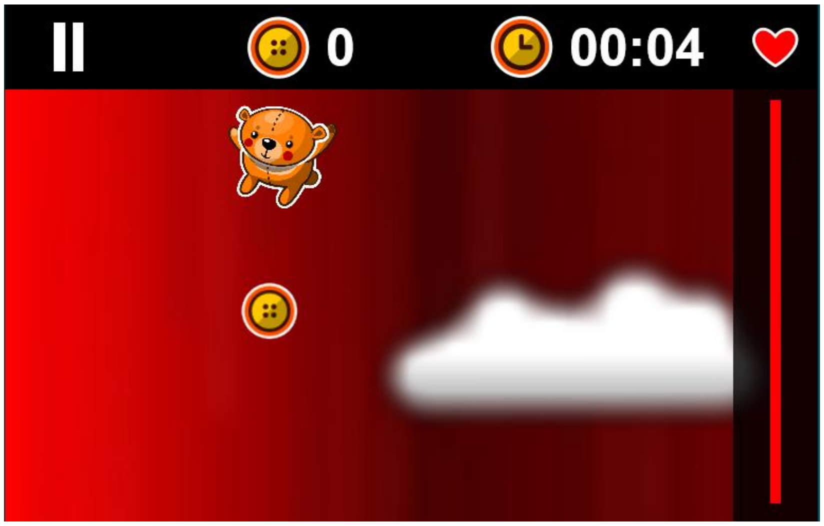
Each monster currently shoots only one type of ammo: a red plasma ball. Be aware, however, that there is a preprogrammed condition in which the last monster in the XML file collection is automatically considered the final boss. This means the **scale**, **maxHealth** and **lifetime** properties must be manually adjusted to reflect the effect. Without doing so, the last monster will, regardless of the XML setting, switch in-game to a triple fireball shot that inflicts three times the damage indicated by the **damage** attribute:



Each monster can have limited ammo as set by the **defaultAmmo** attribute. In the case that the ammo is exhausted before the monster expires, the monster will continue its motion at the bottom of the screen without inflicting direct damage to the main character.

Buttons

These are the least complex entities and only carry an X and a Y position. Given those coordinates, relative to the left margin of the visible area, a button texture is rendered:



Once picked-up, the button counter is increased by one and the meta-score incremented by the value set in the **meta** tag at the beginning of the level XML as long as the feature is activated in the code-behind.

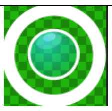
Power-ups

To make the game more fun, there are bonus elements that can be picked up by the bear in order to enhance its performance or protection. These elements are declared in the **<powerups/>** collection. First and foremost, it is important to declare whether the power-up is positive or negative. In the current version of FallFury, only positive power-ups are included. Nonetheless, the harness for negative ones is already integrated in the parser. Therefore, the **category** attribute should be set to 1 if the power-up has a positive effect and 0 for a negative effect. This value will only have an effect over the sound played when the bonus is collected.

The power-up type can be one of the following (enum located in **PoweupType.h**):

```
enum class PowerupType
{
    HEALTH = 0,
    HELMET = 1,
    PARACHUTE = 2,
    BUBBLE = 3,
    CAPE = 4,
    AXE = 5,
    BOOMERANG = 6,
    HAMMER = 7,
    KNIFE = 8,
    PLASMA_BALL = 9,
    CIRCLE = 10
};
```

The table below shows the power-up texture appearance. Behaviors are already defined in the game and influenced by only the **effect** and **lifespan** (seconds) attributes:

HEALTH	Restores the character health, incremented by the value in **effect**. The **lifespan** attribute is ignored.	
HELMET	Adds a helmet to the bear, setting the maximum health to the **effect** value. Active for the duration of **lifespan**.	
PARACHUTE	Slows down the fall of the bear, setting the descent velocity to the **effect** value. Active for the duration of **lifespan**.	
BUBBLE	Wraps the character in a protective bubble, setting the maximum health to the value of the **effect** attribute. Active for the duration of **lifespan**.	
CAPE	Accelerates the descent by multiplying the velocity by the value of **effect**. Active for the duration of **lifespan**.	
AXE	Sets the current character weapon to an axe. The damage is determined by the **effect** attribute and **lifespan** is ignored.	

BOOMERANG	Sets the current character weapon to a boomerang. The damage is determined by the **effect** attribute and **lifespan** is ignored.	
HAMMER	Sets the current character weapon to a hammer. The damage is determined by the **effect** attribute and **lifespan** is ignored.	
KNIFE	Sets the current character weapon to a knife. The damage is determined by the **effect** attribute and **lifespan** is ignored.	
PLASMA_BALL	Sets the current character weapon to a plasma ball. The damage is determined by the **effect** attribute and **lifespan** is ignored.	

CIRCLE is a helper power-up that has no effect on the bear and is instead used as an additional texture overlay along with any other power-up in order to create a pulsating circle effect.

Conclusion

FallFury ships with a dozen of sample levels that showcase all of the elements described in the article. At the moment, a level editor is in the works, but it isn't too complicated to build XML files manually. To do so, you need to consider the pixel-based locations and ensure that they're all in the visible area—the game engine will automatically handle all other displacements and adjustments.

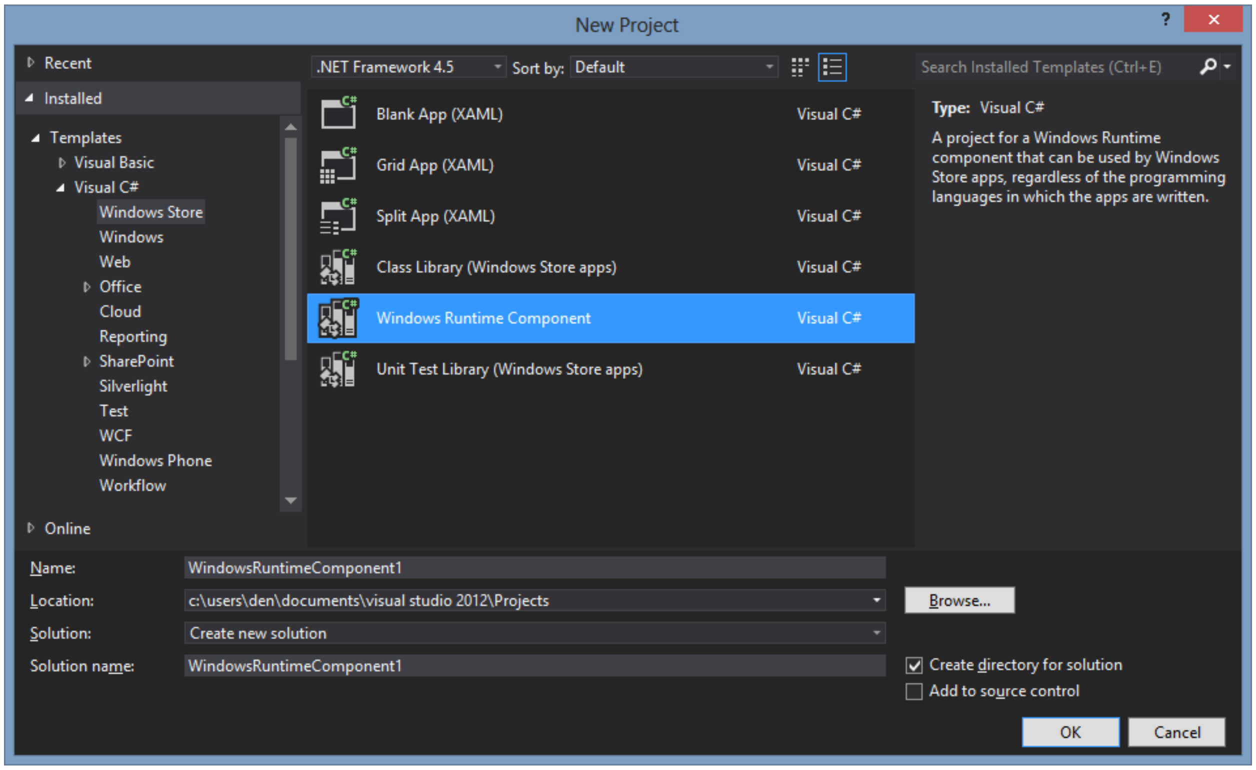
Chapter 6 – Rendering Level Elements

In the previous article, you learned how to build level XML files in order to create playable levels. This article discusses how the internal parser works and how XML nodes become items on the game screen.

Project Interop – The C# Parser

Thanks to classes such as XmlSerializer and XDocument, parsing XML in .NET Framework is not a complicated task. Similarly, in the WinRT world there are alternatives such as Windows::Data::Xml::Dom. That being said, using the parser skeleton in C# this project leverages the existing codebase, adapting it to the specific level reading needs.

Start by creating a new WinRT project that is the part of the existing solution. Make sure that you create a Windows Runtime Component:



New Project

Recent
Installed
Templates
Visual Basic
Visual C#
Windows Store
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Windows Phone
Workflow
Online

.NET Framework 4.5   Sort by: Default   Search Installed Templates (Ctrl+E)

Blank App (XAML)	Visual C#
Grid App (XAML)	Visual C#
Split App (XAML)	Visual C#
Class Library (Windows Store apps)	Visual C#
Windows Runtime Component	Visual C#
Unit Test Library (Windows Store apps)	Visual C#

Type: Visual C#

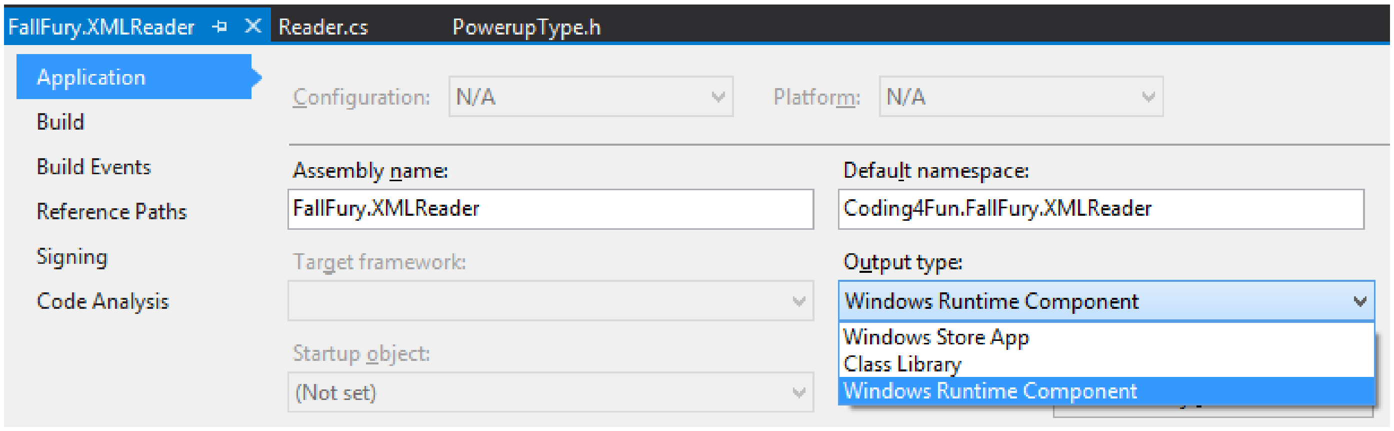
A project for a Windows Runtime component that can be used by Windows Store apps, regardless of the programming languages in which the apps are written.

Name: WindowsRuntimeComponent1
Location: c:\users\den\documents\visual studio 2012\Projects   Browse...
Solution: Create new solution
Solution name: WindowsRuntimeComponent1

☑ Create directory for solution
☐ Add to source control

OK   Cancel

There are major differences between the .NET and WinRT stacks, especially when it comes to building code that can be invoked from any potential WinRT project type, as is the case here. Make sure that the output of the newly created project is a Windows Runtime Component (WinMD file):



Configuration: N/A
Platform: N/A
Assembly name: FallFury.XMLReader
Default namespace: Coding4Fun.FallFury.XMLReader
Target framework:
Output type: Windows Runtime Component
Startup object: (Not set)

The XMLReader project reads much more than the level data—it also manages internal XML metadata such as user scores. This article, however, focuses only on level-related aspects of the engine.

Reader is the one core class used here. It contains the **ReadXml** method, which receives the file name and the type of the XML file to be read. Regardless of the name, the XML file type determines the parsing rules, the root lookup location, and the produced output. The tier (level set) and individual level data are stored internally in the application folder, and the score metadata is outside the sandbox in the application data folder:

```
async Task<Object> ReadXml(string fileName, XmlType type)
{
    StorageFile file = null;
    StorageFolder folder = null;

    if (type == XmlType.LEVEL || type == XmlType.TIERS)
    {
        folder = Windows.ApplicationModel.Package.Current.InstalledLocation;
        folder = await folder.GetFolderAsync("Levels");
    }
    else if (type == XmlType.HIGHSCORE)
    {
        folder = ApplicationData.Current.LocalFolder;
        folder = await folder.GetFolderAsync("Meta");
    }

    file = await folder.GetFileAsync(fileName);

    Object returnValue = null;
    string data;

    using (IRandomAccessStream readStream = await file.OpenAsync(FileAccessMode.Read))
    {
        using (Stream inStream = Task.Run(() => readStream.AsStreamForRead()).Result)
        {
            using (StreamReader reader = new StreamReader(inStream))
            {
                data = reader.ReadToEnd();
            }
        }
    }

    if (type == XmlType.LEVEL)
    {
        returnValue = ExtractLevelData(data);
    }
    else if (type == XmlType.HIGHSCORE)
    {
        returnValue = ExtractScoreData(data);
    }
    else if (type == XmlType.TIERS)
    {
        returnValue = ExtractTierData(data);
    }

    return returnValue;
}
```

Before the level data is extracted, FallFury needs to know about the existing tiers pointing to those levels. That's where **ExtractTierData**, an internal method that accepts a raw XML string and return a **TierSet** instance, comes in:

```
public sealed class TierSet
{
    public Tier[] Tiers { get; set; }
}
```

A **Tier** class has the following structure:

```
public sealed class Tier
{
    public string Name { get; set; }
    public string[] LevelNames { get; set; }
    public string[] LevelFiles { get; set; }
}
```

I am using the most fundamental types – arrays instead of generic collections, to maximize the portability of the code. The **ExtractTierData** method handles the XML-to-Object transformation:

```
TierSet ExtractTierData(string tierString)
{
    TierSet set = new TierSet();

    XDocument document = XDocument.Parse(tierString);

    set.Tiers = new Tier[document.Root.Elements().Count()];

    int tierCounter = 0;

    foreach (XElement element in document.Root.Elements())
    {
        Tier tier = new Tier();
        tier.Name = element.Attribute("name").Value;

        tier.LevelFiles = new string[element.Elements("level").Count()];
        tier.LevelNames = new string[element.Elements("level").Count()];


        int count = 0;
        foreach (XElement lElement in element.Elements("level"))
        {
            tier.LevelFiles[count] = lElement.Attribute("file").Value;
            tier.LevelNames[count] = lElement.Attribute("name").Value;
            count++;
        }

        set.Tiers[tierCounter] = tier;
        tierCounter++;
    }

    return set;
}
```

I am running a coupled array block here, utilizing one array for level names and another for file locations. This is a very basic implementation of a key-value pair that depends on code-based coupling. **ExtractTierData**, however, is not exposed publicly and the C++ layer of FallFury will not access it because **TierSet** is not exposed through a compatible async method. Looking back at **ReadXml**, might seem like the answer to this problem. But a **Task<Object>** is not an interop-compatible type, which means I use **ReadXmlAsync**, which proxies the call through an IAsyncOperation:

```
public IAsyncOperation<Object> ReadXmlAsync(string filename, XmlType type)
{
    return (IAsyncOperation<Object>)AsyncInfo.Run((CancellationToken token) =>
                                                    ReadXml(filename, type));
}
```

AsyncInfo.Run starts a WinRT async operation and handles the returned result, regardless of the selected file path or type.

Level-based data is extracted in a similar manner to the tier data. The difference lies in the used internal models, as well as the node reading order:

```
Level ExtractLevelData(string levelString)
{
    XDocument document = XDocument.Parse(levelString);

    Level level = new Level();
    level.LevelMeta = new Meta();

    level.LevelMeta.Score =
        Convert.ToInt32(document.Root.Element("meta").Attribute("score").Value);
    level.LevelMeta.ButtonPrice =
        Convert.ToInt32(document.Root.Element("meta").Attribute("buttonPrice").Value)
    level.LevelMeta.LevelType =
        (LevelType)Convert.ToInt32(document.Root.Attribute("type").Value);

    int count = document.Root.Element("obstacles").Elements().Count();
    level.Obstacles = new Obstacle[count];

    count = 0;
    foreach (XElement element in document.Root.Element("obstacles").Elements())
    {
        Obstacle obstacle = new Obstacle();
        obstacle.HealthDamage = Convert.ToSingle(element.Attribute("healthDamage").Value);
        obstacle.InflictsDamage =
            Convert.ToBoolean(element.Attribute("inflictsDamage").Value);
        obstacle.Rotation = Convert.ToSingle(element.Attribute("rotation").Value);
        obstacle.Scale = Convert.ToSingle(element.Attribute("scale").Value);
        obstacle.X = Convert.ToSingle(element.Attribute("x").Value);
        obstacle.Y = Convert.ToSingle(element.Attribute("y").Value);
        obstacle.Type = (ObstacleType)Convert.ToInt32(element.Attribute("type").Value);
        level.Obstacles[count] = obstacle;
        count++;
    }

    count = document.Root.Element("monsters").Elements().Count();
    level.Monsters = new Monster[count];

    count = 0;
    foreach (XElement element in document.Root.Element("monsters").Elements())
    {
        Monster monster = new Monster();
        monster.CriticalDamage = Convert.ToSingle(element.Attribute("criticalDamage").Value);
        monster.Damage = Convert.ToSingle(element.Attribute("damage").Value);
        monster.DefaultAmmo = Convert.ToInt32(element.Attribute("defaultAmmo").Value);
        monster.MaxHealth = Convert.ToSingle(element.Attribute("maxHealth").Value);
        monster.X = Convert.ToSingle(element.Attribute("x").Value);
        monster.Y = Convert.ToSingle(element.Attribute("y").Value);
        monster.VelocityX = Convert.ToSingle(element.Attribute("velocityX").Value);
        monster.Lifetime = Convert.ToSingle(element.Attribute("lifetime").Value);
        monster.Type = (MonsterType)Convert.ToInt32(element.Attribute("type").Value);
        monster.Bonus = Convert.ToInt32(element.Attribute("bonus").Value);
        monster.Scale = Convert.ToSingle(element.Attribute("scale").Value);

        level.Monsters[count] = monster;
        count++;
    }

    count = document.Root.Element("buttons").Elements().Count();
    level.Buttons = new Button[count];

    count = 0;
    foreach (XElement element in document.Root.Element("buttons").Elements())
    {
        Button button = new Button();

        button.X = Convert.ToSingle(element.Attribute("x").Value);
        button.Y = Convert.ToSingle(element.Attribute("y").Value);

        level.Buttons[count] = button;
        count++;
    }

    count = document.Root.Element("powerups").Elements().Count();
    level.Powerups = new Powerup[count];

    count = 0;
    foreach (XElement element in document.Root.Element("powerups").Elements())
    {
        Powerup powerup = new Powerup();

        powerup.X = Convert.ToSingle(element.Attribute("x").Value);
        powerup.Y = Convert.ToSingle(element.Attribute("y").Value);
        powerup.Category =
            (PowerupCategory)Convert.ToInt32(element.Attribute("category").Value);
        powerup.Type = (PowerupType)Convert.ToInt32(element.Attribute("type").Value);
        powerup.Lifespan = Convert.ToSingle(element.Attribute("lifespan").Value);
        powerup.Effect = Convert.ToSingle(element.Attribute("effect").Value);

        level.Powerups[count] = powerup;
        count++;
    }

    Bear bear = new Bear();
    XElement bearElement = document.Root.Element("bear");
    bear.CriticalDamage = Convert.ToSingle(bearElement.Attribute("criticalDamage").Value);
    bear.Damage = Convert.ToSingle(bearElement.Attribute("damage").Value);
    bear.DefaultAmmo = Convert.ToInt32(bearElement.Attribute("defaultAmmo").Value);
    bear.MaxHealth = Convert.ToSingle(bearElement.Attribute("maxHealth").Value);
    bear.StartPosition = Convert.ToSingle(bearElement.Attribute("startPosition").Value);
    bear.Velocity = Convert.ToSingle(bearElement.Attribute("velocity").Value);

    level.GameBear = bear;

    return level;
}
```

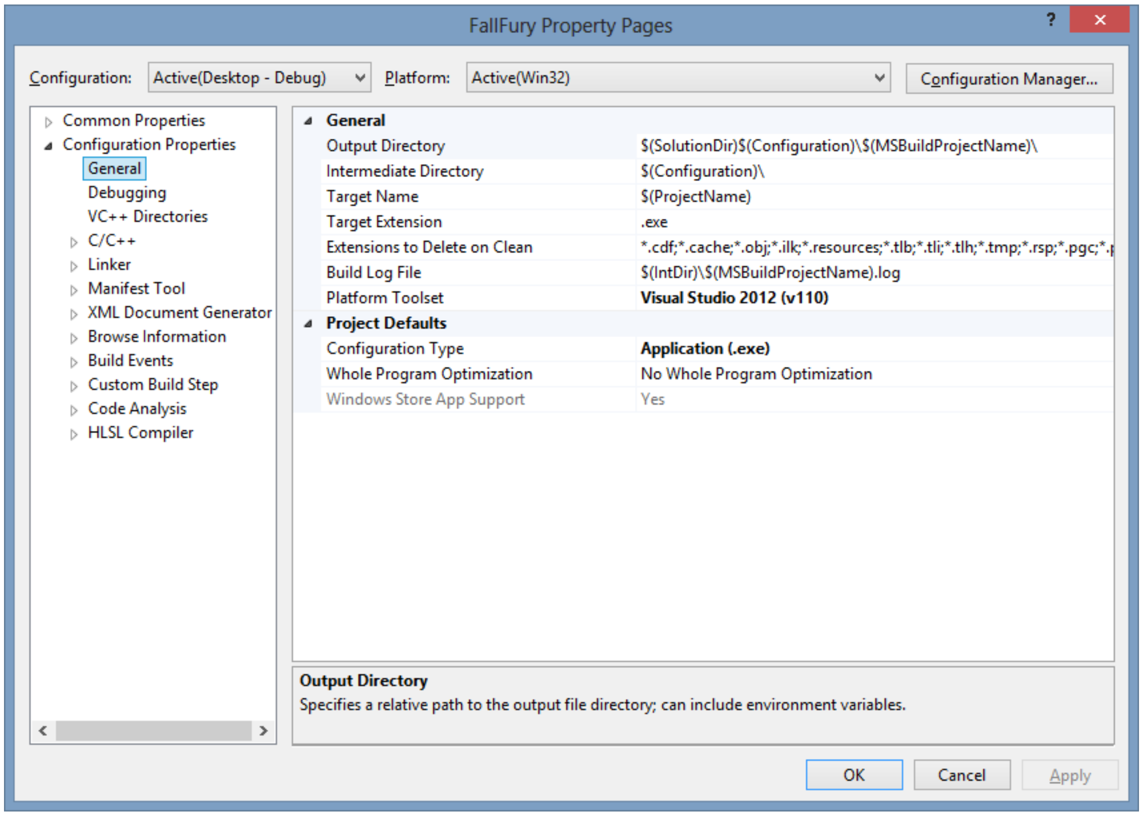
As a result, the **Level** instance has an internal counterpart in the C++ project:

```
public sealed class Level
{
    public Bear GameBear { get; set; }
    public Meta LevelMeta { get; set; }
    public Obstacle[] Obstacles { get; set; }
    public Monster[] Monsters { get; set; }
    public Button[] Buttons { get; set; }
    public Powerup[] Powerups { get; set; }
}
```

From C# to C++ - Leveraging the WinRT Component

At this point, the C#-based WinRT component is complete and can be integrated into the C++ project. This can be done in two ways: either by adding a reference to the project itself or by adding a reference to the generated WinMD file. Both methods will ultimately produce the same output, but it's easier to debug and modify coupled projects on the go, so I went with the first option.

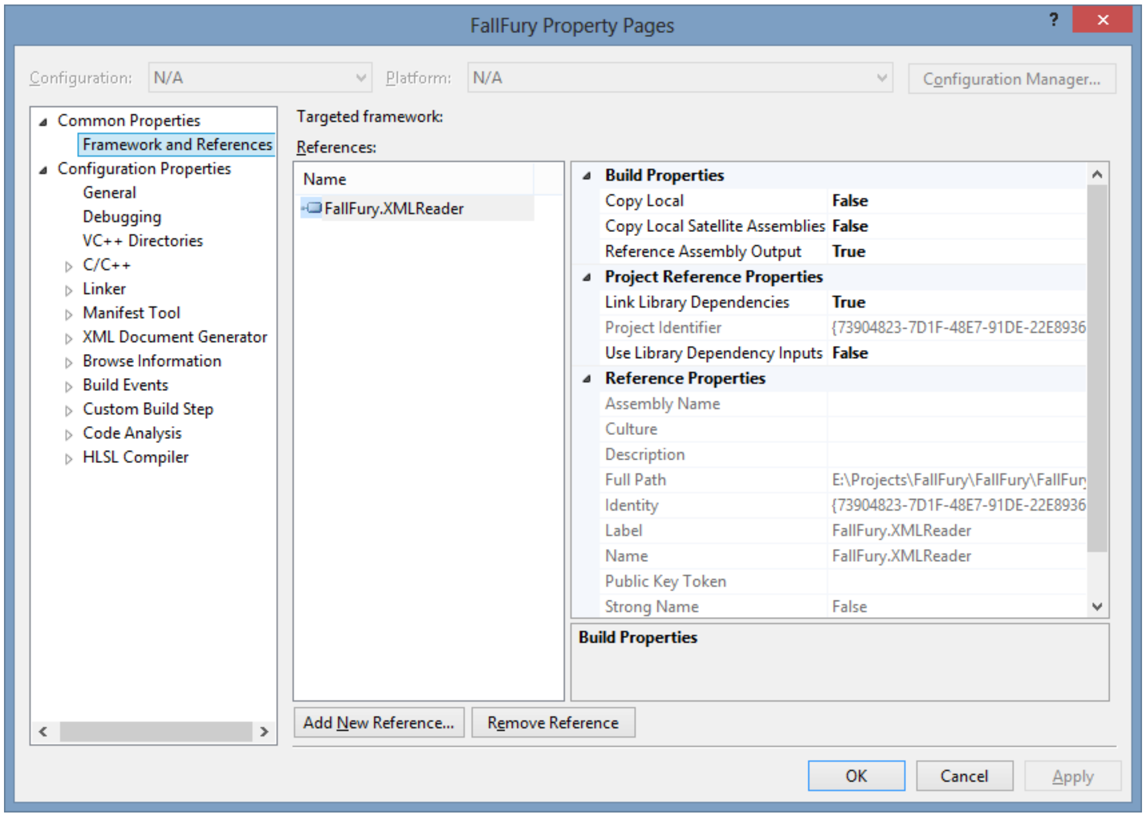
To add a reference to an internal Windows Runtime Component project, right click on the C++ project in **Solution Explorer** and select **Properties**. You will see a dialog like this:



General	
Output Directory	$(SolutionDir)$(Configuration)\$(MSBuildProjectName)\
Intermediate Directory	$(Configuration)\
Target Name	$(ProjectName)
Target Extension	.exe
Extensions to Delete on Clean	*.cdf;*.cache;*.obj;*.ilk;*.resources;*.tlb;*.tli;*.tlh;*.tmp;*.rsp;*.pgc;*.p
Build Log File	$(IntDir)\$(MSBuildProjectName).log
Platform Toolset	**Visual Studio 2012 (v110)**
Project Defaults	
Configuration Type	**Application (.exe)**
Whole Program Optimization	No Whole Program Optimization
Windows Store App Support	Yes

Output Directory
Specifies a relative path to the output file directory; can include environment variables.

Select the **Common Properties** node in the tree on the left and open the **Framework and References** page:


FallFury Property Pages

Configuration: N/A   Platform: N/A   Configuration Manager...

Targeted framework:

References:

Name
FallFury.XMLReader

Build Properties	
Copy Local	**False**
Copy Local Satellite Assemblies	**False**
Reference Assembly Output	**True**
Project Reference Properties	
Link Library Dependencies	**True**
Project Identifier	{73904823-7D1F-48E7-91DE-22E8936
Use Library Dependency Inputs	**False**
Reference Properties	
Assembly Name	
Culture	
Description	
Full Path	E:\Projects\FallFury\FallFury\FallFur
Identity	{73904823-7D1F-48E7-91DE-22E8936
Label	FallFury.XMLReader
Name	FallFury.XMLReader
Public Key Token	
Strong Name	False

Build Properties

In the sample screen capture above, the **FallFury.XMLReader** project is already added as a reference. For a new project, simply click on **Add New Reference** and add any compatible project or third-party extension. As soon as the reference is added, the publicly exposed methods can be accessed.

In **DirectXPage.cpp** I have a method called **LoadLevelData** that allows me to load the list of the registered levels from **core.xml** as well as build the visual tree for the menu items, whichallows players to select a level:

```
void DirectXPage::LoadLevelData()
{
    Coding4Fun::FallFury::XMLReader::Reader^ reader =
         ref new Coding4Fun::FallFury::XMLReader::Reader();

     Windows::Foundation::IAsyncOperation<Platform::Object^>^ result =
         reader->ReadXmlAsync("core.xml",
             Coding4Fun::FallFury::XMLReader::Models::XmlType::TIERS);

    result->Completed =
        ref new AsyncOperationCompletedHandler<Platform::Object^>(this,
           &DirectXPage::OnLevelLoadCompleted);
}
```

Following the normal asynchronous pattern, as well as the structure of the method exposed in the Reader class, I am calling **ReadXmlAsync** and getting the result in **OnLevelLoadCompleted** when the operation reaches its final stage. It is worth mentioning that the associated AsyncOperationCompletedHandler is invoked even when the reading fails; therefore, the invocation of that callback does not on its own mean that the necessary data is obtained:

Here is what happens when **OnLevelLoadCompleted** is called:

```
void DirectXPage::OnLevelLoadCompleted(IAsyncOperation<Platform::Object^> ^op, AsyncStatus s)
{
    if (s == AsyncStatus::Completed)
    {
        auto set = (Coding4Fun::FallFury::XMLReader::Models::TierSet^)op->GetResults();

        auto tiers = set->Tiers;

        int levelCounter = 0;

        for (auto tier = tiers->begin(); tier != tiers->end(); tier++)
        {
            StackPanel^ panel = ref new StackPanel();

            TextBlock^ levelTierTitle = ref new TextBlock();
            levelTierTitle->Text = (*tier)->Name;
            levelTierTitle->Style =
                (Windows::UI::Xaml::Style^)Application::Current->Resources
                    ->Lookup("LevelSelectTierItemText");
            levelTierTitle->RenderTransform = ref new TranslateTransform();
            panel->Children->Append(levelTierTitle);

            levelNames = (*tier)->LevelNames;
            auto levelFiles = (*tier)->LevelFiles;

            int max = levelNames->Length;

            for(int i = 0; i < max; i++)
            {
                MenuItem^ item = ref new MenuItem();
                item->Tag = levelCounter;
                item->Label = levelNames[i];
                item->HorizontalAlignment = Windows::UI::Xaml::HorizontalAlignment::Left;
                item->OnButtonSelected +=
                    ref new MenuItem::ButtonSelected(this, &DirectXPage::OnLevelButtonSelected);
                item->RenderTransform = ref new TranslateTransform();

                m_renderer->Levels.Insert(levelCounter,levelFiles[i]);

                panel->Children->Append(item);
                levelCounter++;
            }

            stkLevelContainer->Items->Append(panel);
        }
    }
}
```

Here, AsyncStatus can also be **Cancelled** or **Error**, so checking for **Complete** ensures that the expected result is processed further inside the context of the callback.

As the returned **TierSet** exposes the **Tiers** array, I am simply iterating through each of the existing items to create independent tier blocks, grouped in **StackPanel** elements, coupled with level-specific **MenuItem** instances that present the user with a choice of playing a given level. The level ID is carried in the Tag property and it will be used to identify the selected button when **DirectXPage::OnLevelButtonSelected** is triggered:

Reading Level-specific Data

Level data is read in the **GamePlayScreen** once the selected menu button passes the name and level identifiers. The **LoadLevelXml** method is called as the screen begins to load, preparing all assets for the start of the game session:

```
void GamePlayScreen::LoadLevelXML()
{
    Coding4Fun::FallFury::XMLReader::Reader^ reader =
        ref new Coding4Fun::FallFury::XMLReader::Reader();

    Platform::String^ LevelName = Manager->Levels.Lookup(StaticDataHelper::CurrentLevelID);

    Windows::Foundation::IAsyncOperation<Platform::Object^>^ result =
       reader->ReadXmlAsync(LevelName,
           Coding4Fun::FallFury::XMLReader::Models::XmlType::LEVEL);

    result->Completed =
        ref new AsyncOperationCompletedHandler<Platform::Object^>(this,
            &GamePlayScreen::OnLevelLoadCompleted);
}
```

The selected ID passes from the menu screen to the game screen through **CurrentLevelID**, an intermediary value preserved in a **StaticDataHelper** class. The level file name is looked up based on the ID and then passed to **ReadXmlAsync** with the **XmlType** set to **LEVEL**. When the load completes, **OnLevelLoadCompleted** is invoked, and an additional helper class— **LevelDataLoader**—sets up the game components based on the received data:

```
void GamePlayScreen::OnLevelLoadCompleted(IAsyncOperation<Platform::Object^> ^op, AsyncStatus
s)
{
    if (s == AsyncStatus::Completed)
    {
       InitializeSpriteBatch();
       m_loader =
           ref new BasicLoader(Manager->m_d3dDevice.Get(), Manager->m_wicFactory.Get());
       CreateBear();

        LevelDataLoader^ loader =
            ref new LevelDataLoader(
                (Coding4Fun::FallFury::XMLReader::Models::Level^)op->GetResults(), this);
       loader->SetupBear(GameBear);
       loader->SetupObstacles(m_obstacles);
       loader->SetupMonsters(m_monsters);
       loader->SetupButtons(m_buttons, m_buttonPrice);
       loader->SetupPowerups(m_powerups);
       m_currentLevelType = (LevelType) loader->CurrentLevel->LevelMeta->LevelType;

       StaticDataHelper::CurrentLevel = loader->CurrentLevel;
       StaticDataHelper::ButtonsTotal = loader->CurrentLevel->Buttons->Length;

       LoadTextures();

        CreateMonster();
        CreatePowerups();

       IsLevelLoaded = true;

       GameBear->TurnRight();
       m_particleSystem.CreatePreCachedParticleSets();
       LoadSounds();
    }
}
```

Each object is separated in its own method, such as **SetupBear** or **SetupObstacles**. **SetupBear**, for example, transforms the exposed managed **Bear** model to a native C++ **Characters::Bear** one:

```
void LevelDataLoader::SetupBear(Bear ^gameBear)
{
    gameBear->Position = float2(GetXPosition(CurrentLevel->GameBear->StartPosition), 0);
    gameBear->MaxHealth = CurrentLevel->GameBear->MaxHealth;
    gameBear->CurrentHealth = CurrentLevel->GameBear->MaxHealth;
    gameBear->CurrentDamage = CurrentLevel->GameBear->Damage;
    gameBear->Velocity.y = CurrentLevel->GameBear->Velocity;
    gameBear->MaxCriticalDamage = CurrentLevel->GameBear->CriticalDamage;
    gameBear->Rotation = 0.0f;
}
```

Conclusion

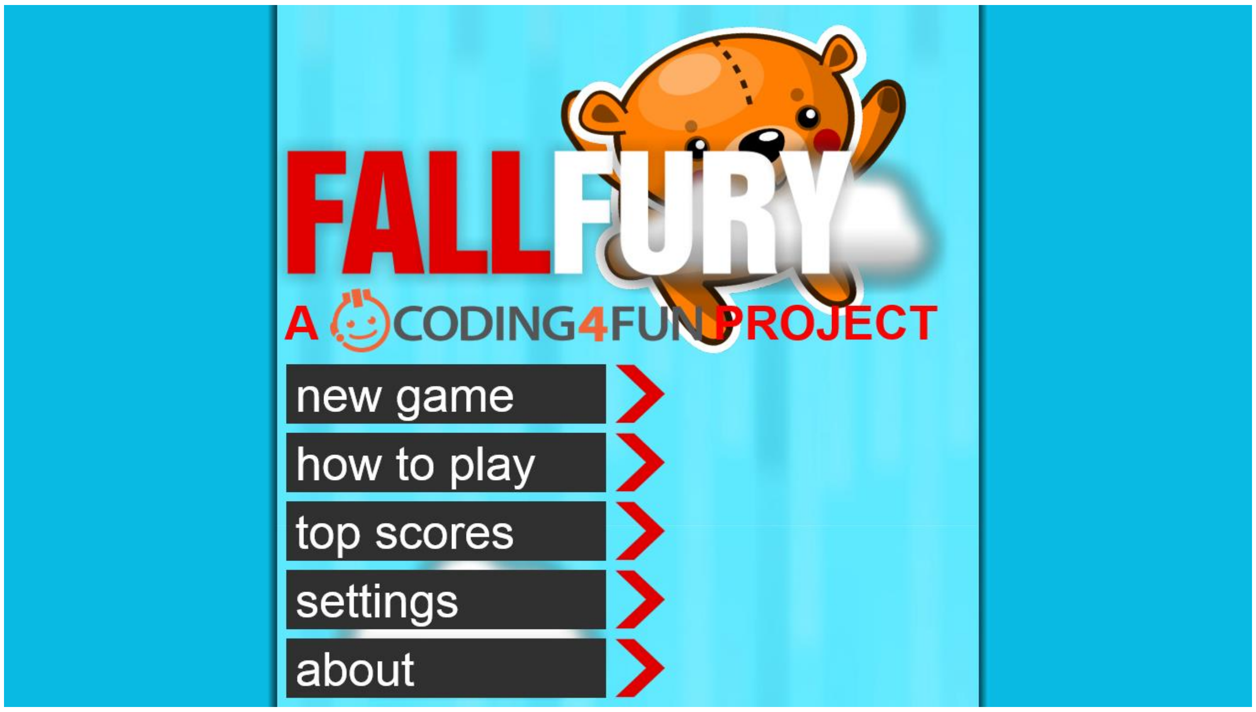
Mixing C# and C++ components is not a complicated process. Nonetheless, it comes with specific restrictions and considerations, such as the format of the publicly exposed asynchronous calls. The above C#-based level loading engine highlights the fact that WinRT interoperability allows you to leverage the languages you know best in order to efficiently accomplish project tasks.

Chapter 7 - Animations

FallFury relies on a lot of dynamic content. As you already aware of how **SpriteBatch** is invoked inside the FallFury rendering stack, this article focuses on how dynamic activities are handled on existing textures and entities. If you need a quick look at what was already covered, refer to Part 3 of the series.

Menu Screen

The first animated element shown in the menu screen is the bear, which is positioned in the top half of the viewport. Notice that the bear moves its paws as well as moving across the screen:



The bear is implemented in the **GameBear** class and is the normal playable entity that is invincible when animated outside the scope of the gameplay screen. Its full body image is composed of four textures:

```
Microsoft::WRL::ComPtr<ID3D11Texture2D>            m_head;
Microsoft::WRL::ComPtr<ID3D11Texture2D>            m_leftPaw;
Microsoft::WRL::ComPtr<ID3D11Texture2D>            m_rightPaw;

Microsoft::WRL::ComPtr<ID3D11Texture2D>            m_body // Exposed through LivingEntity;
```

Each of these textures is internally dependent on the coupled float rotation and position values passed to the sprite batch in the host container (the parent screen):

```
HostContainer->CurrentSpriteBatch->Draw(
    m_rightPaw.Get(),
    m_rightPawPosition,
    PositionUnits::DIPs,
    float2(522.0f, 141.0f) * Scale,
    SizeUnits::Pixels,
    m_shading,
    m_rightPawRotation - 0.5f);

HostContainer->CurrentSpriteBatch->Draw(
    m_body.Get(),
    Position,
    PositionUnits::DIPs,
    float2(400.0f, 400.0f) * Scale,
    SizeUnits::Pixels,
    m_shading,
    Rotation);

HostContainer->CurrentSpriteBatch->Draw(
    m_leftPaw.Get(),
    m_leftPawPosition,
    PositionUnits::DIPs,
    float2(522.0f, 141.0f) * Scale,
    SizeUnits::Pixels,
    m_shading,
    m_leftPawRotation + 0.5f);
```

Let's take a look at how the position and rotation values are modified. All item animations are performed with the help of timers set to brief intervals in which the values are cycled through two value exchange operations. Here is a snippet taken from inside the Update method in the **GameBear** class that is responsible for bear arm movement:

```
if (m_armRotationTimer < 0.6f)
{
    m_rightPawRotation -= 0.005f;
    m_leftPawRotation += 0.005f;
}
else if (m_armRotationTimer > 0.6f && m_armRotationTimer < 1.2f)
{
    m_rightPawRotation += 0.005f;
    m_leftPawRotation -= 0.005f;
}
else
{
    m_rightPawRotation = 0.0f;
    m_leftPawRotation = 0.0f;
    m_armRotationTimer = 0.0f;
}
```

m_armRotationTimer is a float value that is initially set to **o.of**. In the **Update** loop and increments each iteration by the value of **timeDelta**, which represents the time difference between two loop cycles. As this value is less than 600 milliseconds (0.6 seconds), the right paw rotation is incremented and the left paw rotation decremented. The difference in the value adjustment direction is caused by the fact that one paw is rotated clockwise and the other is rotated counterclockwise. Once the timer tracks a value above 600 milliseconds but below than 1200 milliseconds, the process reverses and the paws rotate in the opposite direction. After 1.2 seconds, the timer resets, as does the rotation.

Since this rotation is automated, **GameBear->Update** is called in the menu screen:

```
m_showBear->Update(timeTotal, timeDelta, float2(Manager->m_windowBounds.Width / 2,
                   Manager->m_windowBounds.Height / 2));
```

Outside the context of the bear update cycle, there is a similar timed mechanism used to randomly move the bear across the screen:

```
if (m_positionYAdj == 0.0f)
{
    m_positionYAdj = RandFloat(0.1f, 1.2f);
    m_positionXAdj = RandFloat(-1.0f, 1.0f);
}

if (m_positionTimer < 4.0f)
{
    m_showBear->Position.x += m_positionXAdj;
    m_showBear->Position.y += m_positionYAdj;
    m_showBear->Rotation += 0.001f;
    m_showBear->Scale += 0.0008f;

    m_showMonster->Scale += 0.002f;
}
else if (m_positionTimer > 4.0f && m_positionTimer < 8.0f)
{
    m_showBear->Position.x -= m_positionXAdj;
    m_showBear->Position.y -= m_positionYAdj;
    m_showBear->Rotation -= 0.001f;
    m_showBear->Scale -= 0.0008f;
}
else
{
    m_positionYAdj = 0.0f;
    m_positionTimer = 0.0f;
}
```

The X and Y adjustments are randomly generated and for the duration of a single timer cycle (in this case, 4 seconds) the bear's horizontal and vertical positions are adjusted using those values. The rotation and scale are minimally adjusted to create a 3D motion effect. Because the displacement is minimal, the bear is always visible and does not move much outside the viewport.

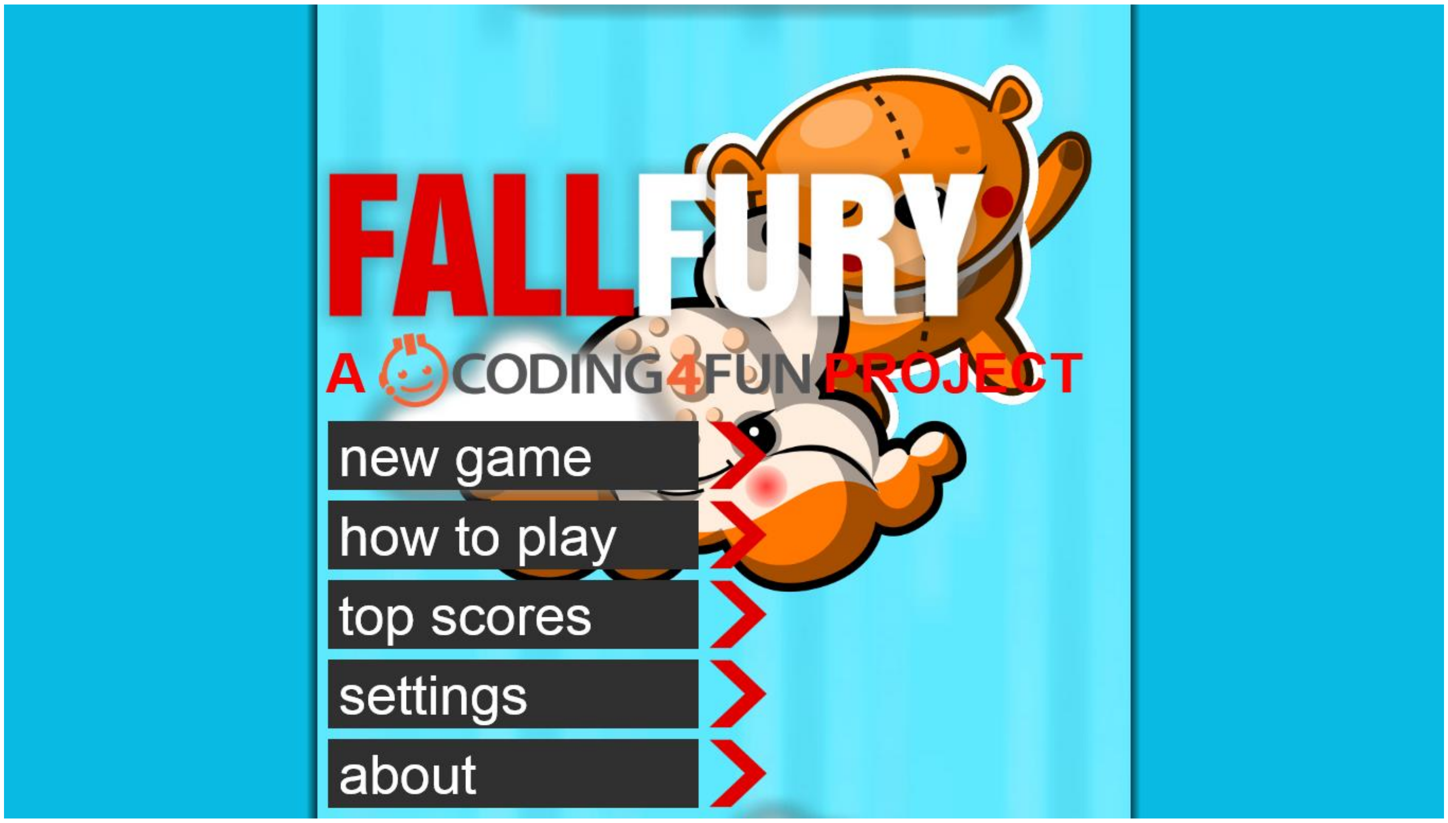
There is also a flying monster animation in the main menu screen. Monster creation happens on top of the same instance that is replaced when an initial monster flies out of bounds:

```
m_showMonster->Position.y -= m_showMonster->Velocity.y;

if (m_showMonster->Position.y < -m_showMonster->Size.y)
{
    CreateNewMonster();
}
```

When a new monster is created, a random monster type is selected and its position on the X axis is randomized. The Y position is set to be right under the bottom screen boundary:

```
void MenuScreen::CreateNewMonster()
{
    m_showMonster =
       ref new Monster(this, (MonsterType)(rand() % (int)MonsterType::MT_CANDYLAND_E), true);
    m_showMonster->Scale = 0.3f;
    m_showMonster->Velocity.y = 1.0f;
    m_showMonster->Load();

    m_showMonster->Position =
        float2(RandFloat(LoBoundX, HiBoundX),
               Manager->m_windowBounds.Height + m_showMonster->Size.y * m_showMonster->Scale);
}
```

Monster scaling is done on the same timed loop as the bear. The results look like this:



Gameplay Screen

Bear

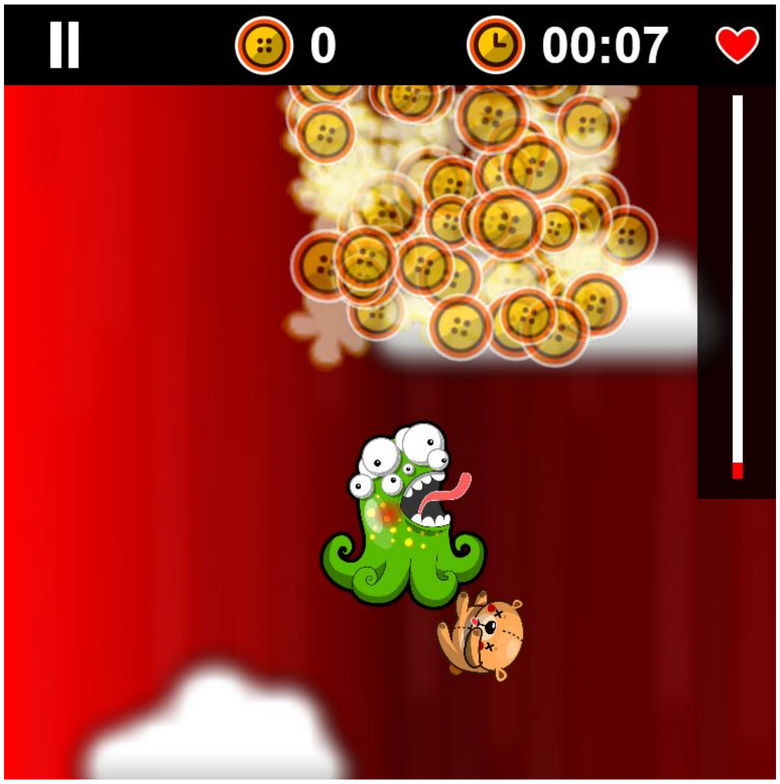
The gameplay screen carries the most animations in FallFury, including animations related to ammo displacement, monster movement paths, power-up animations, and secondary game character animations that are triggered in special cases such as character death.

Let's start with character entrance. When the game begins, the teddy bear falls into the screen and stops at a specific point close to the top boundary of the gameplay screen. Once that transition is complete, the bear is no longer vertically displaced:

```
if ((GameBear->Position.y / HiBoundY) < 0.19f)
{
    GameBear->Position.y += GameBear->Velocity.y * (3.2f);
}
```

If the limit is not hit, the original bear velocity is multiplied by a fixed value, after which the condition is ignored for the rest of the gameplay.

Let's now take a look at what happens when an enemy shell kills the bear. Generally, when the main character is killed, there is not much sense in maintaining other in-game activities. In FallFury, a killed bear results in a time freeze—the fall stops and the monsters are no longer shooting or moving. Not only that, but the bear is also flipped on its back:



In the **Update** loop, the position for the paws and the head are adjusted for the new body texture layout:

```
if (IsDead)
{
    if (m_armRotationTimer < 1.0f)
    {
        m_rightPawRotation += 0.001f;
        m_leftPawRotation -= 0.001f;
    }
    else if (m_armRotationTimer > 1.0f && m_armRotationTimer < 2.0f)
    {
        m_rightPawRotation -= 0.001f;
        m_leftPawRotation += 0.001f;
    }
    else
    {
        m_rightPawRotation = 0.0f;
        m_leftPawRotation = 0.0f;
        m_armRotationTimer = 0.0f;
    }

    m_rightPawPosition = Position + float2(40.0f, -50.0f) * Scale;
    m_leftPawPosition = Position + float2(10.0f, 150.0f) * Scale;
    m_headPosition = (Position - float2(-160.0f, 0.0f) * Scale);
}
```

The proper textures are assigned via the **Kill** method:

```
void Bear::Kill()
{
    IsDead = true;
    Rotation = -1.0f;
    m_head = m_deadHead;
    m_body = m_deadBody;
    m_leftPaw = m_deadLeftArm;
    m_rightPaw = m_deadRightArm;
}
```

The bear will continue falling as long as **StopBackground** in **GamePlayScreen** is called:

```
void GamePlayScreen::StopBackground()
{
    m_isBackgroundMoving = false;
}
```

This will cause the internal state check to fail in the **Update** loop, sending the bear off screen limits. Once the bear completes the fall, the state is set to **GS_GAME_OVER**:

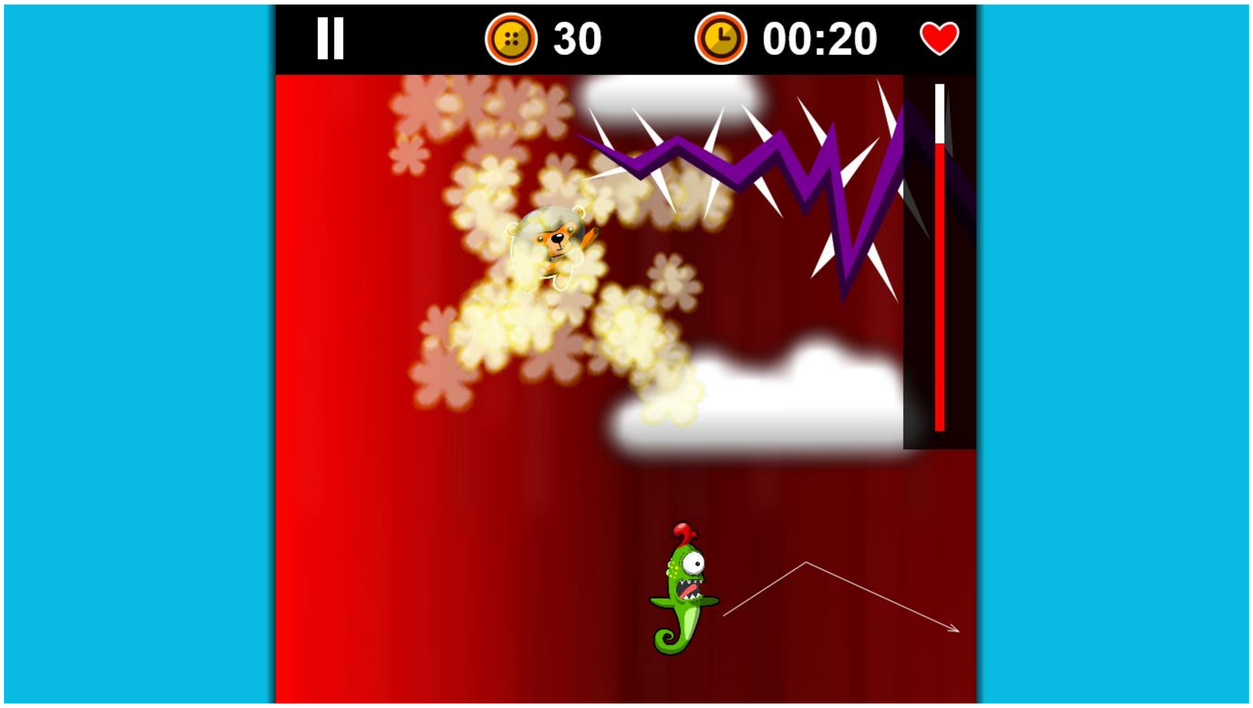
```
if (!m_isBackgroundMoving)
{
    if (GameBear->Position.y > m_screenSize.y)
    {
        Manager->CurrentGameState = GameState::GS_GAME_OVER;
    }
    else
    {
        GameBear->Position.y += GameBear->Velocity.y * 1.5f;
    }
}
```

Monsters

Monsters, on the other hand, are being constantly moved during the duration of the game. Their basic displacement is performed by MoveMonsters:

```
void GamePlayScreen::MoveMonsters(float timeTotal ,float timeDelta)
{
    for (auto monster = m_monsters.begin(); monster != m_monsters.end();)
    {
        Monster^ currentMonster = (*monster);

        if ((currentMonster->Position.y - GameBear->Position.y) <
             -Manager->m_renderTargetSize.Height / 2)
        {
            monster = m_monsters.erase(monster);
        }
        else
        {
            currentMonster->Velocity.y = GameBear->Velocity.y;

            currentMonster->Update(timeTotal, timeDelta, GameBear->Position,
                                   GetScreenBounds());
            CheckForCollisionWithAmmo(currentMonster);
            ++monster;
        }
    }
}
```

This method adjusts the monster position relative to the bear:



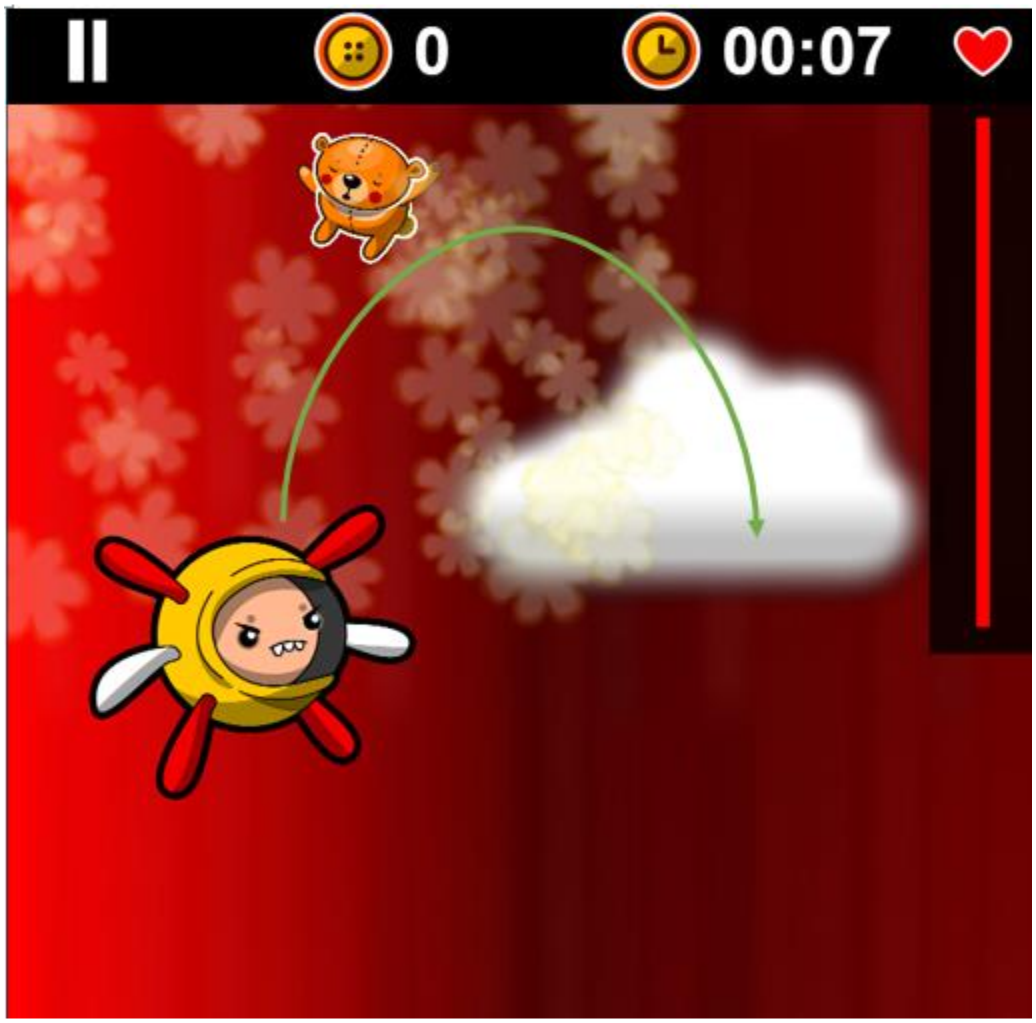
The zig-zag like motion is defined in the **Monster** class in the **Update** method:

```
float adjustment = 0.0f;
adjustment = m_goingRight ? Position.x + Velocity.x : Position.x - Velocity.x;

if (adjustment >= (HostContainer->LoBoundX + (Size.x * Scale) / 2.0f)
    && adjustment <= (HostContainer->HiBoundX - (Size.x * Scale) / 2.0f))
{
    Position.x = adjustment;
}
else
{
    m_goingRight = !m_goingRight;
}
```

The snippet above determines the direction in which the enemy has to move depending on which screen boundary is hit first. While the monster is active and is visible, its vertical adjustment is performed in a timed loop, like this:

```
m_jumpingTimer += timeDelta;
if (m_jumpingTimer > 0.0f && m_jumpingTimer < 1.0f)
{
    Position.y -= 1.0f;
}
else if (m_jumpingTimer >= 1.0f && m_jumpingTimer < 2.0f)
{
    Position.y += 1.0f;
}
else
{
    m_jumpingTimer = 0.0f;
}
```

Here you can either use the Y velocity or introduce a static value. The only condition has to be a number low enough that the monster does not leave the screen, which would prevent the bear from being able to kill it. When the monster is killed, it flies out by following an arch-like path:



The "death arc" is implemented via another timer:

```
Position.x += Velocity.y * 1.3f;
if (m_deathArcTimer > 0.4f)
{
    Position.y += Velocity.y;

    if (Scale > 0.1f)
        Scale -= 0.01f;
}
else
{
    Position.y -= Velocity.y;
    Scale += 0.01f;
    m_deathArcTimer += timeDelta;
}
```

Regardless of the monster position, the horizontal displacement is positive and the entity moves to the right. For 400 milliseconds its vertical position is decreased, moving the texture up, and the scale is also adjusted to create the proximity effect. After this time interval, the monster drops out of the screen boundaries.

As previously mentioned, each monster is composed of three cycled textures, which are loaded and stored in three ID3D11Texture2D containers:

```
Microsoft::WRL::ComPtr<ID3D11Texture2D>            m_spriteA;
Microsoft::WRL::ComPtr<ID3D11Texture2D>            m_spriteB;
Microsoft::WRL::ComPtr<ID3D11Texture2D>            m_spriteC;
```

Each of these is assigned to the main body texture container every 300 milliseconds, replacing the previously assigned asset:

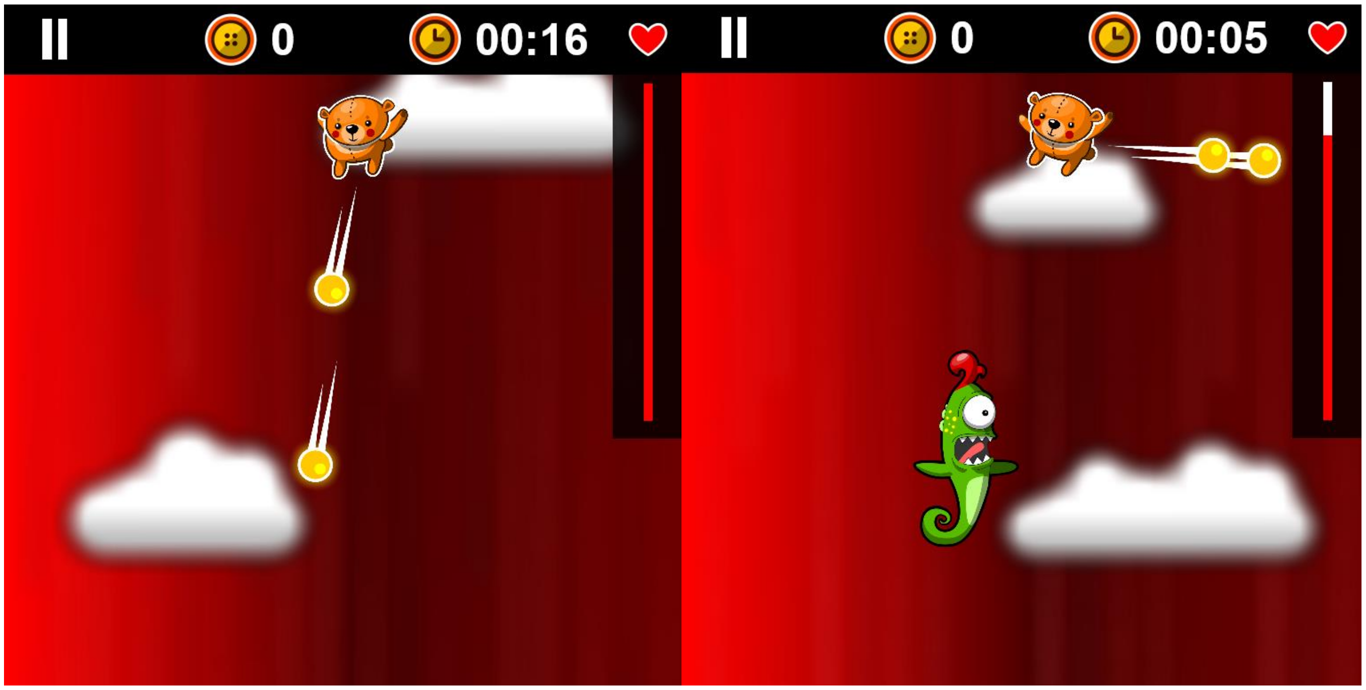
```
m_stateChangeTimer += timeDelta;
if (m_stateChangeTimer < 0.3f)
{
    m_body = m_spriteA;
}
else if (m_stateChangeTimer > 0.3f && m_stateChangeTimer < 0.6f)
{
    m_body = m_spriteB;
}
else if (m_stateChangeTimer > 0.6f && m_stateChangeTimer < 0.9f)
{
    m_body = m_spriteC;
}
else if (m_stateChangeTimer > 0.9f)
{
    m_stateChangeTimer = 0.0f;
}
```

This cycle can be disabled when the monster is not active.

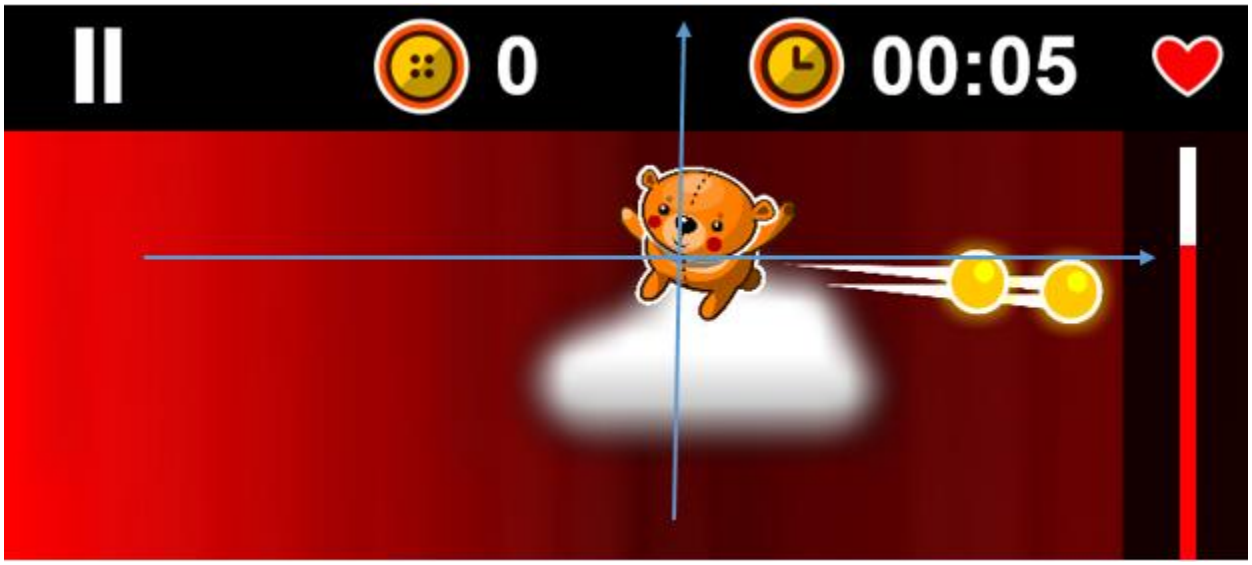
Ammo

There are different types of ammo used in the game, and each behaves in its own way. The stock ammo type is a plasma ball.

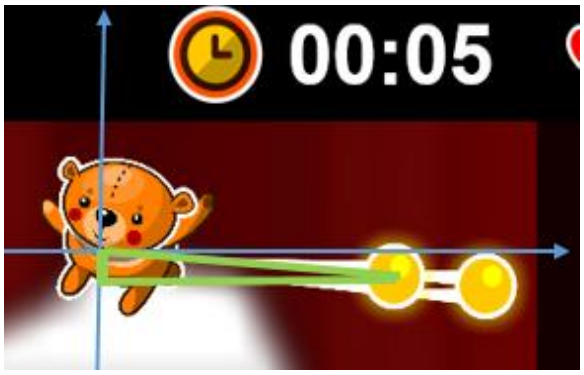
The default shooting behavior releases a shell when the user taps anywhere on the screen (other than the pause button area). Given the texture of the shell, when it is released the tail must be oriented towards the bear at an angle equal to the one created by the bear and the target position. To get a better idea, take a look at the images below:



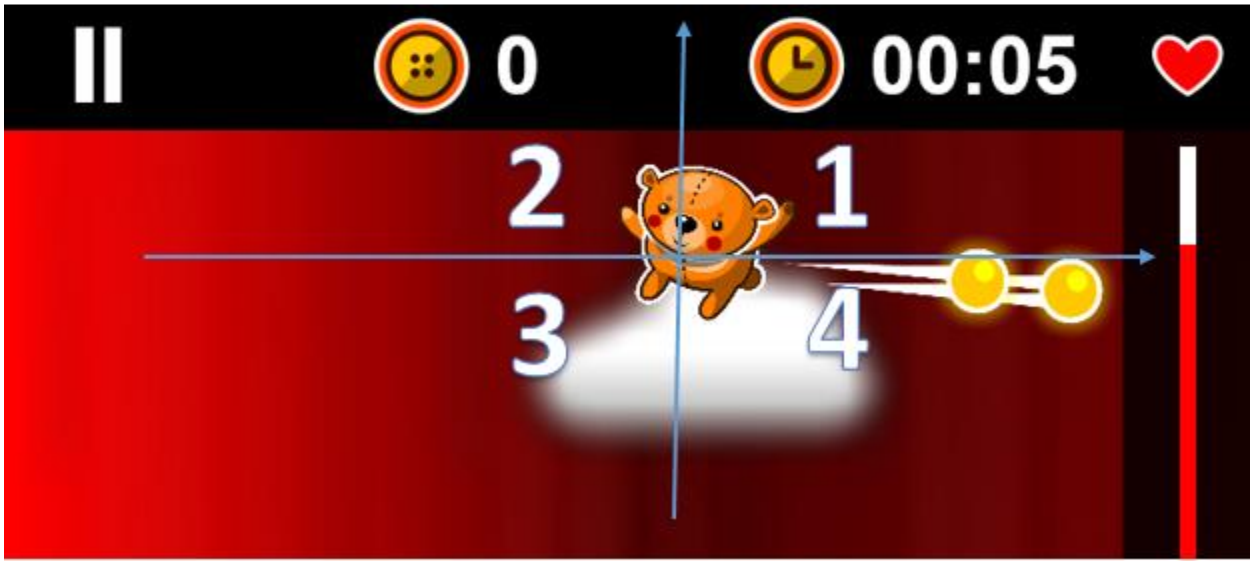
In the image on the left, the player tapped at the bottom of the screen. In the one on the right, the player tapped at the right edge of the screen. The shell accordingly rotates depending on the tapped position. To get a better understanding of how the rotation is performed, imagine the XY coordinate grid with the bear located at the intersection of the axes:



Thinking back to trigonometry, notice the triangle formed by the shell and the bear origin point:



Remember also the trig concept of quadrants:



The quadrant in which the shell is located determines how the angle is calculated and the tail rotated. If the shell is in quadrant 2 or 3, calculate the rotation radians by following this algorithm:

1. Find the hypotenuse from the X and Y components of the velocity, which can be calculated by finding the difference between the bear and the position of the tap. To do this, us the well-known Pythagorean theorem:

   $$a^2 + b^2 = c^2$$

   The hypotenuse formula can be deduced from the formula above and is the square root of the sum of the velocity component squares:

   $$c = \sqrt{a^2 + b^2}.$$

2. Once calculated, it is necessary to find the sine of the target angle. The value can be obtained by dividing the Y component of the velocity by the hypotenuse.

3. Find the angle by calculating the arcsine from the resultant sine value. This will return the final angle in radians.

That said, for shells launched in quadrants 1 or 4, there is one extra step that needs to be performed. In addition to the rotation value obtained from the steps listed above, the rotation should be incremented by the radian value of a 180-degree angle minus twice the rotation value previously obtained. This ensures that the tail is correctly flipped relative to the axis origin.

In C++, the implementation is done inside the **AmmoItem** class. For flexibility purposes, it triggers inside the **Update** loop—that way, it is possible to modify the trajectory of the shell and not worry about turning it again manually after launch:

```
if ((Velocity.x > 0 && Velocity.y > 0) || (Velocity.x > 0 && Velocity.y <0))
{
    Rotation = CalculateRadians(Velocity);
    Rotation += CalculateRadiansFromAngle(180) - 2 * Rotation;
}
else
{
    Rotation = CalculateRadians(Velocity);
}
```

CalculateRadians is the method that transforms the velocity components in a rotation value. It is located in the **BasicMath.h** helper:

```
inline float CalculateRadians(float2 velocity)
{
    float hypothenuse = sqrt(velocity.x * velocity.x + velocity.y * velocity.y);
    float sine = velocity.y / hypothenuse;
    float angle = asin(sine);
    return angle;
};
```

There is a potential problem with the implementation above. As the user taps on different parts of the screen, the X and Y components are different, each resulting in a different hypotenuse. As the target is set, the shell flies faster the further away from the bear the user taps. To avoid this, the triangle legs should be normalized to a near-constant value:

Take a look at the **ShootAtTarget** function in the Bear class:

```
void Bear::ShootAtTarget(float2 lastTargetTrace)
{
    OnPulledTrigger(Position.x, Position.y,
        GetVelocityLegs(lastTargetTrace).x, GetVelocityLegs(lastTargetTrace).y,
        CurrentDamage, IsFriendly, false, HostContainer->CurrentSpriteBatch);
}
```

Notice that the triangle legs are not passed as a raw value, but are proxied through **GetVelocityLegs**, which forces the resulting vector to be produced from a constant triangle with the velocity constant at 10 pixels per iteration:

```
float2 LivingEntity::GetVelocityLegs(float2 lastTargetTrace)
{
    float bottomLeg = 0.0f;
    float sideLeg = (Position.y - lastTargetTrace.y) / 100.0f;

    bottomLeg = (Position.x - lastTargetTrace.x) / 100.0f;

    float requiredVelocity = 10.0f;

    float hypothenuse = sqrt(bottomLeg * bottomLeg + sideLeg * sideLeg);

    float proportionalX = 0.0f;
    float proportionalY = 0.0f;

    proportionalX = (requiredVelocity * bottomLeg) / hypothenuse;
    proportionalY = (sideLeg < 0.0f ? -1 : 1) *
                    sqrt(requiredVelocity * requiredVelocity - proportionalX * proportionalX);
    return float2(proportionalX,proportionalY);
}
```

For ammo that does not have a visual rotation dependency, such as the plasma ball, simply increment the rotation to make the thrown item continuously spin:

```
if (Type == PowerupType::PLASMA_BALL)
{
    if ((Velocity.x > 0 && Velocity.y > 0) || (Velocity.x > 0 && Velocity.y <0))
    {
        Rotation = CalculateRadians(Velocity);
        Rotation += CalculateRadiansFromAngle(180) - 2 * Rotation;
    }
    else
    {
        Rotation = CalculateRadians(Velocity);
    }
}
else
{
    Rotation += 0.2f;
}
```

Conclusions

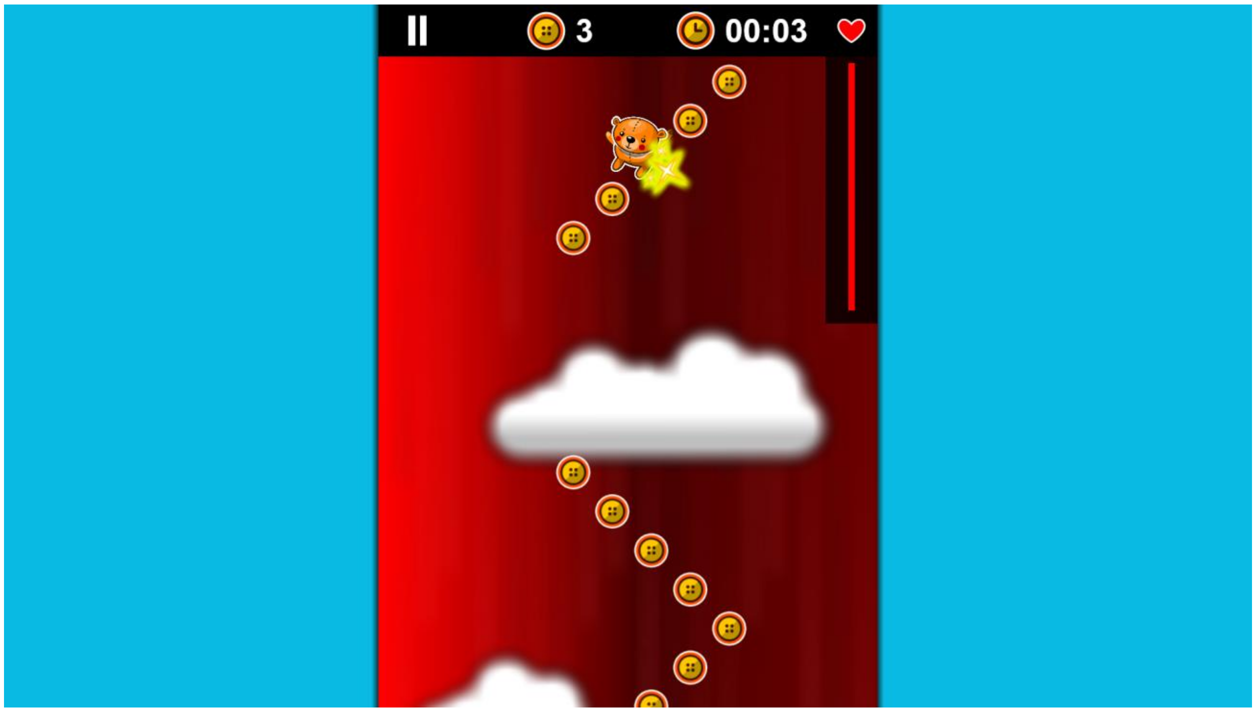
FallFury mostly relies on sprite-based animations in which there are several textures cycled through in order to create the desired dynamic effect. These are not really hard to build, given that there is a possibility to integrate them in a timed loop. Be aware, however, that with slower machines the timing might be off and the time delta value between loops might be higher. In that case the sprite cycling will not be as smooth as it should be, which is why it's important to perform animation testing on a variety of hardware with different OS loads.

Chapter 8 – Element Interaction

During the gameplay multiple entities interact with each other to make the gaming experience what it is. The bear collides with obstacles and collects buttons, monsters shoot shells that can fly off-screen or hit the bear—all this is possible with the help of the basic collision detection techniques that are implemented in FallFury.

Buttons

Buttons are bonus-boosters that can be placed by the level designer anywhere on the screen in game mode. These are relatively small entities, which are displaced vertically with each cycle of the **Update** loop and move in the opposite direction, but with the same velocity, as the main character.



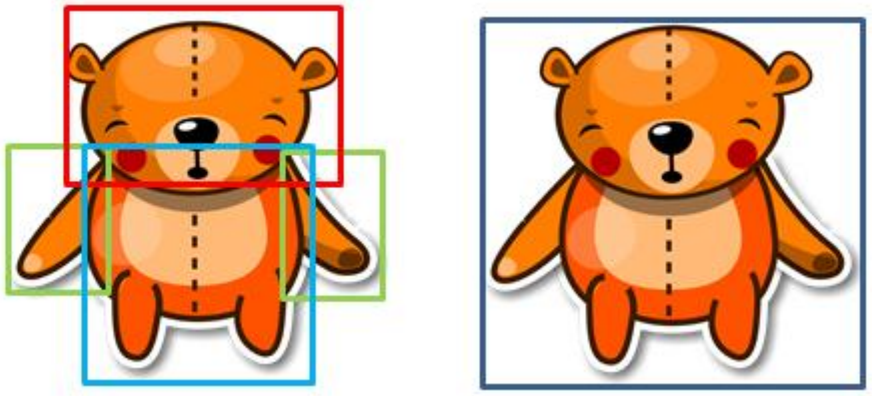
Looking at the Update function in the GamePlayScreen class, you will notice this call:

```
UpdateButtons();
```

UpdateButtons can be considered the button manager function responsible for removing the collected buttons from the rendering stack, counting them, and checking for a button collision when the bear is in close proximity. The implementation looks like this:

```
void GamePlayScreen::UpdateButtons()
{
    Windows::Foundation::Rect livingEntityBoundingBox = GameBear->GetBoundingBox();

    for (auto button = m_buttons.begin(); button != m_buttons.end();)
    {
        (*button)->Position.x = (*button)->PixelDiff + LoBoundX;
        (*button)->Position.y -= GameBear->Velocity.y;

        if (Geometry::IsInProximity(GameBear->Position,(*button)->Position, 100))
        {
            Windows::Foundation::Rect obstacleRect = (*button)->GetBoundingBox();
            if (livingEntityBoundingBox.IntersectsWith(obstacleRect))
            {
                AudioManager::AudioEngineInstance.StopSoundEffect(Coin);
                AudioManager::AudioEngineInstance.PlaySoundEffect(Coin);

                m_particleSystem.ActivateSet("Sparkle",
                    (*button)->Position,float2(RandFloat(-6.0f,6.0f),RandFloat(-10.0f, -5.0f)));

                StaticDataHelper::ButtonsCollected++;

                button = m_buttons.erase(button);
            }
            else
                ++button;
        }
        else
        {
            ++button;
        }
    }
}
```

For performance reasons, FallFury supports composite bounding box creation as well as simple box creation. As mentioned earlier in the series, the main character is not composed of a single texture, but rather multiple sprites that are cross-positioned to create a single visual entity. To give you a better idea of what composite vs. simple boxing looks like, take a look at the images below:



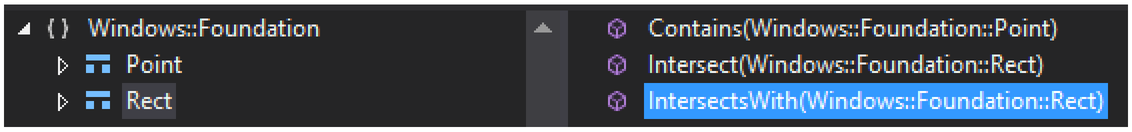
The image on the left shows how each part of the bear has its own bounding box, and each will be used for collision checking. In the image on the right, the bear has a single bounding box, creating minor potential gaps, but gaining performance.

Going back to **UpdateButtons**, once the bounding box is obtained, I iterate through the button collection and make sure that each item is located in the proper space:

```
(*button)->Position.x = (*button)->PixelDiff + LoBoundX;
(*button)->Position.y -= GameBear->Velocity.y;
```

The constant position checks are necessary because FallFury supports dynamic orientation changes. When the user switches from portrait to landscape mode and vice-versa, rendered elements on the screen are not automatically repositioned. Setting the X position is easy as long as there is a fixed button margin (from the left side of the screen: **LoBoundX**) and adding it to the current **LoBoundX** value results in a proper X location. There is no need to do the same check on the Y-axis because the level length remains the same regardless of the current screen orientation. The adjustment made relative the Y position is bound to the bear velocity. If the bear moves slower, buttons will also scroll slower.

Given that all buttons are properly positioned, a proximity check is performed on each button passed through the loop. If the bear position is at least 100 pixels away from the current button, the corresponding bounding box is obtained and an intersect check is performed. In simple boxing mode, this is done via **Windows::Foundation::Rect::IntersectsWith**:


Windows::Foundation
Point
Rect
Contains(Windows::Foundation::Point)
Intersect(Windows::Foundation::Rect)
IntersectsWith(Windows::Foundation::Rect)

If a collision occurs, the appropriate sound effect is played and a particle set is activated to create a visual notification of the action. After the button counter is incremented, the button is removed from the local collection, effectively being removed from the rendering stack.

Power-ups

As the bear flies towards the end of the level, it might encounter bonuses to improve its ability to fight incoming enemies or protect from damage caused by enemy ammo or obstacles. The process behind displaying power-ups on the screen and determining whether there was a collision with the main character is similar to **UpdateButtons**.

The core function for this task is **UpdatePowerups**:

```
void GamePlayScreen::UpdatePowerups(float timeDelta)
{
    if (m_powerups.size() > 0)
    {
        for (auto powerup = m_powerups.begin(); powerup != m_powerups.end(); powerup++)
        {
            (*powerup)->Update(timeDelta);
            (*powerup)->Position.x = (*powerup)->PixelDiff + LoBoundX;
            (*powerup)->Position.y -= GameBear->Velocity.y;
        }
    }

    CheckForCollisionWithPowerups();
}
```

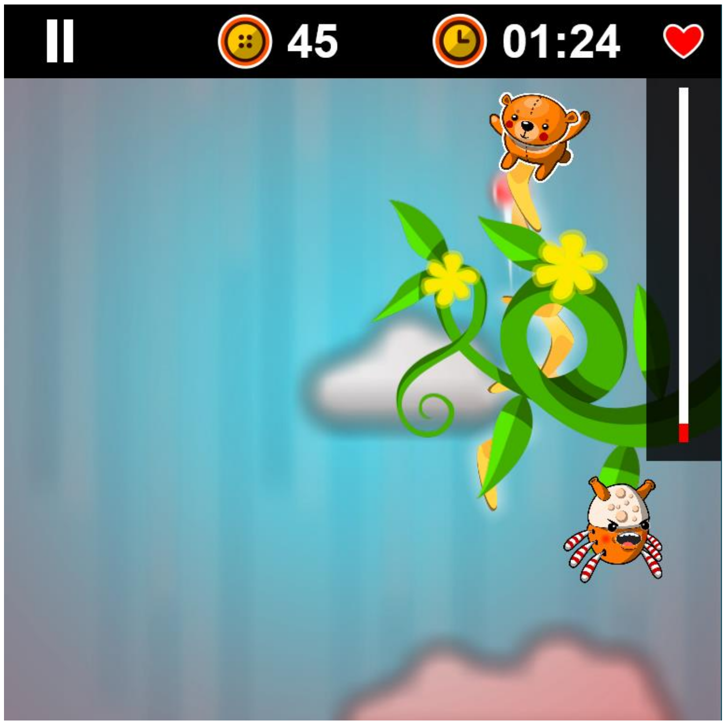
One difference you probably noticed in the snippet above is the fact that the collision check is now done through a separate function—**CheckForCollisionWithPowerups**:

```
void GamePlayScreen::CheckForCollisionWithPowerups()
{
    Powerup^ currentPowerup;
    Windows::Foundation::Rect livingEntityBoundingBox = GameBear->GetBoundingBox();

    for (auto powerup = m_powerups.begin(); powerup != m_powerups.end();)
    {
        currentPowerup = (*powerup);

        if (Geometry::IsInProximity(GameBear->Position,currentPowerup->Position, 100))
        {
            Windows::Foundation::Rect obstacleRect = currentPowerup->GetBoundingBox();
            if (livingEntityBoundingBox.IntersectsWith(obstacleRect))
            {
                AudioManager::AudioEngineInstance.PlaySoundEffect(GenericPowerup);

                GameBear->PickupPowerup(currentPowerup, &m_particleSystem);

                powerup = m_powerups.erase(powerup);
            }
            else
                ++powerup;
        }
        else
        {
            ++powerup;
        }
    }
}
```

If an intersection is detected between the boxed bear and the power-up texture, the current power-up is passed to the **Bear** instance and the appropriate type of action is selected:

```
void Bear::PickupPowerup(Powerup^ powerup, ParticleCore* ParticleSystem)
{
    switch (powerup->Type)
    {
    case PowerupType::PARACHUTE:
        {
            m_previousVelocity = Velocity.y;
            SetParachute(powerup->Lifespan);
            Velocity.y -= powerup->Effect;

            ParticleSystem->ActivateSet("ScalableParachute", Position, 0.0f, false, true);

            break;
        }
    case PowerupType::HEALTH:
        {
            CurrentHealth = MaxHealth;

            ParticleSystem->ActivateSet("ScalableHeart", Position, 0.0f, false, true);

            break;
        }
    case PowerupType::BUBBLE:
        {
            if (IsHelmetEnabled)
            {
                IsHelmetEnabled = false;
                DamageDivider = 1.0f;
            }
            else
            {
                DamageDivider = powerup->Effect;
            }

            IsBubbleEnabled = true;
            m_maxBubbleCounter = powerup->Lifespan;
            ParticleSystem->ActivateSet("ScalableBubble", Position, 0.0f, false, true);

            break;
        }
[.…]
    case PowerupType::BOOMERANG:
        {
            m_weaponType = powerup->Type;

            m_weaponTexture = m_boomerangTexture;
            CurrentDamage = powerup->Effect;

            m_weaponSize = float2(225.0f, 205.0f) * 0.5f;

            ParticleSystem->ActivateSet("ScalableBoomerang", Position, 0.0f, false, true);
            break;
        }
    }
}
```

Depending on the power-up, textures are added to the bear model and later passed to the rendering stack (if the power-up type is **PARACHUTE**), some textures and capabilities are replaced (**BOOMERANG**), or the bear capabilities are temporarily modified (**BUBBLE**):



If one of the temporary power-ups is enabled n the **Update** loop, dedicated timers ensure that ability enhancement does not last longer than necessary. As an example, here is the snippet that controls the bubble:

```
if (IsBubbleEnabled)
{
    m_currentBubbleCounter += timeDelta;
    if (m_currentBubbleCounter > m_maxBubbleCounter)
    {
        IsBubbleEnabled = false;
        DamageDivider = 1.0f;
        m_currentBubbleCounter = 0.0f;
    }
}
```

Ammo Collisions

There are also objects that direct ammo at either the enemy or the bear. After release, each shell follows a linear path towards the target, and while the user directs the shells that originate from the bear, those released by enemy entities automatically target the bear. In this case, the ammo needs to collide with an object to either damage or kill it. When a shell is released, it is added to the general ammo collection that is later updated internally:

```
void GamePlayScreen::UpdateAmmo(float timeDelta)
{
    for (auto shell = m_ammoCollection.begin(); shell != m_ammoCollection.end(); shell++)
    {
        (*shell)->Update(timeDelta, &m_particleSystem);
    }
}
```

At this point, the rendering system does not differentiate between friendly and enemy ammo—all it knows is that each item in the collection must have a new position when a new frame is rendered. The **CheckForCollisionsWithAmmo** function checks for ammo collisions:

```
void GamePlayScreen::CheckForCollisionWithAmmo(LivingEntity^ entity)
{
    if (entity != nullptr)
    {
        Windows::Foundation::Rect livingEntityBoundingBox = entity->GetBoundingBox();

        if (entity->IsFriendly)
        {
            for (auto ammo = m_ammoCollection.begin(); ammo != m_ammoCollection.end();)
            {
                if (!(*ammo)->IsFriendly)
                {
                        Windows::Foundation::Rect ammoBoundingBox = (*ammo)->GetBoundingBox();

                        if (livingEntityBoundingBox.IntersectsWith(ammoBoundingBox))
                        {
                                m_particleSystem.ActivateSet("SmallExplosion",entity->Position,
                                                                true);

                                entity->InflictDamage((*ammo)->HealthDamage);
                                GameBear->RedShade();

                                AudioManager::AudioEngineInstance.PlaySoundEffect(OuchA);
                                AudioManager::AudioEngineInstance.PlaySoundEffect(
                                                                                HardSoftCollision);

                                CheckBearHealth();

                                ammo = m_ammoCollection.erase(ammo);
                        }
                        else
                        {
                                ++ammo;
                        }
                }
                else
                {
                        ++ammo;
                }
            }
        }
        else
        {
            for (auto ammo = m_ammoCollection.begin(); ammo != m_ammoCollection.end();)
            {
                if ((*ammo)->IsFriendly)
                {
                        Windows::Foundation::Rect ammoBoundingBox = (*ammo)->GetBoundingBox();

                        if (livingEntityBoundingBox.IntersectsWith(ammoBoundingBox))
                        {
                                Monster^ monster = ((Monster^)entity);

                                if (!monster->IsDead && monster->IsActive)
                                {
                                        m_particleSystem.ActivateSet("SmallExplosion",
                                                                entity->Position, true);
                                        entity->InflictDamage((*ammo)->HealthDamage);

                                        monster->RedShade();
                                        monster->CheckIfAlive();

                                        AudioManager::AudioEngineInstance.PlaySoundEffect(
                                                                                SharpSoftCollision);
                                }
```

```
                    ammo = m_ammoCollection.erase(ammo);
                }
                else
                {
                    ++ammo;
                }
            }
            else
            {
                ++ammo;
            }
        }
    }
  }
}
```

When the function is called, it is usually run against an entity that is present on the screen, such as the main character. Regardless of whether the enemy or the friendly character fired the shot, the shot cannot inflict damage to its source, and that's why the function implements the ammo-to-entity crosscheck. If the ammo collides with any shells intersecting the entity bounding box, a collision is counted and health verification is performed to ensure that the character is still alive and that the game should continue. The bear's health is checked via **CheckBarHealth**:

```
void GamePlayScreen::CheckBearHealth()
{
    if (GameBear->CurrentHealth <= 0)
    {
        m_particleSystem.ActivateSet("Buttons",GameBear->Position, true);
        GameBear->Kill();

        StopBackground();
    }
}
```

That said, not all ammo will collide with entities on the screen. Some of it will go out-of-bounds, and without an explicit cleanup process in place out-of-bounds ammo is constantly re-rendered even though the end user has no way of seeing it. To avoid this, there is a helper function—**CheckForOutOfBoundsAmmo**:

```
void GamePlayScreen::CheckForOutOfBoundsAmmo()
{
    for (auto l_Iter = m_ammoCollection.begin(); l_Iter != m_ammoCollection.end(); /* nothing
                                                                                     here */ )
     {
        if (!GamePlayScreen::Manager->IsWithinScreenBoundaries((*l_Iter)->Position))
        {
           l_Iter = m_ammoCollection.erase(l_Iter);
        }
        else
        {
            ++l_Iter;
        }
     }

}
```

If any shell flies outside the screen bounding box, its instance is erased from the collection and the renderer no longer worries about allocating memory for an irrelevant item. To give you an idea of how that happens, here is a snippet that shows how the **RenderScreen** function handles the current ammo set:

```
if (!m_ammoCollection.empty())
{
    for (auto shell = m_ammoCollection.begin(); shell != m_ammoCollection.end(); shell++)
    {
        if (!(*shell)->IsFriendly)
        {
            (*shell)->Render();
        }
        else
        {
            GameBear->RenderShell((*shell)->Position, (*shell)->Rotation);
        }
    }
}
```

Conclusion

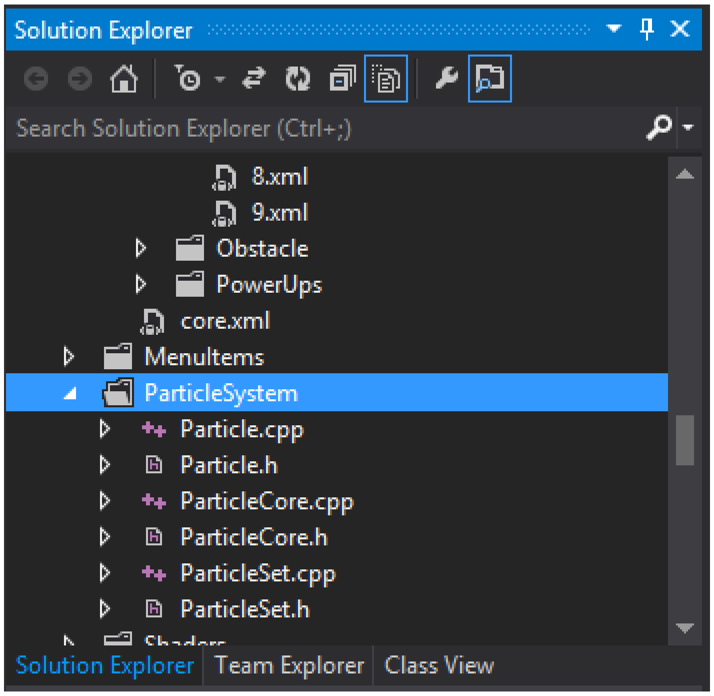
Element interaction is a core part of the FallFury experience. Separate handlers are implemented for each of them to ensure maximum flexibility when it comes to adding or removing components without breaking major parts of the code-base. Handling is mainly accomplished in the Update loop by iterating through registered entity sets, such as ammo, and verifying whether an action should be taken. Be cautious when implementing this kind of scenario with large entities and data sets—having multiple loops running simultaneously might tax machine performance, especially on low-power configurations such as ARM.

Chapter 9 – Particle System

During gameplay there are scenarios during which users need to be visually notified that something has happened, such as a collision with an obstacle or an enemy shell. One way to do this is by having explosion or item breaking simulation, which brings us to the next large component in FallFury—the sprite-based particle system. It doesn't offer as much power as a full-fledged particle system would, but it allows for effects that fit well within the overall theme and layout of the game.

The Core

ParticleSystem is the dedicated folder in the project that contains everything needed to render multiple textures at once and displace them to create the desired effect:



A single particle carries information regarding its size, position, velocity, color shading, rotation, circular velocity, and scale. As with any other rendered entity, it has a bounding box that can be used to detect its intersection with other elements on the screen. In FallFury, this functionality is not used.

Its structure is as follows:

```
#pragma once
#include "pch.h"

struct Particle
{
    Particle(float2 size);
    Particle(float2 size, float4 shading);
    float2 Size;
    float2 Position;
    float2 Velocity;
    float4 Shading;
    float  Rotation;
    float  RotationVelocity;
    float  Scale;

    bool IsWithinScreenBoundaries(float x, float y, Windows::Foundation::Rect screenBounds);
    Windows::Foundation::Rect GetBoundingBox();
};
```

The **Particle** class also happens to have two constructors—one that sets the particle to have the default shading, effectively removing the effect, and one where shading is dynamic. Note that a particle on its own doesn't do much—it neither carries the associated texture nor has an internal loop that can be used in any given application part to display it.

The next core class is **ParticleSet**. It is used as a container for all the particles associated with a specific effect. For example, if I want to create flying stuffing when the bear hits an obstacle, I create a new ParticleSet instance and define the necessary particle properties:

- **Lifespan** – a particle set does not constantly animate. It displays the particles for a limited amount of time, and displaces them by the given velocity values, and then self-destructs.
- **Texture** – all particles in a **ParticleSet** have the same texture. Going back to the stuffing example, there is a single PNG file used to render multiple variable-sized particles on collision.
- **IsAlive** – this is the flag that shows whether the particle set should be rendered in the first place. If it is set to **false** then, regardless of the conditions, this **ParticleSet** instance is ignored.
- **ShouldScale** – this flag determines whether particles will automatically increase their scale as they are being displaced, creating the effect of a particle approaching the screen. This effect is applied to each particle in the set.

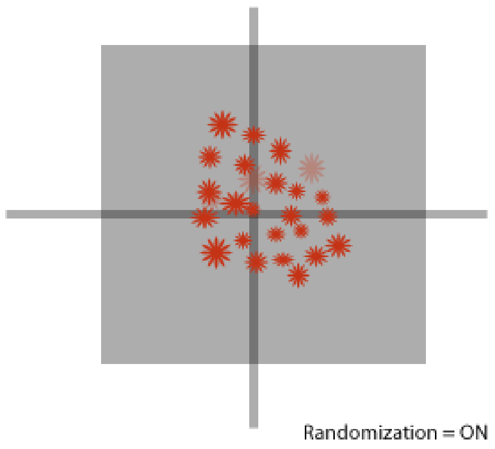
The container class is used to update the particles through the Update function:

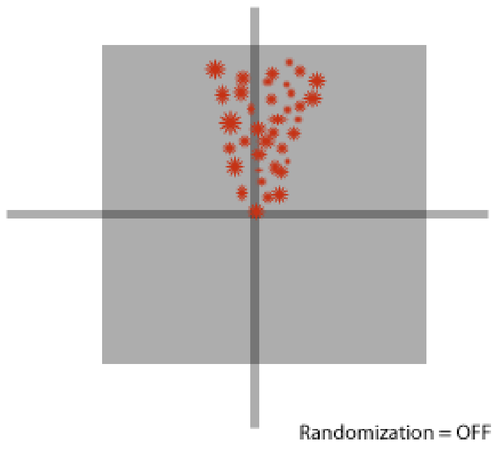
```
void ParticleSet::Update(float timeDelta)
{
    float quat = _lifespan / 0.016f;
    float decrement = 1.0f / quat;

    if (_totalTime <= _lifespan && _isAlive)
    {
        _totalTime += timeDelta;

        for (auto particle = _particles.begin(); particle != _particles.end(); particle++)
        {
            if (_shouldScale)
                particle->Scale += 0.2f;

            particle->Shading.a -= decrement;
            particle->Position = float2(particle->Position.x + particle->Velocity.x,
                particle->Position.y + particle->Velocity.y);

            if (!_shouldScale)
                particle->Rotation += particle->RotationVelocity;
        }
    }
    else
    {
        _totalTime = 0.0f;
        _isAlive = false;
    }
}
```

As it relies on the **_isAlive** flag, the **Update** loop is only used when the particle displacement is activated via the **Activate** function:

```
void ParticleSet::Activate(float2 position, float2 velocity, bool randomize, bool scale)
{
    for (auto particle = _particles.begin(); particle != _particles.end(); particle++)
    {
        particle->Position = position;
        if (randomize)
            particle->Velocity = float2(RandFloat(-5.0f,5.0f), RandFloat(-5.0f, 5.0f));
        else
            particle->Velocity = float2(velocity.x + RandFloat(-0.6f, 0.6f), velocity.y +
                                        RandFloat(-0.6f, 0.6f));
    }

    _shouldScale = scale;
    _isAlive = true;
}
```

When a set is activated, several user-defined parameters come into play. The position is set no matter what and is used to create the source point from which the particles start appearing. The velocity, on the other hand, can be randomized between the values of -5 and 5 pixels per update loop, on both the X-axis and the Y-axis. If randomized, large amount of particles will create an explosion that starts from the center point and expands towards all quadrants. When the velocity is not randomized, it creates a triangular expansion grid on which particles deviate from the center point in one of the given directions:





When it's time to render the particles, the sprite batch associated with the current game screen is used to pass a texture for each particle registered in the set:

```
void ParticleSet::Render(SpriteBatch ^spriteBatch)
{
    for (auto particle = _particles.begin(); particle != _particles.end(); particle++)
    {
        if (GamePlayScreen::Manager->IsWithinScreenBoundaries(particle->Position))
            spriteBatch->Draw(_texture.Get(), particle->Position, PositionUnits::DIPs,
            particle->Size * particle->Scale, SizeUnits::Pixels,
            particle->Shading, particle->Rotation);
    }
}
```

Although there is no flag check inside the Render method that would make sure that the set is alive, this can be done outside of it by calling **IsAlive**, which will return the flag value:

```
bool ParticleSet::IsAlive()
{
    return _isAlive;
}
```

Let's now take a look at the class that managed the particle flow—**ParticleCore**.

The Particle Manager

ParticleCore is the class that manages internal particle sets, and is also the proxy for set activation, update, and rendering. Here is its structure:

```
#pragma once
#include "pch.h"
#include "ParticleSet.h"
#include <list>
#include <map>

class ParticleCore
{
public:
    ParticleCore();
    ParticleCore(Coding4Fun::FallFury::Screens::GameScreenBase^);
    virtual ~ParticleCore();

    void CreatePreCachedParticleSets();
    void ActivateSet(Platform::String^, float2);
    void ActivateSet(Platform::String^, float2, float2);
    void ActivateSet(Platform::String^, float2, bool);
    void ActivateSet(Platform::String^, float2, float2, bool);
    void ActivateSet(Platform::String^, float2, float2, bool, bool scale);
    void Update(float);
    void Render();

private:
    std::list<ParticleSet> _renderParticleSets;
    std::map<Platform::String^, ParticleSet*> _particleSetCache;
    std::map<Platform::String^, Microsoft::WRL::ComPtr<ID3D11Texture2D>> _textureCache;
    Coding4Fun::FallFury::Screens::GameScreenBase^ _screenBase;
};
```

As previously mentioned, set activation can be done with some omitted parameters inferred by the system, such as randomization of velocity or particle scaling, which is the reason why you see multiple overloads for **ActivateSet**.

ParticleCore is also the container for pre-defined sets that have specific textures and properties that are dumped in the particle set cache (internal **_particleSetCache**). The cache is reusable and even though sets can be activated and destroyed, the cache remains intact for the duration of the game unless explicitly reset or modified. The texture cache is an addition to the particle set cache and is used as a helper container to temporarily store the textures used for individual particles.

Taking a look under the hood at the **CreatePreCachedParticleSets** function, you can see multiple sets that can be used in the game, each with different properties. Here is a snippet that shows how the bear stuffing explosion set is created:

```
auto smallExplosionSet = new ParticleSet(_textureCache["Stuffing"], LIFESPAN);
for (int i = 0; i < 20; i++)
{
    float size = RandFloat(50.0f, 100.0f);
    Particle particle(float2(size, size));
    smallExplosionSet->AddParticle(particle);
}
```

Once all the particles are in place for that specific set, it is added to the global particle set cache:

```
_particleSetCache["SmallExplosion"] = smallExplosionSet;
```

The particle set cache is not explicitly used to render anything on the screen. Rather, that task is delegated to the rendering cache. When a set is activated, the cache is inspected for the given key, its Activate function is called, marking it as alive, and the set itself is pushed on the rendering stack:

```
void ParticleCore::ActivateSet(Platform::String^ name, float2 position, float2 velocity, bool
randomize, bool scale)
{
    ParticleSet set = ParticleSet(*_particleSetCache[name]);
    set.Activate(position, velocity, randomize, scale);
    _renderParticleSets.push_back(set);
}
```

The Render loop takes care of invoking the proper **SpriteBatch** drawing functions for each set that is in that stack:

```
void ParticleCore::Render()
{
    for (auto set = _renderParticleSets.begin(); set != _renderParticleSets.end(); ++set)
    {
        if ((*set).IsAlive())
        {
            (*set).Render(_screenBase->CurrentSpriteBatch);
        }
    }
}
```

Notice that, as previously mentioned, the set does not perform the life check on itself when rendering. Instead, the manager class performs this action. The same applies to the **Update** loop:

```
void ParticleCore::Update(float timeDelta)
{
    for (auto set = _renderParticleSets.begin(); set != _renderParticleSets.end();)
    {
        if ((*set).IsAlive())
        {
            (*set).Update(timeDelta);
            ++set;
        }
        else
        {
            set = _renderParticleSets.erase(set);
        }
    }
}
```

The textures for each set are individually loaded in the **CreatePreCachedParticleSets**. Each instance is internally pushed into the cache and also added to the sprite batch associated with the current game screen, where **ParticleCore** is used:

```
Loader->LoadTexture("DDS\\stuffing.dds", &_textureCache["Stuffing"], nullptr);
_screenBase->CurrentSpriteBatch->AddTexture(_textureCache["Stuffing"].Get());
```

Once everything is loaded, the **ParticleCore** is ready to go and you can use as many particles as necessary in any part of the application. In **GamePlayScreen**, particle sets are activated in many cases. For example, if the bear dies:

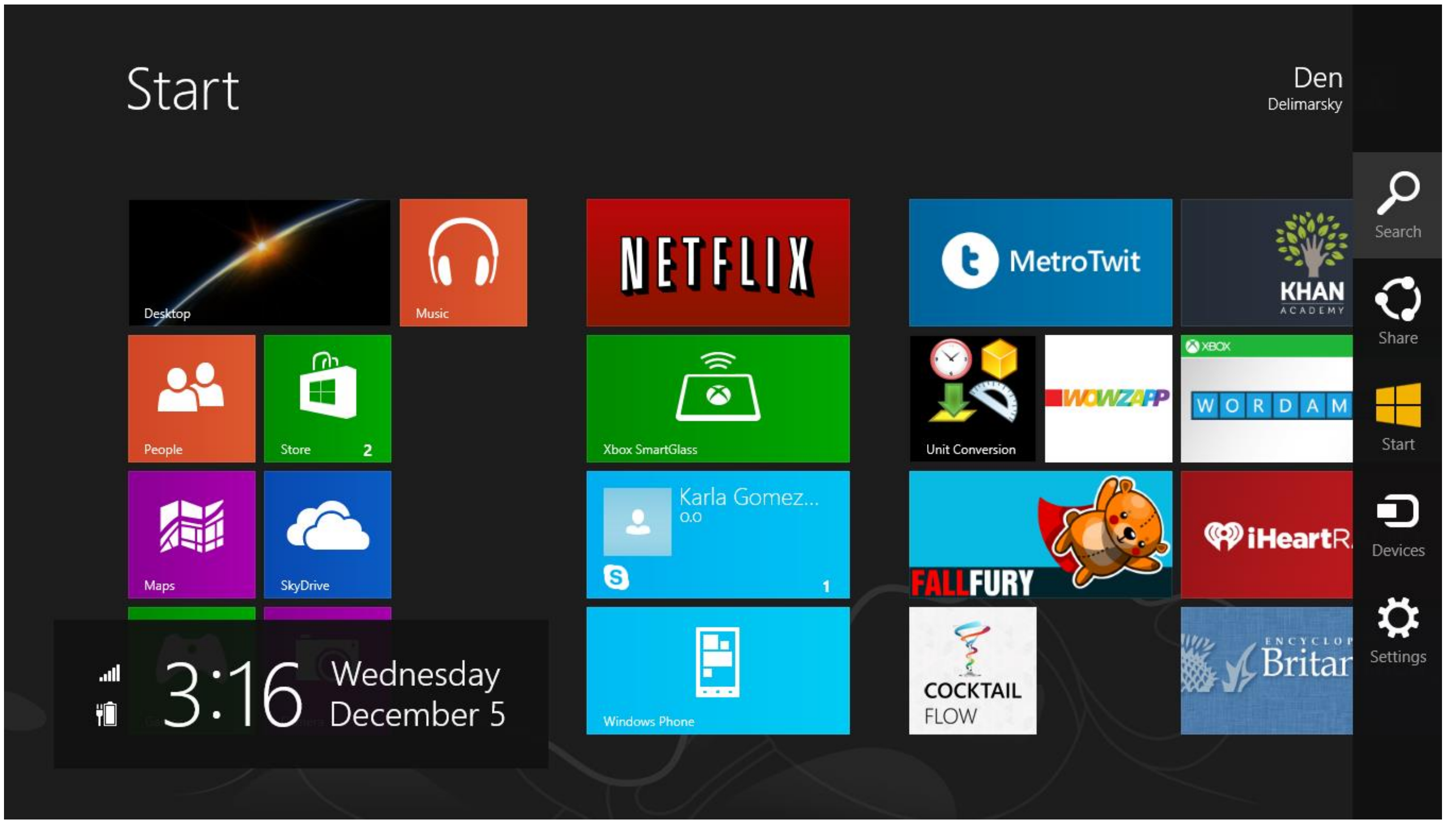
```
void GamePlayScreen::CheckBearHealth()
{
    if (GameBear->CurrentHealth <= 0)
    {
        m_particleSystem.ActivateSet("Buttons",GameBear->Position, true);
        GameBear->Kill();

        StopBackground();
    }
}
```

Conclusion

Implementing a sprite-based particle system is not complicated, but it requires depending on a number of assumptions. For example, when a particle set is created, you might consider the fact that some hardware can handle drawing only a given number of sprites at the same time. If a particle set is rendered on a desktop machine, there is no guarantee that the same set will successfully render on an ARM device. Therefore, plan accordingly. For each particle set, create a lifespan that fits the scenario without wasting resources on rendering unnecessary particles. As an additional failsafe, you might want to disable specific particle set types when a Direct3D feature level below 10.0 is detected.

Chapter 10 – Charms

With the release of the Windows 8 operating system, applications are now able to easily integrate with each other and have a unified way to control their workflow through unique system-wide contracts called Charms. Out of the multitude of available options, FallFury leverages two charms—Share contract and Settings. This article describes how the integration is implemented.

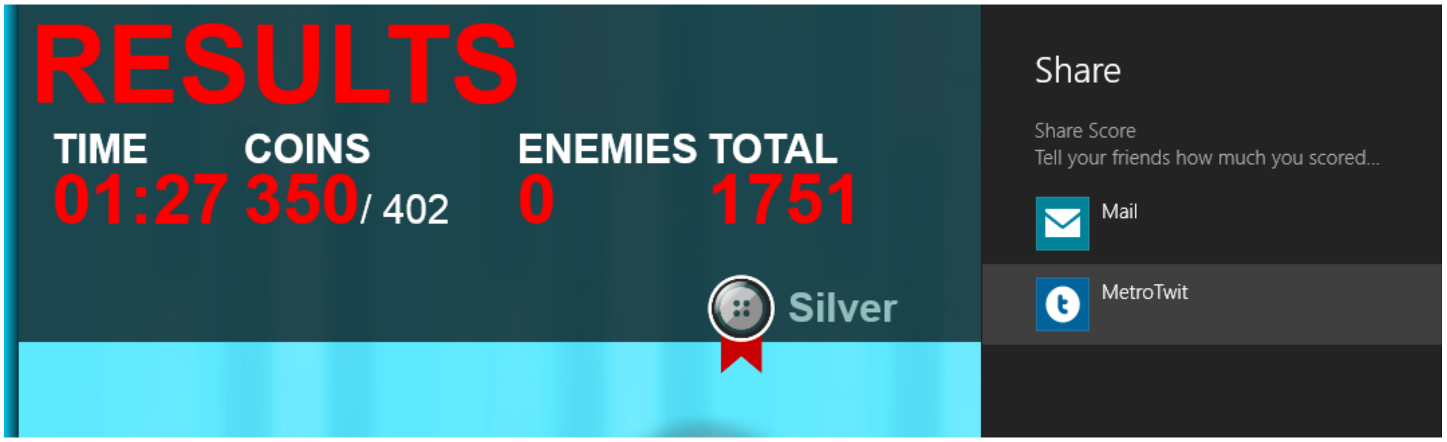


The Share Contract

When a user achieves a specific score in the game, he might decide to share it with his social circle. In Charms, that's the purpose of the Share contract, which integrates directly with the OS.

Windows Store applications have the capability to expose their sharing capabilities and register as a share target. For example, if there is a Twitter client out, it can register itself as an app through which content can be shared.

FallFury, on the other hand, acts as a consumer that aggregates existing share targets and lets the user pick the one through which he wants to let the message out:



Let's take a look at how this process is built in the code-behind. The core is located in **DirectXPage.xaml.cpp**—the class responsible for XAML content manipulation in FallFury. First and foremost, you need to get the current instance of the DataTransferManager class:

```
auto dataTransferManager =
Windows::ApplicationModel::DataTransfer::DataTransferManager::GetForCurrentView();
```

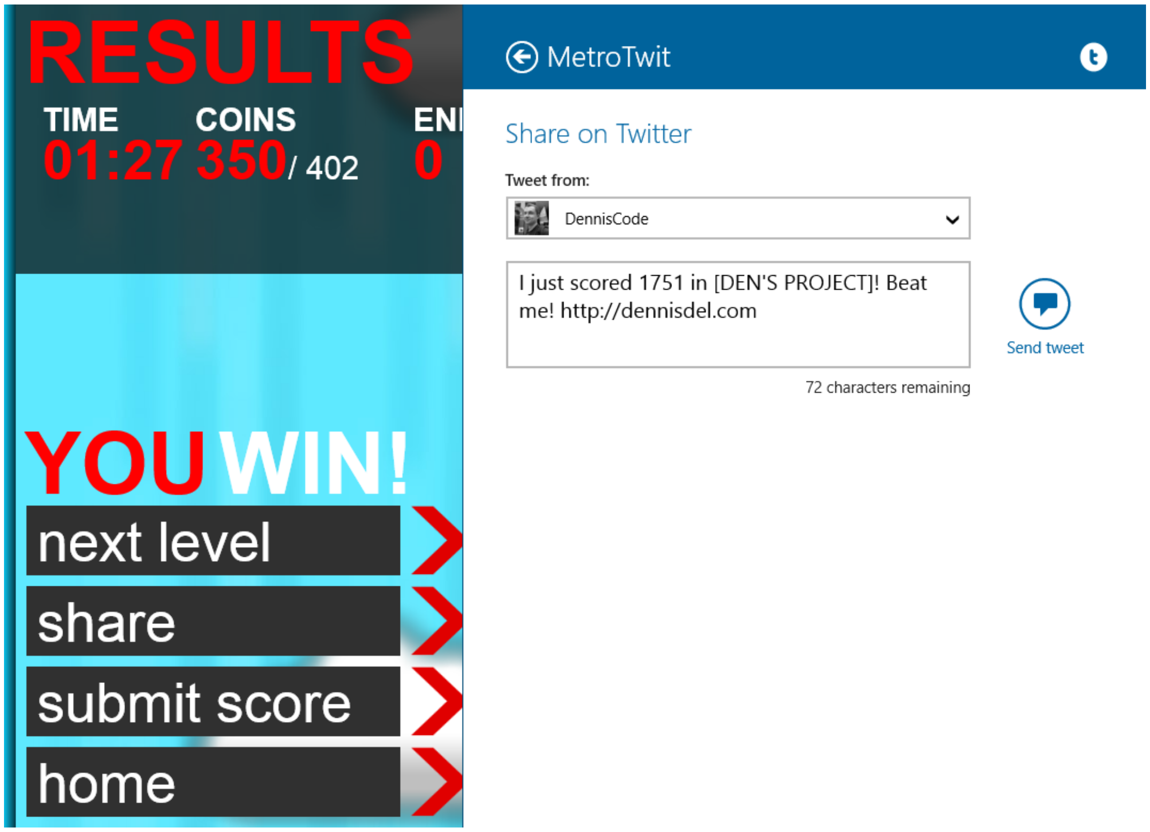
Consider this a proxy that allows you to pass content between your app and other Windows Store apps that are executed in the context of the same sandbox. As the instance is obtained, hook it to the **DataRequested** event handler that will handle the scenario where the user invoked the sharing capability:

```
dataTransferManager->DataRequested +=
   ref new TypedEventHandler<Windows::ApplicationModel::DataTransfer::DataTransferManager^,
   Windows::ApplicationModel::DataTransfer::DataRequestedEventArgs^>(this,
                                                       &DirectXPage::OnShareDataRequested);
```

In the **OnShareDataRequested**, specify the information that goes into the sharing popup:

```
void DirectXPage::OnShareDataRequested(
    Windows::ApplicationModel::DataTransfer::DataTransferManager^ manager,
    Windows::ApplicationModel::DataTransfer::DataRequestedEventArgs^ params)
{
    auto request = params->Request;
    request->Data->Properties->Title = "Share Score";
    request->Data->Properties->Description =
                                 "Tell your friends how much you scored in [DEN'S PROJECT]!";
    request->Data->SetText("I just scored " + StaticDataHelper::Score.ToString() + " in [DEN'S
PROJECT]! Beat me! http://dennisdel.com");
}
```

From that point, the control is in the user's hand. The application cannot force the share, so unless you implement a direct API hook to a social service, the Share charm will only expose the endpoints available for sharing and will let you set the shareable content. You also don't have to worry about the way the content will be shared—that will be handled by the target application:



I just scored 1751 in [DEN'S PROJECT]! Beat me! http://dennisdel.com

72 characters remaining

You can show the popup triggered by the Share charm from your application without having the user open the Charms Bar. To do this, call the ShowShareUI method:

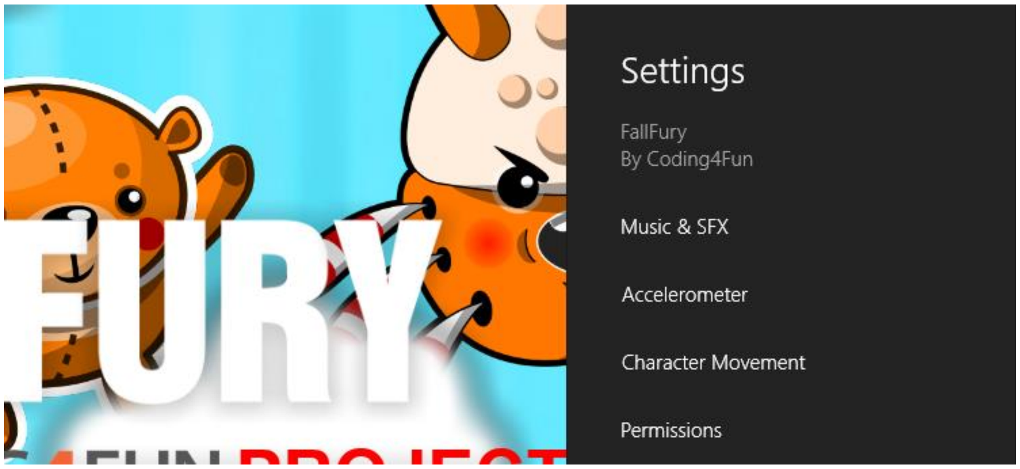
```
void DirectXPage::ShareTopScore_Selected(MenuItem^ sender, Platform::String^ params)
{
    Windows::ApplicationModel::DataTransfer::DataTransferManager::ShowShareUI();
}
```

This is exactly what the Share button does in the screenshot above. You should make this behavior predictable.

The Settings Charm

As you just saw, integrating basic sharing capabilities in FallFury is not too complicated. Working with settings is also a fairly easy task, though it involves some XAML work. While with sharing capabilities the work focused mostly on OS-based endpoints and application-specific popups, settings allow for full control over how they're displayed.

For all Windows Store applications, settings should be handled via the Settings charm and not through dedicated application screens. Consider which settings that directly affect the user experience might be changed in FallFury:



- **Music and SFX** – the user can enable or disable music and sound effects, as well as control their volume.
- **Accelerometer** – depending on personal preferences, the user might decide to disable the accelerometer (it is enabled by default). The accelerometer can also be inverted—if the device is tilted to the right, the character will move to the left and vice-versa. Last but not least, even with dynamic screen rotation enabled on the device, the user can disable that rotation on the application level and lock the screen to one orientation, such as portrait or landscape.
- **Character Movement** – the character can be easily controlled via touch (swipe) or mouse. This behavior is enabled by default, but if the user decides to only use the mouse to direct shells, he can easily disable this feature here.

As seen in the image above, the operating system provides the basic shell used to list the possible settings. Once one option is selected, however, further UI displays are delegated to the developer.

As with the share UI, the settings UI can be shown to the user from the application and not from the Charms Bar. Here is how to do this:

```
void DirectXPage::Settings_Selected(MenuItem^ sender, Platform::String^ params)
{
    SettingsPane::GetForCurrentView()->Show();
}
```

SettingsPane is the core class that handles the settings display. It does not control how settings are stored or activated. When the main page loads, you need to make sure that you hook the current **SettingsPane** to a **CommandRequested** event handler. It will be triggered when the Settings capability is invoked:

```
SettingsPane::GetForCurrentView()->CommandsRequested +=
    ref new TypedEventHandler<SettingsPane^, SettingsPaneCommandsRequestedEventArgs^>(this,
                                                                   &DirectXPage::OnSettingsRequested);
```

OnSettingsRequested is the function where the core setting selections are defined and hooked to their own event handlers:

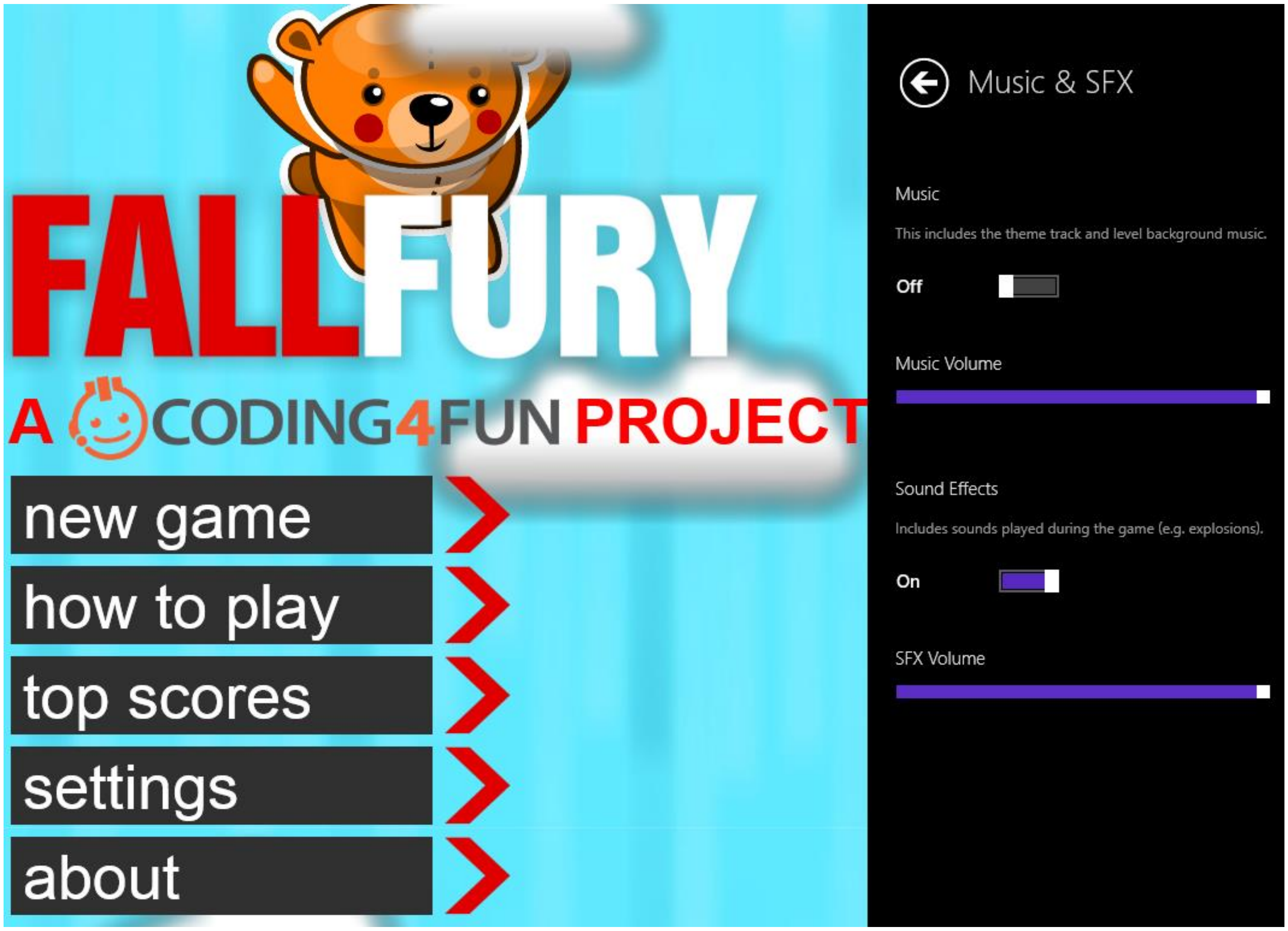
```
void DirectXPage::OnSettingsRequested(Windows::UI::ApplicationSettings::SettingsPane^
settingsPane, Windows::UI::ApplicationSettings::SettingsPaneCommandsRequestedEventArgs^
eventArgs)
{
    if (m_renderer->CurrentGameState == GameState::GS_PLAYING)
        StaticDataHelper::IsPaused = true;

    UICommandInvokedHandler^ handler = ref new UICommandInvokedHandler(this,
                                       &DirectXPage::OnSettingsSelected);

     SettingsCommand^ generalCommand = ref new SettingsCommand("musicSfx", "Music & SFX",
                                      handler);
     eventArgs->Request->ApplicationCommands->Append(generalCommand);

    SettingsCommand^ accelerometerCommand = ref new SettingsCommand("accelerometer",
                                            "Accelerometer", handler);
    eventArgs->Request->ApplicationCommands->Append(accelerometerCommand);

    SettingsCommand^ charMovementCommand = ref new SettingsCommand("charMovement", "Character
                                            Movement", handler);
    eventArgs->Request->ApplicationCommands->Append(charMovementCommand);
}
```

Each SettingsCommand is an item in the list displayed in the settings pane. When one is selected, **OnSettingsSelected** is called:

```
void DirectXPage::OnSettingsSelected(Windows::UI::Popups::IUICommand^ command)
{
    if (command->Id->ToString() == "musicSfx")
    {
        stkMusicSfx->Width = 346.0f;
        grdSubMusicSfx->Height = m_renderer->m_renderTargetSize.Height;
        stkMusicSfx->IsOpen = true;
    }
    else if (command->Id->ToString() == "accelerometer")
    {
        stkAccelerometerSettings->Width = 346.0f;
        grdAccelerometerSettings->Height = m_renderer->m_renderTargetSize.Height;
        stkAccelerometerSettings->IsOpen = true;
    }
    else if (command->Id->ToString() == "charMovement")
    {
        stkCharacterMovement->Width = 346.0f;
        grdCharacterMovement->Height = m_renderer->m_renderTargetSize.Height;
        stkCharacterMovement->IsOpen = true;
    }

    WindowActivationToken = Window::Current->Activated +=
         ref new WindowActivatedEventHandler(this, &DirectXPage::OnWindowActivated);
}
```

Looking back at **OnSettingsRequested**, each command has a string identifier. When a command is invoked, that string identifier is returned through the IUICommand instance in the Id property. Based on that, I decided which popups to open. Since each has a similar structure, I am going to cover the implementation of just one—**Music & SFX**.

Here is what the end result looks like:



Music

This includes the theme track and level background music.

Off

Music Volume

Sound Effects

Includes sounds played during the game (e.g. explosions).

On

SFX Volume

The panel on the left is a Popup, with two ToggleButton controls used to enable or disable generic music and sound effects. There are also two Slider controls that are used to adjust the volume. The XAML for the above layout looks like this:

```
<Popup HorizontalAlignment="Right" IsLightDismissEnabled="True" x:Name="stkMusicSfx" >
    <Grid Background="Black" x:Name="grdSubMusicSfx"  Width="346">
        <Grid.Transitions>
            <TransitionCollection>
                <RepositionThemeTransition />
            </TransitionCollection>
        </Grid.Transitions>

        <Grid.RowDefinitions>
            <RowDefinition Height="120"></RowDefinition>
            <RowDefinition Height="*"></RowDefinition>
        </Grid.RowDefinitions>

        <StackPanel Grid.Row="0" Orientation="Horizontal" Margin="24,12,0,0" >
            <Button Margin="0" VerticalAlignment="Center" x:Name="dismissAudioSettings"
                    Click="dismissAudioSettings_Click"
                    Style="{StaticResource BackButtonStyle}"></Button>
            <TextBlock Margin="12,0,0,12" Height="Auto" VerticalAlignment="Center"
                       Text="Music &amp; SFX"
                       Style="{StaticResource SubheaderTextStyle}"></TextBlock>
        </StackPanel>

        <StackPanel Grid.Row="1" Margin="24,24,0,0">
            <StackPanel>
                <TextBlock Text="Music" Style="{StaticResource BodyTextStyle}"></TextBlock>
                <TextBlock Width="346"
                           Text="This includes the theme track and level background music."
                           Style="{StaticResource CaptionTextStyle}" TextWrapping="Wrap"
                           Margin="0,12,12,12" ></TextBlock>
                <ToggleSwitch x:Name="tglMusic" Toggled="tglMusic_Toggled"
                              IsOn="{Binding ElementName=XAMLPage,Path=MusicEnabled}"
                              Margin="-6,0,0,0"></ToggleSwitch>
            </StackPanel>

            <StackPanel Margin="0,24,0,0">
                <TextBlock Text="Music Volume"
                           Style="{StaticResource BodyTextStyle}"></TextBlock>
                <Slider x:Name="sldMusicVolume" ValueChanged="sldMusicVolume_ValueChanged"
                        Value="{Binding ElementName=XAMLPage,Path=MusicVolume, Mode=TwoWay}"
                        Minimum="0" Maximum="100" Margin="0,0,12,0"></Slider>
            </StackPanel>

            <StackPanel Margin="0,24,0,0">
                <TextBlock Text="Sound Effects"
                           Style="{StaticResource BodyTextStyle}"></TextBlock>
                <TextBlock Width="346"
                           Text="Includes sounds played during the game (e.g. explosions)."
                           Style="{StaticResource CaptionTextStyle}" TextWrapping="Wrap"
                           Margin="0,12,12,12" ></TextBlock>
                <ToggleSwitch x:Name="tglSFX" Toggled="tglSFX_Toggled"
                              IsOn="{Binding ElementName=XAMLPage,Path=SFXEnabled}"
                              Margin="-6,0,0,0"></ToggleSwitch>
            </StackPanel>

            <StackPanel Margin="0,24,0,0">
                <TextBlock Text="SFX Volume"
                           Style="{StaticResource BodyTextStyle}"></TextBlock>
                <Slider x:Name="sldSFXVolume" ValueChanged="sldSFXVolume_ValueChanged"
                        Value="{Binding ElementName=XAMLPage,Path=SFXVolume, Mode=TwoWay}"
                        Minimum="0" Maximum="100" Margin="0,0,12,0"></Slider>
            </StackPanel>

        </StackPanel>

    </Grid>
```

```
</Popup>
```

For every Popup instance used for settings, make sure that **IsLightDismissEnabled** is set to true. This allows the user to dismiss the panel with a touch outside its boundaries, just like the stock system panels. Other than that, you are working with the standard XAML control set and can include virtually anything in your settings.

Notice, that the switches and sliders are bound to internal properties, such as **SFXEnabled** and **MusicEnabled**, that perform the binding via dependency property references:

```
DependencyProperty^ DirectXPage::_musicEnabled =
    DependencyProperty::Register("MusicEnabled", bool::typeid, DirectXPage::typeid, nullptr);
DependencyProperty^ DirectXPage::_sfxEnabled =
    DependencyProperty::Register("SFXEnabled", bool::typeid, DirectXPage::typeid, nullptr);
```

The properties themselves are declared in the **DirectXPage** header file:

```
static property DependencyProperty^ SFXVolumeProperty
{
    DependencyProperty^ get() { return _sfxVolume; }
}
property int SFXVolume
{
    int get() { return (int)GetValue(SFXVolumeProperty); }
    void set(int value)
    {
        SetValue(SFXVolumeProperty, value);
    }
}

static property DependencyProperty^ MusicVolumeProperty
{
    DependencyProperty^ get() { return _musicVolume; }
}
property int MusicVolume
{
    int get() { return (int)GetValue(MusicVolumeProperty); }
    void set(int value)
    {
        SetValue(MusicVolumeProperty, value);
    }
}
```

Let’s take a quick look at how settings are stored. I have a class called **SettingsHelper** that allows me to save, read, and check if specific settings exist. Here is the implementation:

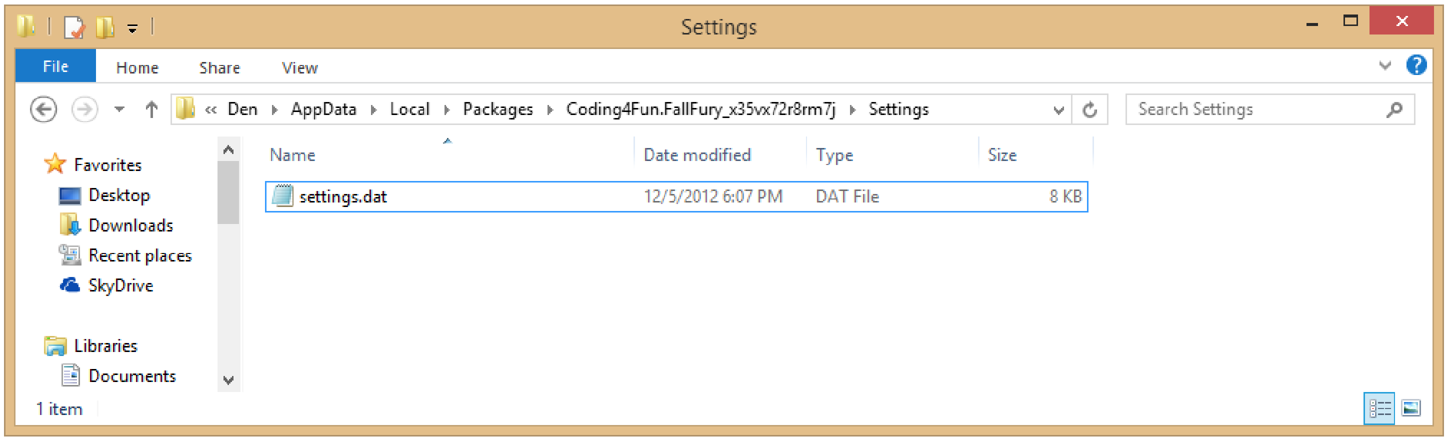
```
#include "pch.h"
#include "SettingsHelper.h"

using namespace Windows::Storage;
using namespace Coding4Fun::FallFury::Helpers;

SettingsHelper::SettingsHelper(void)
{
}

void SettingsHelper::Save(Platform::String^ key, Platform::Object^ value)
{
    ApplicationDataContainer^ localSettings = ApplicationData::Current->LocalSettings;
    auto values = localSettings->Values;
    values->Insert(key, value);
}

Platform::Object^ SettingsHelper::Read(Platform::String^ key)
{
    ApplicationDataContainer^ localSettings = ApplicationData::Current->LocalSettings;
    auto values = localSettings->Values;
    return values->Lookup(key);
}

bool SettingsHelper::Exists(Platform::String^ key)
{
    ApplicationDataContainer^ localSettings = ApplicationData::Current->LocalSettings;
    auto values = localSettings->Values;
    return values->HasKey(key);
}
```

It is clear that storage and retrieval heavily relies on the ApplicationDataContainer class, the container class for local settings that eliminates the need for the developer to create his own setting files, instead delegating this task to the OS and utilizing a centralized storage for all Windows Store applications.

A typical scenario that utilizes the class above is executed when the toggle that manages the sound effects is switched:

```
void DirectXPage::tglSFX_Toggled(Platform::Object^ sender,
                                 Windows::UI::Xaml::RoutedEventArgs^ e)
{
    SettingsHelper::Save("sfxEnabled", tglSFX->IsOn);
    SFXEnabled =  tglSFX->IsOn;

    AudioManager::IsSFXStarted = SFXEnabled;
    if (SFXEnabled)
    {
        AudioManager::AudioEngineInstance.StartSFX();
    }
    else
    {
        AudioManager::AudioEngineInstance.SuspendSFX();
    }
}
```

The Boolean value above will be automatically serialized and stored. The files will be located at **C:\Users\YOUR_USER_NAME\AppData\Local\Packages\PACKAGE_ID\Settings\settings.dat:**



Name	Date modified	Type	Size
settings.dat	12/5/2012 6:07 PM	DAT File	8 KB

Conclusion

Implementing sharing through the OS channel in Windows 8 is extremely easy seeing as the developer does not necessarily have to worry about connecting the app to third-party API endpoints. Instead, the user controls the sharing, allowing flexibility of choice without requiring a major addition to the existing code base. It's hard to predict which services might appear, and modifying the app to support each one of them would be next to impossible otherwise.

You can read more about settings in Windows Store applications here.

Chapter 11 – Hardware Testing

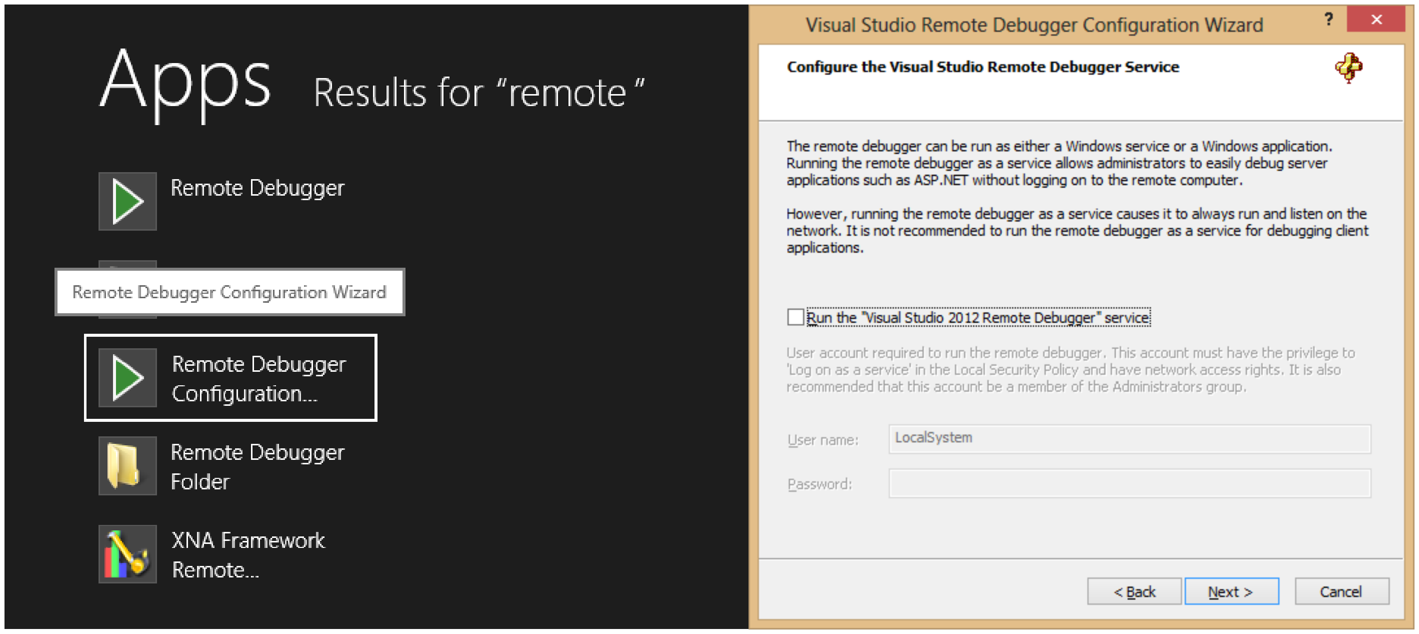
As previously mentioned, FallFury runs on multiple types of hardware as long as that hardware supports Windows 8. This article describes the project's general testing and debugging process, including setting the debug configurations and remote debugging.

Remote Debug

When working on an application that targets different machines, it's probably out of the question to install Visual Studio on each instance and move the solution from one source to another in order to run it and diagnose potential problems. Here is where Remote Tools for Visual Studio 2012 come into play. Microsoft offers you three separate builds—tools for x86 systems, x64 systems, and ARM systems—also known as Surface RT.

Once the tools are running on the machine you want to debug, you have two choices. You can either install the remote debugger as a service, allowing it to constantly run in the background, or you can use the debugger on a per-launch basis. From a developer perspective, the choice doesn't affect how your application is executed on the remote machine. From a security perspective, however, you need to be sure that you properly configure it so that no unwanted apps are remotely deployed.

Now you can start the Remote Debugger Configuration Wizard:


Apps
Results for "remote"
Remote Debugger
Remote Debugger Configuration Wizard
Remote Debugger Configuration...
Remote Debugger Folder
XNA Framework Remote...
Visual Studio Remote Debugger Configuration Wizard
< Back
Next >
Cancel

Configure the Visual Studio Remote Debugger Service

The remote debugger can be run as either a Windows service or a Windows application. Running the remote debugger as a service allows administrators to easily debug server applications such as ASP.NET without logging on to the remote computer.

However, running the remote debugger as a service causes it to always run and listen on the network. It is not recommended to run the remote debugger as a service for debugging client applications.

☐ Run the "Visual Studio 2012 Remote Debugger" service

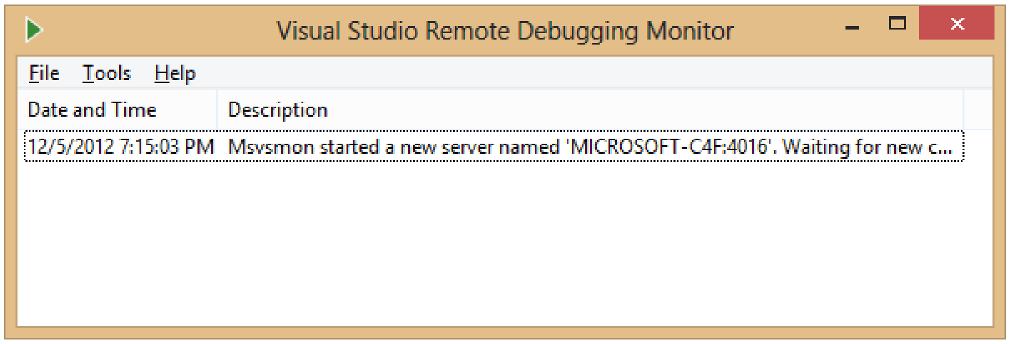
User account required to run the remote debugger. This account must have the privilege to 'Log on as a service' in the Local Security Policy and have network access rights. It is also recommended that this account be a member of the Administrators group.

User name: LocalSystem

Password:

This works on both the initial start-up and also any subsequent launch as a way to easily and quickly set the necessary remote debugger settings. Specifically, it is useful to configure the machine's network settings, allowing cross-domain communications for debugging purposes.

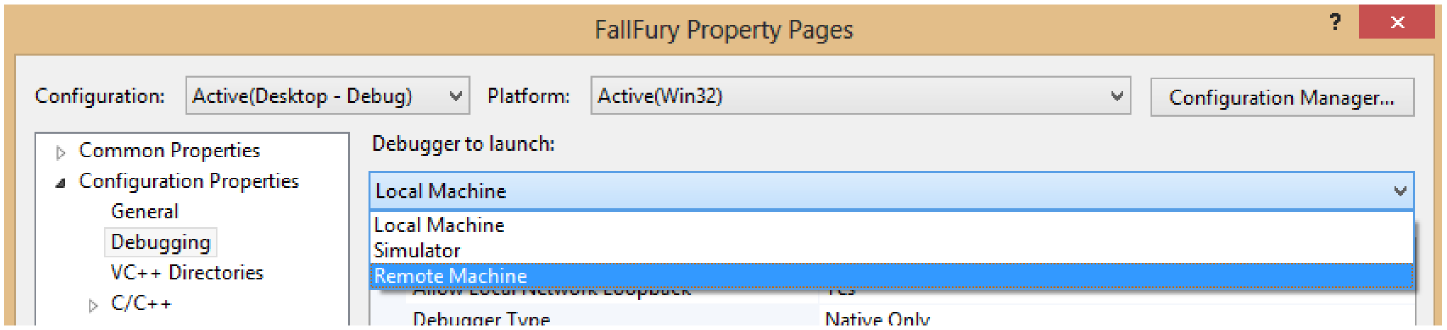
Once the wizard is completed, launch the Remote Debugger Monitor. Notice that it lists your machine name and the port on which it's running. This is necessary for configuring the project to send the package to a remote machine instead of the local one:



Date and Time	Description
12/5/2012 7:15:03 PM	Msvsmon started a new server named 'MICROSOFT-C4F:4016'. Waiting for new c...

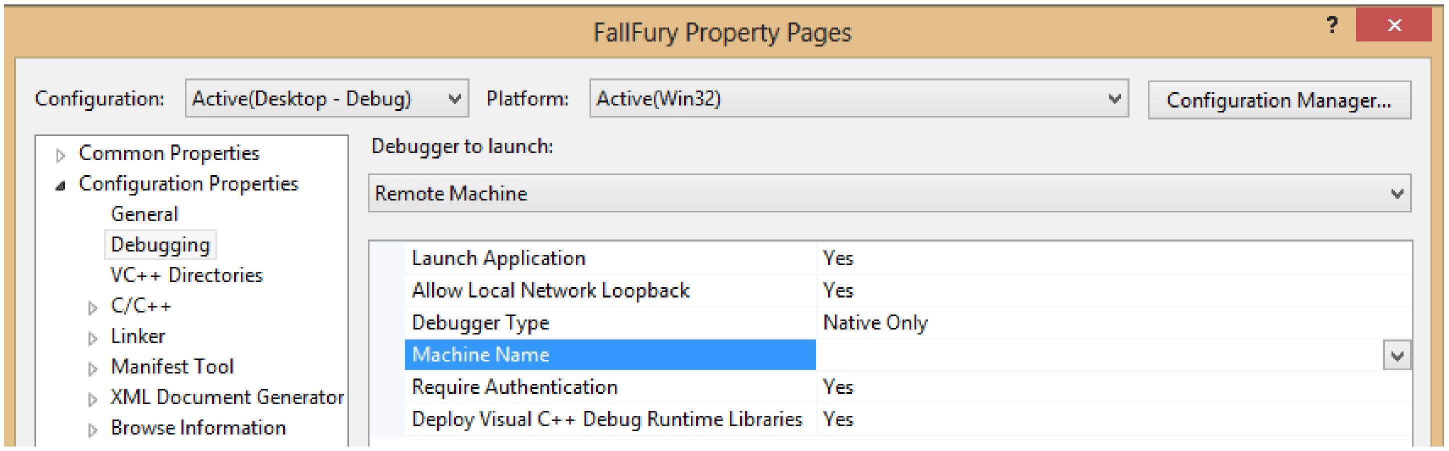
The configuration depends on the network settings on both the local machine and the subnet as a whole. For example, in some cases, and especially on domain-joined machines, remote debugging is better done with the authentication disabled. Windows Authentication is used by default.

Since FallFury is a C++ project, the configuration for a remote session is different than, say, that of a C# Windows Store application. To configure the session, right click on the project in **Solution Explorer** and open the **Debugging** section. Make sure that Remote Machine is selected as the type of debugger to launch:



Configuration: Active(Desktop - Debug)
Platform: Active(Win32)
Debugger to launch: Local Machine

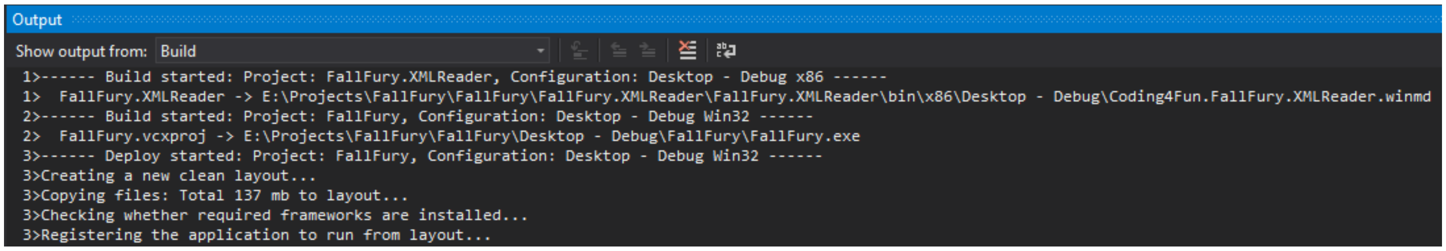
Next, specify the machine name as well as whether the current session will require authentication:



Configuration: Active(Desktop - Debug)
Platform: Active(Win32)
Debugger to launch: Remote Machine

Launch Application	Yes
Allow Local Network Loopback	Yes
Debugger Type	Native Only
Machine Name	
Require Authentication	Yes
Deploy Visual C++ Debug Runtime Libraries	Yes

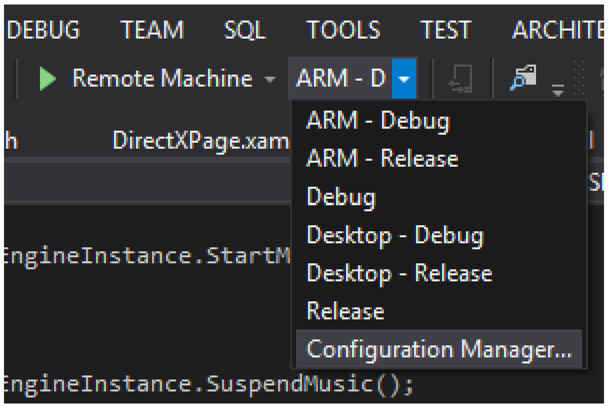
If you cannot specify the machine name, use the direct IPv4 address of that computer, minus the port (unless you've explicitly set it to a port other than 4016, which is the assumed default). Once the source machine connects to the remote one, you will see Visual Studio performing the deployment:



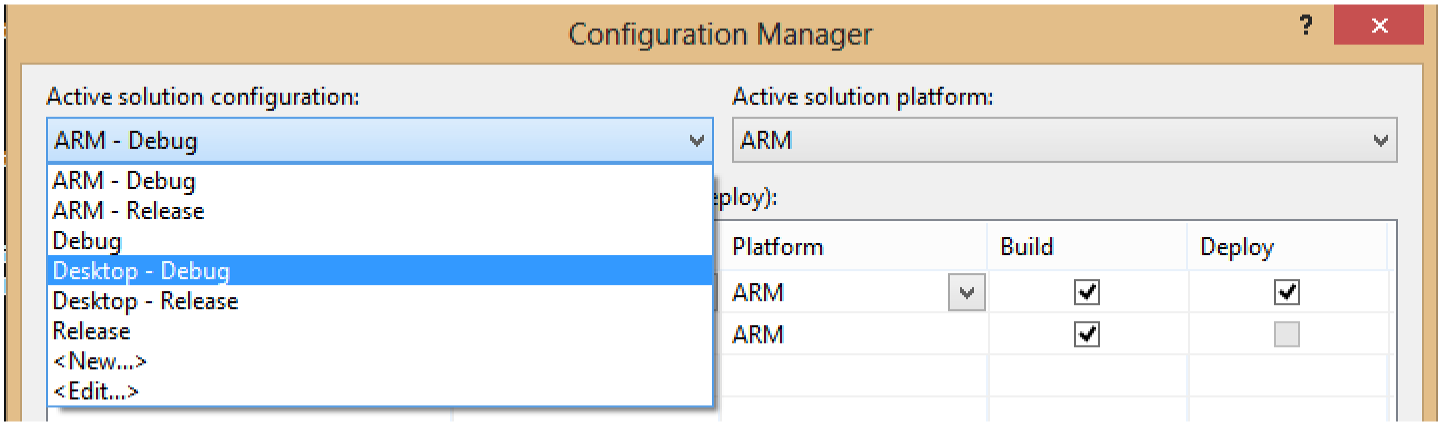
```
1>------ Build started: Project: FallFury.XMLReader, Configuration: Desktop - Debug x86 ------
1>  FallFury.XMLReader -> E:\Projects\FallFury\FallFury\FallFury.XMLReader\FallFury.XMLReader\bin\x86\Desktop - Debug\Coding4Fun.FallFury.XMLReader.winmd
2>------ Build started: Project: FallFury, Configuration: Desktop - Debug Win32 ------
2>  FallFury.vcxproj -> E:\Projects\FallFury\FallFury\Desktop - Debug\FallFury\FallFury.exe
3>------ Deploy started: Project: FallFury, Configuration: Desktop - Debug Win32 ------
3>Creating a new clean layout...
3>Copying files: Total 137 mb to layout...
3>Checking whether required frameworks are installed...
3>Registering the application to run from layout...
```

Configurations

As a matter of convenience, it is always good to have different debug configurations that will define how your projects are built, especially if the project targets multiple platforms (such as ARM and x86). Visual Studio provides a Configuration Manager that can be accessed from the debug target dropdown:

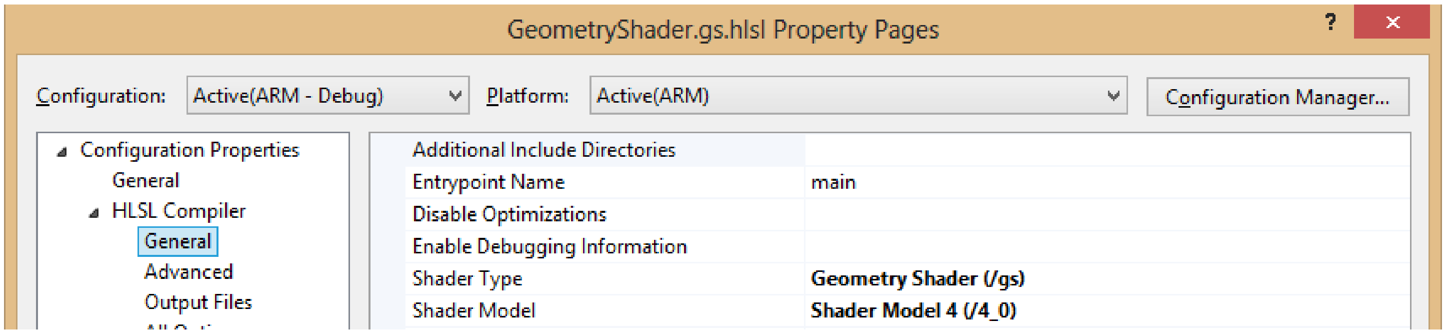


FallFury includes two separate projects—the game core, and the C#-based XML reader. Both need to be explicitly associated with separate target platforms in order to be correctly debugged. For that, profiles were created for both local and remote sessions:



Platform	Build	Deploy
ARM	✓	✓
ARM	✓	

As with standard Debug/Release configurations, the ones shown above determine whether the project will carry debug symbols and support debugging commands. It is important to mention that as you switch configurations, you must be careful how you configure the graphic shaders. For each shader in the container folder, explicitly set the HLSL shader type and model. Otherwise the application deployment will fail:


GeometryShader.gs.hlsl Property Pages

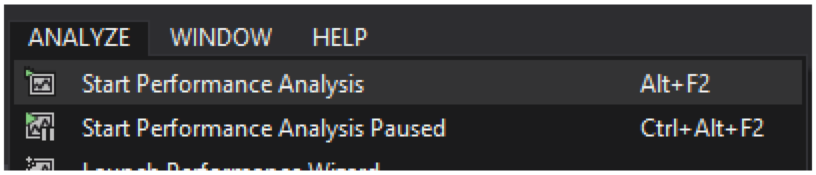
Configuration: Active(ARM - Debug) Platform: Active(ARM) Configuration Manager...

Additional Include Directories	
Entrypoint Name	main
Disable Optimizations	
Enable Debugging Information	
Shader Type	**Geometry Shader (/gs)**
Shader Model	**Shader Model 4 (/4_0)**

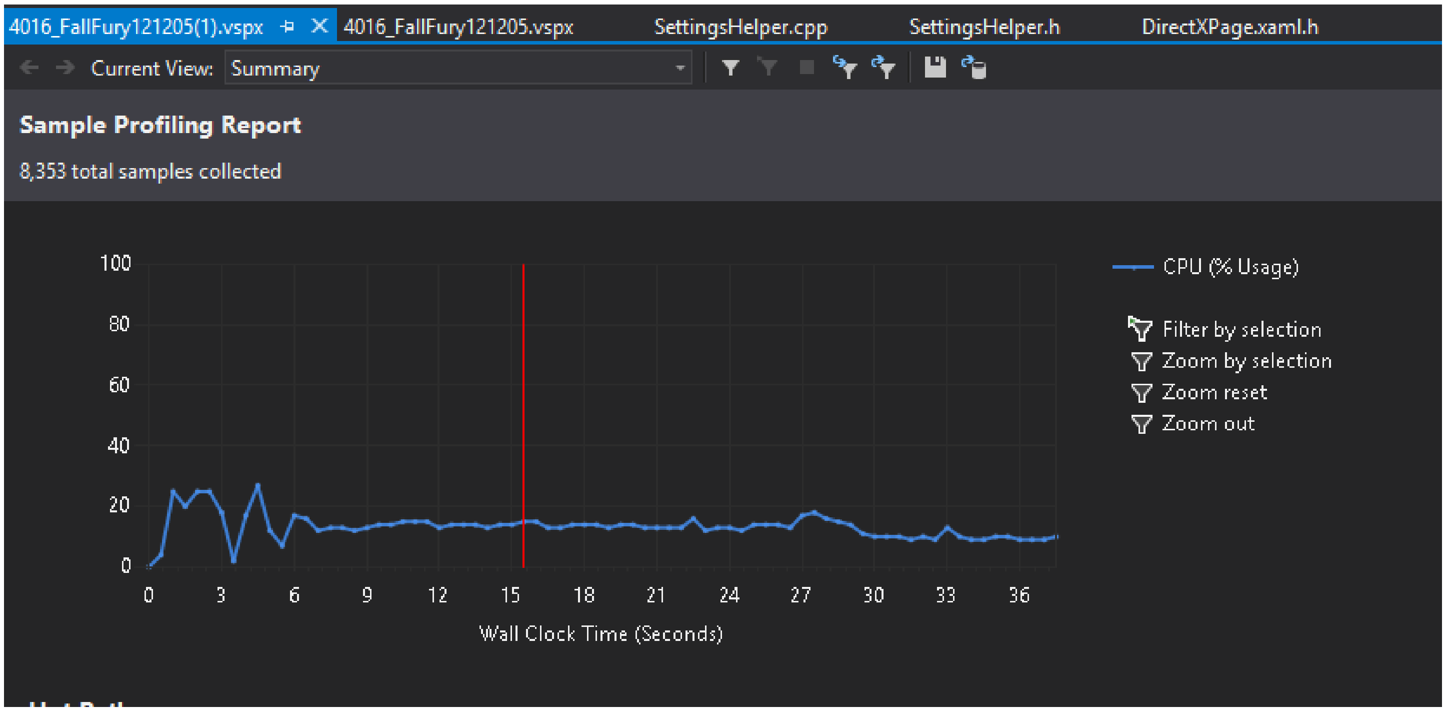
Remote Profiling

Last but not least, when working with different hardware configurations it might be useful to perform application profiling or performance review. Fortunately, Visual Studio also provides this capability through the vsperf tool, which is already integrated in the IDE if you are using Visual Studio 2012 Professional or above.

To initiate a profiling session on a remote device, the Remote Debugger Monitor must be active. Make sure that the **Remote Machine** debug target is selected, and go to **Analyze** > **Start Performance Analysis**:


ANALYZE
WINDOW
HELP
Start Performance Analysis Alt+F2
Start Performance Analysis Paused Ctrl+Alt+F2

On the remote machine, allow the **vsperf** process to run with administrative privileges. Once the profiling session completes and the data is analyzed, you can review the same performance indicators as you would when having a standard application run the profiler on the local machine:



Conclusion

Testing hardware outside the boundaries of a desktop machine is often a necessity, especially if the application relies on specific sensors, such as NFC, touch, or accelerometer. The remote debugging process is fairly streamlined and intuitive, with developing a proper network configuration allowing communication between machines requiring the most significant amount of effort. If you have problems getting the debugger to work, consult this article on MSDN.

Chapter 12 – Project Conclusions

Developing FallFury was a fun and educational activity. Besides the fact that I got to work with an amazing team of people who were willing to assist me with the testing and development, I learned some important points about the development process as a whole and have outlined them in this article.

Go Outside Your Comfort Zone

This past summer, I decided to tackle a new challenge—game development. Never before had I built a game from the ground up. Moreover, as I could already find my way around C# code, I decided that I wanted to make C++ the language of choice, and so worked with DirectX instead of XNA. Additionally, the project needed to be constructed with Windows 8 as the target platform and I'd have to rely on the WinRT stack.

The concepts were new to me, but I figured that learning about a new platform and a new language would be more interesting than sticking to my comfort zone. Through FallFury I extended my knowledge about pointers, trigonometry, how DirectX handles a lot of the graphics legwork, and how to write HLSL shaders.

It was a challenging but very rewarding experience.

Communicate

The most important part of any project is communication. There are always people who know more than you in one field or another, so don't hesitate to reach out to other developers and ask questions. Your code is not perfect, so talk about best practices used in production and how you can improve on what you already have—communication should not be limited to your team, floor, or even building.

One bonus of my experience at Microsoft was discovering that a lot of developers, when not in a meeting or in the middle of an important assignment, are more than happy to help a fellow employee with a piece of advice or some code review.

Plan the Small Details

Before you even touch the keyboard to write your first line of code, outline the components of your application and have a general idea of how you'll implement those components as well as how they'll interact with each other. It is tempting to start coding and then add or modify features, but doing so will create more problems than it will solve.

Build Small Prototypes

FallFury is a complex project composed out of blocks such as:

- XML level/metadata reader and writer
- The Share and Settings charm components
- SpriteBatch sub-system
- Level renderer
- Screen sub-system
- Menu interaction sub-system

Each block is responsible for its own set of tasks and each can be individually tested with either mocked data or a loose connection to the existing codebase. The necessity for small prototypes comes from individual requirements tied to the platform and possible integration scenarios. For example, when I began development, I created a non-hybrid DirectX project, which meant that if I needed to introduce XAML integration, I had to rewrite small parts of my swap chain preparation stack as well as work with a lot of XAML content in the code-behind. The Settings charm integration heavily relied on XAML, and I would have had to write the popup and control-related code in C#.

Obviously, that was not an option, so I created a hybrid project instead and ported most of my code into the new solution. Lesson learned here: prototype—and prototype early—in order to make sure that what you plan to implement will be implemented in an efficient manner.

Test Often, Test with Other People

When working on an application, it is important to understand the needs and expectations of your target audience, and a big part of this is making sure that the application provides a way for the user to easily learn about an action without a helper booklet. The menu system is a good example of this in FallFury. As I was designing the buttons on my screens, I tried to implement the approach used in games such as Dance Central—the user simply slides the button-panel and it takes him to the screen that was linked to that specific button. However, as I was testing the application with the local development crowd, I noticed a pattern—people were attempting to simply tap the button instead of sliding it. To avoid this confusion, I added a pulsating arrow on the right side of the button in order to highlight the fact that the button should be displaced in order to be activated.

The XML level loading system is another example of how secondary testing improved FallFury. Initial element positioning relied on a given level length, as well as on a ration-based placement. For example, if I wanted to put a button in the middle of the screen, I could position it with a value, such as .5. The level loader would then translate this value relative to the screen size. Rick Barraza was creating some test levels when he notified me about a big flaw—as the level extended, some elements were mis-positioned by a large margin compared to the original intended location. This was because when a level length was reset, the ratio was applied on a larger/smaller value, and some elements would therefore overlap. Ultimately, I rewrote the level structure to rely on pixel values, and designed the level-length to calculate automatically based on the included elements and their size.

Test on Different Hardware

FallFury started with portable hardware in mind. Its original design targeted Windows 8 tablet devices, though it was later decided that the project should support desktop machines as well. That automatically added thousands of possible hardware combinations on which the game might potentially run. Here are just some of the test machines we got to use:

- Samsung Series 7 Slate
- ARM Developer Tablet (Microsoft Surface)
- ASUS EP121 Slate

Each of these machines came with its own specific conditions that had to be accommodated. While Samsung Series 7 Slate featured an accelerometer, at the time of development the ASUS EP121 tablet did not have Windows 8 drivers for its sensor. When I sideloaded the game, I noticed that I could no longer control the character the way I wanted to. Also, though FallFury has keyboard controls, I had to develop controls for machines without keyboards. This is why I came up with the concept of touch-based, on-screen controls.

Moving on to the ARM device, first and foremost I noticed that my shader configurations were not compatible with the platform capabilities. All my testing was done on a machine that supported DirectX Feature Level 10, which was not the case for a low-power ARM machine that supports DirectX Feature Level 9.1. I had to work with some MSDN samples to find a way to properly modify the required shaders so they would work on Windows RT. ARM devices are built with battery life in mind. Therefore, their capabilities are reduced compared to those of a full-sized Windows tablet. This had an impact on the general game performance—some parts of the game, such as particle rendering, were working fine on a desktop machine, but were quite laggy on the ARM tablet. Knowing the possible source of the problem, I had to optimize the particle storage and rendering stack to ensure they didn't affect the user experience on Windows RT hardware.

There Is Never Enough Time - Learn To Cut

When I started planning FallFury, I worked up a huge list of features. As the deadline approached, however, I noticed that I somewhat underestimated the allocated project time—a lot of time was spent on testing and eliminating bugs in existing features. That was the point at which I had to cut my feature set.

When designing the feature list for your app, make sure that you have a clear vision of which features get the priority and which can be cut if necessary. For example, FallFury was originally planned to be a multiplayer game. Although it was tempting to start working on it as such from the very beginning, I knew that it wasn't a core feature and that it wouldn't necessarily affect the general play experience on release day. Ultimately, multiplayer capability was dropped from initial development along with a number of other not-immediately-integral features.

Expect the Unexpected

When allocating time for the project, anticipate situations that might derail the process and cause a delay in the planned delivery date. It is easy to estimate that creating an accelerometer-based motion system will take you a day or two, but you also need to allow for the potential possibility of your code not working as expected, hardware accidents, natural disasters, etc. Doing so will help you focus on the right features at the right time while diminishing the risk of missing the deadline.

Platform Samples Are the Best Documentation - Learn to Read Others' Code

While building the core of FallFury, I leveraged multiple features that hadn't yet become mainstream among Windows developers. Therefore, the documentation was scarce and in some instances the only help I could get was internal. Luckily, Microsoft bundled hundreds of samples that demonstrate what the new platform can do. A lot of the development approaches gained during my internship were learned from looking at the official, publicly released sample code. The lesson here: if it's not on MSDN or blogs, check the official developer samples. For example, I used the DirectX sprite drawing sample to see how to build a proper sprite rendering mechanism with behavior similar to the one available in XNA.

Atomic Commits

When I started, source control wasn't new to me, but there were proper usage practices that I didn't apply before the internship. One of them is atomic commits, which I learned from Clint Rutkas.

With every new addition to the application, chances are something might go wrong. It could be a faulty third-party assembly, a misplaced class, or a broken reference that brings down the entire solution. If the commits include longer time intervals, for example, you will ultimately have to roll back to a build without a lot of the new functionality. Using atomic commits, however, you'll be able to avoid unnecessary, redundant work in the case that something goes wrong during development.

A Non-standard Approach Is Not Necessarily a Bad Approach

For loading high scores, I needed the UI layout to display the registered items. To do so, I had could either use data binding or write a simple routine that would generate XAML items on the fly in the code-behind. Data binding is a pretty common practice among XAML developers, so my initial thought was to use that, but I would have had to create a C++ binding harness in addition to the existing data retrieval code. The second approach was much cleaner because it didn't require me to rewrite or add much code. The lesson here: pick the best approach for the job and don't blindly follow programming trends.

Focus on the User Experience

At the end of the day, the user doesn't care whether you wrote the application in native DirectX or XNA, whether you applied the MVVM pattern, or whether you have easily readable code. The user cares about having a great time while playing the game, regardless of its backbone. It's your goal as a developer to deliver that user experience and ensure that the user feels comfortable with the product. This means that the development cycle goes beyond simply writing code. In FallFury, levels are dynamic, which means the source code doesn't necessarily need to be modified in order to create a different playable experience. But that experience matters in any case, and level design is a huge part of FallFury's final product. Users need to be immersed in the game world, so levels must be planned in such as way that the user returns to the game and doesn't abandon it after a couple of lost rounds.

Conclusion

FallFury was my first major project that I wrote almost entirely in C++, targeting the newly released Windows 8 operating system. Although the development process was challenging, it was proven to be a great educational experience that taught me about the approaches and practices that are used internally at Microsoft.