Porting tips for Windows Store with Unity

This document is evolving constantly with new and updated information. It is still a work in progress. If you need answers that this document does not address, try the Unity Windows Development Forum: http://forum.unity3d.com/forums/50-Windows-Development

Contents

Introduction

This write-up provides detailed guidance and coding samples on techniques that will be helpful when porting your Unity game to the Windows Store.

To get the most out of this document, you should first read the Getting started on Windows Store with Unity overview; that document presents the context and motivation for some of the tips in this write-up.

Sample Code

A Sample Unity Project Github Repository, Windows 8.1 solution and Windows Phone 8 solution have been provided to highlight many of the techniques highlighted in this section.

Common tasks

There are a number of common tasks you will likely encounter while porting your game to the Windows Store.

- Getting your app to compile in Unity
- Loading your game gracefully
- Orientation Support
- Writing Platform Specific Code
- Graphics Issues
- Pausing and Resuming your Game
- Window Resizing
- Text Input on Windows
- Performance Analysis

We are going to discuss the tasks individually, in a random order influenced by the complexity of the task, the frequency, and the perceived relevance to you.

Getting your app to compile in Unity

Windows Store apps will run against the .NET Core profile instead of Mono.
.NET Core profile is a subset of the full .NET Framework, therefore, during your port, you will likely run into a few classes available in Mono (and available in the full version of .NET) that are not in the .NET core profile subset allowed in Windows Store.

If you are faced with a compiler error inside your Unity code, this will be because

- The class itself is missing e.g. .Hashtable
- There is a method that is missing or has an unsupported overload e.g. .String.Format

When porting, you want to minimize the amount of changes you make to the existing code base so the recommended approach is to create the missing class with the same name, or create extension methods with the missing/unsupported method overload. This approach will maintain highest portability with other platforms; it will minimize risk of introducing new bugs, and be easiest way to 'rollback' your workaround when Unity or Microsoft make the types available later.

Using the Legacy classes

The sample project provides legacy class implementations of some older .Net namespaces and classes. These are referenced using a slightly different namespace to avoid conflicts with the Mono/.Net assemblies in the Unity editor.

So rather than System.IO, the namespace would be *LegacySystem*.IO. This means you only have to change the using directives in your game code to support these classes.

```
#if UNITY_METRO && !UNITY_EDITOR
    using LegacySystem.IO;
#else
    using System.IO;
#endif
```

Here is a high level aggregation of the most common missing types, along with suggested workarounds for dealing with these. The list is not all inclusive.

Collections

You will find that a few popular classes in the System.Collections namespace are missing. The missing types include Hashtable, ArrayList, OrderedDictionary, SortedList, Queue, Stack and a few others.

In the meantime, the sample project in this write-up includes implementations for these. You can find sources at */UnityPorting/tree/master/PlatformerPlugin/MyPluginUnity/Legacy/System/Collections*

File storage and IO

Windows Store has a rich set of File IO APIs, but the programming model is asynchronous and the namespaces are different, so you will not find a few of the classes in System.IO namespace – the most frequently missed ones are System.IO.File, System.IO.StreamReader, System.IO.Directory.

Full wrappers for these missing types are not available, but Unity ships two classes (namely File and Directory) in the UnityEngine.Windows namespace that can help you implement the basic IO functionality needed for games.

The sample project in this write-up also includes mostly implementations for File and Directory classes at: */UnityPorting/tree/master/PlatformerPlugin/MyPluginUnity/Legacy/System/IO*

If these wrappers do not suffice you can extend further, look at Windows.Storage namespace in WinRT and also look at the WindowsRuntimeStreamExtensions class that makes it easy to convert WinRT streams to .NET streams.

Socket-based networking APIs

The networking classes in System.Net are not available on .NET core.

The sample project in this write-up also includes mostly complete implementations for System.Net.TCPClient using Windows.Networking.Sockets WinRT namespace:

/UnityPorting/tree/master/PlatformerPlugin/MyPluginUnity/Legacy/System/Net

WinRT has new sockets APIs in Windows.Networking namespace and you should be able to extend on the approach demonstrated in the sample project functionality. There is also the option of looking at 3rd party solutions such as Photon.

Crypto

Some APIs in System.Cryptography namespace are missing. The most common task when using Crypto is computing hashes, so Unity provides you with a few small wrappers in UnityEngine.Windows

for computing MD5 and SHA1 hashes. If these do not suffice, use the Windows.Security.Cryptography classes.

Adding other classes

The list above is not all-inclusive, but it outlines the most popular missing types and should give you the right context on amount of work needed to port your game. Look at the sample project to get familiar with the techniques to implement and include missing functionality without disrupting your existing code substantially here:

/UnityPorting/tree/master/PlatformerPlugin/MyPluginUnity/Legacy

When implementing replacement classes/methods, you might need to rely on native APIs (available in Windows Store but not in .NET core) and for this you will need to reference the "Writing Platform Specific Code" section below which details approaches for communicating between Windows and Unity.

A good example of this is the System.IO.Directory implementation in the sample project in

/UnityPorting/tree/master/PlatformerPlugin/MyPluginUnity/Legacy/System/IO/Directory.cs

3rd Party Plugins

Besides the core classes covered above, you might need to reference 3rd party plugins. Some of the popular plugins (such as NGUI or Toolkit2D) have already been ported to work on Windows Store. Other plugins might not have been ported yet.

If you have plugins that are shipping as source code (such as NGUI), then the compiler will catch most issues for you. If you have plugins that ship as a binary, you will likely get errors at run-time when you try to load them or use them. A good way to check if a plug-in is compatible is to run it through the http://scan.xamarin.com. You can then see if a plugin is compatible with Windows Store APIs.

If your plugin is not compatible, contact the plugin author or email jaimer@microsoft.com and we will try to contact the author.

Once you have implemented all the missing classes or methods your game should then be able to compile into a Windows Store app.

Reporting progress as your game loads

When your game launches, it's important to show the user a progress indicator so the user knows that the game is not frozen or stalled.

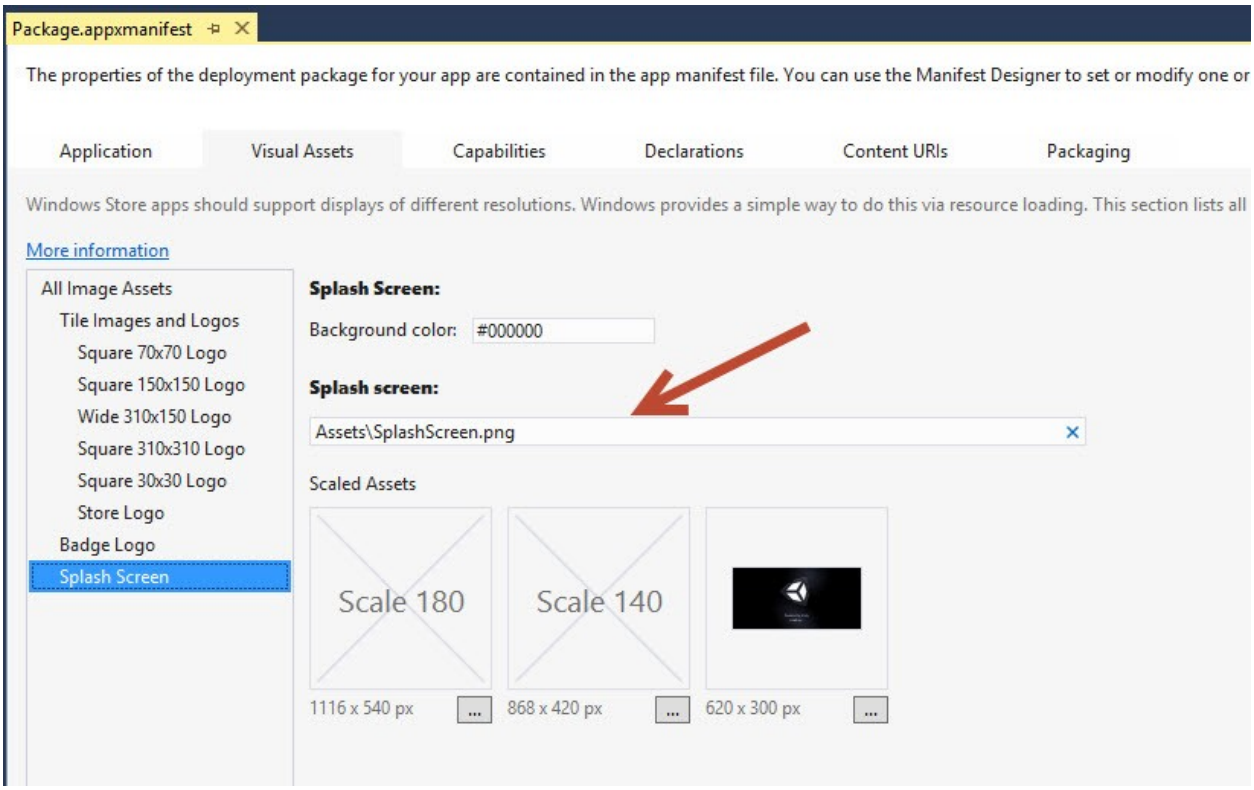
In this write up, we will assume you are using a XAML project. If you are not, most of the concepts here will still apply to a DirectX project, but the implementations will be quite different.

There are 2 key stages to the loading of a game on Windows Store/Windows Phone using Unity

1. App Splash Screen
2. Extended Splash Experience

App Splash Screen

The initial splash screen image will be shown by the OS and you can configure it by updating your applications manifest as follows:

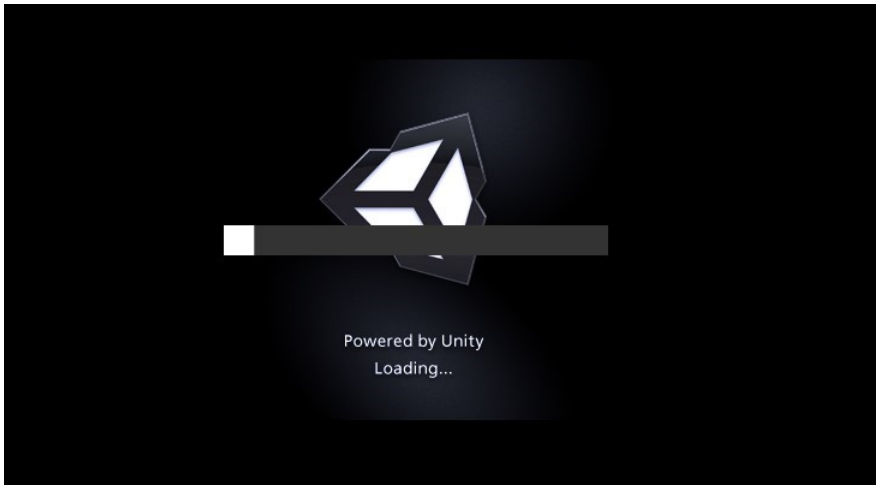


Extended Splash Experience

Windows will show the initial app splash screen only during the time it takes the OS to load the Page (written in XAML markup) that will host your Unity scenes.

Usually, the unity scene will take longer to load so what you will want to do is use XAML to show the same splash image again with a progress indicator on top to the user while your unity scene loads, this is the extended splash experience.

The Sample Unity Project has a basic example of this. The progress bar is displayed whilst the game loads along with the splash screen and then when Unity has finished loading, it is removed and the game appears.



In MainPage.xaml you can see there is a progress bar added alongside the splash screen image.

Note: Make sure the maximum is set to a value which is longer than it takes the game to load on your slowest device to ensure the progress bar never maxes out. You can also just make the ProgressBar indeterminate – an indeterminate progress bar shows the animated dots moving around the screen, but it does not incremental progress.

```
    <SwapChainBackgroundPanel x:Name="DXSwapChainBackgroundPanel">
        <Grid x:Name="ExtendedSplashGrid">
            <Image x:Name="ExtendedSplashImage" Source="Assets/SplashScreen.png"/>
            <ProgressBar x:Name="SplashProgress" Foreground="#FFFFFFFF"
Background="#FF333333" Maximum="10000" Width="320" Height="25"/>
        </Grid>
    </SwapChainBackgroundPanel>
```

In MainPage.xaml.cs –the code-behind file for the XAML UI- use the constructor to start a timer which will make the progress bar tick along providing some visual feedback to the user.

Next, wire up to a delegate in the Unity side called WindowsGateway.UnityLoaded(). WindowsGateway is explained further in the "Writing Platform Specific Code" section below,

essentially it allows you to decide when Unity is finished loading and the user should see the Unity scene.

When WindowsGateway.UnityLoaded() is fired, we set a private Boolean to say Unity is done loading. Then the progress bar timer will detect this on its next Tick event and remove the progress bar gracefully showing the game.

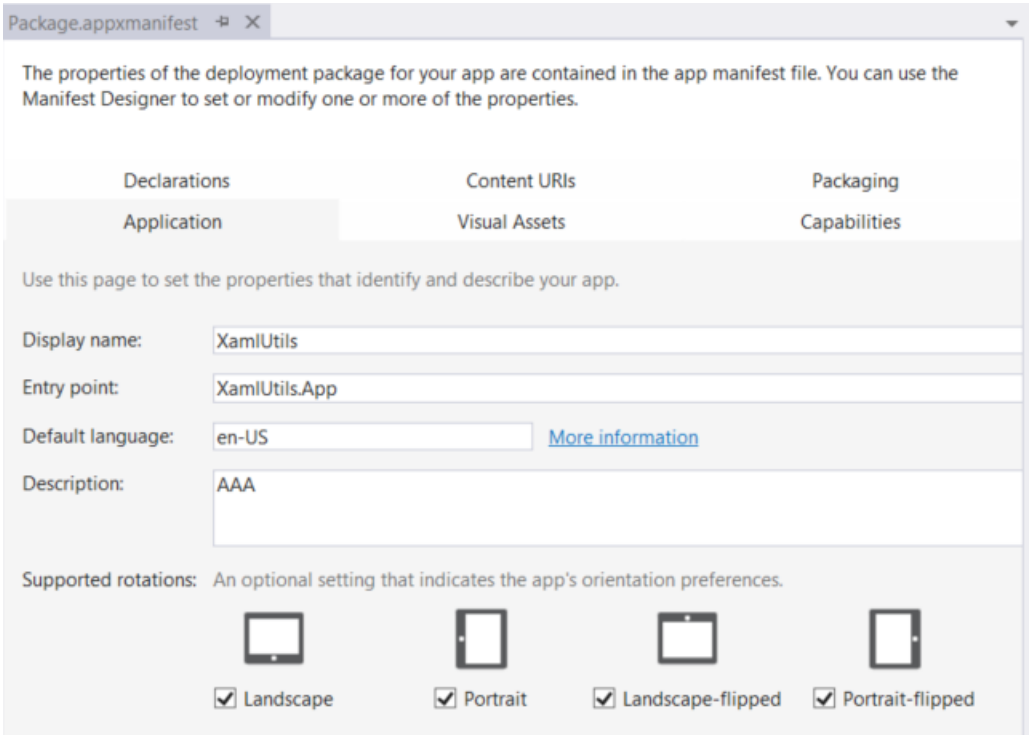
Note: There is also an event called AppCallbacks.Initialized() but this can tend to fire too early. Often you will want explicit control of the loading experience to inform the app when the game is playable.

```
        private DispatcherTimer extendedSplashTimer;
        private bool isUnityLoaded;

        public MainPage(SplashScreen splashScreen)
        {
            // ensure we listen to when unity tells us game is ready
            WindowsGateway.UnityLoaded = OnUnityLoaded;

            // create extended splash timer
            extendedSplashTimer = new DispatcherTimer();
            extendedSplashTimer.Interval = TimeSpan.FromMilliseconds(100);
            extendedSplashTimer.Tick += ExtendedSplashTimer_Tick;
            extendedSplashTimer.Start();

        }

        /// <summary>
        /// Control the extended splash experience
        /// </summary>
        async void ExtendedSplashTimer_Tick(object sender, object e)
        {
            var increment = extendedSplashTimer.Interval.TotalMilliseconds;
            if (!isUnityLoaded && SplashProgress.Value <= (SplashProgress.Maximum -
increment))
            {
                SplashProgress.Value += increment;
            }
            else
            {
                SplashProgress.Value = SplashProgress.Maximum;
                await Task.Delay(250); // force delay so user can see progress bar
maxing out very briefly
                RemoveExtendedSplash();
            }
        }

        /// <summary>
        /// Unity has loaded and the game is playable
        /// </summary>
        private async void OnUnityLoaded()
        {
            isUnityLoaded = true;
        }

        /// <summary>
        /// Remove the extended splash
        /// </summary>
        public void RemoveExtendedSplash()
        {
            if (extendedSplashTimer != null)
            {
                extendedSplashTimer.Stop();
            }
            if (DXSwapChainBackgroundPanel.Children.Count > 0)
            {
                DXSwapChainBackgroundPanel.Children.Remove(ExtendedSplashGrid);
            }
        }
```

Orientation Support

Due the predominantly widescreen aspect ratio on new devices, the Unity editor will export projects that default to landscape orientation. If your game supports portrait orientation, you will need to change the app manifest in Visual Studio –you can't do this in the Unity editor-.

To change the manifest, double click in the Package.appxmanifest file, which you can find on the root of your Windows Store Visual Studio project. In the manifest editor, just check the orientations you want to support.



The amount of work you will have to do to support landscape will depend on the game you are porting. Even if the game has been developed primarily for portrait you may find that relatively minor adjustments to the camera position within the game, along with changes to on screen menu items may get you most of the way there.

Starting with Unity 4.3, the orientation APIs (Input.deviceOrientation and screen.orientation) work well within Unity. You can use these to query the device's orientation and even set the orientation for specific scenes.

With the sample project the sample plugin provides an example of this within the WindowsPlugin.cs class. Your Unity code can then make use of a call to the OrientationChanged handler

```
        public WindowsPlugin()
        {
#if NETFX_CORE
            Dispatcher.InvokeOnUIThread(() =>
            {
                DisplayInformation.GetForCurrentView().OrientationChanged +=
WindowsPlugin_OrientationChanged;
            });
#endif
        }

        /// <summary>
        /// Will allow Unity to respond to orientation changes
        /// </summary>
        public EventHandler OrientationChanged;

        /// <summary>
        /// Allows Unity to set auto rotation preferences
        /// </summary>
        void SetOrientationPreferences ( int value )
        {
#if NETFX_CORE
            Dispatcher.InvokeOnUIThread(() =>
            {
                Windows.Graphics.Display.DisplayProperties.AutoRotationPreferences =
(Windows.Graphics.Display.DisplayOrientations)value;
            });
#endif
        }

        public void Dispose()
        {
#if NETFX_CORE
            Dispatcher.InvokeOnUIThread(() =>
            {
                DisplayInformation.GetForCurrentView().OrientationChanged -=
WindowsPlugin_OrientationChanged;
            });
#endif
        }

#if NETFX_CORE
        void WindowsPlugin_OrientationChanged(DisplayInformation sender, object args)
        {
            var eh = OrientationChanged;
            if (eh != null)
            {
                Dispatcher.InvokeOnAppThread(() =>
                {
                    eh(this, null);
                });
            }
        }
#endif
```

Writing Platform specific code

Porting your code is only part of the task when writing a game. You will also want to use platform APIs to implement tasks such as in-app purchasing, or settings, etc.

There are several different ways to call platform APIs from within your Unity scripts:

Since we are running .NET on both sides (your Unity scripts and the host process), we can have a direct connection between the Unity engine and the app. This technique is very simple but it is a little limited with regards to the types and communications you can expose.

You can create a Unity plugin which affords greater reusability and opens up the types and API you can reference. This technique requires more up front effort.

We are discussing both of these techniques by example! Within the Unity Sample project you can find a class at /Assets/Scripts/Windows/WindowsGateway.cs. This class provides communication between Windows Store and Unity. You will see both the direct approach and the plugin approach. It's created as an abstraction between the Unity game code, and all integration with Windows specific code.

Direct Communication

The direct communication approach is very simple. Within your Unity script, you can create a class that the host can reference and access.

For example, in WindowsGatweway, we expose an Action (action is a void delegate with no parameters) like this:

```
    static WindowsGateway()
    {

#if UNITY_METRO

        // unity now supports handling size changed in 4.3
        UnityEngine.WSA.Application.windowSizeChanged += WindowSizeChanged;

#endif

        // create blank implementations to avoid errors within editor
        UnityLoaded = delegate {};

    }

    /// <summary>
    /// Called from Unity when the app is responsive and ready for play
    /// </summary>
    public static Action UnityLoaded;
```

And from within our host code in Visual Studio, you can just call the code directly:

```
// ensure we listen to when unity tells us game is ready
WindowsGateway.UnityLoaded = OnUnityLoaded;

/// <summary>
/// Unity has loaded and the game is playable
/// </summary>
private async void OnUnityLoaded()
{
       /* here you can use WinRT APIs that your unity scripts did not know about and
would not have been able to reference. */
}
```

The technique is simple; and it is mostly "call back driven". Here are the lower level details you should understand for the implementation:

Use Compiler Directives

Use #if UNITY_METRO && !UNITY_EDITOR to ensure your gateway classes and any code that makes use of them is only executed in the context of a Windows Store run.

Note: You can also use UNITY_WINRT to cover both Windows Store and Windows Phone 8 or you can use UNITY_WP8 for just Windows Phone 8.

Marshall Unity calls in the Windows Store app

Always ensure that when calling into the Unity side, that you marshal back onto the App Thread. Here's an example where we are calling back into Unity after the window is resized.

```
AppCallbacks.Instance.InvokeOnAppThread(() =>
{
        // back to Unity
}, false);
```

Conversely use, the InvokeOnUIThread() method to call into the Windows Store side and switch from the Unity app thread to the Windows UI thread. For example, using the Windows Store APIs for in app purchases will require you to be on the UI thread to avoid exceptions.

```
AppCallbacks.Instance.InvokeOnUIThread(() =>
{
       // UI Thread
}, false);
```

Keep it simple and leak free

The direct connection between your app and Unity is often used for callbacks. In some instances you might need an event (multicast delegate) but often you don't and you can use a simple Func or Action to get the job done.

You do have to be very cautious about making sure there is no leaks. If your host holds on to a scene, that can leave a lot of memory behind, so whether you use Func or events, make sure you do not create a leak.
There are some techniques such as weak references that you can implement if your code is not straight forward enough to easily release a reference or unsubscribe to an event. We kept it simple in our sample to illustrate a point.

Windows Store Unity Plugins

A plugin is a binary dll (in .NET called assembly) that you can reference from within your Unity script. The principle is that a plugin would have platform specific code that you can't reference directly from your Unity scripts (since Unity does not reference the WinRT APIs directly) and the plugin can be used to encapsulate the logic into a reusable component.

Guidance on how to create Windows Store and WP8 plugins for Unity is provided here:
https://docs.unity3d.com/Documentation/Manual/windowsstore-plugins.html
http://docs.unity3d.com/Documentation/Manual/wp8-plugins-guide-csharp.html

An example plugin called "MyPlugin" has been provided as part of the Unity Sample Project solution. Here, we will walk through the structure and details on how Unity uses the plugins.

There are three plugin related projects alongside the Windows Store app

- MyPluginUnity - .Net 3.5 class library.

- MyPluginWindows - Windows 8.1 class library.
- MyPluginWP8 – Windows Phone 8 class library (for Windows Phone 8)

Each of the plugin projects outputs an assembly with the name MyPlugin.dll. Having the same name is a Unity requirement. After building, the outputs are copied automatically via a post build script to a predetermined folder structure in Unity as follows:

- /Assets/Plugins/MyPlugin.dll is generated from MyPluginUnity and will be used within the Unity Editor.
- /Assets/Plugins/Metro/MyPlugin.dll is generated from MyPluginWindows will be added as a reference to the Windows Store Visual Studio project and used at run-time.
- /Assets/Plugins/WP8 < MyPlugin.dll generated from MyPluginWP8 will be will be added as a reference to the Windows Store Visual Studio project and used at run-time.

Our sample plugin allows for a ShowShareUI(), which you can find in the /Assets/Scripts/ShareManager.cs Unity script.

Note that it works for both Windows 8 and Windows Phone 8

```
/// <summary>
/// Handles Share Integration
/// </summary>
public class ShareManager : MonoBehaviour
{
    void OnGUI()
    {
        if (GUI.Button(new Rect(Screen.width - 120, 20, 100, 20), "Share"))
        {
#if UNITY_WINRT
            MyPlugin.WindowsPlugin.ShowShareUI();
#endif
        }
    }
}
```

You can inspect the code for each plugin or run the sample and notice the implementations are different between Windows Phone and Windows 8, yet within our Unity code, it is the same call and a single code path.

The assembly generated by MyPluginUnity for the editor experience has to expose the same binary contracts (the same APIs) that your Unity scripts will compose. For the most part, the functions in the editor dll will be no-ops and do nothing, but they do have to exist for Unity to compile your scripts. The swap happens later, and without the editor dll, your project will not compile.

Marshalling calls in a Windows Store plugin

You do not have access to AppCallbacks within your plugin, so this means that we have to do some work to afford marshalling to the UI and Unity App Threads. Fortunately, that work has been done for you in the sample project.

Within the MyPluginWindows Windows Store plugin project, you will see a class called Dispatcher, which has a couple of static properties allowing us to get us onto the App and UI Threads.

```
/// <summary>
/// Handles dispatching to the UI and App Threads
/// </summary>
public static class Dispatcher
{
    // needs to be set via the app so we can invoke onto App Thread
    public static Action<Action> InvokeOnAppThread
    { get; set; }

    // needs to be set via the app so we can invoke onto UI Thread
    public static Action<Action> InvokeOnUIThread
    { get; set; }
}
```

In order for this to work inside our plugin, we need to set these properties from within our app. The best place to do this is straight after Unity has initialized in the App.xaml.cs.

Here we set the UIDispatcher and the AppDispatcher to methods which wrap around the existing marshalling methods of AppCallbacks, allowing the Unity engine to do all the work.

```
public App()
{
       this.InitializeComponent();
       appCallbacks = new AppCallbacks(false);
       appCallbacks.Initialized += appCallbacks_Initialized;
}

void appCallbacks_Initialized()
{
       MyPlugin.Dispatcher.InvokeOnAppThread = InvokeOnAppThread;
       MyPlugin.Dispatcher.InvokeOnUIThread = InvokeOnUIThread;
}

public void InvokeOnAppThread(Action callback)
{
       appCallbacks.InvokeOnAppThread(() => callback(), false);
}

public void InvokeOnUIThread(Action callback)
{
       appCallbacks.InvokeOnUIThread(() => callback(), false);
}
```

Always ensure that when calling into the Unity side, that you marshal back onto the App Thread. Here's an example where we are calling back into Unity within a plugin using the above approach

```
Dispatcher.InvokeOnAppThread(() =>
{
        // back to Unity
}, false);
```

Conversely use, the InvokeOnUIThread() to get onto Windows UI thread within our plugin. For example, raising the Facebook login window in our sample project will require you to be on the UI thread to avoid exceptions.

```
Dispatcher.InvokeOnUIThread(() =>
{
       // UI Thread
}, false);
```

Plugins or Dependency Injection?

Both are valid approaches, it's not usually a case of one over the other all of the time. You are likely to end up using both depending on the type of app/platform specific code you need to write.

Plugins

- Great for encapsulating reusable platform specific code into binaries.
- Generally for more abstracted scenarios
- They take a little bit more time to setup and maintain
- Recommended way of dealing with platform specific code with Unity

Dependency Injection

- Quicker to understand and implement
- Simply add directly to Unity scripts and app classes
- Supports two-way communication between app and Unity
- Not great re-use, leading to copy and paste approach between projects

Graphics Issues

At the moment of writing (Unity 4.3 release), there are some issues with a few of the built-in shaders in Unity. The most common issue is that fixed function shaders are not working on Feature Level 9.1. The majority of shader issues can be detected in the editor by setting the graphics emulation level. (Unity main menu -> edit -> Graphics Emulation)

When your build target is set to Windows Store App, the graphics emulation level choices are:

- DirectX 11 9.3 (shader model 3)
- DirectX 11 9.1 (shader model 2) No fixed function. This feature level is used for Surface RT first generation devices, so it is recommended that you support down to this level.

If you write your own shaders, keep in mind that semantics are required on all variables passed between shader stages.

The example below will create an error:

```
struct vertOut {
    float4 pos:SV_POSITION;
    float4 scrPos;            //ERROR! no semantic
};
```

The fix is easy:

```
struct vertOut {
    float4 pos:SV_POSITION;
    float4 scrPos: TEXCOORD0; // <-- FIX! add semantic.
};
```

The convention is to use TEXCOORD[n] for a general purpose semantic, so that is a good choice if none has been assigned.

Here are a couple other known issues with shaders:

- Fixed function shaders are not supported for shader model 2.
  Unity shaders come in three flavors: surface shaders, vertex and fragment shaders, and fixed function shaders. Surface shaders are not supported for shader model 2.
- Fog doesn't work on devices with feature level <9.3>
  You need to implement it manually. Unity has shared a sample fog shader at http://files.unity3d.com/tomas/Metro/Examples/MyCustomFog.shader. There is also a couple more fog shaders in the sample provided with this write-up. The shaders are at */UnityPorting/blob/master/Resources/ShaderIssueExamples.unitypackage*

Pausing and resuming your game

In Windows, an application window can be switched out at any time. When this happens you should ensure your game pauses appropriately. To pause the game loop, call the UnityPause method.

To detect that your game is being sent to the background and a new app is taking the foreground, use the VisibilityChanged event on the App's main window.

This is a simplified example which restarts the game when the game is brought to the foreground, in a real situation you will want to surface a resume menu to allow the user to enter back into the game gracefully.

```
Window.Current.VisibilityChanged += OnWindowVisibilityChanged;

private async void OnWindowVisibilityChanged(object sender, VisibilityChangedEventArgs
e)
        {
            if (e.Visible)
            {
                if (AppCallbacks.Instance.IsInitialized())
                    AppCallbacks.Instance.UnityPause(0);
                return;
            }
            else
            {
                if (AppCallbacks.Instance.IsInitialized())
                {
                    AppCallbacks.Instance.UnityPause(1);
                }
            }

        }
```

Window Resizing

In Windows 8, applications run in one of 3 windowing modes being Snapped (320px on the side), Filled (displayed next to a snapped app), or Full (taking up the whole screen).

In Windows 8.1, things are more flexible and your game may run with any width from a minimum of 320px or 500px which you can define in the app manifest. You can learn more about resizing here.

Most games are likely to opt for a 500px minimum width as this will afford maximum playability and it is also the required one for Windows 8.1 (320 pixels is optional).

When the user changes the width of the window, the recommended practice is to pause the game so that the user can decide what to do next.

The previous section provides guidance on you to pause and resume your game using the Unity APIs, so you can use the code, and just detect Window size changes.

Unity 4.3 now supports handling of windowing changes directly within the runtime via the UnityEngine.WSA namespace. This is demonstrated in our WindowsGateway class as follows:

```
  static WindowsGateway()
    {

#if UNITY_METRO

        // unity now supports handling size changed in 4.3
        UnityEngine.WSA.Application.windowSizeChanged += WindowSizeChanged;
#endif

    }

#if UNITY_METRO

    /// <summary>
    /// Deal with windows resizing
    /// </summary>
    public static void WindowSizeChanged(int width, int height)
    {
           // TODO deal with window resizing. e.g. if <= 500 implement pause screen
        if (width <= 500)
        {
            SnapModeManager.Instance.Show();
        }
        else
        {
            SnapModeManager.Instance.Hide();
        }
       }

#endif
```

Text Input on Windows

Windows offers two different ways to enter text input: Users can type using a physical keyboard or (when using touch) they can use the soft keyboard (also known as Soft Input Panel, or SIP). If you are coming from a mobile platform like iOS/Android, where keyboards are not prevalent, and your game might also have a custom version of a soft keyboard.

The Microsoft recommendation is that your game run well with keyboard and mouse, as well as touch, so you should take input from the keyboard. As you consider this, understand the permutations you might encounter.

Using your own custom soft keyboard will have no notion of key input, so a user would not be able to type using keyboard. You could try listening for key strokes, but then you would end up having to deal with locales, special keys, etc. it is doable but a bit of work.

A different solution would be to let Windows take care of input. The Unity GUI.TextField and GUI.TextArea APIs support keyboard input when a physical hardware keyboard is present: When the user taps on a GUI.Text* element, focus is set on that element and physical keyboard input works great. When the user does not have a physical keyboard, the expected behavior would be that the user taps on the element using touch or mouse and the soft keyboard would come up, but this is not the behavior you will see out of the box with a Unity game. Due to restrictions on Windows API, Unity does not have a way to show the soft keyboard. How can you work around this?

One option, since XAML UI seamlessly composes with your Unity UI, is to use a XAML TextBox and "overlay" it on your Unity UI. The XAML TextBox will have the right behavior on SIP & hardware keyboard and it can be styled to seamlessly integrate with your Unity UI – you can change the background color; you can control if there is a border, and you can control the font (color, size and family).

If you overlay XAML UI on top of Unity UI, you should take into account if the device is high DPI. Windows automatically scales high-DPI devices so that the touch targets are still hittable comfortably. For example, Surface 2 devices have a native resolution of 1920x1080, but this resolution is scaled, and XAML ends up reporting 1371.4x771.4 of screen real-estate, 1/1.4 scale.

At run-time, you can find out the scale the system is using by inspecting the Windows.Graphics.Display.DisplayProperties.ResolutionScale property. If the value is Scale100Percent, then it is a 1:1 ratio, at Scale140Percent, then a XAML unit is going to be 1/1.4 the Unity scale, and at Scale180Percent, it will be 1/1.8 scale.

With the sample project the sample plugin provides an example of using a TextBox overlay, see the TextBoxOverlayPlugin.cs class.

If your Unity game has a lot of text, or it just needs input all the time, or positioning the overlay is not practical, there is an alternate technique to take input; you can create a PeerAutomation element and

attach it to the Swap Chain Panel that Unity is using to draw the scene. There is a (non-Unity) working sample of this technique in this Windows SDK sample.

Debugging and Performance Analysis

You may hit problems with building your game that require you to dig a little deeper.

This section covers debugging your app, accessing the fully Unity log files and lastly on using the Unity Profiler to analyze your app's performance in real time.

Debugging your App

You can use Visual Studio can be used to debug both your app and Unity C# scripts. This can be very helpful in narrowing down issues.

- Add Unity's Assembly-CSharp project as follows to your Visual Studio solution
  File -> Add > Existing Project, browse to the Unity project folder and select Assembly-CSharp.csproj file.
- Optionally uncheck Build flag for Assembly-CSharp since it has already been built by Unity.
- Under Project > Properties > Debug, make sure the debugger type is set to Mixed (Managed and Native).

Now you are free to add breakpoint(s) to your project or Unity script file(s), and hitF5 to build, deploy, run and debug your app.

Note: If you are using source control, always remember to remove the Unity project before committing!

The Unity Log File

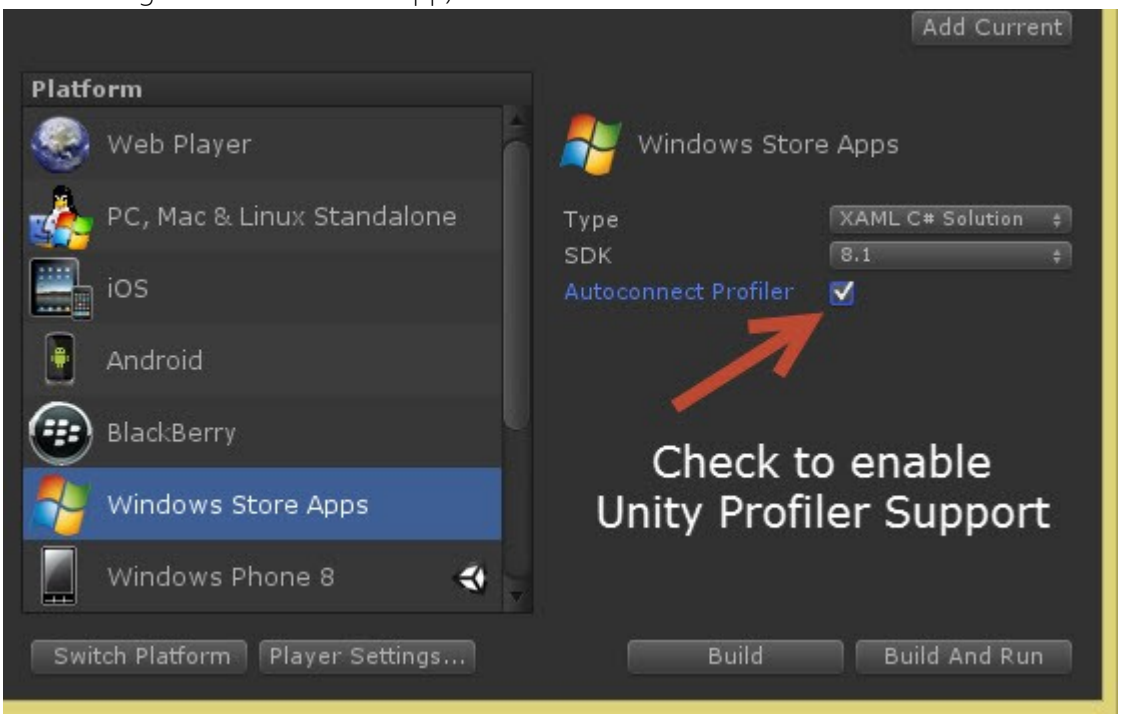
If debugging doesn't help to resolve your issue it might be useful to examine UnityPlayer.log file. It's located here <user>\AppData\Local\Packages\<productname>\TempState\UnityPlayer.log and can be found directly using Windows Explorer. Don't forget to include this file with your bug reports to Unity.

Performance Analysis

Unfortunately, you will need a second device (PC, Surface RT, Surface Pro etc.) to deploy to, as due to the way Windows Store apps work, you cannot profile an app running on your local development machine.

Here's a quick guide to getting it working with two machines:

- Configure windows firewall on both devices appropriately, in particular make sure that ports 54998 to 55511 are open in the firewall's outbound rules – these are the ports used by Unity for remote profiling.
- Ensure that you have the capabilities Private Networks (Client & Server) and Internet (Client & Server) enabled in your Windows Store App Manifest
- Check the Autoconnect Profiler checkbox when you do the Windows Store build (via File > Build Settings > Windows Store App).



- Ensure the remote debugger is running on your target machine (and yes, you can run the remote debugger on your Surface RT)
- Configure your Windows Store project to use remote debugger (via Project Properties > Debug)
- F5 from your Windows Store project in Visual Studio (run in debug or release, either works, but be consistent)
- The initial deploy of your app can take a while the first time if you have a large package over Wi-Fi, then it's incremental after further changes.

Once deployed Window > Profiler will show you the performance information flowing from your app.


Profiler
Add Profiler
Record
Deep Profile
Profile Editor
Active Profiler
Clear
Frame: 1733 / 1778
CPU Usage
Rendering
Scripts
Physics
GarbageCollector
VSync
Others
66ms (15FPS)
33ms (30FPS)
16ms (60FPS)
13.6ms
6.2ms
10.1ms
GPU Usage
Opaque
Transparent
Shadows/Depth
Deferred PrePass
Deferred Lighting
PostProcess
Other
GPU profiling is not supported by the graphics card driver. Please update to a newer version if available.
Rendering
Draw Calls
Triangles
Vertices
50
5.8k
5.4k
Memory
Total Allocated
Texture Memory
Mesh Count
Material Count
Object Count
68.9 MB
1.1
1.3k
23.9 MB
14

Feedback & Revision history

There is a lot more to cover. Check out the rest of the series and out suggested references. To let us know what missed or what you want to hear more about, drop an email to jaimer@microsoft.com.

Revision	Date	Changes	Contributors
1.0	11/15/2013	Seeding this conversation with a big brain dump. Sharing for comments.	Jaime Rodriguez (Microsoft), Keith Patton (Marker Metro), the Marker Metro team.
1.1	12/08/2013	Plugin file reference changes, renamed direct dependency to dependency injection which is more accurate	Jaime Rodriguez (Microsoft), Keith Patton (Marker Metro), the Marker Metro team.
1.2	12/20/2013	Formatting & TOC update	JC Cimetiere (Microsoft)