Building a 360 video player for VR

With the release of Unity 5.6 all of this became much easier, Unity now has a very competent media player baked in with extensions that allow you to import a 360 video and play it back on any type of headset or screen with minimal effort.

We're going to explore a couple of different examples, building for PC as a normal desktop app, PC VR and mobile VR. Please note this is a very introductory tutorial, but it'll help you with some of the basics.

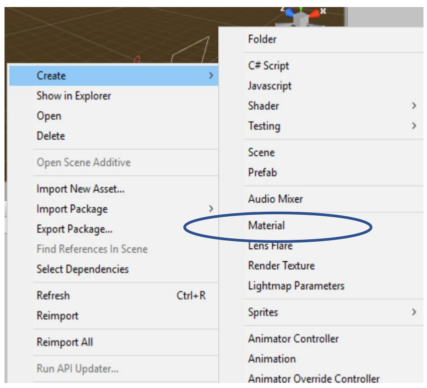
What is a '360' video

A 360 video has a lot of different forms and formats, the most commonly seen now is a movie that is filmed in a spherical manner, so we can biew 360 degrees at once. If we saw it as a completely flat video it would be twice as wide as tall – for instance the video I used is 4096 x 2048, but many videos are 3840 x 1920. We then use something called equirectangular projection to convert the image into a suitable format- think of a globe being unwrapped into a flat map, it's a similar concept.

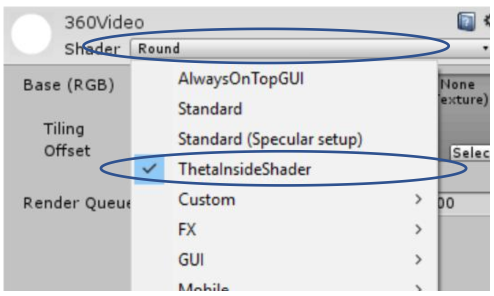
Starting off

We'll need a few things to start us off, the first being a 360 video file, you can grab these from a number of places online. I chose a simple dragster example from 360heroes.com but you can choose your own, preferably a shorter example as otherwise test build times will be longer.

Another thing you need is a sphere to map this onto, taking the flat map and "folding" it back into a globe, or the flat image back into a spherical one that you can look around. You have two choices, you can use a sphere with the normals inverted in a 3d modelling backage (flipped inside out essentially) or, we can use one of Unity's builtin sphere and use a shader to do this, which is the version I've included. The shader I've included with this tutorial is one created by Shanyuan Teng for the Ricoh Theta Community, but it works just as well here. The code is below or the shader file is on MooICT.



If you're using the below create a new shader called "ThetaInsideShader" and replace it's code with the code from the next page. Then create a new material by right clicking and select that new shader in the dropdown at the top.



```
Shader "Custom/ShaderTest" {
	Properties {
		_Color ("Color", Color) = (1,1,1,1)
		_MainTex ("Albedo (RGB)", 2D) = "white" {}
		_Glossiness ("Smoothness", Range(0,1)) = 0.5
		_Metallic ("Metallic", Range(0,1)) = 0.0
	}
	SubShader {
		Tags { "RenderType"="Opaque" }
		LOD 200

		CGPROGRAM
		#pragma surface surf Standard fullforwardshadows
		#pragma target 3.0
		sampler2D _MainTex;

		struct Input {
			float2 uv_MainTex;
		};

		half _Glossiness;
		half _Metallic;
		fixed4 _Color;
		void surf (Input IN, inout SurfaceOutputStandard o) {
			fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
			o.Albedo = c.rgb;
			o.Metallic = _Metallic;
			o.Smoothness = _Glossiness;
			o.Alpha = c.a;
		}
		ENDCG
	}
	FallBack "Diffuse"
}
```

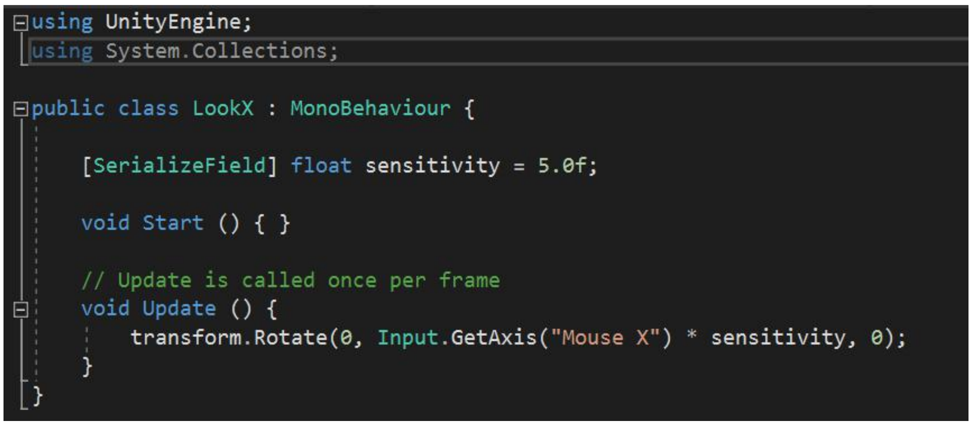
Now, on the sphere you need to do two things, one is to drag the material you've just created onto the sphere. The second is to click add component with the sphere selected, then, go to Video and add a VideoPlayer component and Audio and add an Audio Source. As you might have guessed, one deals with the video, one deals with the audio. You now need to fill in a few boxes on the components, the most obvious one is the video clip source, drag the video clip from the project view to the video player's video clip. You need to check that the Render Mode is set to material override and the Material Property is the sphere's material. This means that the video essentially gets converted frame by frame into an image which is wrapped around the sphere.

Finally, the audio source needs to be filled in, select the sphere and drag it into the Audio source field of the video player, this should auto fill the sphere's audio source into the appropriate field.

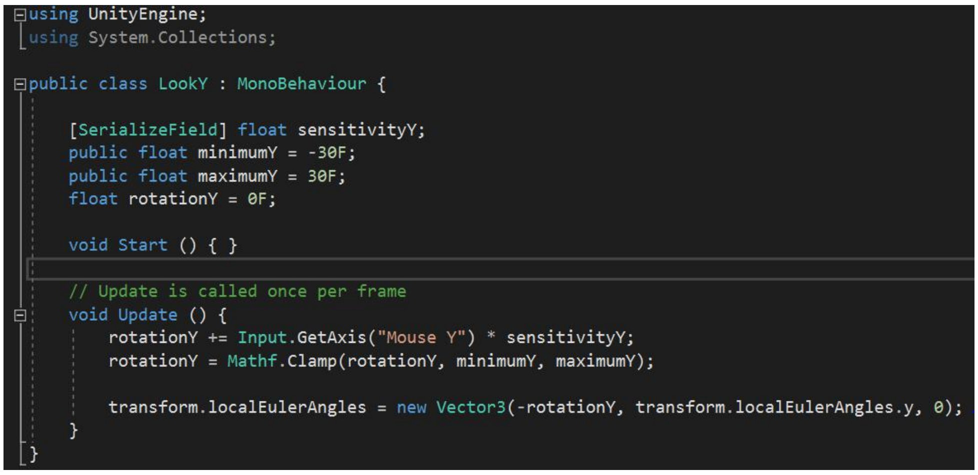
For a Basic player, that's it, almost, if you hit play you can see the video being played but can't move your view around, can't look around.

Adding in some controls

Luckily adding in controls is straightforwards, we just need to allow some movement of the camera. We can do this using a standard look script (see other tutorials for explanation of these) create two new scripts "lookX" and "lookY" (same as before, right click, create C# Script). The code for both is below. Create these and attach them to the camera, then test.



```
using UnityEngine;
using System.Collections;

public class LookX : MonoBehaviour {

    [SerializeField] float sensitivity = 5.0f;

    void Start () { }

    // Update is called once per frame
    void Update () {
        transform.Rotate(0, Input.GetAxis("Mouse X") * sensitivity, 0);
    }
}
```

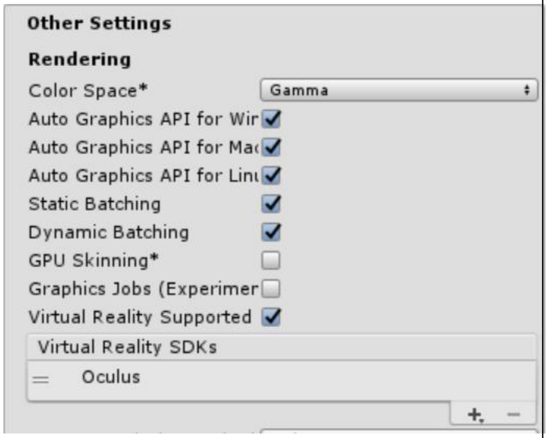


```
using UnityEngine;
using System.Collections;

public class LookY : MonoBehaviour {

    [SerializeField] float sensitivityY;
    public float minimumY = -30F;
    public float maximumY = 30F;
    float rotationY = 0F;

    void Start () { }

    // Update is called once per frame
    void Update () {
        rotationY += Input.GetAxis("Mouse Y") * sensitivityY;
        rotationY = Mathf.Clamp(rotationY, minimumY, maximumY);

        transform.localEulerAngles = new Vector3(-rotationY, transform.localEulerAngles.y, 0);
    }
}
```

What now?

Now, you should be able to look around your 360 video on the computer, but what about in VR? Well there are lots of ways of adding VR to this project, lots of platforms to look at but, thankfully, Unity again makes this fairly easy for us.

Oculus Rift and the Vive/ OpenVR



Other Settings

Rendering

Color Space*	Gamma
Auto Graphics API for Wir	✓
Auto Graphics API for Mac	✓
Auto Graphics API for Linu	✓
Static Batching	✓
Dynamic Batching	✓
GPU Skinning*	
Graphics Jobs (Experimen	
Virtual Reality Supported	✓

Virtual Reality SDKs

Oculus

I'm going to start with the rift, it's probably one of the easiest to develop and test as integration is built into Unity.

Go to Edit->Project Settings-> Player and in the setting for PC/ Mac & Linux Standalone in other settings you can choose Virtual Reality Supported. Clicking the plus lets you choose from any of the built in integrations (these will require the correct SDKs installed, Oculus Home, SteamVR, Google's android sdk, etc.). Then, you merely ensure Home is running, the headset is plugged in and press play, everything should work straight off with head tracking handled natively. SteamVR is the same (OpenVR is the integration for the VIve). You can then build this as an exe and it should work straight off with both platforms. If you have problems with Oculus, ensure in settings within home you have unknown sources allowed.

Google Cardboard and GearVR

Next up is the cardboard and GearVR, both have similar functionality and development processes, the GearVR being slightly more involved as it requires an extra step.

With regards to the cardboard (and less so the GearVR) you do need to be careful, there are still quite a few phones out there that do not support 4k video. You can, in the videos' import settings transcode the video down to a different resolution, which may be reccomended for higher quality videos. 1440p is a good medium.

Once more, this is as simple as changing the supported VR SDK in player settings, but for android you'll need to do a few more things (assuming the android sdk is setup in Unity).

In other settings, ensure the package name is updated to something reasonable, something like "com.companyName.appName" or "com.Moo.360Test". There are a few more settings you can tweak that you can look at within player settings, but that's enough for cardboard.

For GearVR you need one more thing, an osig file, this requires an Oculus developer account, the link to this is below, this guides you through creating and adding an osig. You'll need to do this for every device.

"https://developer3.oculus.com/documentation/publish/latest/concepts/publish-mobile-app-signing/"

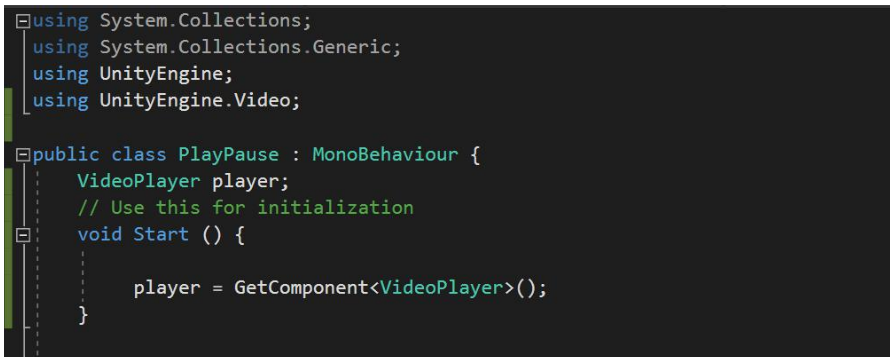
Now you can connect your device, ensure you have install apps from unknown sources activated and test away.

Adding Interactivity

At the moment the app works fine, we can look around and enjoy our video but we can't do anything, one thing you may want to do is play or pause the video, let's add that on a button press.

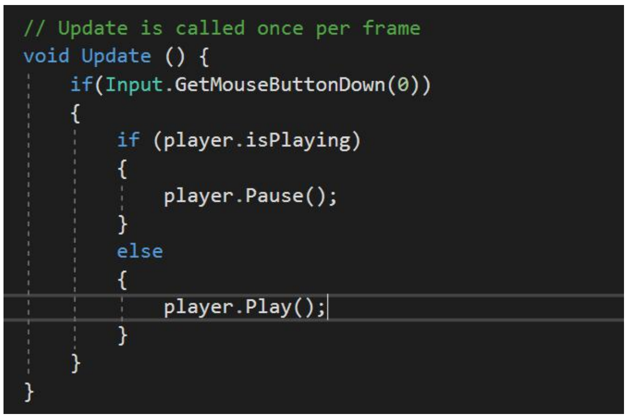
Now, every headset will have a different way of interacting, cardboard headsets have a magnetic or capacative button (usually), GearVRs have a touch panel and Oculus has a remote, controller or touch. If we had to implement controls for each of this with Unity (and we can) then even with assets to help us this would be tedious and time consuming. Instead, Unity encapsulates and abstracts a lot of this, hiding it behind the scenes, we take advantage of this by using one such feature. That is, mouse emulation, in Unity a screen tap of a phone, a tap on the GearVR and a press of the Oculus remote all are considered the same as clicking our mouse.

So, create a new script, "PlayPause" and attach it to the sphere the videoplayer is attached to. We are going to start by adding a line at the top "using UnityEngine.Video" a line underneath the class declaration "VideoPlayer player" and in start "player =GetComponent<VideoPlayer>(); The reason we do this is to 1) tell Unity we're using the videoplayer component so it can load associated scripts, 2) create a reference to the videplayer so we know which it's using 3) link the videoplayer we want to that reference by getting the videoplayer component attached to that object.



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Video;

public class PlayPause : MonoBehaviour {
    VideoPlayer player;
    // Use this for initialization
    void Start () {

        player = GetComponent<VideoPlayer>();
    }
```

Underneath this we want to actually write the code to handle the button presses, we do this in Update which runs every frame. This means we can check every frame if the button has been pressed down, here, mouse button 0 is LMB.



```
// Update is called once per frame
void Update () {
    if(Input.GetMouseButtonDown(0))
    {
        if (player.isPlaying)
        {
            player.Pause();
        }
        else
        {
            player.Play();
        }
    }
}
```

Then, we check if the player is playing something, if it is then we call pause, if it's not we call play.

Very simple but very straightforwards and will work relatively easy cross platform. Obviously this isn't ideal but it's enough for a basic introduction.