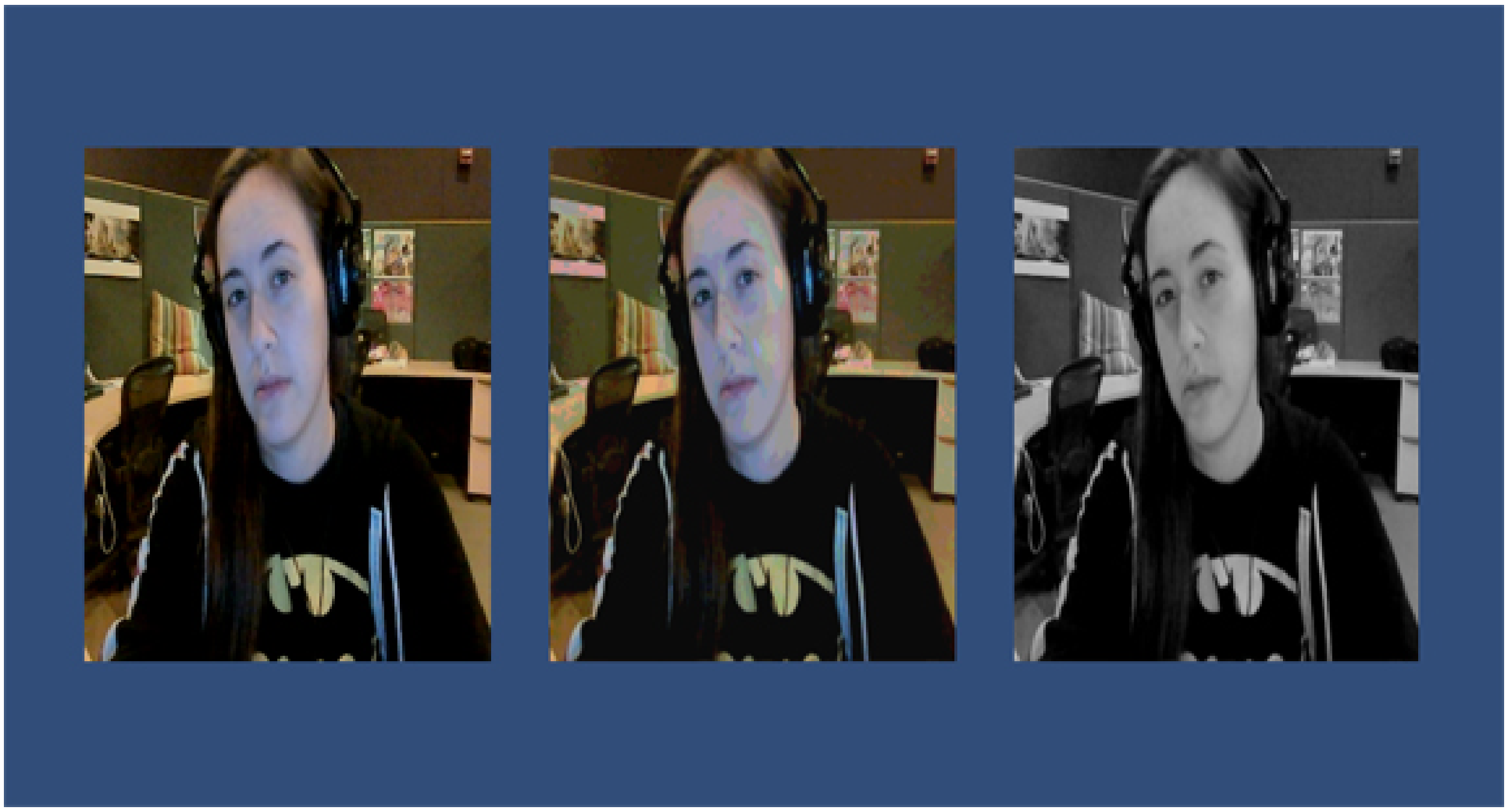
THESIS:

Real Time Webcam Capture & Basic Effects Using C# and Unity



Caitlyn Trout

TROUT CAITLYN TROUT

TECHNICAL ARTIST

GRAPHIC DESIGNER

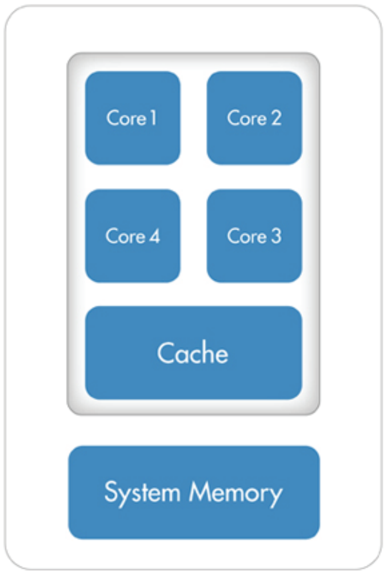
3D ARTIST

Lets take a moment to talk about the CPU versus the GPU. For me, I hear these terms get thrown around by my team of programmers a lot. For the rest of the paper it would be nice to have a basic understanding for both in the back of our minds.

The CPU, or the central processing unit, is what performs the functions of the computer's software. Its sort of the central control that tells the computer what to do as outlined by the software and the user.

The GPU, or the graphics processing unit, is a part of the rendering system of your computer that works along with you graphics card. It has parallel processes that are specialized for rendering parts of images together for faster speeds.

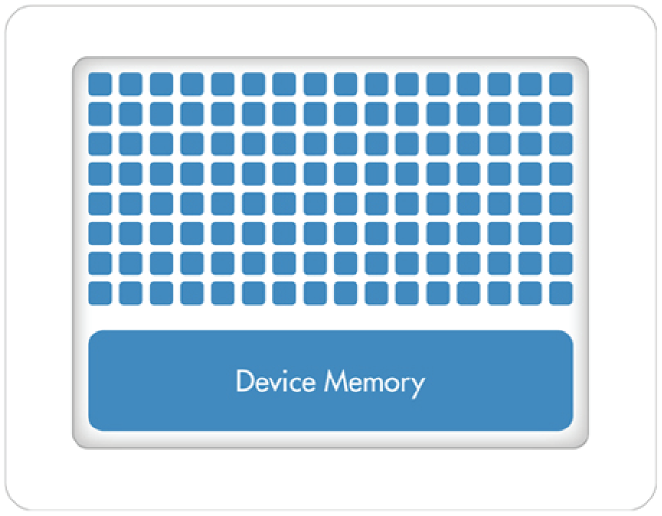
CPU (Multiple Cores)



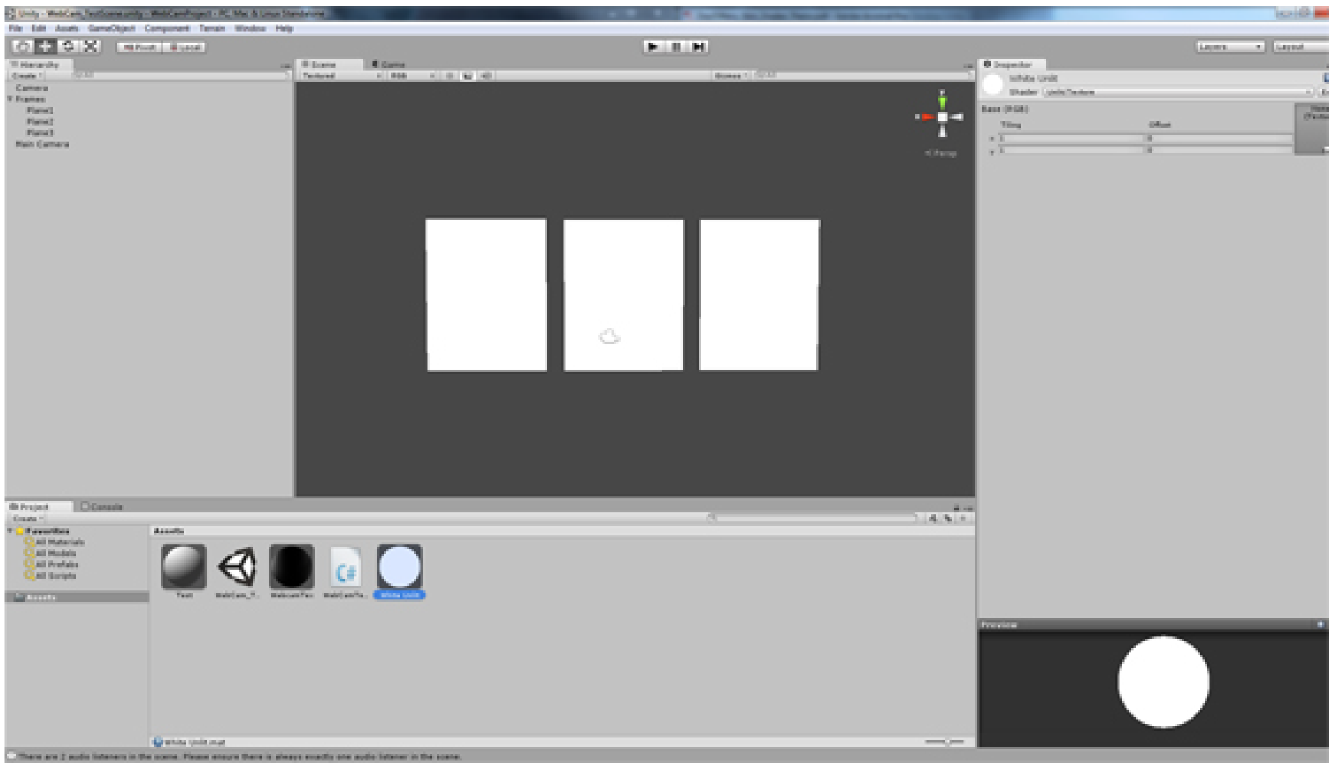
Another term to have in the background is pixel shader. What's a pixel shader? Well, its a shader that effects pixels. Simple enough right? Sort of.

Pixel shaders are graphic functions the work in tandem with vertex shaders to calculate effects on a per pixel basis. It's where we tell the GPU how to render, light, shade, and color our resolution of pixels per frame.

GPU (Hundreds of Cores)



Alright let us open up Unity and start the tutorial.



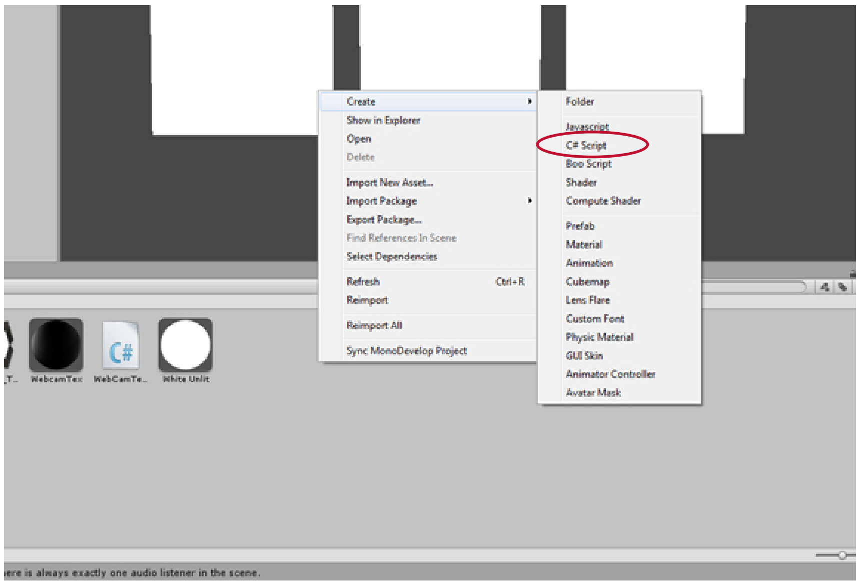
There are 2 audio listeners in the scene. Please ensure there is always exactly one audio listener in the scene.

This step is setting up a simple scene in Unity with a few planes to display our webcam footage on. Start by setting up a new project. This, like most other project setups, will give you all the necessary folders and whatnot. Name it and save it whatever and wherever you like. Just someplace that's easily accessible.

I created three planes, again, name them anything you like. We will reference them in C# using a public variable declaration which is quite nifty. I'm using a white unlit texture to display my webcam footage. You can create and just drag on top of your planes to apply the texture.

With that, right click in the project window and say : create > C# script. Name this something simple.

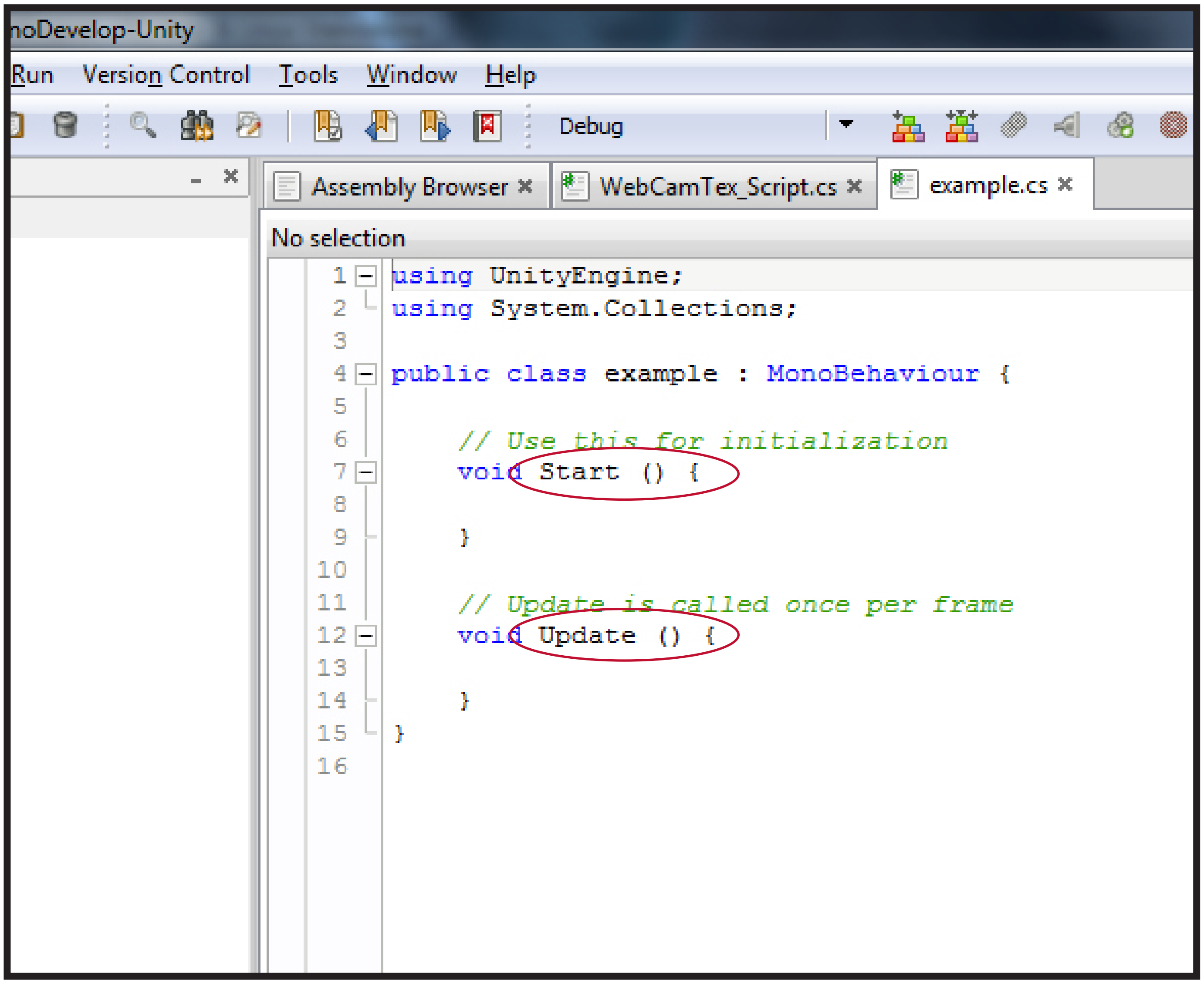
I named mine WebCamTex_Script.



ere is always exactly one audio listener in the scene.

OK, so you've created a new C# script and named it appropriately. A separate window will open if you double click . The MonoDevelop window is where we are going to be doing the bulk of our work in code.

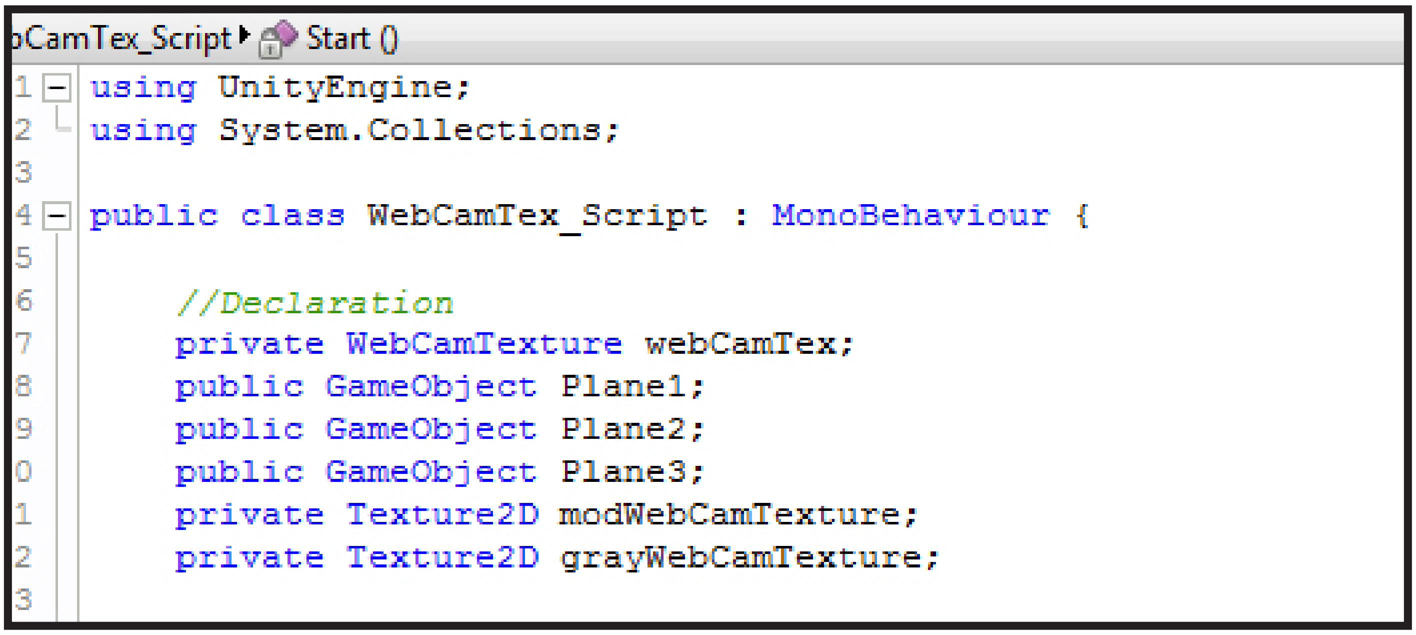
The first thing you will notice is that two functions or methods were automatically generated with the script. The first called Start() and the second called Update(). These do exactly what they say they do. Code in Start() will run at the beginning of the program (much like an __init__). Code in Update() will update every frame. This is super useful and we will be using both methods here in our webcam script.



```
using UnityEngine;
using System.Collections;

public class example : MonoBehaviour {

	// Use this for initialization
	void Start () {

	}

	// Update is called once per frame
	void Update () {

	}
}
```

They even comment them for you in green. Isn't that nice?

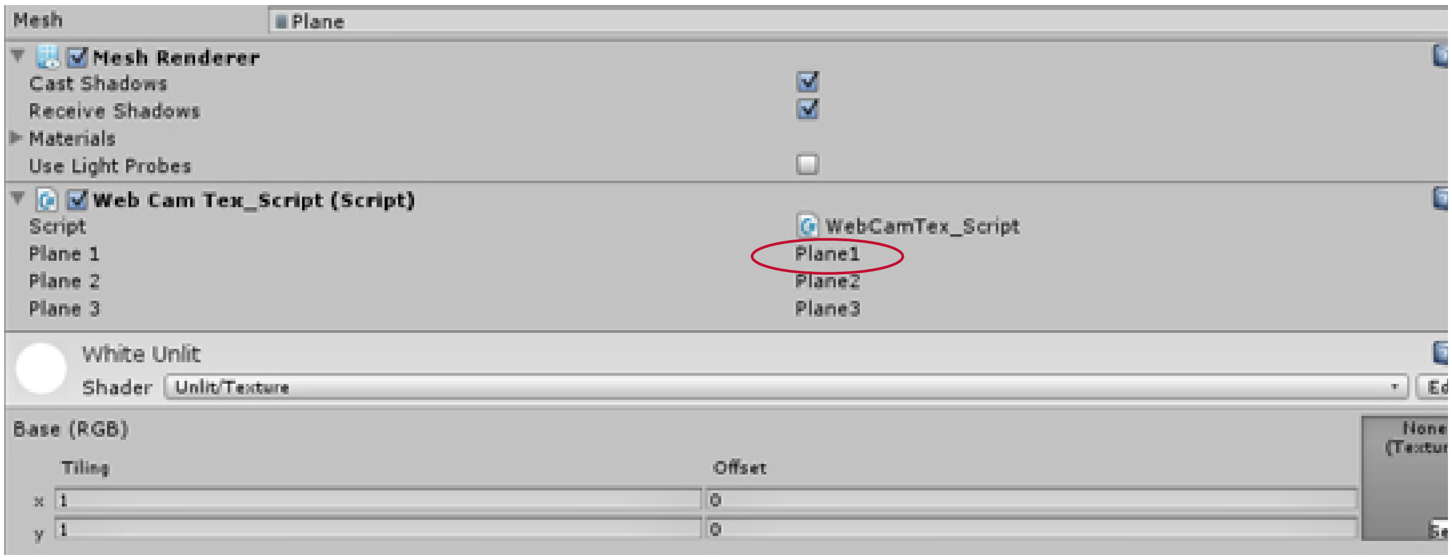
Inside your class you will want to declare needed variables and their data types for use in your script. I'm making three public game objects and calling them Plane1, Plane2 & Plane3. The other three private variables are the webcam texture itself (which you will notice has its own data type in Unity... just a little Unity magic) and two 2d textures. You can call them whatever you like . Of course, you will want the variable names to be self defining. Just good programming practice.



```
using UnityEngine;
using System.Collections;

public class WebCamTex_Script : MonoBehaviour {

    //Declaration
    private WebCamTexture webCamTex;
    public GameObject Plane1;
    public GameObject Plane2;
    public GameObject Plane3;
    private Texture2D modWebCamTexture;
    private Texture2D grayWebCamTexture;

```

Heres a nifty trick. Drag your script file on top of your first plane. Drop down the componet and you will see the public variables you created. You can now drag objects to these public variables. This gives you access to them in code and is pretty self explanitory for anyone else who might be viewing your scene in the future.

Drag game objects from your scene to this position in the scripts component dropdown.



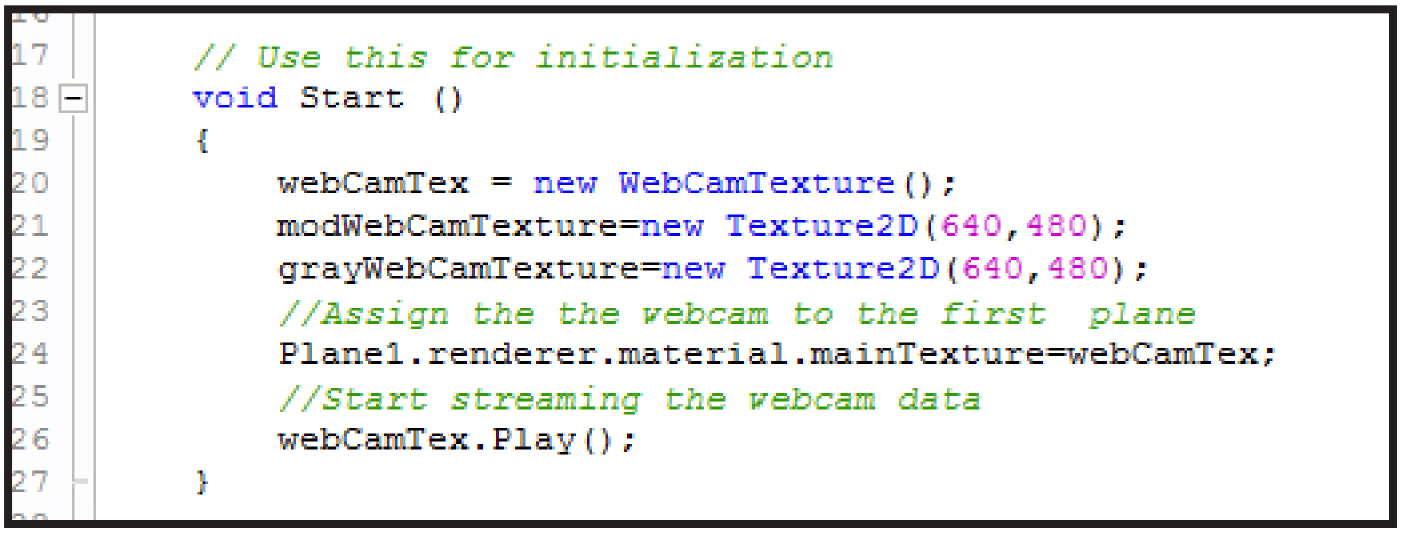
Mesh	Plane
☑ Mesh Renderer	
Cast Shadows	☑
Receive Shadows	☑
Materials	
Use Light Probes	☐
☑ Web Cam Tex_Script (Script)	
Script	WebCamTex_Script
Plane 1	Plane1
Plane 2	Plane2
Plane 3	Plane3

White Unlit
Shader Unlit/Texture

Base (RGB)

	Tiling	Offset
x	1	0
y	1	0

Accessing the webcam on your computer and hooking it up to a texture in your scene is fairly simple at this point. Initialize the webcam in your start function (along with you 2D textures which I'm feeding pixel width and height).



```
    // Use this for initialization
    void Start ()
    {
        webCamTex = new WebCamTexture();
        modWebCamTexture=new Texture2D(640,480);
        grayWebCamTexture=new Texture2D(640,480);
        //Assign the the webcam to the first  plane
        Plane1.renderer.material.mainTexture=webCamTex;
        //Start streaming the webcam data
        webCamTex.Play();
    }
```

Make sure to set the texture of your first plane to your webcam texture and call the play() argument last to start streaming. You can test your code and see if everything works by playing your unity scene.

Thus far we haven't actually done anything with the webcam data. We've only declared and instantiated everything and started streaming. Now lets write something in our update function so that our scene updates every frame.

```
for(int i=0; i<webCamTex.width; i++)
{
        for (int j=0; j<webCamTex.height; j++)
        {
        //Grayscale Effect
                Color pixel = webCamTex.GetPixel (i,j);
                grayWebCamTexture.SetPixel(i,j,new Color(pixel.grayscale, pixel.grayscale, pixel.grayscale, 1f));
        }
}

Plane3.renderer.material.mainTexture=grayWebCamTexture;
grayWebCamTexture.Apply();
```

Sorry for small font (trying to fit full line).

Ok so the short answer for this code is that it is a double for loop to write a texture from pixel data. (Amazing...its like those textures we created in the beginning had a purpose.) We are iterating through the pixels and calling a grayscale argument that will desaturate the color out. Once we have done this for all, we set the pixels for our grayWebCamTexture. Assign the material texture for Plane3 to this newly created gray masterpiece and call the Apply() argument.

Test your scene again and you will find two animated textures based on you webcam input. Pretty cool right?

Ok for the final effect in the scene, I'm going to simplify the color palate giving it a blocked out/ over exposed look. We already have the double for loop from before, lets use it again tor write another texture. Add this code in red:

```
for(int i=0; i<webCamTex.width; i++)
{
            for (int j=0; j<webCamTex.height; j++)
            {
            //Grayscale Effect
                        Color pixel = webCamTex.GetPixel (i,j);
                        float rChannel = Simplify(pixel.r);
                        float gChannel = Simplify(pixel.g);
                        float bChannel = Simplify(pixel.b);
                        modWebCamTexture.SetPixel(i,j,new Color(rChannel,gChannel,bChannel, 1f))
                        grayWebCamTexture.SetPixel(i,j,new Color(pixel.grayscale, pixel.grayscale, pixel.grayscale, 1f));
            }
}

Plane3.renderer.material.mainTexture=grayWebCamTexture;
grayWebCamTexture.Apply();

Plane2.renderer.material.mainTexture=modWebCamTexture;
modWebCamTexture.Apply();
```

Just adding this will not be enough, you'll get a pretty compile error because I haven't introduced our third and final menthod: Simplify(). In this method I'm simplifying color ranges of our webcam pixels. Its not complicated code but does have quite a bit of else if statements. The code is displayed on the next page. The method returns float x which is the rounded value of either our .r, .g or .b channel value. The alpha for our new texture is always 1 so its hard coded in.

```
static float Simplify (float x)
{
        //Switch Function
        x*=255;//put color in an easy to read ratio from 0 to 255
                if(x>=0f && x<=20f)
                {
                        x = 10f/255f;
                }
                else if(x>=21f && x<=40f)
                {
                        x = 30f/255f;
                }
                else if(x>=41f && x<=60f)
                {
                        x = 50f/255f;
                }
                else if(x>=61f && x<=80f)
                {
                        x = 70f/255f;
                }
                else if(x>=81f && x<=100f)
                {
                        x = 90f/255f;
                }
                else if(x>=101f && x<=120f)
                {
                        x = 110f/255f;
                }
                else if(x>=121f && x<=140f)
                {
                        x = 130f/255f;
                }
                else if(x>=141f && x<=160f)
                {
                        x = 150f/255f;
                }
                else if(x>=121f && x<=140f)
                {
                        x = 130f/255f;
                }
                else if(x>=141f && x<=160f)
                {
                        x = 150f/255f;
                }
                else if(x>=161f && x<=180f)
                {
                        x = 170f/255f;
                }
                else if(x>=181f && x<=200f)
                {
                        x = 150f/255f;
                }
                else if(x>=201f && x<=220f)
                {
                        x = 210f/255f;
                }
                else if(x>=221f && x<=240f)
                {
                        x = 230f/255f;
                }
                else if(x>=241f && x<=255f)
                {
                        x = 250f/255f;
                }
        return x;
}
```

So after you have created this method, you save your script and run the game scene in engine. You should see three different textures all based on the same webcam data.

THE CONCLUSION

So in the beginning I discussed the CPU vs the GPU. Heres why that is relevant. Right now, this script is running on the CPU. Which, for the most part ,is fine. Modern CPUs have more processors and are very efficient. But say I wanted to do a more complicated algorithm I would have to find a way to push that to the GPU where it is optimized to render pixel data.

The way that I can think to do this would be to write a pixel shader that communicated to the GPU my desired math. Unfortunately they work together with vertex shaders in the pipe-line (there is no vertex data here, we are already working in 2D).

But it would be possible to pass the pixel shader a position from the vertex shader and do nothing with it. And that way we could cheat our way into the CPU through shader writing.

If I was to push these real time webcam textures further I would probably follow that course of action. None the less, unity gives you access to a pretty neat feature and a possible path to more self immersion in games.

For the project used these resources:

Unity Scripting Documentation

http://docs.unity3d.com/Documentation/ScriptReference/

Nvidia Documentation

http://www.nvidia.com/object/feature_pixelshader.html

Thanks to the following people for bothering to talk to me about C#, webcamTexture and the shader pipeline:

Chris Sosa
Isaiah Walker
Matt Burton
Laura Barton
Adam Grayson