COMP1150/MMCC1011 Week 9 Prac

Topics covered:

- Cameras in 3D
- Post-processing Effects
- Render Textures

Check out your project

This week we **are going back to the COMP1150 3D pracs** project, and we will continue working in the 3D world (terrain scene) that you have been building there.

Warning: The techniques you'll be learning in the remaining pracs (and those covered since we've switched to 3D) all relate to features you can claim in the Game Design Task (GDT) assessment (now available on iLearn). However, you will need to apply those skills in your GDT assignment repository (not the 3D pracs repo) in order for us to be able to mark them as part of that assignment. In other words, the COMP1150 3D pracs repo is where you need to complete the remaining pracs and learn the needed skills, but then you need to implement the features in your GDT assignment repo. You should also be aware that simply porting over your 3D prac work is unlikely to achieve a good mark in that assessment, as the features need to not only be technically correct, but also clearly demonstrate the features in the context of your game design.

Cameras

When we make games in 3D, it is useful to think about the camera as a 3D object like any other, which can be moved and rotated in the Scene (scaling a camera has no effect).

In your terrain scene, **delete** the previously added **Player prefab**. You should notice that the Game view goes blank (if it doesn't then you must have another camera in your scene that you also need to delete). You need a camera in your scene in order to be able to see anything.

Add a new camera object from the GameObject menu **(GameObject > Camera)**. Experiment with moving and rotating it in your scene. Try the following:

- Make a camera pointing straight down at the terrain.
- Make a camera look up from underneath the terrain.
- Place a camera very close to an object in the scene
- Place a camera inside an object in the scene, looking out

One thing you will notice is that sometimes objects disappear when you look at them from certain angles or from very close up. This is caused by two things:

1) Back-face culling
2) The camera frustum

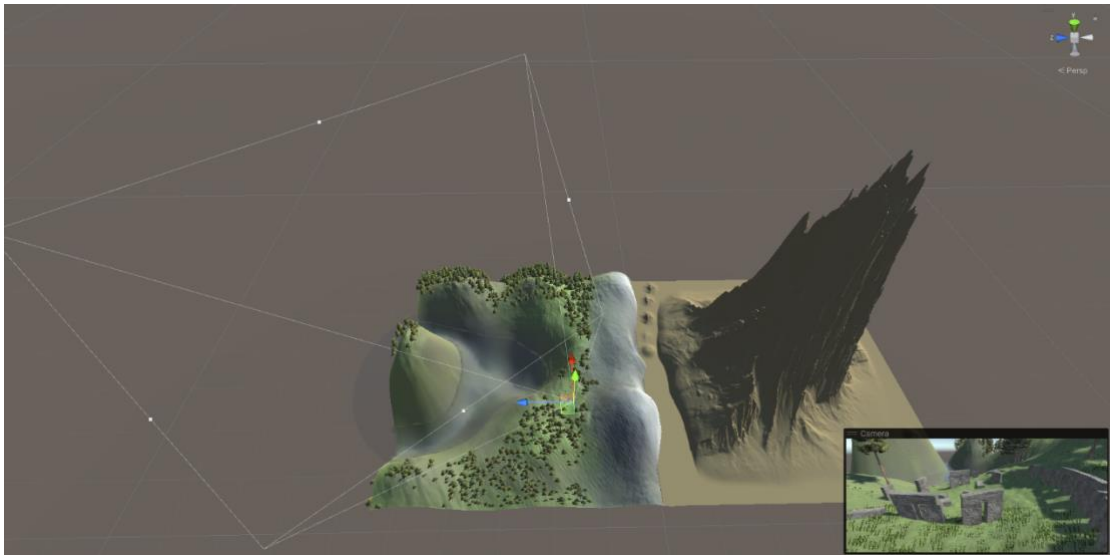
1) The surfaces of a mesh (the collection of polygons that make up the outer surface of an object) have a front and a back. Normally all the 'front' surfaces of the mesh face outwards and back surfaces cannot be seen unless the camera goes inside the object. However, surfaces like the 'Plane' object have no 'inside' and so the back side would be visible if not for 'back-face culling'. 'Back-face culling' is a standard trick to make 3D rendering faster. It simply means that the 'back' surfaces of meshes are not drawn. You may have noticed this in video games when your camera accidentally clips inside a 3D object, making it invisible. Usually this is not a problem, though it can obviously break a player's immersion in a game.

Back-face culling is deeply integrated into Unity and there is no easy way to turn it off. If you want your objects to have both an 'inside' and an 'outside' then you should build your models appropriately, to have forward-facing surfaces on both sides.

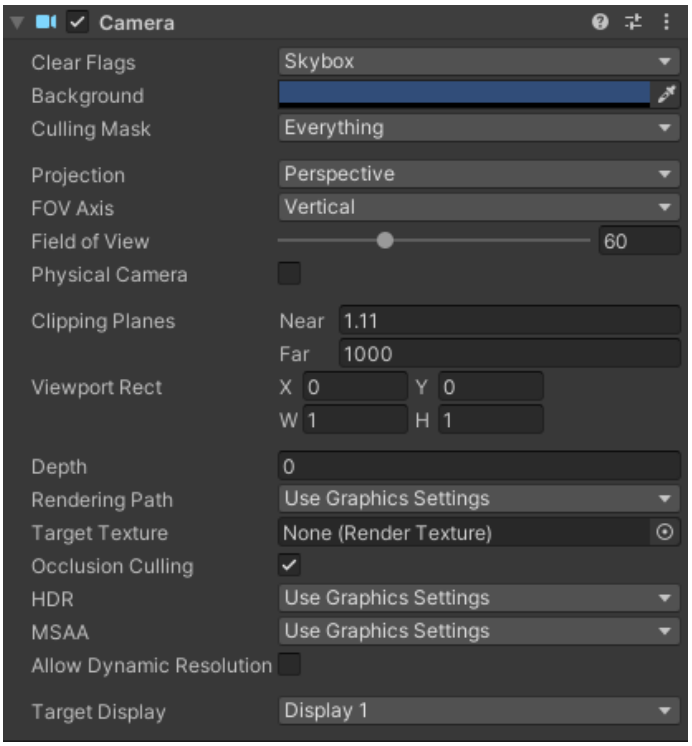
2) The camera 'frustum' is a pyramid shape represented in the Unity interface that shows all the things the camera can see. If you select the camera in the scene view it shows this pyramid with a faint white outline.

If you zoom in far enough you will notice that the box starts a little distance in front of the camera, as seen in the image above. This is the ‘near plane’. Objects closer than this distance are not rendered by the camera, and will disappear.

If you zoom right out (a long way) you will see the camera also has a ‘far plane’. This is the furthest distance the camera will render. Objects beyond this distance will not be shown.



If you examine the camera in the Inspector you will see a Camera component. Among the parameters you can change in this component are the **Near** and **Far Clipping Plane** distances. Changing these values will change the camera frustrum accordingly.



Perspective and Orthographic cameras

In Unity's 3D mode a 'perspective' camera is used by default, as seen in the 'Projection' dropdown on the Camera component. This implementation is similar to what you would normally expect from a camera in real life, such that objects appear smaller the further away they are. A perspective camera is defined by a 'Field of View' angle, commonly referred to as an FoV, and is between 0 and 180 degrees, equivalent to the zoom setting on a normal camera. Play with the Camera's **Field Of View** slider and see what effect it has on your game view and on the camera's frustum.

Before moving on, there are two film terms we that also need to be defined, a **dolly** effect, which is moving the camera closer to an object, and a **zoom** effect, which is changing the camera's field of view.

To understand the difference between the two, implement them yourself now:

1. **Make an animation** that dollies the camera towards an object by changing its Z position.
2. **Make an animation** that zooms the camera towards an object by changing its Field of View (i.e. **Camera** component **> Field of View**).

A very unsettling effect (called a dolly-zoom or 'zolly') can be created by combining the two.

3. **Create an animation** which simultaneously moves the camera closer to an object while widening the Field of View. What does this look like?

The main alternative to a perspective camera is an 'Orthographic' camera. An orthographic camera does not change the apparent size of objects as they get further away. Everything is drawn at the same size. You can see this by changing your camera's **Projection** to **Orthographic** (you may wish to do this on another camera in your scene).

The frustum of an orthographic camera has parallel sides, so rather than having an angular field of view, it has a **Size** value which determines the width of the frustum (for technical reasons the size is half the width). Since the frustrum is so narrow, you may need to change the size to get a decent view of your scene.

Perspective cameras are usually used in 3D games when we want to show a particular character's point of view (e.g. a first-person or third person games). Orthographic cameras are more commonly used for a 'god's eye' view (e.g. in most RTS or Sim games; Lost Ark and Diablo are also good examples). 2D games also use the orthographic camera by default.

The Skybox

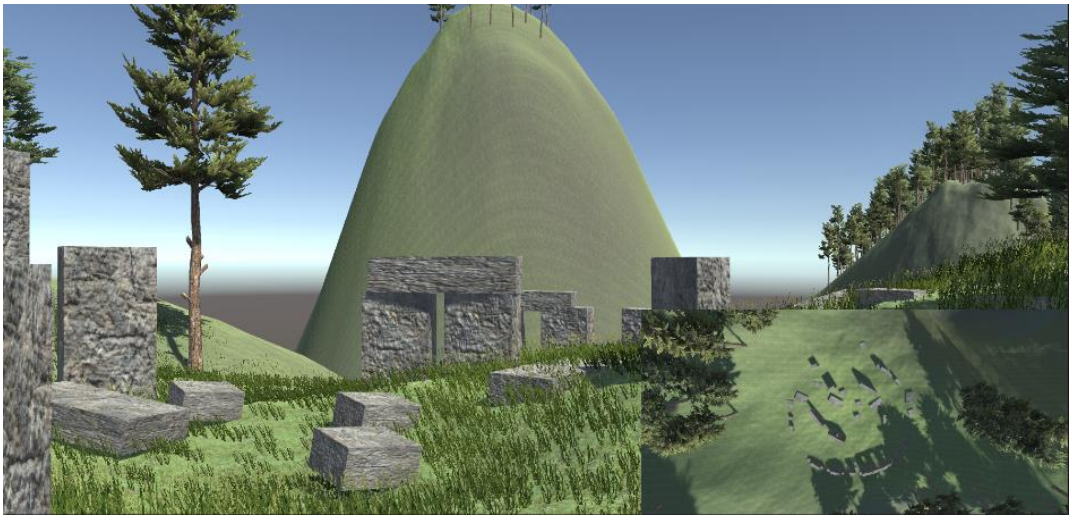
Unity automatically adds skyboxes to your game, which are also part of camera component and by default provide a gradated background shade to the entire scene. This is a great stand-in for the sky if you happen to be making an outdoor scene, but is a nuisance if you are making any other kind of scene. You can turn this feature off by changing the **Clear Flags** setting in the Camera component to **Solid Colour** and then selecting a **Background** colour in the box below this.

Multiple cameras - Creating a minimap

You can add multiple cameras to your scene and overlay them in different ways. This can be used for a variety of tricks. We won't go into them all in detail, but one trick is worth showing, creating a minimap.

1. Change the **Depth** of all cameras remaining in your scene to -1.
2. Drag a copy of your **Player** prefab back into the scene.
3. Create a **new camera** in your scene (or you can use the one that was pointing down at your scene from earlier). Rename it "Minimap camera"
4. Make it **Orthographic** and point it straight down on the scene.
5. Set the Minimap camera's **Depth** to **1**. This means it will draw on top of the Player camera (which has Depth 0).
6. Set the **Viewport Rect** to **X = 0.6, Y = 0, W = 0.4, H = 0.4**. This means that it will only be shown in the bottom-right corner of the screen.

You should end up with something like this, in your Game view:



Experiment with different settings for the **Viewport Rect** to make it bigger or smaller, and place it in other corners of the game window.

This is still a rather crude mini-map, but it is a starting point for development, and it shows what you can do with multiple cameras in your scene.

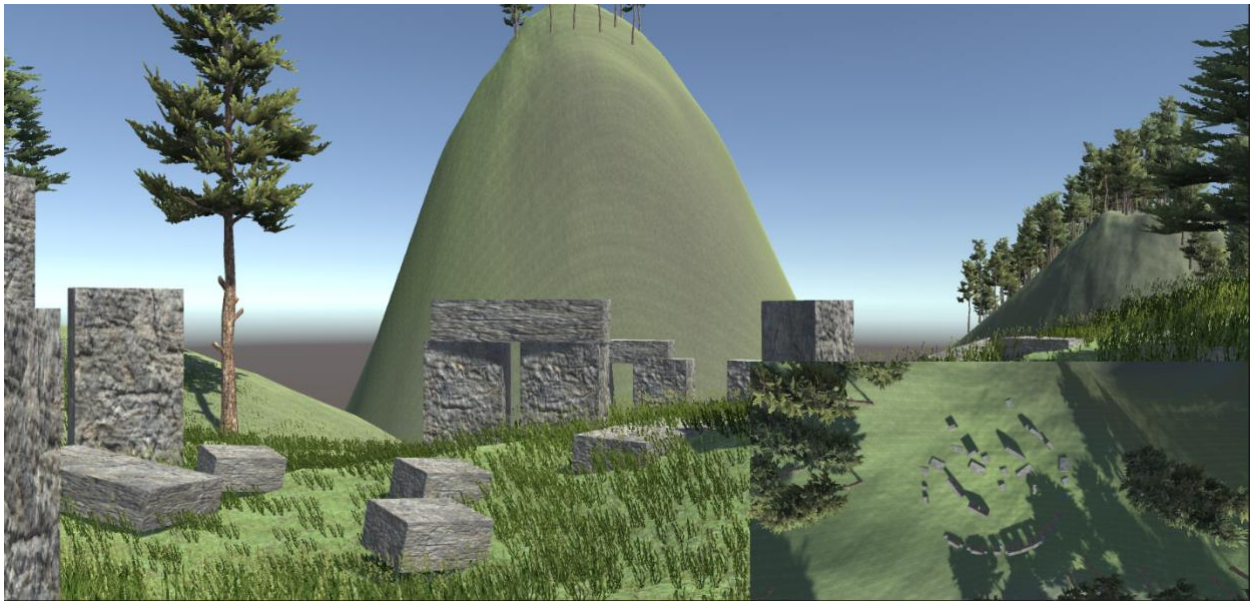
The Culling Mask

Sometimes you want to prevent some objects from being shown in a particular camera view. For instance, you may want an item such as a treasure chest to appear in your main camera view but not in your mini-map view. You can do this by using layers and a culling mask.

Do the following:

1. Create a treasure chest in the centre of your minimap view using a primitive cube **(GameObject > 3D Object > Cube**).
2. Scale it so that it can be clearly seen in your 'minimap camera'. You may also like to give it a new material with a colour that stands out in the scene, like bright pink or yellow.
3. Create a new **Layer** called "Treasure" and assign the treasure chest to this layer (see page 11 of Prac 3 if you forget how to do this).
4. In the **Camera** component attached to 'minimap camera', uncheck the **Treasure** layer within **Culling Mask**.

The treasure chest should now be visible to the main/**Player** camera, but not in the minimap.

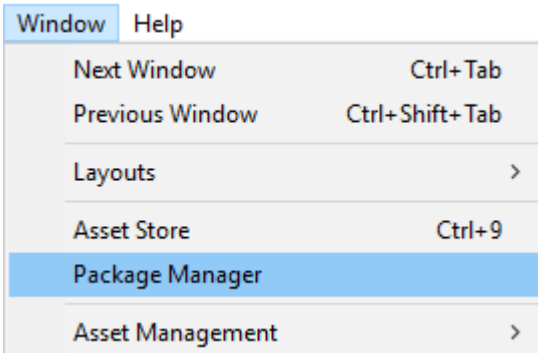


Reminder: Save, Commit and Push to GitHub

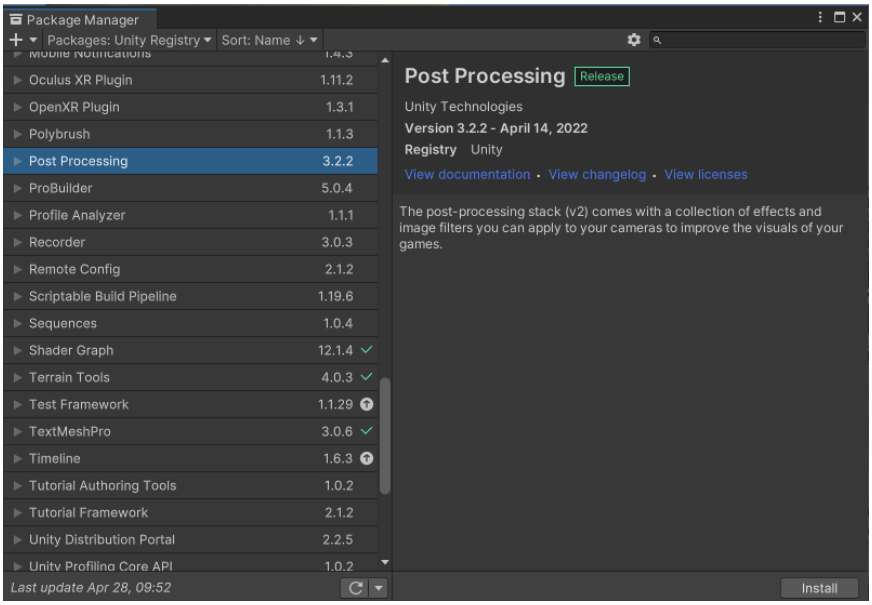
Post-processing Effects

Once a scene has been rendered, we can apply a number of different post-processing effects to adjust the way it looks. These can be used to imitate several camera tricks from film, or other kinds of effects, making your game much more visually pleasing, but they can also be very expensive in terms performance. These kinds of effects are best used sparingly, especially on mobile devices. Some of them will simply not work on browser or mobile games.

To use post-processing effects in Unity you need to get the Post Processing Stack from the Package Manager. Go to **Window > Package Manager**.

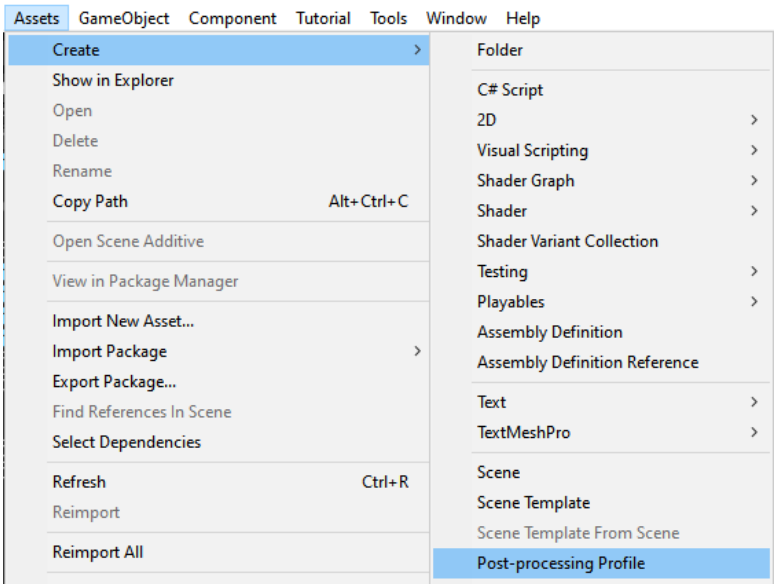


With the Package Manager open, select the **Packages** dropdown and tick the **Unity Registry** option**, then scroll down** and select **Post Processing** from the list, and press the **Install** button in the lower right-hand corner.



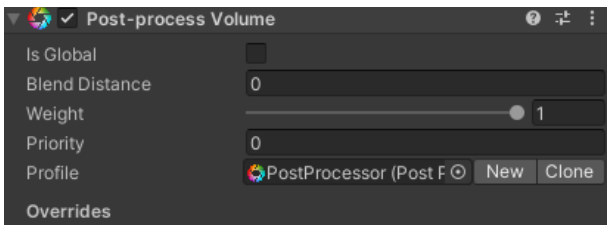
Before you can get started, you first need to create a 'Post-Processing Profile'. Select **Assets > Create > Post-processing Profile** to create a new profile. You can then find this file within your

Assets folder, in the Project window. This ‘Post-processing Profile’ asset will save the look of your post-processing effects.



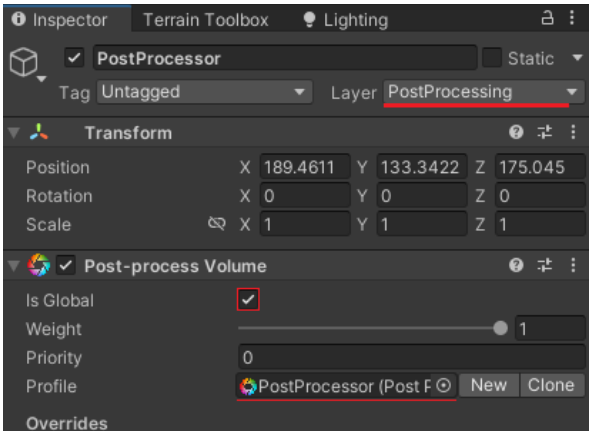
After we’ve created our profile, we need to create an object that will host this profile, and control how the post-processing effects are triggered. This object is known as a **Post-Processing Volume**, using a component with the same name**.** To create one, first create an empty GameObject **(GameObject > Create Empty).** Give this object a meaningful name, something like “PostProcessor”. Add a ‘Post Processing Volume’ to your empty GameObject by selecting **Add Component > Rendering > Post-process Volume.**

Add your ‘Post-processing Profile’ asset to the **Profile** field by clicking the object picker button and selecting it from the menu, or simply drag it from the Project view into the field.



Broadly, ‘Volumes’ can be used in two major ways: as **Local** or **Global.** A ‘local volume’ will trigger when a specified object collides with the volume’s trigger. This is useful for having the view alter when the player enters a certain space (e.g. tinting the screen blue, and slightly blurring the camera when underwater). For today’s class, however, we just want to alter the view for the whole game, so we are going to set this ‘Volume’ as a **Global** volume by ticking **Is Global.** We encourage you to experiment with **Local** Post Processing Volumes in your own time and for the Game Design Task assessment (where both can be implemented in your scene to claim extra marks).

Before we move on, we also need to set our ‘PostProcessor’ (the object containing the Post-process Volume) to the **PostProcessing** Layer. It is good practice to keep our effects on their own layers for both organisational and performance reasons. **Add a new layer** called **“**PostProcessing” and set the layer of the PostProcessor game object to it.



While you now have a ‘Post-process Volume’ that is set up, you still need to have an object to enable it during play. We do this by adding a **‘**Post-process Layer’ to the relevant camera.

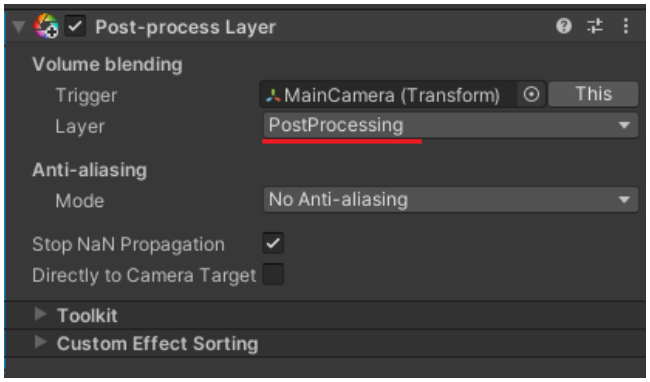
Note: You can quickly filter by specific types of objects within your Hierarchy. A quick way to do this is to click in the little search box at the top of your Hierarchy view and type the component name, in our case, “Camera”. This will find all objects with “Camera” in their name, as well as any objects that have a Camera component.

You can also search by object type by either clicking the magnifying class to the left of the search bar and selecting “Type” or “All”, or by adding “t:” as a prefix (e.g. “t: Camera” to only show objects with Camera components in our scene).

Select the Player camera (**Player > Main Camera**), and in the Inspector view click **Add Component > Rendering > Post-process Layer**.

The ‘Post-process Layer’ is used to turn post processing effects on and off. For instance, we may want to have a ‘Post Process Volume’ triggered when the player enters water, turning the screen blue. Or we may want to trigger a volume when the player enters a cave, darkening our view. In some cases, we may want our Post Process Volume to be triggered by objects other than a camera (e.g. in a top-down game, me way want the player’s avatar to trigger the volume instead).

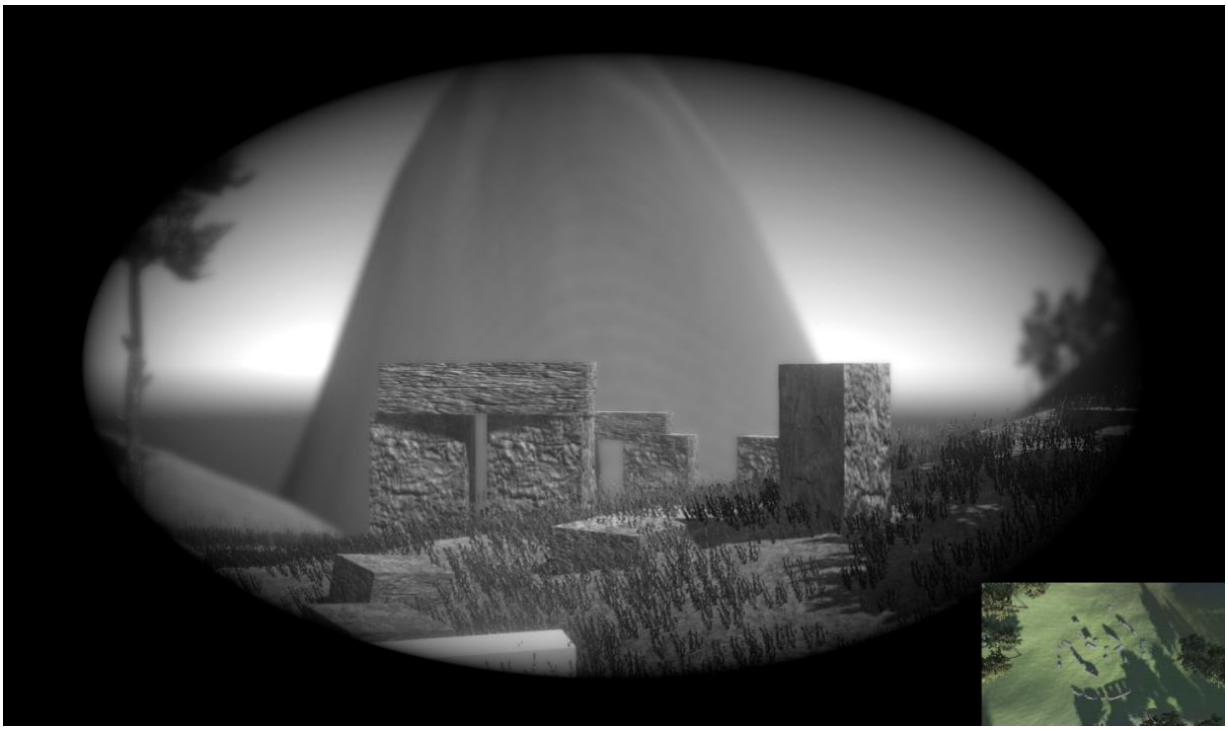
To control the object that turns post-processing on or off, we can change the **Trigger** object on the **Post-process Layer** you’ve just created. By default, the trigger will be set to the same object the component was added to; in this case the **MainCamera** on the **Player** prefab, which is what we want. The ‘Layer’ setting just below this dictates which layers our ‘Post-process Volume’ responds to. Change the **Layer** to **PostProcessing** here, so that it is set to the same layer that our ‘Post-Process Volume’ is outputting to.



With the post-processing volume set up, it is now time to experiment with the 'Post Processing Profile'. Re-select the **Post-processing Profile** asset in your Project panel. From here, you can press **Add effect...** in the Inspector panel to add effects to the profile.

There are several different effects available here. You can add them, activate/deactivate them, and edit them to your liking. Clicking on an effect will give you a drop-down box with the relevant parameters. We're not going to go through the different effects here, but you can find a list of effects and how they work in the Unity manual.

Play around with the effects and see what you can achieve. Try combining **Bloom**, **Vignetting**, **Colour Grading** and **Depth of Field** to create a black-and-white movie effect.

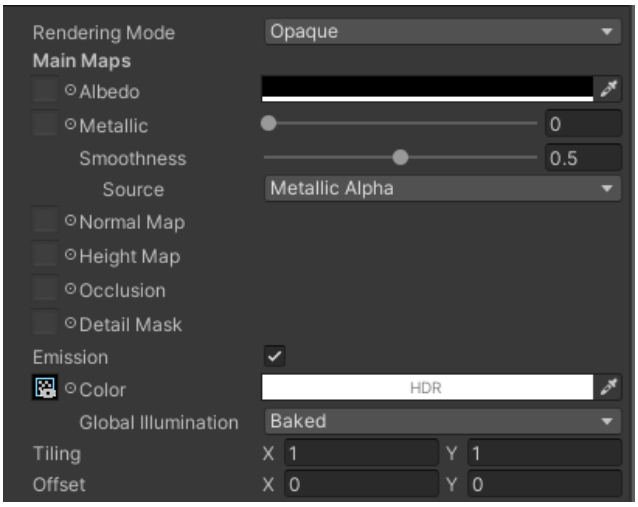


Render Textures

Another fun thing you can do with cameras is to set a camera to render to a texture rather than to the screen. This means that you can take the output from a camera and stream it onto an object in your world, which is a practical way to implement effects like security cameras, a jumbotron or even a magic mirror in your scene. To achieve a Render Texture asset needs to be created and added to the Target Texture slot on the Camera component. The camera will then draw onto the texture rather than display on the screen.

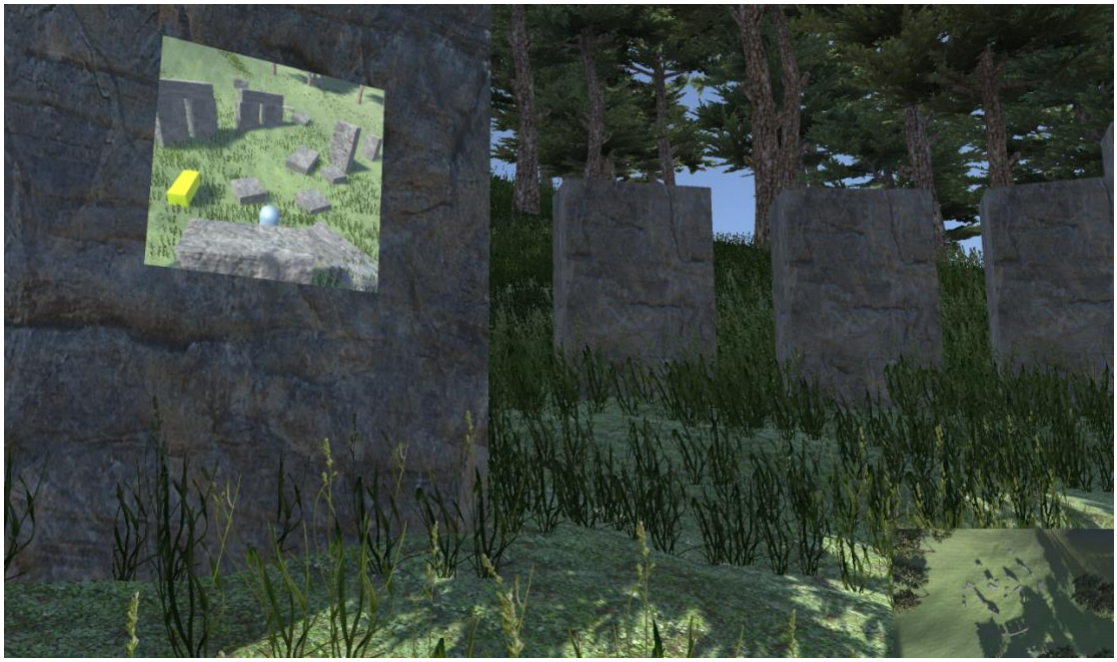
To do this in your scene:

1. Create a Quad (**GameObject > 3D Object > Quad**) and position it so that it is attached to one of your standing stones, or another surface in your level.
2. Create a Render Texture (**Assets > Create > Render Texture**).
3. Create a new Material (**Assets > Create > Material**).
4. Tick the **Emission** checkbox within the material, and **Add the render texture** into the colour slot underneath Emission by clicking the object picker button and selecting it.
5. Inside this material, make your **Emission Albedo** colour **black** or a dark colour. This increases the contrast and prevents it looking washed out.



6. Apply the material to the quad.
7. Create another Camera (**GameObject > Camera**) and position it so that it's capturing something dynamic (e.g. some trees affected by a wind zone), or just select one of your Dolly/Zolly cameras.
8. In the Camera component for your chosen camera, drag the new **RenderTexture** into the **Target Texture** slot.

Now the camera will output (render) to the **RenderTexture** you created, which will in turn be displayed on the Quad, streaming the images like a virtual screen.



Reminder: Save, Commit and Push to GitHub

Show your demonstrator

To demonstrate your understanding of this week's content to your prac demonstrator you might show:

- How to change the frustum of a Perspective and/or Orthographic camera.
- The animations you created to dolly and zoom your camera.
- Your use of multiple cameras to create a minimap.
- Implementation of the Culling Mask to hide objects.
- Global post-processing effects applied to a camera.
- Local post-processing effects applied to a camera (e.g. underwater).
- The rendering of a camera to a texture in your scene.