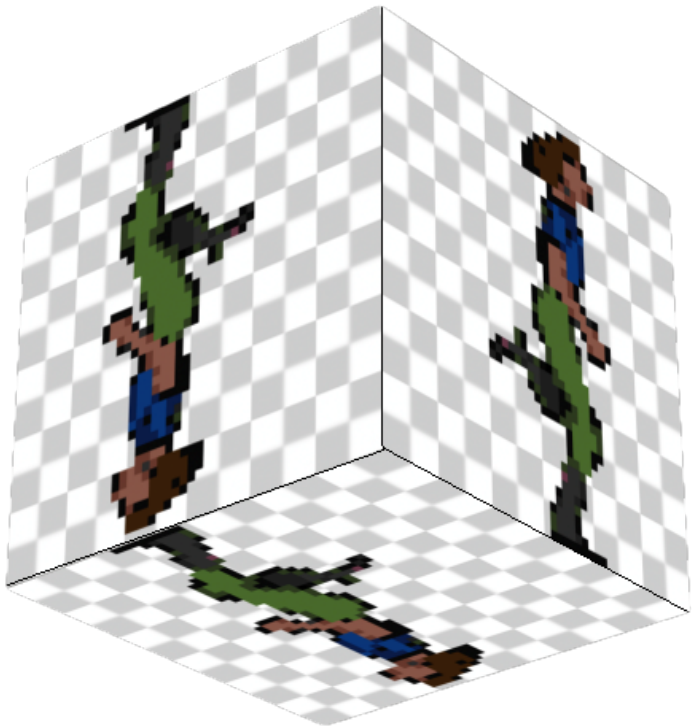
Lab - Graphics



Goals

- Quick look at Pixelmator - one of possible tools for creating/generating art assets
- Introduction to different rendering pipelines in Unity
- Playing with multiple cameras in Unity
- Using materials and shaders in Unity
- Introduction to custom shaders made in Unity's Shader Graph
- Introduction to post-processing volumes

Resources

- testpack.png (testpack.png).
- Graphics (https://docs.unity3d.com/Manual/Graphics.html) section in the Unity Manual

- Video tutorial (https://www.youtube.com/watch?v=84A1FcQt9v4) describing every node of the shadergraph
- Tutorials from Unity (https://learn.unity.com/search/?k=%5B%22tag%3A5d351f0b7fbf7d006af48182%22%2C%22lang%3Aen%22%2C%22tag%3A58088f5f0909150077ca2bc6%22%5D)

Basic challenge

The basic challenge is learning how work with different rendering pipelines and how to create some neat effects through the use of materials and custom made shaders.

Creating assets

The main focus of this tutorial is not the creation of image assets. However, at some point, you will need to produce some original artwork for the game, and this means (at least) using an image manipulating program to create sprite sheets. On the lab machines we provide Pixelmator, which (admiteddly) is not a tool dedicated to sprite sheets, and (unfortunatelly) not the most flexible image manipulating software for sprite sheet creation. Still, it can do pretty much anything that's needed for creating 2D artwork, though not always in the easiest way. And so, we'll devote some time here to a quick demonstration of the type of things you can do with Pixelmator. Other options available - drawing tablets and Inskape are briefly mentioned as well. **If you're already skilled at working with images (even if it's not in Pixelmator) you may choose to skip over to the next section and proceed with learning about Materials/Textures/Shaders in Unity (starting with step 30).**

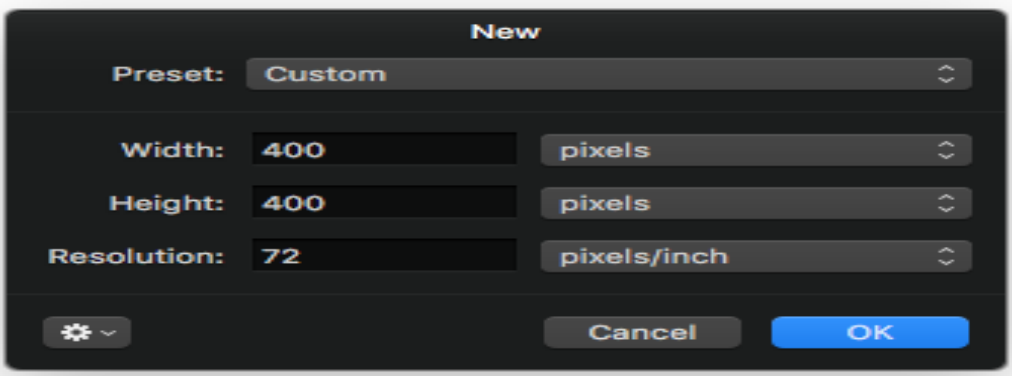
Pixelmator

In this part of tutorial you'll learn how to "bake-in" some of the graphics effect into a sprite sheet. The latter part will teach you how to use the game engine to create similar effects.

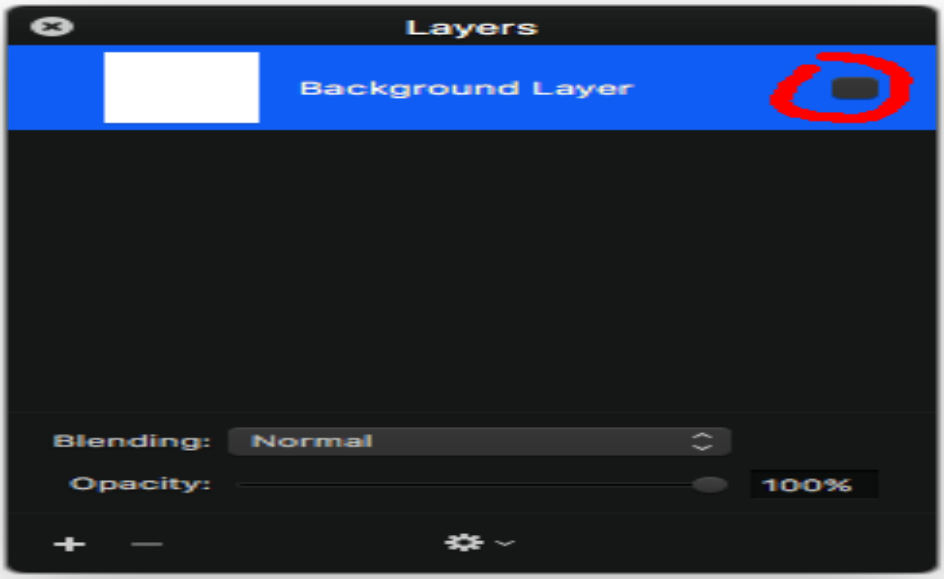
. Open Pixelmator (the  icon in the Launchpad).

. Download testpack.png (testpack.png) and Open it in Pixelmator.

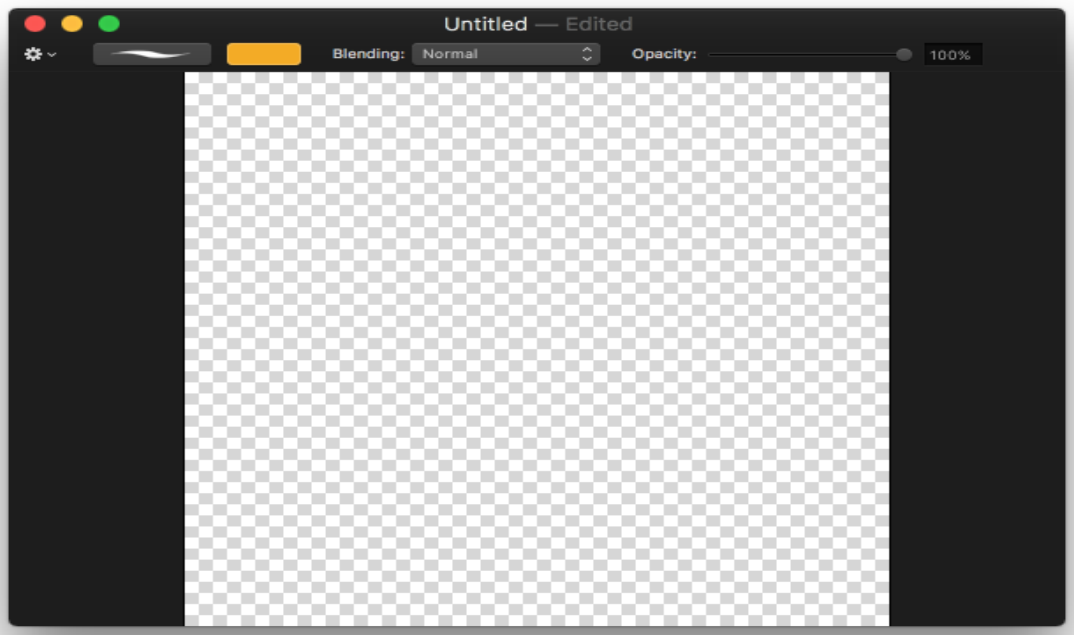
. Rather than working on this image, you'll create a new one to work on. While in Pixelmator, from the main menu select *File / New*. Set the size of the new image to at least 400x400 pixels. The exact size doesn't matter at this stage; you just need enough room to work with. Press "OK".



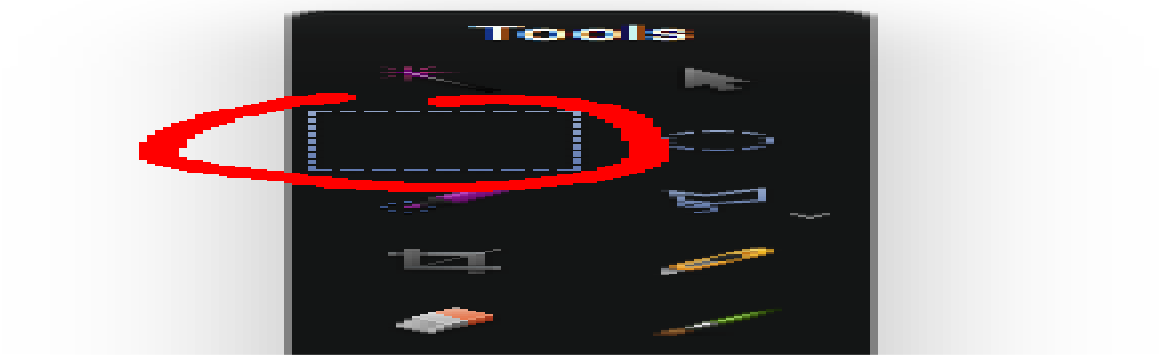
. Find the "Layers" window and make the "Background Layer" invisible by removing the check-mark beside it.



. You should see a new image with a checkerboard pattern. This pattern sits behind your image and it means that the entire image is now transparent.

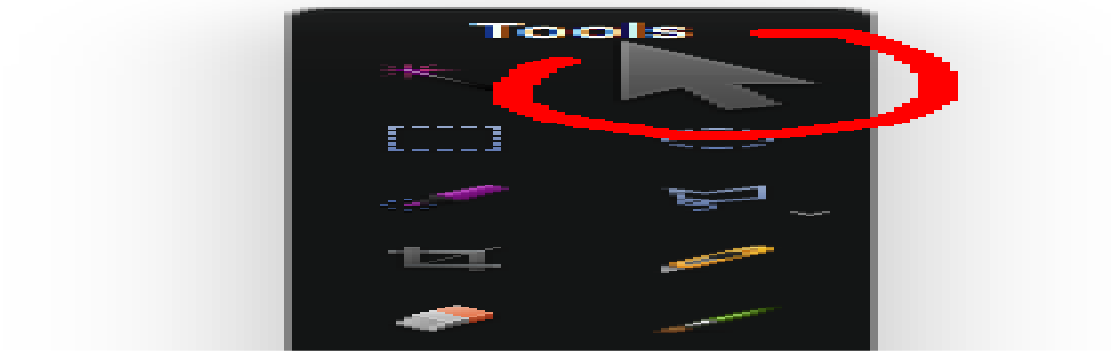


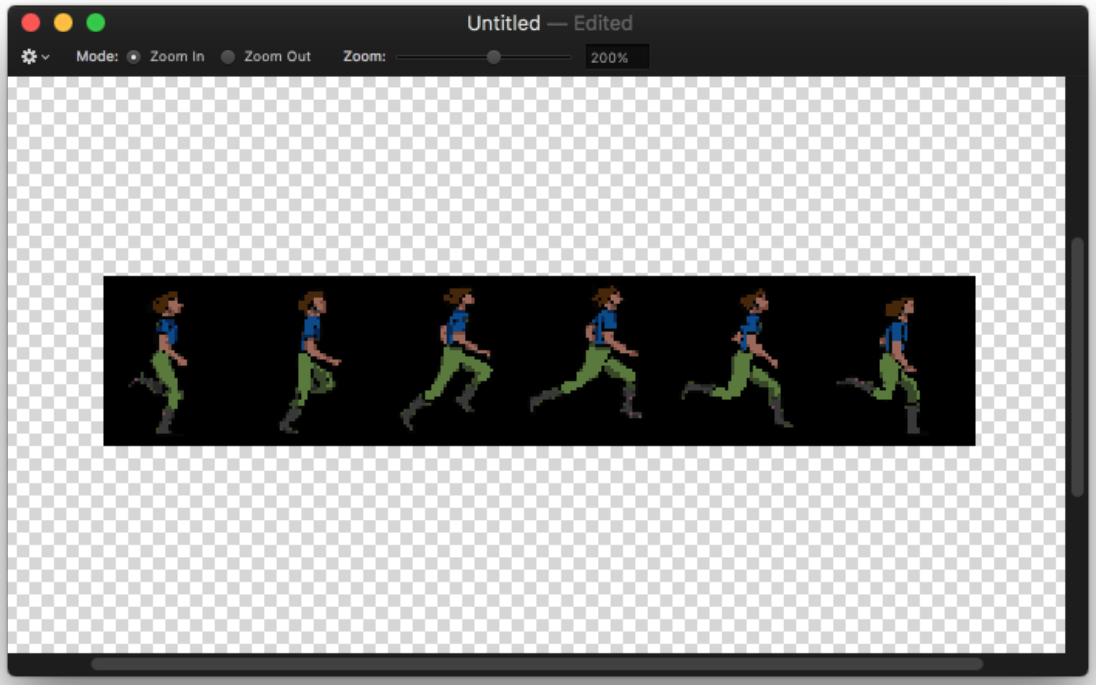
. Go back to the "testpack.png" image (it should still be open in Pixelmator). Find the "Tools" window and select the *Rectangular Marquee Tool*.



. Select the top line of runners sprites from "testpack.png". Copy these to clipboard (⌘command-C), switch to the new image, and paste (⌘command-V).

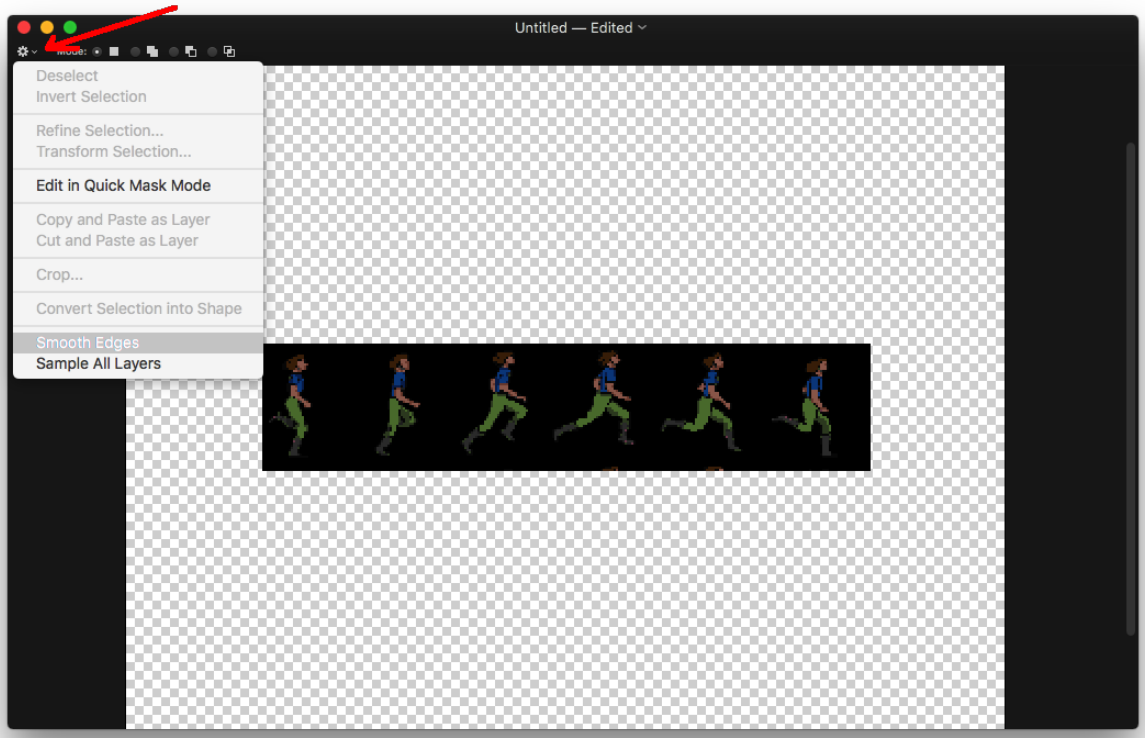
. You can use the *Move Tool* (top-right of the toolbar) and move the runners into the middle of the image. You can use the *Zoom Tool* (bottom-left of the toolbar) to zoom in or zoom out. Zoom in a bit to better see what you're doing.



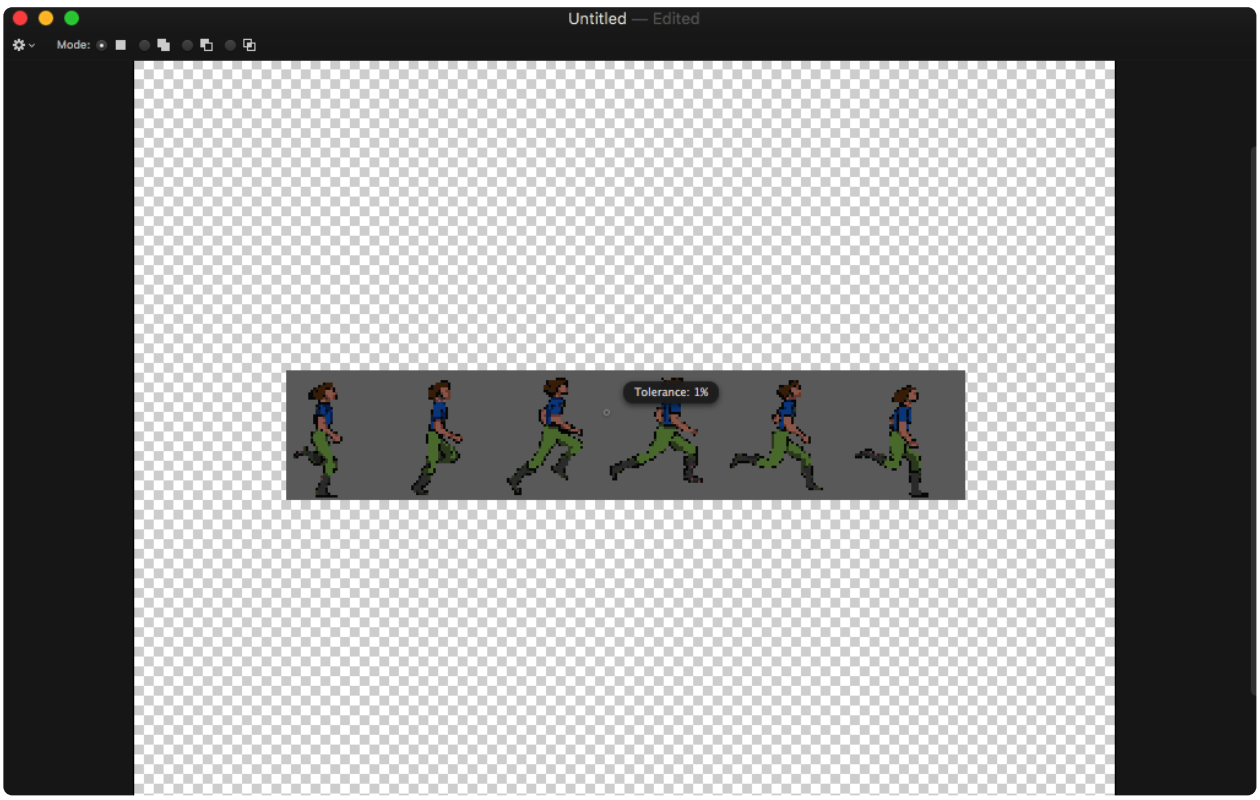


. From the toolbox, select the *Color Selection Tool*.

). The menu bar on the image window is context sensitive -to the selected tool. Right now, it should be showing options for the *Color Selection Tool*. Select the cog-wheel and make sure to disable the "Smooth Edges" option.



1. Click and hold on the black background of the runner image. The *Color Selection Tool* tool select all of the black pixels in the image. While holding the mouse-button down, you can drag the cursor to change the tolerance. Set the tolerance to about 1%.



2. Let go of the mouse button and you should get a selection of the black pixels around the silhouettes of the runner. Press ⌘command-X (or select *Edit->Cut* from the main menu) to remove the background.

3. Find the *Layers* window. It should list two layers: "Background Layer", which you disabled earlier, and "Layer", which is the layer that got created when you pasted the silhouettes. We won't need the "Background Layer" - you can remove it by right-clicking on it and selecting "Delete".

4. Make sure "Layer" is selected in the *Layers* window. From the toolbar select *Rectangular Marquee Tool* and from the image select one of the frames of the runner. Copy to clipboard.

5. Now paste: ⌘command-V or *Edit->Paste*. Initially you may not see any difference, because you've pasted the runner on top of where it was previously. Move (the *Move Tool*) the image so that you can see it.

6. Notice in the *Layers* window that your pasted image created a new layer: "Layer 2".

7. From the main menu select *Edit | Flip Vertical*. Move the inverted runner until it makes a reflection (see below). Note that you can use the arrow keys for fine placement.

8. Now, the upside runner is going to be made into a shadow, so the first thing is to blacken it. From the main menu select *Image | Color Adjustments...*. Find the *Effects Browser* window that should appear, and select "Hue"

.

9. In the pop up dialog, slide the *Lightness* all the way down to -100 to blacken the shadow and press "OK".

10. That's a start, but shadows are softer, less precise in general. In the *Effects Browser* window switch the effect to "Blur" and select *Gaussian*.

11. A dialog will pop up where you can adjust the blur. Watch the main image windows to observe the corresponding effect. Adjust the radius of the blur until you're happy with it and press "OK".

12. In the *Layers* window adjust "Layer 2"'s "Opacity" to about 70% to give you a nice, soft, transparent shadow.

Shadows are cool, how about a reflection?.

13. Select "Layer" in the *Layers* window. Go back to the image, select, copy to clipboard and past another silhouette. The new image will become "Layer 3". Flip it and line it up with the original.

14. Next, you want to distort the reflection slightly. In the *Effect Browser* select "Distortion" effects and double click on "Twirl". The window that will pup up will allow you to manipulate the effect with a combination of "Radius", "Angle", and a dongle-like control that adjusts curvature of the effect.

15. Reflections tend to get less well defined the further away from the source of the reflection. To simulate this, you can create a layer mask with a gradient effect.

16. In the *Layers*, while "Layer 3" is selected, click the cogwheel at the bottom and select "Add Layer Mask".

7. Click on the mask in the *Layers* window and from the toolbar select the *Gradient Tool*.

8. In the image, click near the foot of the reflection, and drag downwards. Drag to near the head of the reflection. You should see the following effect (if you don't, you might have to select a different gradient type in the context-sensitive toolbar):

9. If you're not quite happy with it, you can try to distort and warp things a bit more with filters of your choice.

As already stated, Pixelmator is a decent image processing software, but not dedicated to creating sprite sheets. Hence, creating sprite sheets in Pixelmator from scratch, while possible, is a bit work intensive. There are numerous other software packages, such as Pyxel Edit (https://pyxeledit.com/) or TexturePacker (https://www.codeandweb.com/texturepacker), that might be easier to work with in this regard, though they are not always free, and you'd be responsible for your own license.

Drawing with a tablet

There are two Samsung Galaxy Tab A with S Pen (http://www.samsung.com/us/mobile/tablets/all-other-tablets/samsung-galaxy-tab-a-9-7-16gb-wi-fi-with-s-pen-smoky-titanium-sm-p550nzaaxar/) tablets in the department that you can borrow for a limited time. It's an excellent tool for artwork creation - much easier to draw with the pen than with a mouse.

- Before login on to the tablet, make sure that "Student" account is selected

- Swipe to log on. The tablet has two applications for drawing: Autodesk Sketchbook (https://www.sketchbook.com/?locale=en) and Arftflow (http://artflowstudio.com/). You can find the icons for both programs on the home page once you log-in.

- In either program you can create artwork by drawing with the S-Pen right on the tablet. You can work off existing pictures, create new ones, use layers...etc. There are plenty of tutorials out there on digital painting, such as Ctrl-Paint (http://www.ctrlpaint.com/library/) series for instance. This lab will not provide instructions on how to use the programs - just how to get the images off of the tablet.

- Once you have you're image ready, like the one below done in Artflow, find the option for sharing the image and save it to the device. Remember which directory you saved it to - "Pictures" or "Downloads" or whatever.

- Next, plug-in the provided USB dongle (the little USB connector doesn't go all the way in on the tablet, so don't force it). Then plugin a memstick at the other end of the dongle. After a short delay "File Manager" should come up on the tablet screen. Find the directory where you file has been saved. Tap and hold the file, until selection box appears next to it. Select it and click on "More" in the right-hand corner. You should get an option to "Copy" or "Move" the file. Then copy/move it to your USB.

- If it's a PSD file (default save mode for Artflow), then you can open that file in Pixelmator and all the layers will be preserved...so you can continue working on your artwork - adding various effects, and then saving it as PNG. If you saved the artwork in PNG format, you can probably import it to Unity directly.

Drawing with Inkscape

Inkscape is another graphics creation program. While Pixelmator is mainly based around working with raster images (pixels), Inkscape is more about editing vector based images. Vector based images have the advantage that they scale to any desired resolution without blurring or 'pixelating'. Since version 2018 Unity has support for rendering vector based graphics, SVG files

(see the Vector Graphics (https://docs.unity3d.com/Packages/com.unity.vectorgraphics@2.0/manual/index.html) section of Unity manual for more details). Still, it is often a better option to create assets in vector graphics and save it as a raster image at a resolution appropriate for your game.

One of the coolest sites around...

Rather than write another tutorial like the Pixelmator one, someone else has already put together a series of excellent tutorials based around Inkscape. What's more, they're aimed at programmers as much as artists. Here's a good selection, although it's worth browsing the entire blog (http://2dgameartforprogrammers.blogspot.co.nz/) at some stage.

- Let's get started - with circles (https://2dgameartguru.com/lets-get-started-with-circles-2/)
- Continue the fun with squares (https://2dgameartguru.com/continue-the-fun-with-squares/)
- Bringing in the gradients... (https://2dgameartguru.com/bringing-in-the-gradients/)
- More fun with gradients... (https://2dgameartguru.com/more-fun-with-gradients/)
- Staying in shape... the Clip Tool... (https://2dgameartguru.com/staying-in-shape-the-clip-tool/)
- Creating a game character (https://2dgameartguru.com/creating-a-game-character/)
- Creating a basic face (https://2dgameartguru.com/creating-a-basic-face/)
- Character Animation... (https://2dgameartguru.com/character-animation/)
- Tank Tutorial (https://2dgameartguru.com/tank-tutorial/)
- Back with a BANG! (https://2dgameartguru.com/back-with-a-bang/)

PyxelEdit

PyxelEdit (http://pyxeledit.com/) is an excellent tool for creating pixel-style tiles and animation. Tutorial on how to use it can be found on PyxelEdit's website (http://pyxeledit.com/learn.php)

Materials

Let us turn our attention to graphics capabilities of Unity. For the reminder of this lab youw will work with a scene following steps that showcases some of the capabilities of the the game engine's graphics.

0. Fork the Graphics (https://altitude.otago.ac.nz/cosc360/graphics) project on GitLab, then clone your fork to your hard-drive.
1. Open the Graphics project, the directory of the cloned repository, in the Unity Editor.
2. A "SampleScene" has been already set up with a couple of Sprites in the Background and a "Player" game object that animates a prepared sprite sheet from the previous exercise. You can press the "Play" button and move the player left and run to see the animation.

Let's start with Materials (https://docs.unity3d.com/Manual/Materials.html). In order to render an object in the scene, Unity must know its shape and the appearance of its surface. The shape is defined by a *Mesh*, and the rendering of the surface is governed by a program called a Shader (https://docs.unity3d.com/Manual/Shaders.html). Shaders are instructions to the graphics hardware on how to render pixels on the surface of the mesh. Some shaders provide variables that can customise the look of the surface without the need to dive deep into the shader code. In Unity, a *Material* (not to be confuses with the *Physics material*) is a Shader wrapper that provides an interface where you can specify the value of the variables that control the shader. We'll begin by creating several materials using the built-in shaders provided with the game engine.

3. Right-click inside the "Materials" folder in the **Assets panel** and select "Create->Material". Rename the material to "Grass" and set its "Shader" property (in the **Inspector panel**) to "Unlit/Color". Then set the "Main Color" property of the material to specify the colour - something green would do.

   The "Color" shader we chose is a type of "Unlit" shader. "Lit" shaders react to light whereas the "Unlit" ones do not. Surfaces that react to light are dark unless you specify lighting in the scene. Surface that do not react to light have the same level of brightness, regardless of the presence (or absence) of lights. For most 3D games lighting is of critical importance, but for 2D games it ususally it is not. Hence, in this tutorial we will stick with the "Unlit" shaders so that we don't have to worry about the lighting aspects of the scene.

   The Color shader has only one variable - the "Main Color", which is displayed uniformly over the surface of a mesh. You can see how its going to look in the preview panel (below the **Inspector panel**). By default the previou shows the shader over a spherical mesh. You can switch the preview to another shape by clicking the shape button as shown below.

   The screenshot above has the preview shape set to a quad - which is a flat mesh consisting of 4 points.

4. Right-click in the **Hierarchy panel** and add a "3D->Quad" game object to the scene. Rename it to "grass" and make it a child of the "Background" object. Set its Transform properties: Position (X,Y,Z) to (0,0.2,11) and Scale (X,Y,Z) to (15, 0.6,1). The reason why we position the quad at Z=11 is so that it renders behind the buildings, which are at Z=10.

   A quad is just a flat mesh, perfect for rendering a 2D surface.

5. Set the "Mesh Renderer->Material" property of the "grass" game object to "Grass". You should immediately see the quad change its appearance in the scene.

   Just like that, we can quickly get some visuals going in Unity without sprites or artwork. This is great for placeholders and sometime even useful for final visuals in the game.

6. Create a new material named "Pavement" using the same "Unlit/Color" shader, but this time specify the colour as something greyish. Create a new quad game object (or duplicate the "grass" one) in the scene, make it a child of "Background", set its "Mesh Renderer->Material" property to the "Pavement" material, and set its position and scale as shown in the screenshot below.

Textures

Single colour shaders are easy to set up, but have limited use. At some point, you'll want something more visually appealing. Textures are 2D images that shaders wrap around a mesh. In fact, a sprite is just a quad with a texture image laid over. In this scene, we'll use a simple fence texture to create a continuous fence.

7. Create a new "Material". This time set its "Shader" to "Unlit/Transparent". This shader requires you to specify the texture to wrap over the mesh. Set the texture to the "fence" texture provided in your project's "Assets/Textures".

8. Next, create a new quad game object in the **Hierarchy panel**; set its properties as in the screenshot below, and change its "Mesh Render->Material" to "Fence".

   Is the fence texture stretching over the entire length of the quad? That's not what we want - we want the same texture to be repeated a number of times in the horizontal direction. In order for that to happen, we need to change the configuration of the "fence" texture and the "Fence" material.

3. In the **Project panel** select "Assets->Textures->fence" and set its "Advanced->Wrap Mode" property to "Repeat". This tells the Unity that it is allowed to tile the texture over a material. Press the "Apply" button, otherwise the setting will take no effect.

9. Now, click on the "Fence" material that you have created previously (if you followed the outlined folder structure, then it should be under "Assets->Materials->Fence") and set its "Tiling" X value to 10. That means that we want to repeat the texture 10 times horizontally. You should immediately see the fence texture repeated over the quad in the scene.

   Tiling a repetitive pattern, rather than baking the repetition into the sprite, is not only more flexible (allowing you to change number of repeats at will) but also way more efficient for the graphics engine to handle. Loading a big image is way more taxing for the graphics card than loading a small one and repeating it many times in the frame buffer.

1. Create a new material called "Skyline", set its shader to "Unlit/Transparent" and set its texture to "buildings4" tiled 3 times in the X direction. Then create a game quad game object called "skyline", set its "Transform" properties as shown below and its "Mesh Renderer->Materials" to "Skyline". Don't forget to set the "Wrap Mode" of the "buildings4" texture to "Repeat".

2. Let's add a quad to stand to give us sky. Create a new material called "Sky", set its shader to "Unlit/Color", set the color to a bright yellow, and use this material in the mesh rendered of a quad game object called "sky", with properites as shown in screenshot below.

Rendered textures

The scene we are building is a waterfront scene, and so we need water. You know now you could easily create a blude "Color" texture to stand in for water. However, we'll try something more advanced - a reflection. For this, we will use a secondary camera and direct its captured view to a rendered texture.

3. Right-click in the **Hierarchy panel** and create a "Camera" game object. Name it "ReflectionCamera". Set its "Transform" properties as in the screenshot below. Also set its "Projection" to "Orthographic", "Size" to 4.5, and "Viewport->H" to 0.6. Also, disable the camera's "Audio Listener" - only one camera in the scene is allowed to have its "Audio Listener" enabled and this is typically left best enabled on the "Main Camera".

   The "Orthographic" projection flattens the captured image along the "Y" axis, which is the setting desirable for many 2D games. The whole setup is such that the second camera captures part of the scene from above the pavement line. Also, note the "ReflectionCamera"'s "Target Texture" property, now set to "None".

4. Right-clink in the "Assets->Texture" and select "Create->Render Texture". Name the new texture "reflection".

5. Set the "Target Texture" property of the "ReflectionCamera" to the "reflection" texture.

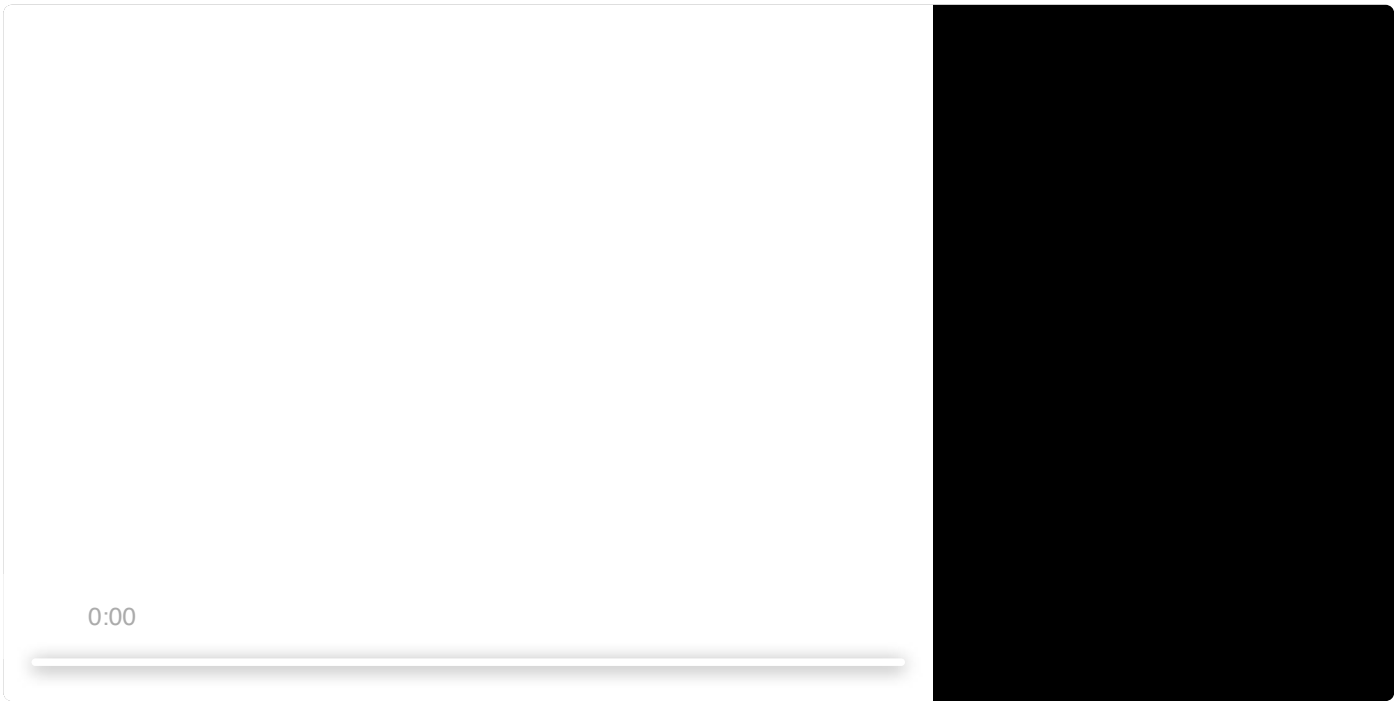
6. If you've been paying attention, you should be able to guess what is coming next. Create a new material, name it "Reflection", set its shader to "Unlit/Transparent" and set "reflection" as its texture. You should see the image captured by the "ReflectionCamera" in your texture preview right away.

7. In the **Hierarchy panel** create a new quad object, name it "water", set its "Transform" properties as in the screenshot below, and set its "Material Renderer->Material" to the "Reflection" material.

3. The reflection doesn't look right. To flip it upside down, negate the "water" game object's "Scale->Y" value.

3. To stretch the texture over the quad, change the "Reflection" texture's "Tiling->Y" property to 0.4. So, rather than stretching the entire length of the texture vertically over the quad, the engine will stretch only a fraction of it.

). And now, if you play the scene, you can see that the Player sprite gets reflected in the water as it moves through the scene.



L. A great feature of not having reflections baked into your textures is that if you add more sprites to the scene, they get incoroprated into the reflection automatically. Add the street lamps (you can find the images in the "Assets->Textures") to the scene as shown below.

2. Play the scene - the reflection camera captures the new added objects.

Scriptable rendering pipeline

The rendered texture from a second camera provides the basis for a water effect, but until we add some distortion (to simulate water ripples) it will look more like a mirror rather than a water reflection. To get the ripples we will need a custom-made shader.

Rendering pipelines

Unity recognises that different games have different graphics requirements and so they provide a choice of different rendering pipelines. The default, so called Built-in Rendering Pipeline (https://docs.unity3d.com/Manual/built-in-render-pipeline.html), is fairly basic, optimised to perform well on a wide range of hardware, but it only supports the pre-build shaders. To make custom shaders we need to switch the Graphics to use the Universal Rendering Pipeline (https://docs.unity3d.com/Manual/universal-render-pipeline.html). This pipeline is (possibly) a bit more taxing on the graphics than the built-in one, but not as much as the High definition rendering pipeline (https://docs.unity3d.com/Manual/high-definition-render-pipeline.html), which brings very advanced graphics capabilities (that are likely not needed for a 2D game)..

Let's change the rendering pipeline.

3. First, you need to install the "Universal RP" package. From the main menu, select "Window->Package Manager". In the window that comes up switch the "Packages" options to "Unity Registry". Find "Unity RP" in the list, and click the "Install" button. This will take a couple of minutes (installing the "Shader Graph" package at the same time). After the package is installed you can close the Package Manger window.

4. Right-click in the "Assets" window of the **Project panel** and select "Create->Rendering->Universal Render Pipeline->Pipeline Asset (Forward Renderer)". The should create assets that you need to reconfigure the Unity's rendereing pipeline.

5. To change the pipeline setting, from the main menu select "Edit->Project Settings", and in the "Graphics" settings change the "Scriptable Render Pipeline Settings" from "None" to "Graphics->ScriptableRenderPipelineAsset (Universal Render Pipeline Asset)".

6. The new pipeline is set, but let's reconfigure one one more thing. In the "Project Settings" window select the "Player" tab and under the "Other Settings->Rendering" section change the "Color Space" from "Gamma" to "Linear". Once this is done uncheck the ""Auto Graphics API" option undenreath and remove the "WebGL 1.0" item from "Graphics APIs".

   The colour space setting is independent of the rendering pipeline setting. It controls how the colour information is encoded in the game engine.

Gamma vs. Linear colour space

Our eyes do not distinguish colours equally well in all parts of the three-channel colour space - we perceive finer distinctions between the clours in the brighter range. Hence, it does not make sense for monitors to render dark colours that we cannot perceive. And this, in turn, means that it does not make sense to send colour information to the monitors that they will not render. The Gamma colour space is a complressed (and therefore more efficient) encoding of the colours that filterrs out the information that is not needed by the monitors (nor our eyes). As a result, software has less data to deal with and the graphics can render things faster. However, image processing on the compressed gamma colour space is limited. For instance, if you wanted to change mid-tones of an image to bring out darker areas into the brighter colour of the spectrum, the information about the colours in that darker part of the

image might not be there in the Gamma colour encoding. Therefore, sometimes it is better to keep the colour data encoded in the Linear space, which presereves all the colour information without any compression, while images are still processed in the rendering pipeline, converting to Gamma space just before sending out the information to the monitor. The linear colour space is more taxing for the game engine, but has potential to produce (visually) better results. An analogy from photography is the difference between the JPEG and RAW image compression. A phone camera takes an image and saves it into a JPEG format. JPEG is a compressed format, so some information, that we do not perceive well anyway, is lost. It is still possible to do some post image processing on a JPEG image, but information that has been lost in the compression cannot ever be brought out. SLR cameras save images in a RAW format, preserving information exactly as recorded by the photosensors. RAW images are incredibly large (Byte-wise), but are more maleable in post-processing.

Linear colour space means more work for the game engine, and is not supported by older libraries, such as WebGL 1.0. That's why you removed WebGL 1.0 from the supported APIs in the settings. The use of Linear colour space is probably not absolutely critical for this exercise, and in the end you will always need to decide on the best trade off between awesome graphics and the need for a wider hardware/library support for your game. However, just to demonstrate the existence of this option, we will stick with the Linear colour space for the rest of the tutorial. You can change it back to Gamma at the end and see if you spot any visual differences.

Shader graph

With the graphics switched to the Universal Rendering Pipeline you can now make custom made shaders. Shader programming is...possibly not super hard once you become practiced with it, but the learning curve is pretty steep. Luckily, Unity provides a tool called Shader graph (https://docs.unity3d.com/Packages/com.unity.shadergraph@latest/index.html) that makes the shader programming relatively easy, and (actually) quite fun. The "Shader Graph" package should have installed itself when you installed the "Universal RP" package - if it did not, navigate to "Pacage Manger" window and install it. In this part of the exercise, you'll use the Shader Graph tool to create a shader that simulates water ripples.

7. Create a new shader by right-clicking in the "Assets->Shaders" folder of the **Project panel** and selecting "Create->Shader->Universal Render Pipeline->Sprite Unlit Shader Graph". Name the new shader "Water".

8. Double click the "Water" shader. It should open the shader in the **Shader graph panel**.

   Shader graph is a tool for visual programming of shaders. You can add various processing nodes, connect them and can inspect visual previews instantly. The two nodes found in the graph by default are the nodes with various properties of the shader - by connecting to their inputs you influence various aspects of your shader. The Vertex node provides inputs that affect the shape of the shader surface (relative to the mesh), whereas the Fragment node inputs affect the appearance of the rendering of the mesh surface. The "BaseColor(3)" input of the Fragment node accepts the colour information about the surface (the (3) in the brackets indicates that it expects 3 numbers, a 3D vector for RGB values). The "Alpha(1)" input of the Fragment node accpets a single value (between 0 and 1) that specifies the transparency of the surface - currently accepting a constant value of 1, which we will not touch. The "Graph Inspector" window shows various bits of information about selected node or the overall shader "Graph Settings".

   Navigating shader graph with a Mac mouse is a bit tricky: scrolling zooms; to pan around you press the (⌥option) button, click on the window and drag.

9. Let's start by turing the "Water" shader into a simple texture shader. To add a new node to the graph, right-click somewhere in the empty area of the shader graph window and select "Create Node" (you can also invoke the "Create Node" dialog by pressing the "Space bar". In the "Create Node" dialog start typing "Texture" and you should see several matching block options.

9. Select the "Sample Texture 2D" option.

   The node in the shader graph display inputs on the left and outputs on the right-hand side. One of the inputs of the "Sample Textured 2D" node is "Texture(T2)", which accept a texutre (you can simply select an image from your assets in the small rectangle connected to the input). The other input is "UV(2)", which has to do with the manner of mapping of the texture to the mesh surface. We will come back to the "UV(2)" in a moment. One of the outputs of the "Sample Texture 2D" node is "RGBA(4)", which represents a 4-dimensional vector of 3 colours + transparency. The other outputs are just the indidviual components of that vector "R(1)", "G(1)", "B(1)" and "A(1)", which (presumably) need to be split often enough that it makes sense to provide two options for outputing the colour information.

10. Click on the input rectangle connected to the "Texture(T2)" input of the "Sample 2D Array" node and select the "reflection" texture. Recall, that's the texture rendered by the second camera in the scene. You should see a preview of the texture captured by the cameara in the node right away. Click on the "RGBA(4)" output of the "Sample 2D Array" node and drag a connection to the "BaseColor(3)" of the "Fragment" node. To save your graph click on the "Save Asset" button.

    Note that you made a connection from a (4) output to a (3) input. Shader graph allows this - it silently drops the 4th coordinate from the connection. Since the 4th component of the "RGBA(4)" output is the alpha channel, the connection to "BaseColor(3)", which accepts "RGB" vector works as intended.

11. Let's check how the shader works in the scene. Recall that the "water" game object is a quad mesh using "Reflection" material for its surface. In the **Project panel**, in the "Assets->Materials", find the "Reflection" material . Change its shader to the newly created shader, which can be found under "Shader Graphs/Water".

    The reflection in the scene should change immediatelly to the image produces by the new shader. The main difference you will notice is that the image now is not scaled properly. That's because the previous shader had the lower 0.4 of the vertical aspect of the texture stretched over the entire quad in the scene (recall, the setting of the Tiling property of the previous shader). We need to add the tiling setting into the new shader.

12. Create a new node in the shader graph of the "Water" shader and type in "Tiling". You should see the option for "Tiling and Offset" node.

13. Add the "Tiling and Offset" node to the graph. Set the Y attribute of its "Tiling" input to 0.4. Connect the "Out(2)" output of the "Tiling and Offset" node to the "UV(2)" input of the "Sample Texture 2D" node. You should instantly see the texture stretched in the preview window.

    The "Tiling and Offset" node you've just set up, takes the "UV(2)" coordinates from the UV0 channel. The UV0 channel gives the UV-coordinates of the texture mapping, with (0,0) marking the bottom-left corner of the mesh and UV (1,1) to the top-right coordinate of the mesh. The "Tiling(2)" and "Offset(2)" inputs specify how much tiling and offset there should be in the texture map; the output "Out(2)" produces UV coordinates matching the set of all input UV coordinates with the appropriate tiling and offset effects incorporated. This is fed through the connector to the "UV(2)" channel of the "Sample Texture 2D".

UV coordinates

The UV-coordinate system is used to refer to specify locations of the flat texture map as it is stretched over a mesh. The (U,V)-coordinate (0,0) relates to the bottom-left of the map and (1,1) to the right-top of the map. The stretching becomes somewhat complicated over 3D meshes, but in a case of a quad it's pretty straight forward - the (U,V) square gets stretched over the shape of the quad so that (0,0) related to the bottom-left of the mesh and (1,1) relates to the top-right of the mesh.

The texture mapping can be tiled and offset. The tiling (X,Y) value of (1,1) maps the texture over the mesh so that the one entire width and one entire height of the texture stretches over the mesh.

The tiling (X,Y) value of (2,1) maps the texture over the mesh so that the two entire widths and one entire height of the texture stretches over the mesh. If the texture wrap is set to the clamped mode, the texture ends up covering left-half of the mesh (in the image below we show the right-half being empty, but in Unity the last column of pixels just gets repeated).

If the texture wrap is set to the repeat mode, the tiling (X,Y) value of (2,1) will repeat the texture twice in the horizontal direction, and just once in the vertical direction.

If the texture wrap is set to the repeat mode, tiling (X,Y) to (2,1) and offset (X,Y) to (0,0.25), then the texture will repeat twice in the horizontal direction, once in the vertical direction, and it also will be pushed up quarter of the way; the top quarter of the texture will be repeated at the bottom (since the texture is tiled over the mesh).

5. Don't forget to press the "Save Asset" button in the shader graph. Check out the reflection in the scene. It should now be identical to the one we had before switching over to the custom made shader.

6. Time to mess with the texture offsets to get a distortion that looks like water ripples. In the "Water" shader graph, create a new "Sample Texture 2D" node, set its texture to "cloudnorm" (which you have in your "Assets/Textures") and connect its "RGBA(4)" output to the "Tiling and Offset" nodes's "Offset(2)" input.

   The "Tiling and Offset", as well as the other "Sample Texture 2D", nodes' previews shoud be now distorted. Here's how this effect comes about. The "cloudnorm" texture can be visualised as a colour image, but in fact the information it contains is not really about colours, but more about 3D vectors. Each pixel of the "cloudnorm" texture has a 3D vector, which when interpreted as values of RGB, renders as shown in the preview of the left-most "Sample Texture 2D" node. By connecting that node's "RGBA(4)" output to the "Tiling and Offset" node's "Offset(2)" input, only the first two numbers (the RG values) are used and treated as offset. Since these 2D values change from pixel to pixel, we end up with different offsetting of different pixels in the texture which gives a distortion of the texture.

7. At the moment we have too much distortion. We need to tone it down a bit by multiplying the distorting texture by a fraction. In the "Water" shader graph, create a new "Multiply" node. Delete the connection between the left-most "Sample Texture 2D" and the "Tiling and Offset" nodes (right click the connection line and select "Delete") and put the new "Multiply" node in between. Connect the "Sample Texture 2D"'s "RGBA(4)" output to the "Multiply"s "A" input. The "Multiply" node should immediately recognise that your input is a 4D vector and change other inputs and outputs to 4-dim vectors as well. Connect the "Multiply" node's "Out(4)" output to the "Tiling and Offset" node's "Offset(2)" input.

   In the "Multiply" block we don't want to multiply all channels independently. It would be easiest to control the strength of the distortion with just one number. For that you will create an input value.

8. In shader graph, locate the window entitled "Water" that lists "Properties" and "Keywords". Click on the plus sign and add a "Float" property. That creates a variable in the shader. Name the variable "Amplitude". Then right-click the variable and drag it out to an area in the graph near the "B(4)" input of the "Multiply" node. You should get a new node that has single output "Amplitude(1)".

9. Connect the "Amplitude(1)" output to "Multiply" node's "B" input. Even though you are connecting a 1-dim value to a 4-dim input, shader graph figures out what to do - it will just multiply every value of the 4-dim input with that single 1-dim input. Click on the "Amplitude" property in the "Properties" window and find the "Graph Inspector" window. In there you should now see various characteristics of your property, including its "Default" value. Set it to 0.05. Click on the "Save Asset" button.

   Now the distortion in the preview of the right-most "Sample Texture 2D" should be a bit more subtle...and kind of looking like a water ripple, wouldn't you agree?

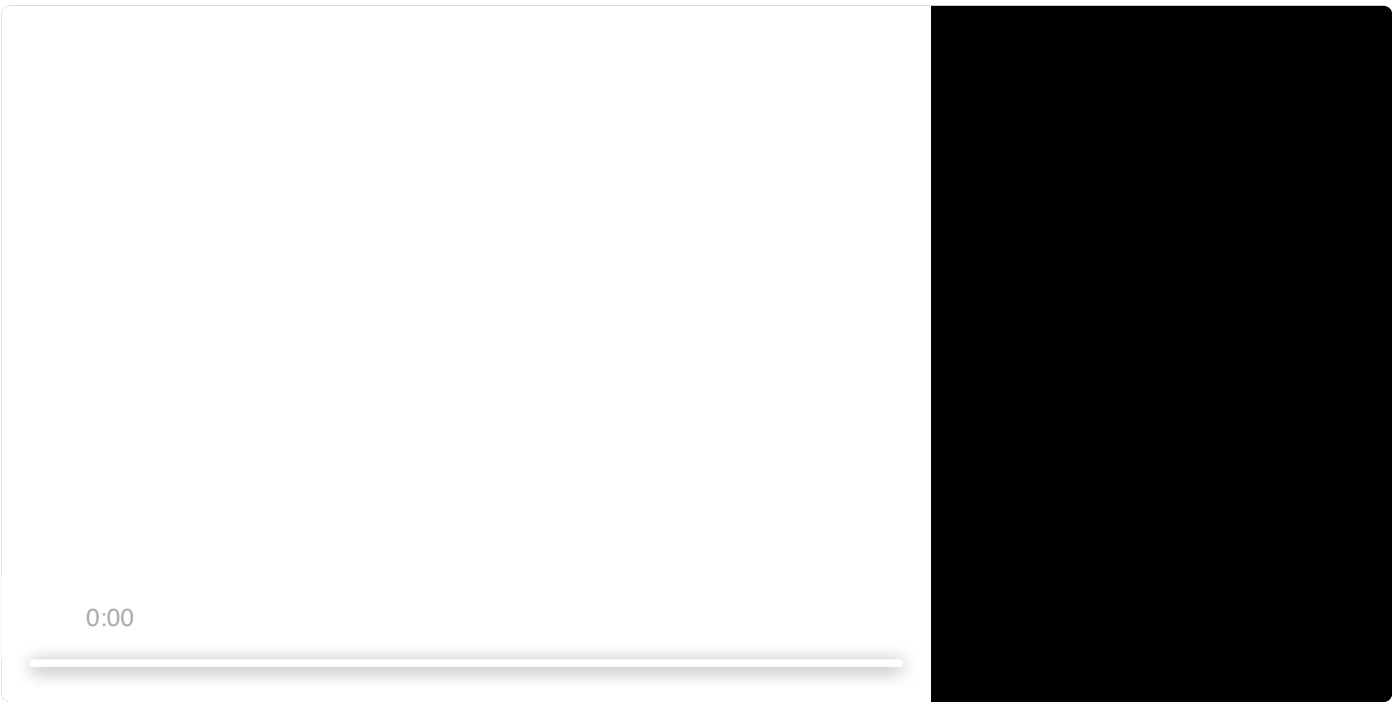
10. Check how the reflection looks like in the scene. Not bad, but still static - water ripples should move.

11. To get the ripples moving, you need to have the distorting offsets change with time. In the "Water" shader graph, to the left of the "Sample Texture 2D" with the "cloudnorm" texture, create another "Tiling and Offset" node as well as the "Time" node. Connect them as shown in the screenshot below and click the "Save Asset" button.

    At this point the cloudnorm texture has likely disappeared from the "Sample Texture 2D" preview. What happened? Well, the "Time(1)" output of the "Time" node increases constantly (with time). Since this is used as the offset to our distortion texture, the offset changes (for both X and Y offset at the same time) with the time variable. If you have followed the steps of this tutorial faithfully, then the wrap property of the "cloudnorm" texture is the "Clamped" mode. That means that, as soon as "Time" vlaue gets larger than 1 (and that seems to be happening quite fast) the texture offset very quickly gets out of the UV range and the result is complete shift of the texture out of the mesh frame.

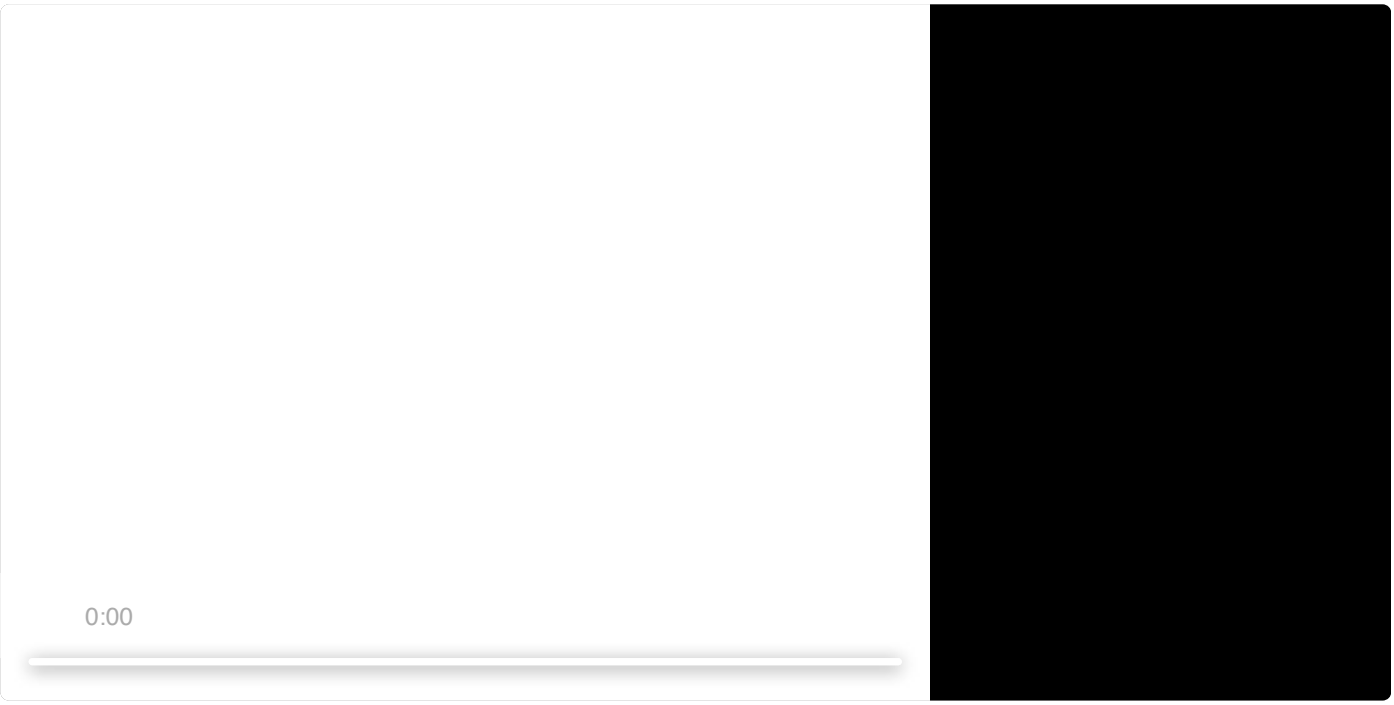
12. Find the "cloudnorm" texture under "Assets/Textures" and set its "Wrap Mode" to "Repeat".

13. Play the scene now - the ripples should be moving. That's due to the offset of the "cloudnorm" distortion being moved with time. Since the distortion texture is tiled over in repeat mode, the continuous offsetting ends up giving a distortion that looks like it moves diagonally across the quad.

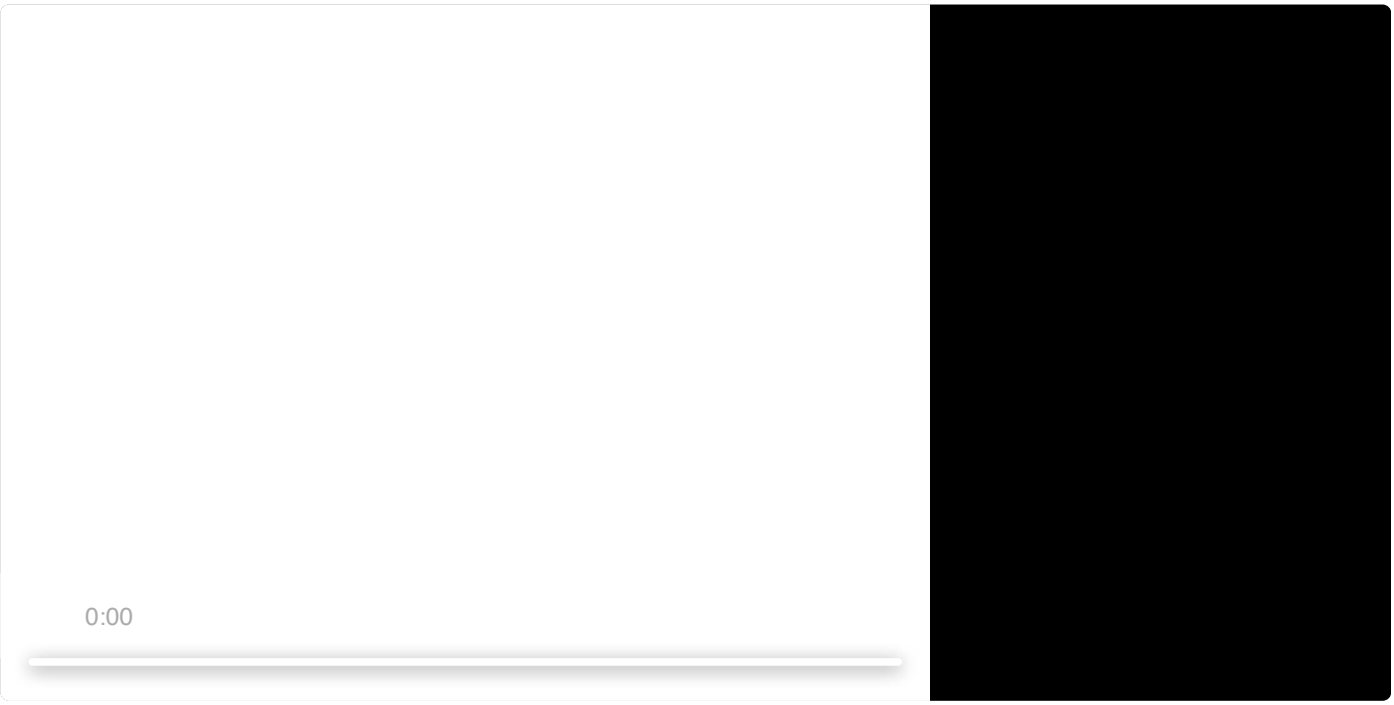


14. That ripple effect goes way to fast. To dial it down a bit get back into the "Water" shader graph and create a new "Float" property and name it "Frequency". Create another "Multiply" node and stick it between the "Time" and "Tiling and Offset" nodes; drag the "Frequency" property out into the graph and connect everything as shown in the screenshot below. Set the "Frequency" default value to 0.1. Don't forget to press the "Save Asset" button.

5. Play the scene now, the ripples should be a bit slower.



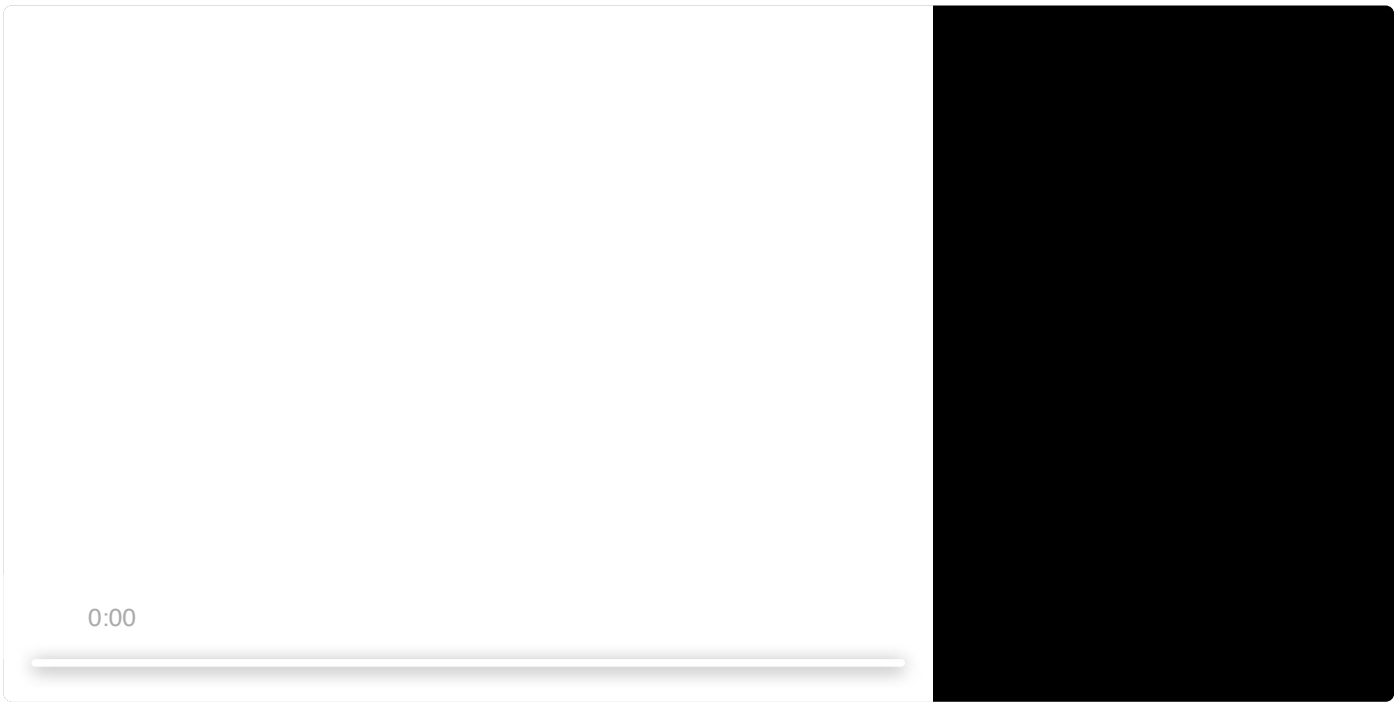
6. The overall ripple effect is still a bit too much - need to adjust the amplitude and frequency a bit more. You can do that while watching the effect in the scene. Play the scene, and then, while the scene is playing, find the "water" game object in the hierarchy. In the **Inspector panel** expand its "Reflection (Material)" property and you should see the values of the properties of your shader. Remember, Material is just a UI wrapper for the shader, so all the properties you created in the shader graph show up there and can be controlled from the inspector. Play with the values and "Frequency" and "Amplitude" until you find a combination that works for you. The process of my fiddling with the variables is shown in the video below.



7. There is one more thing that doesn't quite work (for me) in this effect. I would like those ripples to move only vertically and not diagonally. For that, we will need only the Y offset of the distortion texture to be changing with time, while the X offset is kept at zero. Back in the shader graph of the "Water" shader, add the "Combine" node and connect to other nodes as in the screenshot below.

The "Combine" node block takes 4 inputs and builds a 4-D vector. By connecting the multiplied time-dependent value to the 2nd input ("G(1)") and leaving the others set to zero, we have created a 4D vector with its second coordinate varying with time. Connecting "Combine" node's "RGBA(4)" output to "Tiling and Offset" node's "Offset(2)" input, the last two components get dropped - you have effectively created an input of 2D, where the first coordinate (the X offset) is 0 and the second coordinate (the Y offset) is changing with time.

3. Play the scene to see the new effect. In my setup the water seemed to be flowing slowly upwards...and that didn't seem right. So I fiddled with the values and found out that making "Frequency" a negative number flipped the Y offset of the ripple giving me the desired effect of the ripples flowing downwards. This process is shown in the video blow.



And that is (one) potential way of getting a reasonably looking water ripple effect. The whole "Water" shader graph is shown in the image below (open it in new tab for better resolution).

9. Play the scene with the reflection effect based on the custom made "Water" shader.

Post-processing

Time to put some finishing touches on the ripple effect. This could probably be done just as well in the shader graph, but we'll use the post-processing capabilities of Unity. Post-processing allows application of various photo processing effects to the rendered image. Typically this is done on the entire image of the scene, but we'll set it up so that it only applies to the view of the ReflectionCamera.

0. The Universal Rendering Pipeline provides post-processing capabilities, which are configured through Volumes. In the "Assets" window right-click and "Create->Volume". Name it "ReflectionVolume".

1. In the **Hierarchy panel** select "ReflectionCamera". Make sure its "Camera->Rendering->Post Processing" property is enabled.

2. To make sure that the post processing doesn't change the view of the main camera, make sure to disable the "Camera->Rendering->Post Processing" property of the "Main Camera".

3. The "Scene" view is generate from its own camera, separate from any of the cameras in the scene. To disable post processing on the view of this camera, expand the "Toggle skybox, fog and various other effects" icon and remove the tick by the "Post Processing". The screenshot below will show you where to find this setting.

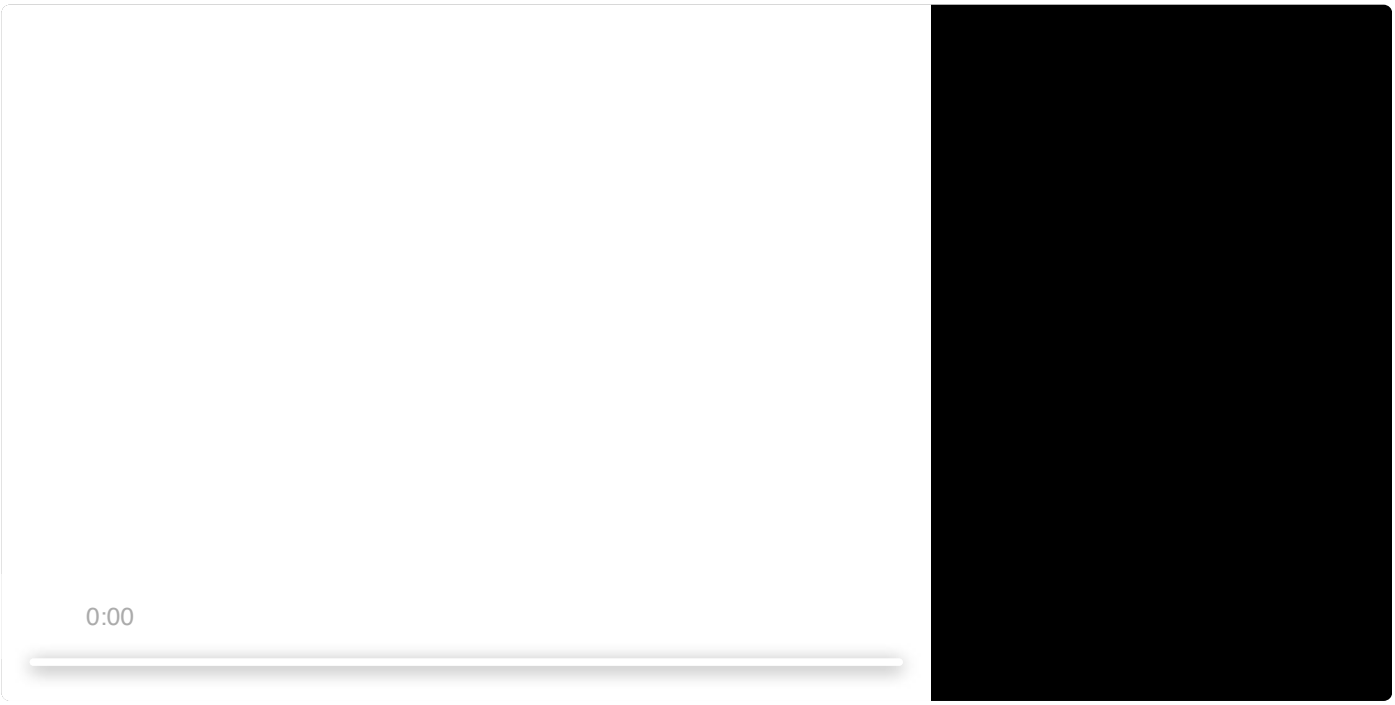
4. Now, right-click in the **Hierarchy panel** and create a post processing volume game object by selecting "Volume->Global Volume". Name the new game object "ReflectionVolume". Set tis "Volume->Profile" property to the "Reflection Volume" asset.

   Now you can add effects to the post processing volume and they should only affect the view of the second camera, which is fed into the reflection texture.

5. In the **Inspector panel** showing properties of the "ReflectionVolume" game object, click on the "Add Override" button and select "Post-processing->Color Adjustments". In the new subcomponent that shows up, enable the "Color Filter" and set its colour to something blueish. The idea is to make the reflection slightly blue, since it's reflecting off of water.

6. Just to try something different, I have also added the "Post-processing->Bloom" override with the settings as shown below. It gives the water few reflections, a bit like sun glistening in the wave.

7. Play the scene to see the final result with post-processing on the reflection.



There are many options for post processing effects, so have a bit of play with them.

Animating a shader

The potential for awesome special effects is even greater when we throw a bit of scripting to the mix. We'll create a shader for the sky at sunrise and animate it to seem that the sky is getting brighter over time. Post-processing characteristics can be controlled from the scripts as well, but in this final example we'll focus on a script that controls the properties of a shader.

3. Create a new shader by right-clicking in the "Assets->Shaders" window and selecting the "Create->Shader->Universal Render Pipeline->Unlit Shader Graph". Name the new shader "Sky".

9. Double-click on the new shader to open it in the shader graph. Create the graph as shown in the screenshot below. Make sure the "Space" setting of the "Position" node is set to "Object". It is also very important that the "Reference" field of the "Origin" float property is renamed to "Origin". These reference names are how scripts refer to shader properties, and by default Unity create a complicated random string (not sure why). The C# script supplied in this project will attempt to control shader variable referenced as "Origin". Also, don't forget to click the "Save Asset" button at the end.

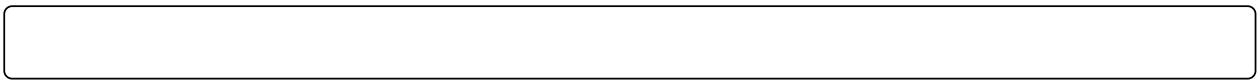
This graph creates a shader that starts at the bottom of the mewsh with the "BottomColor" and gradually changes to the "TopColor" along the vertical dimension of the mesh. Here's how this effect falls out of the above shader graph. The "Position" node outputs coordiantes of the points on the mesh (since the chosen space is "Object", these coordinates are relative to the origin point on the mehs, and are not affected by the position of the object in the scene). The "Split" node is used to split the 3D position vector and fish out the second coordinate, the Y value, or the vertical offset. The "Subtract" node gives the result of subtracting the value of the "Origin" property from Y. Essentially all the values Y>"Origin" end up as a positive number, and values Y<"Origin" end up as anegative number. Then, in the "Divide" node, the result of the sbtraction is divided by the value of "Spread". The larger the spread, the smaller the outputs of the "Divide" node and the more gradual the change from one colour to the other. The "Clamp" node ensures that any values coming out of the "Divide" node that are bigger than 1 are set to 1 and those that are less than 0 are set to 0. Thus, we end up with a range of values going from the bottom to the top of the mesh inexed with numbers between 0 and 1. This range is feed into the "T" input of the the "Lerp" node. "Lerp" stand for "**L**inear int**erp**olation" - it mixes inputs "A" and "B" according to the formuala (1-T)*A+T*B. Since "A" is connected to the "BottomColor" colour property and "B" is connected to the "TopColor" colour property, the results is that wherever "T" is 1 the output of "Lerp" is the "TopColor", whenever "T" is 0 the output is the "BottomColor", and whenever 0<"T"<1 the output is a colour that is a mix between the two. The output of "Lerp" is fed into

the "Fragment" node's "Color(3)" input. Since values of "T" are relative to the vertical position within the mesh, the colour mixing ends up giving gradual change from "BottomColor" to "TopColor" over the vertical dimension. For description of the function of individual nodes refer to the shader graph manual (https://docs.unity3d.com/Packages/com.unity.shadergraph@latest/index.html).

Here is the pseudocode representation of the logic implemented by the above shader graph (since the Alpha channel is eventually dropped in the connection to the "Fragment" node, we will ignore it in this pseudocode):

```
/*Given float values: Origin, Spread; and colours: BottomColor=(Rb,Gb,Bb), TopColor=(Rt,Gt,Bt)...*/

for every position point P=(X,Y,Z) on the mesh  /* Position node with object space */
  /* Subtract node */
  S = Origin - Y

  /* Divide node */
  D = S / Spread

  /* Clamp node */
  if D > 1
    T = 1
  else if D < 0
    T = 0
  else
    T = D
  end if

  /* Lerp node */
  R = T * Rb + (1-T) * Rt
  G = T * Gb + (1-T) * Gt
  B = T * Bb + (1-T) * Bt

  /* Fragment node */
  set the colour of pixel at P=(X,Y,Z) to (R,G,B)
end for
```

If this is still not clear, here's an illustration of a pixel by pixel evaluation of the logic of the above shader graph with a specific choice of the values for the float and colour variables. The Position P=(X,Y) is the position of the pixel with respect to the left-bottom corner (with X going horizontally from left to right, and Y going vertically up). To simplify the presentation, the Z coordinate of the position is ignored, we are assuming the X and Y values range from 0 to 1, and we ignore the alpha channel of the colour variables.



). In the **Hierarchy panel** find the "sky" game object and navigate to its Material in the **Inspector panel**. Change the material shader to "Shader Graph/Sky", the newly created shader. The sky background should change into a gradient blending two colours.

L. Add a new component to the "sky" game object - the "Sky Control" script found in "Assets->Scripts". Change the "Sky (Material)" properties, so that "Origin"=0 and "Spread"=0.8.

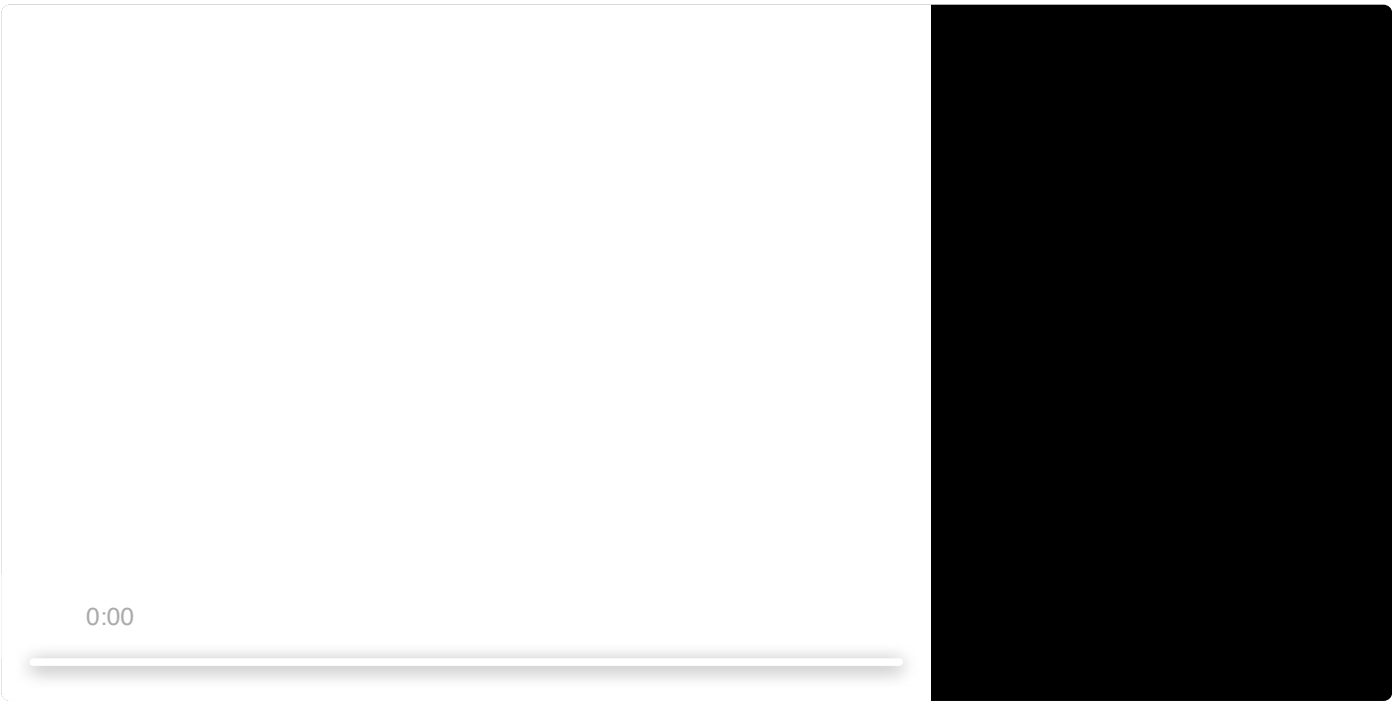
For renference, here's the "SkyControl.cs" script.

SkyControl.cs

```
01: using System.Collections;
02: using System.Collections.Generic;
03: using UnityEngine;
04:
05: [RequireComponent(typeof(MeshRenderer))]
06:
07: public class SkyControl : MonoBehaviour
08: {
09:
10:     public float speed = 0.1f;
11:     MeshRenderer mesh;
12:
13:     void Start()
14:     {
15:         // Get reference to mesh renderer
16:         mesh = gameObject.GetComponent<MeshRenderer>();
17:     }
18:
19:     // Update is called once per frame
20:     void Update()
21:     {
22:         // Get the current origin offset from the material
23:         float offset = mesh.material.GetFloat("Origin");
24:
25:         // Update the origin offset based on time passed since last update
26:         offset -= Time.deltaTime * speed;
27:
28:         // Set the current Origin offset in the material
29:         mesh.material.SetFloat("Origin", offset);
30:     }
31: }
```

This code can be attached only to a game object that has a "MeshRenderer" component. On "Start()" it fetches a reference to the "MeshRenderer" component. On every "Update()" it reads the value of the "Origin" variable of its material. This value is then reduces it by a value computed based on the time difference since the last update and the "speed" variable. Lastly, the new, reduced value, is written back to the material's "Origin" property. The resulting effect is that the "Origin" property of the "Sky" shader gets decremented over time, with the "speed" value controlling how quickly. This changes the location of the change over from the "BottomColor" to "TopColor" in the "Sky" shader, giving an effect similar to the brightening of the sky.

2. Play the scene. The sky should gradually change colour (in my case from dark red to light blue).



This is all for this basic challenge. Remember, the point of this exercise is not just to walk through the steps of the reflection effect, but rather to give you a glimpse of some graphics capabilities of the Unity game engine. Hopefully, this is just the beginning of your learning adventure on graphics/materials/shaders and Unity's shader graph. Go beyond this lab - experiment, test, gather ideas from other tutorials (found online) and get creative, so that you can create awesome graphics for your own game.

Assessment

Show your work to the demonstrator for assessment.

Intermediate challenge

Intermediate challenge is about creating your own digital content/graphics/special effects. Here are few potential ideas:

- Learn how to use particle emitters in Unity and create some awesome particle effects (preferably a suite of reasonably complex effect - simple stuff that you can whip up in half an hour won't do).

  or

- Lear how to create awesome effects with Pro Builder (https://docs.unity3d.com/Packages/com.unity.probuilder@latest/index.html)

  or

- Develop some creative shaders (it's ok to use them in your game later)

  or

- Devise and complete your own Graphics Intermediate Challenge - just check with the lecturer or the demonstrator first, whether the scope of the work will be sufficient for the awarded skill.

Assessment

Show your work to the demonstrator for assessment.

Master challenge

- Devise and complete your own Graphics Master Challenge - just check with the lecturer or the demonstrator first, whether the scope of the work will be sufficient for the awarded skill.

Assessment

Show your work to the demonstrator for assessment.