CAMOSUN COLLEGE

COMPUTER SCIENCE DEPARTMENT

ICS 123 - GAMING AND GRAPHICS CONCEPTS

LAB 6 – MAKING IT PRETTY

DUE: DEMO BY END OF LAB MON FEB 27
You must work individually for this lab.
Submission Instructions are on the last page of the lab.

Objective:

To have the students experiment with textures, skyboxes, 'props', and particle systems.

Preparation:

Supplemental Video Tutorials:

https://unity3d.com/learn/tutorials/topics/graphics/textures
https://unity3d.com/learn/tutorials/topics/graphics/standard-shader
https://unity3d.com/learn/tutorials/topics/graphics/particle-system

Background and Theory:

Art Assets:

So far, the labs have concentrated on game functionality which is the focus of this course. But in this day and age, how pretty a game looks is as important as its functionality (although it seems Minecraft is an exception?). In this lab, you'll investigate how to make your game look pretty.

Right now, our game has several Assets: Scripts, Materials, 3D Models, and Textures. We've been concentrating on Scripts – what controls the game's functionality. Materials, 3D Models, and Textures are considered *Art Assets*. Art Assets are the assets that make up the visual content of the game. Let's re-examine these to be sure we understand what they are:

- *Materials* – defines the surface properties of the 3D object the material is attached to. It can define things like the colour, shine, or smoothness. Our LaserBeam and Enemy both use a custom material. The other 3D objects in the Scene use a basic default material provided by Unity.

- *3D Models* – all the 3D objects that make up the game. This includes the player, enemy, floor, walls, and laserbeam. The *mesh* is the geometry that makes up each model. This includes the vertices, edges, faces, and polygons. Since a 3D Model is literally its mesh, the terms are sometimes used interchangeably. The player, the

floor, and the walls are primitive built-in 3D models (simple shapes). The enemy was imported. It is a more complex 3D model which contains more vertices, edges, faces, and polygons than the primitive ones. Creating a complex 3D Model is a time-consuming process and thus is a computer field all on its own (someone who does this professionally is called a 3D Modeller or 3D Artist). Models can be stored in a variety of file formats but the most common is **FBX** (Filmbox). FBX supports both Mesh and Animation data. Whatever 3D tool you might be using, always try to export your work as FDX files for use in Unity (Blender, discussed below, supports FBX exporting).

- *Textures* – textures are 2D images used to enhance 3D graphics. They are typically displayed on the surface of a 3D model. You'll experiment with textures in this lab. Currently, only the Enemy we imported is using a texture.

There are two more art assets we haven't dealt with yet:

- *Animations* – animations define the 'small' movement of an associated 3D object. This is different from the movement you've been defining in code which moves a 3D object around the Scene. The movement defined by an animation are subtle – things like making a humanoid 3D object's legs and hands move like its walking. Animations are created ahead of time and played at certain times in a game. Like 3D Modelling, 3D Animating is a time-consuming process and thus a field all on its own (someone who does this professionally is called a 3D Animator).

- *Particle Systems* – a particle system is used for visual effects in a game. It is used to create and control a large number of (usually) small objects. Examples are smoke, fire, rain, snow, explosions, and water spray. Unlike 3D modeling and animating, particle effects can be created right in Unity.

Unity is not designed for creating complex 3D Models and Animations and thus external software is usually required. Three of the most popular tools are **Autodesk 3ds Max**, **Autodesk Maya**, and **Blender**. Of the three only Blender is free. 3ds Max is Windows only. Maya and Blender are cross-platform. Which is the best? It depends on who you talk to. Professional studios usually use 3ds Max or Maya. Each cost a whopping $1,500/year. There is a light version called Maya LT that is geared towards indie developers that is $240/year. 3D Modelling and Animation is a huge topic on its own. We won't have time to cover this topic in this course.

Note from the Instructor: If you are interested in trying 3D Modelling with Blender for this course, go ahead. But be warned, the UI is intimidating. I also can't offer support or instruction on using Blender. Perhaps there might be a 3D artist in the class that can! I hate to say it, but if you prefer 3D Modelling over Game Programming, then you are probably in the wrong program. Instead, you should enroll in a 3D Art program. In BC, BCIT or the Art Institute of Vancouver offer such a program.

Whiteboxing:

Whiteboxing is the process of building the basic geometry for a Scene. Since a scene is akin to a level in a game, people who do whiteboxing are called Level Designers (you may have seen job ads for a level designer!!). It's called whiteboxing because the level designer builds the basic structure of the level using plain white boxes. If you haven't figured it out already, you did this in Lab 2. For more complex games, whiteboxing also involves placing enemies and items. Typically, a Level Designer will design a level on a piece of paper, whitebox it, and then hand it to both the Art team to prettify it, and the Game Developers to test their code with.

Texturing:

Texturing is the process of applying 2D images over 3D graphics to visually enhance them. There are a variety of file formats you can use for your 2D texture images but two of the most common are ***TGA*** (Targas) and ***PNG*** (Portable Network Graphics). Both TGA and PNG have lossless compression (no loss in image quality) and both can have an *alpha channel* (an alpha channel means that the image can also store transparency information). The Robot Kyle enemy model uses TGA textures (if you chose a different model, it too may use TGA textures). That being said, we'll use PNG wherever possible. Whereas TGA is legacy, PNG is a modern and commonly used format. When texturing a large area (like a floor and wall), you'll want your image to be *tileable*. A *tileable* (sometimes called *seamless*) image means that it can be repeated side by side and its edges will match up so that there are no visible seams between the repeated images. Texture images should be sized in powers of 2 because graphics cards are designed to handle them that way. For example, 256, 512, 1024, 2048, or 4096 (units are pixels). For optimum performance, they should be square or rectangle: 256x256, 512x256, 2048x1024, 2048x2048, etc. Which size you choose depends on several factors: is the object in question going to be viewed by the camera close up? Is it a large or small object? What kind of graphics hardware will users have? These are some of the factors. For modern games and hardware, you don't want to go lower than 256 as the quality will be too low. You also never want to go higher than 4096 (and 4096 is fairly excessive; a high-performance, high-memory graphics card is required). In this course, we'll use 512x256 and 512x512 (which, so happens, to be about the max you want for mobile games).

A Note about Texels: A texture image's pixel dimensions are constant and are often differentiated from pixels by calling them *texels* (texture pixels). Why? Because the actual number of pixels used to render a texture in a Scene varies because the size of the object the texture is applied to varies. For example, suppose you place a box in a Scene that uses a 256x256 texel texture. You then play your Scene and walk your player so she's right in front of the box. This means the box will be huge and will mostly likely take up almost all the space in the camera (the view of the world). If you are playing your Scene on a Monitor with a resolution of 1920x1080 pixels, the box would be displayed with way more pixels than what the original texture provided! The graphics card will 'average' out each pixel from the texture and display them using several pixels on the Monitor. This causes a blurred effect. In contrast, if your player were to move far back from the box, so it appears quite small in the camera, the graphics card will have to combine pixels from the original texture causing a loss of detail. The further back the player moves, the smaller the box will appear on the Monitor and the more detail is lost.

For a professional game, a 3D Artist would custom design the texture images. But for our purposes, we'll download pre-made free tileable images from the internet. The site we'll use in this course is, surprise, textures.com (Note: this site requires you to sign up. If you don't want to do this, some textures will be provided to you).

When applying textures, you don't actually apply them directly to a 3D object. Instead, you create a Material and attach the Texture to the material. Then, you attach the Material to a 3D object. This may seem counter-intuitive but it does allow you to re-use textures with different materials. Recall also that materials define the surface properties of 3D objects. Since textures are applied to the surface, it makes sense that textures are attached to an object's material!

You will apply textures to the floor and walls of your Scene in Task 2.

Texture Mapping:

The textures you used for the floor and walls are pretty simple: the 2D image is applied to the 3D object's surface by tiling or stretching. But what about more complex objects? If you look at the texture image for the enemy or props you have imported into your project, you will probably see something that looks more like a jigsaw puzzle!

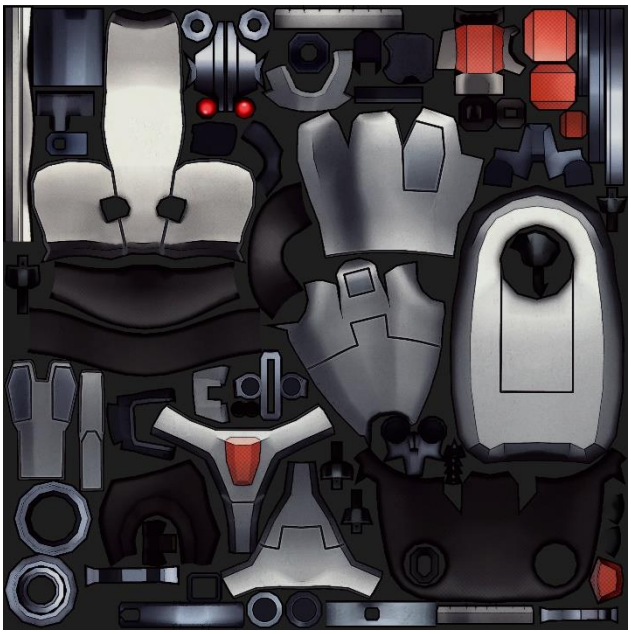

Figure 1: Robot Kyle (Enemy) Texture

When texturing more complex 3D objects, *texture coordinates* are used. In a texture, instead of X and Y, U and V are used. Therefore, it is sometimes called *UV coordinates*. Specific coordinates in the texture correspond to specific coordinates on the 3D object's surface. This process of determining which areas of the texture are drawn on which areas of the surface of the 3D object is called *texture mapping*. In essence, what is going on is the surface of the 3D object is unfolded or unwrapped into a 2D image (think of the texture as wrapping paper and the 3D object as the box being unwrapped). That is why it also called *texture unwrapping*. How do you do this? It's all done in the 3D Modelling software. You won't be doing this in this course but it is worthwhile understanding what it is.

Texture Compression:

3D Graphics hardware are not designed to use Texture files directly. Instead, they are designed to render textures stored in one of many texture compression formats. Textures take up

an enormous amount of memory and disk space. By compressing them beforehand and then de-compressing them in real-time right before they are rendered on screen, both memory and disk space are reduced. The disadvantages are that de-compressing them requires some processing power and there is typically loss in texture quality. By default, Unity will compress any texture image that is imported into your project to a format called DXT1. For a list of other formats, refer to this page: https://docs.unity3d.com/Manual/class-TextureImporterOverride.html

Skyboxing:

You can think of a Skybox as a giant cube that surrounds your 3D world. Its purpose is to fill the camera's view where none of your 3D objects fill the Scene. Think of it as the background of your Scene. No matter what way the camera is facing and if there is empty space (no 3D object blocking the view) the Skybox will fill the rest of the view. Unity will render everything else first and then render the Skybox behind all the other 3D objects. Currently, your Scene is using the default Skybox. This Skybox is just a simple blue gradient. You'll notice that the horizon is blurry and that the Skybox goes from a light blue to dark blue giving it a distant, 3D look. This is all handled by Unity automatically.

Like with textures, in a professional game, a 3D Artist would be hired to design a custom Skybox. For our purposes, we'll download pre-made ones. You can find Skyboxes on the internet. One place is http://www.custommapmakers.org/skyboxes.php.

Once you have downloaded a Skybox you'll notice it comes with six images (one for each side of the Skybox cube). It will also come with another file that has a *.shader* extension. This is a custom Shader used with this particular Skybox. Unfortunately, it isn't written specifically for Unity and won't work without significant editing. Instead, we'll use a built-in Shader that Unity provided for Skyboxes. Therefore, you can safely delete this file. Shaders are explained below.

You'll apply a new skybox to your 3D Scene in Task 4.

Shaders:

A *shader* is a script that calculates the colour of each pixel that is rendered in the 3D view based on the lighting and material. The script is written in a language called **HLSL** which stands for *High-Level Shader Language*. We will not be studying this language in this course. Unity comes with a built-in *Standard Shader*. The Standard Shader is designed to deal with most real-world hard surfaces. It uses a technique called *Physically Based Shading* that mimics reality as close as possible.

If you are trying to mimic reality the Standard Shader does the job. You'd want to use a different Shader if you are looking for a special effect or some sort of unnatural look to a material. Non-hard objects would also require a different Shader.

So far in your project, you've been using only the Standard Shader. The Skybox you'll add though is a non-hard, non-standard material and thus requires its own Shader.

You'll apply a skybox Shader in Task 4.

Tasks:

Tasks Part 1 – Import Lab 5's Assets into a New Project

1. You will start a new Project for Lab 6 but you will want to import the Scene and all the other Assets from Lab 5. First, open your Lab 5 project in Unity. Go up to the menu bar and select *Assets -> Export Package*. This should open the Exporting Package window:

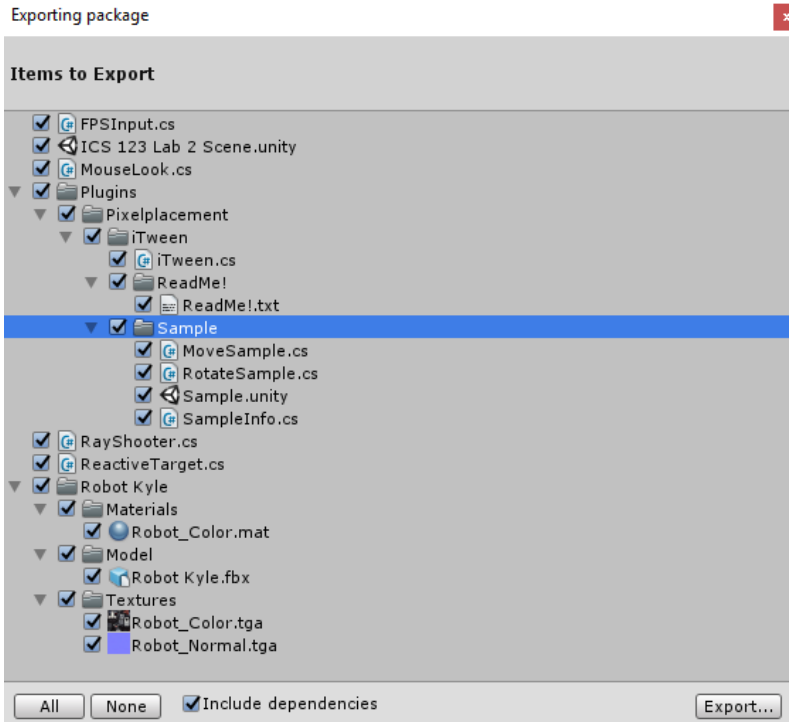

Figure 2: The Exporting package window

   Ensure that all your Assets are checked and that *Include dependencies* is checked. Hit the *Export…* button.

2. Save this package under your Lab 6 folder (create one if you haven't yet!). Save this new package as 'ICS123Lab5Assets'. A *.unitypackage* extension will automatically be added.

3. Now go up the menu bar and select *File -> New Project*. Name this new project 'ICS 123 Lab 6'. Hit the 'Create project' button. This will close your current Lab 5 Project and relaunch Unity with a new empty Lab 6 Project.

4. Once the new empty project is loaded in Unity, go up to *Assets* and select *Import Package -> Custom Package*. Browse for the package you created in Step 2. An Import Unity Package window will pop-up. Ensure everything is checked and hit the Import button.

5. In the Project Tab under Assets, you can double click on your Scene to load it up into the Scene Tab. You're done!

Tasks Part 2 – Texturing the Floor and the Walls

1. First, get a texture that will be used for the floor and a texture that will be used for the walls. You can either use the provided ones in D2L or sign up for an account on textures.com and download ones from there. If using the provided textures, skip to step 4.

2. Start with the floor texture. Go to textures.com and sign in. Along the left-hand side are the Categories you can choose from. There is a *Floors* Category but you don't have to pick that one. Some of the other categories also contain textures suitable for the floor. For example, the floor texture uploaded to D2L was chosen from Metal->Floors->Rusted. Whatever category you choose, make sure to select the *Show Seamless Textures Only* option (the tileable textures).

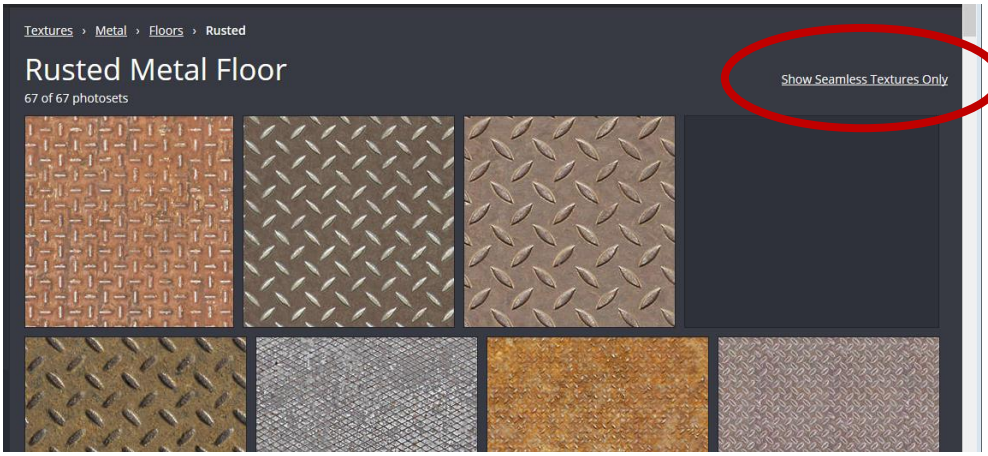

Figure 3: Rusted Metal Floor Category with Show Seamless Textures Only option circled

3. Select the Texture you want and then click on it. If the texture you chose has multiple images, make sure to select the seamless one. Download the texture that has a small size (most larger sizes cost money). Follow a similar procedure to download a texture that you can use for the walls.

4. Even though you downloaded the small sized textures, they probably have more pixels than you want or one of the dimensions isn't divisible by two. In addition, it's a JPG and not a PNG. To fix these issues, we need to re-size the image and save it as a PNG. We'll use the open source 2D Image Editor called **GIMP** (GNU Image Manipulation Program).

5. Open up GIMP. It may open several windows but the one you want to pay attention to is the central one that has the menu bar. Go up to File in the menu and open up the texture you'll use for the floor.

6. Once you have the texture open, select *Image->Scale Image*. This will open the scale image window.

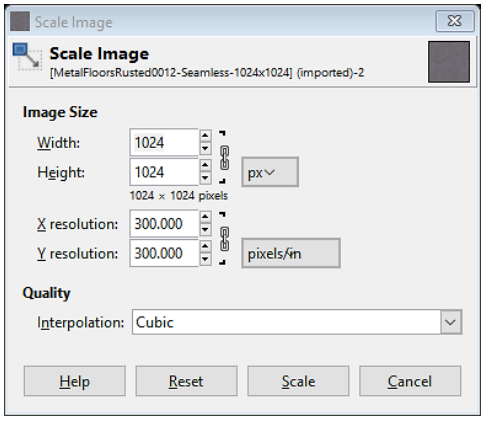

Figure 4: Scale Image Window

7. Change the Image Size to 512x512 or 512x256. Note that if your texture isn't perfectly 1:1, 2:1, or 1:2 you may need to press the chain icon beside the Width and Height boxes. This allows you to scale the width and height independently. The downfall is you will end up stretching or shrinking one dimension of the original image. Leave the resolution setting the same but change Interpolation to *Sinc (Lanczos3)* (which gives the best image quality when shrinking).

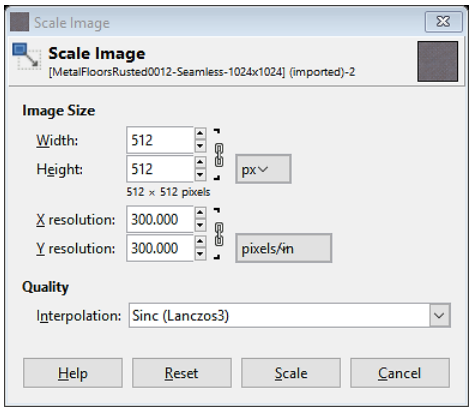

Figure 5: Texture with Image Size and Interpolation Adjusted

8. Hit the Scale button. This will re-scale the image. Next, you want to save it as a PNG. Go up to File and select *Export As…*. This will open the Export Image window. To save the image as a PNG, just change the extension of the image to PNG.

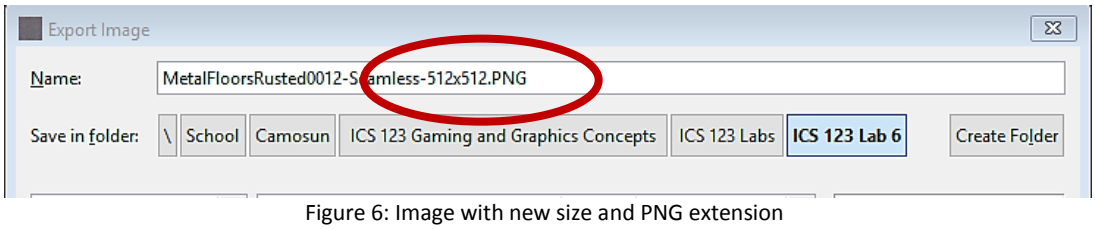

Figure 6: Image with new size and PNG extension

9. Hit the Export button at the bottom. This will bring up the Export Image as PNG window. Leave options as Default and hit Export. You can close GIMP. When it prompts you to save changes to your original image select Discard Changes (so you have the original as a

backup). Repeat steps 4 to 9 for all the textures you downloaded.

10. Now you need to add your textures to your project. In Unity, go up to *Assets* and select *'Import New Asset…'*. Navigate to one of your textures to import (either wall or floor). Do the same for the other texture.

Tasks Part 2b – Applying the Texture:

11. Now you need to create the materials for the textures to attach to. Simply grab the texture you imported for the floor in the Assets window and drag it onto the floor in the Scene Tab. Several things will happen: first, a new Materials folder will be created in your Assets window; second, a new Material will be created inside this folder with your texture attached; and finally, the Material will be attached to the floor and be shown in the Scene Tab.

12. Currently, it probably doesn't look that good. That's because tiling is turned off. Click on your material and in the Inspector Tab look for the Tiling X and Y values under *Main Maps* (**not** Secondary Maps which allows for advanced secondary texture effects). When these are set to 1, it means no tiling. If you set them each to 16, for example, it means render 16 times by 16 times of the texture on the floor. Experiment with a number that looks good to you.

13. Now, grab the texture you imported for the walls and drag and drop it onto one of the walls. Like the floor, this will create a new material in the Materials folder. For the rest of the walls, you want to drag and drop this material on them and **not** the texture (since the material has already been created). As you may have now guessed, this means you could use different materials (and thus different textures) for different walls!

14. Finally, adjust the tiling for the Material(s) you used for your walls so that they appear good! It probably can be a smaller number than the floor since it covers less of an area. You may also only need to tile in one direction (for example X=8 and Y=1).

Tasks Part 3 – Doing a Bit of Housekeeping

1. At this point, your Assets window is probably starting to look quite cluttered. There are several materials, scripts, and textures. It would be nice to de-clutter this window a bit and organize each of these assets in folders. As long as you move assets into folders ***WITHIN*** Unity, Unity will know where the assets are located in your project. In Task 2, a Materials folder was created when you applied textures. Start by moving the *LaserColour* material to this folder. All you have to do is drag and drop it on the folder right in the Assets window.

2. Next, create a folder called *Scripts*. Right-click in the Assets window to bring up the Assets pop-up menu. Select *Create->Folder* and rename it to Scripts. Drag and drop all your Scripts into this folder.

3. Next, create a folder called *Textures*. Once again, right-click in the Assets window to bring up the Assets pop-up menu. Select *Create->Folder* and rename it to Textures. Drag and drop all your texture assets into this folder. Once you have finished this, your Assets

window should look like the figure below.

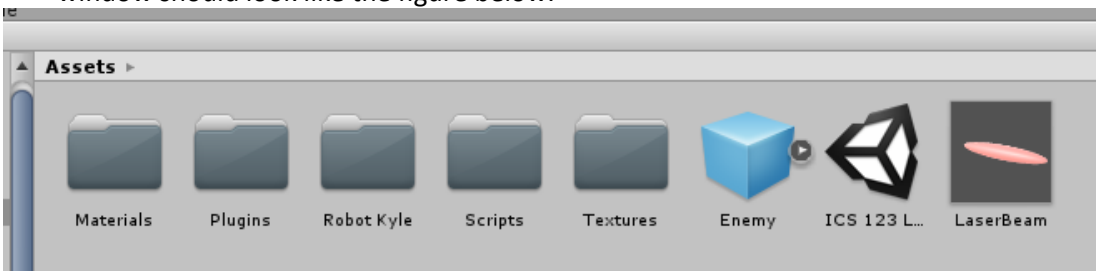

Figure 7: The de-cluttered Assets window

4. This should leave only the Scene asset and custom 3D object assets as root objects in your Project. All the other assets are now in sub-folder. You can play your Scene just to make sure everything still works. As long as you did all the folder and asset operations within the Visual Editor, Unity will keep track of your project files and everything should still work!

Tasks Part 4 – Adding a New Skybox

1. Start by creating a new material. Double-click on your Materials folder to open it up in the Assets window. Now right-click and select Create->Material. Name it *Skybox*.

2. As discussed in the *Preparation* section of this lab, the Skybox requires a different Shader than the Standard one. Select your new Skybox Material and in the Inspector Tab, you'll see a Shader dropdown. Select Skybox->6 sided. You'll notice that there are now six texture slots where you'll place the texture images that make up your Skybox.

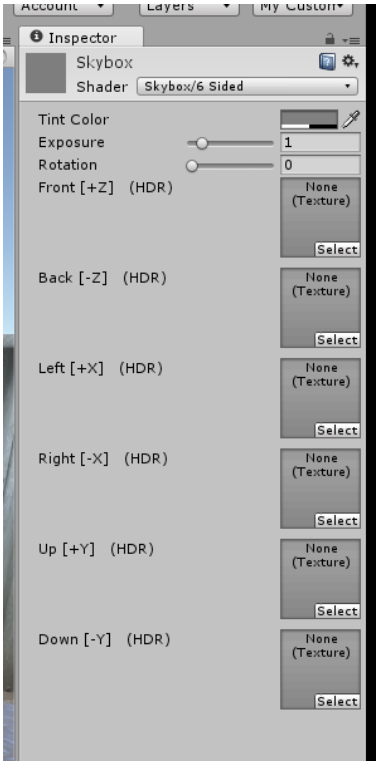

Figure 8: The 6-sided Skybox Shader

You can see that each of the six textures is for one of the X, Y, or Z axis in either the positive or negative direction. The HDR stands for *High Dynamic Range*. In a nutshell, HDR allows details to still be seen in a Scene where there is a high contrast ratio. It basically allows a greater range of bright and dark in a Scene without anything appearing "washed out". In the case of the Skybox, it should allow for bright skies and dark shadows without any 3D objects losing detail.

3. The next thing to do is to import the textures for the Skybox. Double-click on your Textures folder in Unity to open it up. It might be a good idea to create a new sub-folder just for the Skybox textures. Right-click and select *Create->Folder*. Name this Folder *SkyboxTextures*.

4. Now import each texture. There are two ways to do this. You can use the same method that you used to import your textures for Task 2 or you can shift select all your textures in Windows Explorer and drag and drop them into your *SkyboxTextures* folder in the Unity Visual Editor.

5. You'll need to select each skybox texture and change the Wrap Mode from Repeat to Clamp. Normally, repeat works when tiling textures because it causes the textures to blend together. For the skybox however, this tends to cause a faint line in the sky. When you change each one, make sure to press the *apply* button.

6. Now you can drag and drop each texture into the appropriate square of the Inspector Tab matching each file with the proper axis.

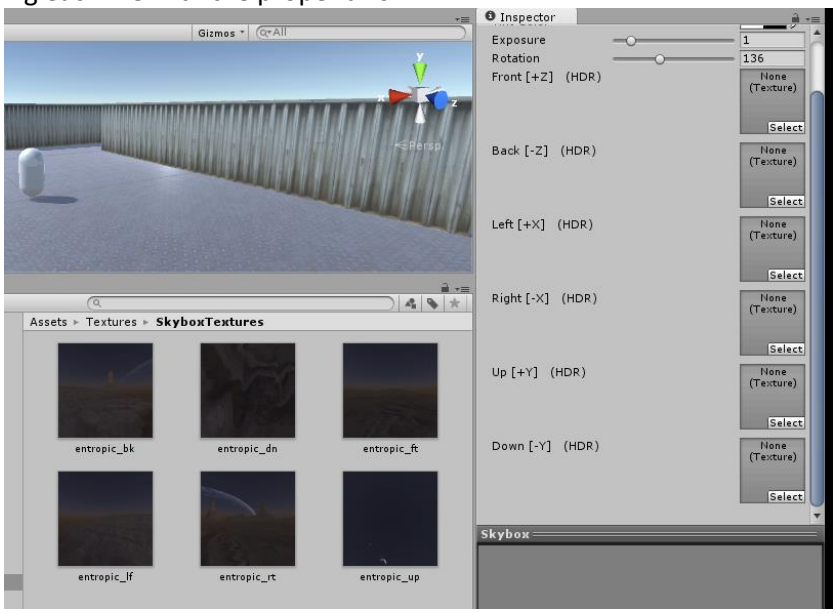

Figure 9: Skybox Textures

In the example above, the _bk file refers to Back (-Z axis), _dn refers to Down (-Y axis), and so on.

7. After you have assigned all of the textures, you can apply your new Skybox. Go up to the menu bar and select *Window->Lighting*. Right now the Default-Skybox is selected, change this to your Skybox Material. Click on the small circle to the right of the box and select

your Material. You should see the new skybox in your Scene Tab.

8. Try playing your Scene! The new Skybox should add a lot of atmosphere to your game.

Tasks Part 5 - Adding 'Custom' 3D Models to Your Scene

1. As discussed in the *Preparation* section of this lab under *Art Assets*, designing custom 3D models is a time consuming process that is done in 3D modelling software (Blender, Maya, or 3ds Max) by a 3D Modeller. In a professional setting, several 3D Modellers (also sometimes called Asset or Content Designers) would create custom 3D models for the game. In our case, we can use free, pre-made models available in Unity's Asset Store (just like you did for the Enemy). Hit Ctrl+9 to open the Asset Store Tab.

2. Use the Asset Store's Filter tools to search for 3D Models. For example, pressing the *Free Only* button and selecting *3D Models->Props* in the menu on the right-hand side should give you lots of possible models. Don't get too ambitious! Pick something fairly simple.

3. When you find a Prop you like, download and import it into your project. Make sure you select everything: the model (.fbx), textures, materials, and prefabs.

4. For the example in this lab, I have downloaded and imported a 3D model simply called *Barrel*. When I click on the Prefab, it has the default Transform component and an Animation component. I don't want the Animation component (I won't be animating it) but I do want a Collider so the Player and Enemy can't walk through it. To edit a Prefab, you have to drag it somewhere into your Scene.

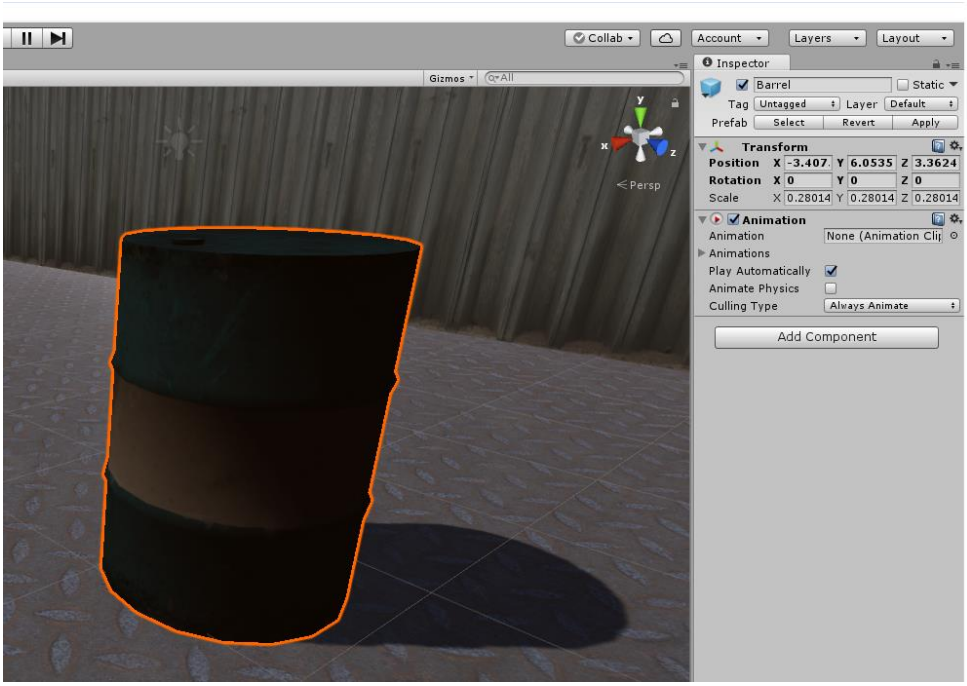

Figure 10: Barrel Prefab dragged into Scene showing Animation and Transform Components

5. I'm going to remove the Animation component (recall: click on the small gear at the upper

right of the component in the Inspector Tab) and add a Collider component. I Click on *Add Component -> Physics -> Capsule Collider*. I could have used a Mesh Collider but they require more processing power and I am already using one with the Enemy. In the case of the Barrel, I found a Radius of 2, a Height of 5, and a Y of 2.5 worked well.

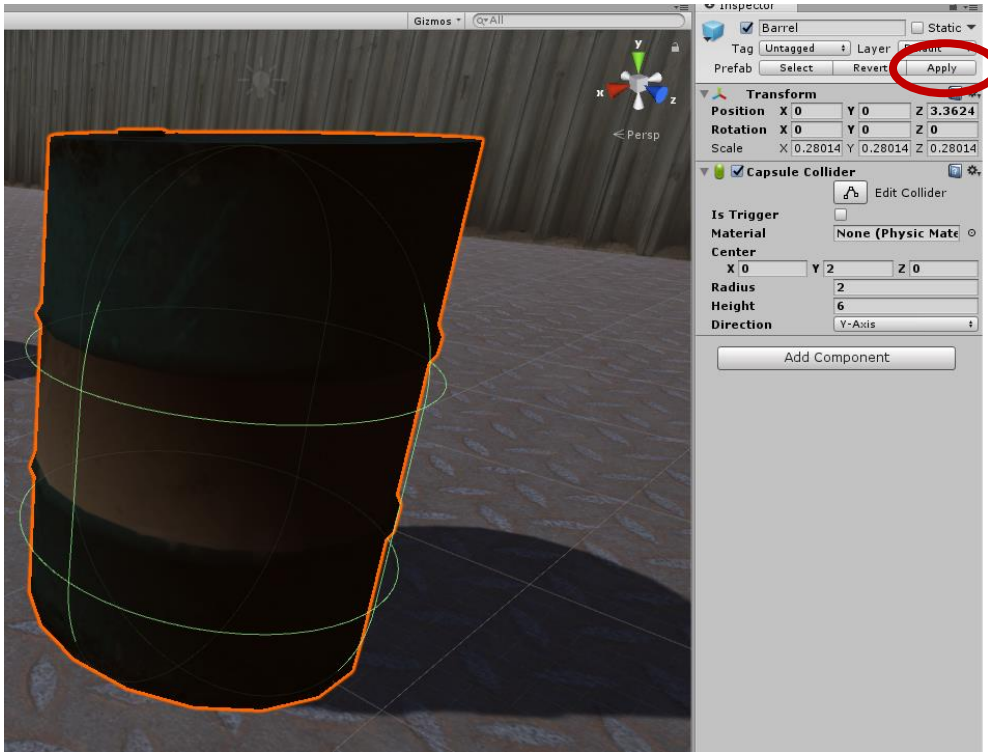

Figure 11: Barrel Prefab with the Capsule Collider Added

6. Since I have made changes to the Barrel Prefab I need to re-save it with the changes. I can either select *GameObject -> Apply Changes to Prefab* or press the Apply button in the Inspector Tab of the Prefab (circled in the figure above). Now I can place several copies of this Prefab in the Scene.

7. Now it's your turn! Add a Collider to your Prefab if it doesn't have one and place a few Props around the Scene.

8. The figure on the next page shows how I added a stack of barrels to my Scene. You'll notice the color of the barrels are different. This particular asset included three different materials so I have edited the model (not the prefab) of the different barrels to use different colors. The barrels were rather dark so you'll notice I also added a point light near them to lighten them up (if you decide to do the same, refer to Lab 2). Lastly, in the Hierarchy Tab I selected *Create -> Empty* to create a new empty *GameObject* called *BarrelStack* and I dragged the 3 barrels into it. You did the same back in Lab 2 for the Room. With this in mind, you may want to stack some of your Props.

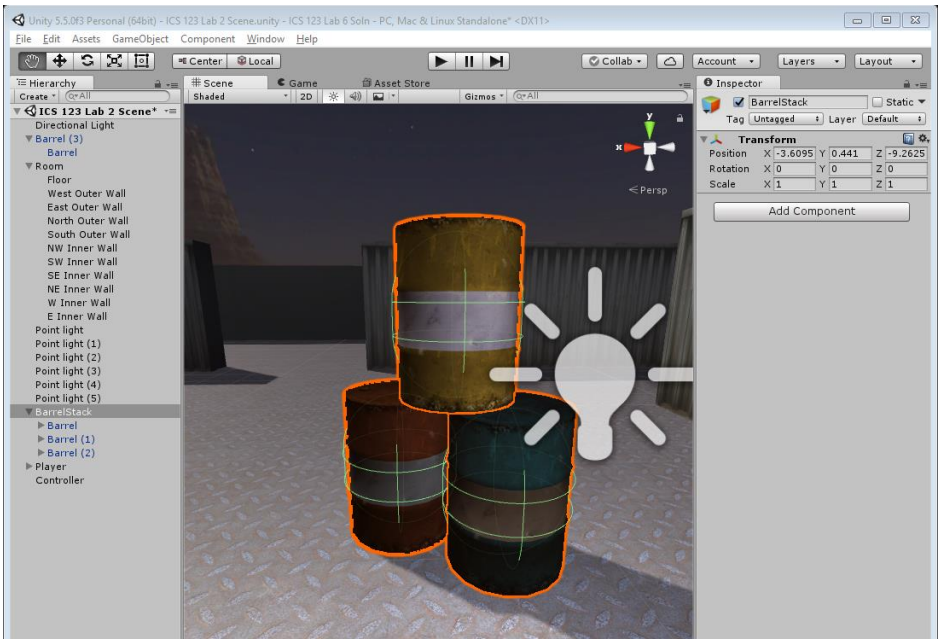

Figure 12: BarrelStack showing different Materials and the added Point Light

Tasks Part 6 – Adding a Particle System

1. The last visual content you'll add to your Scene is a particle system. Go up to the *GameObject* menu item and select *Particle System*. Right away you should see several flying white blobs with a small Particle Effect window in the lower right corner. Pretty neat!

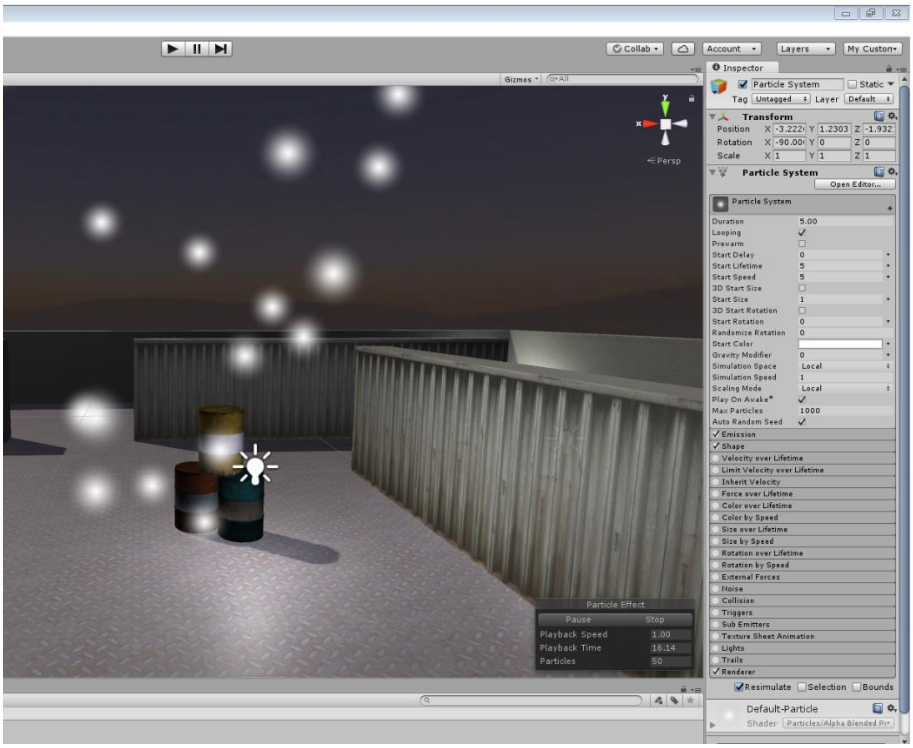

Figure 11: Particle System. Notice the *tons* of settings in the Inspector Tab

2. You are going to change some of the settings to change the Particle System to look like a flame. As you change settings, you'll see the result in real-time in the Scene Tab. Each of the grey boxes in the Inspector Tab are called Modules and contain several settings. The first module, called Particle System, is already expanded. Leave everything as default here except change the *Start Lifetime* to 1. This means each particle will only last for one second. You can minimize this module by clicking on its grey box at the top.

3. Now click on the *Emission* module. Change the Rate over Time to 200. That's the number of particles emitted per second.

4. Next expand the *Shape* module. This defines how the particles are emitted. Change this to Box. All other settings here can remain as default.

5. Now check the circle for the *Size over Lifetime* module (which activates it). Expand this module. This controls the size of particles over their lifetime. For a fire, we can reduce the size of its particles over its lifetime to get that 'pointed' effect. Click on the grey box where it says Size and you will notice a red line appear. At the bottom of the Inspector Tab, you'll also see a *Particle System Curves* window.

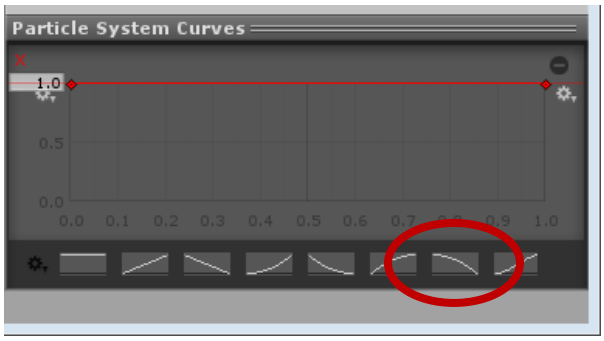

Figure 12: Particle System Curves window

For the effect we want, choose the pre-set circled in the figure above. This means that the size of the particles will reduce linearly over its lifetime. After applying all these steps your 'fire' should look something like the figure below.

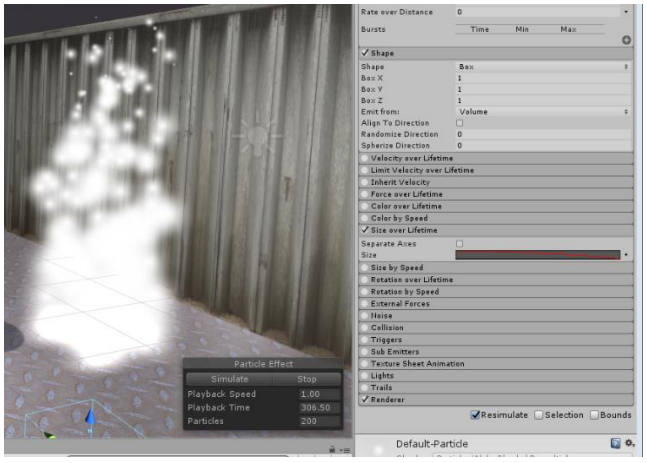

Figure 13: The White Fire

6. Like any other 3D object in the Scene, you can apply a texture (via a material) to a Particle System. Import the ParticleFirecloud.png image into your Project (if you forget how to do this, refer to Tasks Part 2b).

7. Now drag this image from the Assets window onto the white flames in the Scene Tab. This will automatically create a material and attach the texture image to it (if this sounds familiar, it should. You did this before when applying textures in Tasks Part 2b). Don't worry if your fire looks a little messed up right now!

8. In your Materials folder, select the *ParticleFirecloud* Material (this was automatically created for you in Step 7). Change the *Shader* to Particles/Additive. Then, click on the grey box beside the Tint Color label. This will bring up a Color Picker. Enter 118 for R (Red), 16 for G (green) and 16 for B (blue). Leave the A (alpha or transparency) at 128. This will tint the color of the texture giving a neat two-tone color to the Particle System making it appear more realistically like a fire.

9. Now place your fire somewhere in your Scene. In the example figure below, I placed it in with my barrels and shrank it slightly (Scale to 0.8) giving this flaming barrels look!

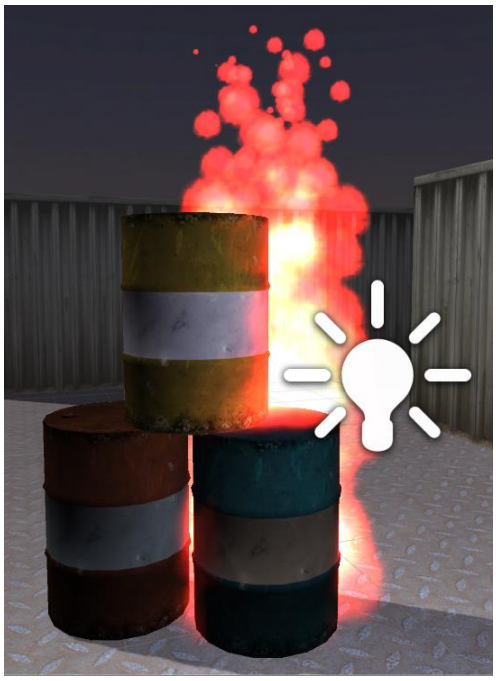

Figure 14: Flaming barrels!

Submission:

Demo to the Lab Instructor:

- **Show that the Floor and Walls have textures.**
- **Show that you have some 3D Props in the Scene and the Player properly collides with them.**
- **Show that you have added a Particle System.**

- A successful demo scores you 10 marks.