1 Abstract

The focus of this project is primarily an exploration of the Unity game engine with emphasis on scripting in C and rendering using the HLSL language or Unity’s shader graph, which leads to the learning of other techniques and theory within the computer graphics domain.

Rendering is the process of generating 2D or 3D images from models by means of a computer program. The graphics pipeline describes the necessary steps a graphic system needs to go through in order to actually render an image, and I am concerned with the vertex and fragment processing steps, which utilizes shaders to describe the traits of either a vertex or fragment (a pixel in the screen space) accordingly (Figure 1). A shader is a program that runs in the graphics pipeline to inform the computer’s GPU on how to render each pixel based on information provided in the model and are used to control color, lighting, shadows, and other properties. It can be used for effects such as bump mapping, stylized toon shading, and texture mapping.

I wanted to strengthen my knowledge in graphics while gaining applicable skills in the VR/AR and gaming industries. With that in mind, I decided to create a game in Unity with an emphasis on rendering and game play. I initially planned to spend more time on rendering rather than get an actual game together, but scripting and understanding the game engine become a very large and timely focus of the project as I went to achieve some of my initial goals. Because of that, I decided to also aim for an MVP for my game where the player has to grow certain trees in order to attract and collect certain animals.

2 Introduction

This project was a collaboration between Julia Shi and me. We decided to make a game to combine our interests in computer graphics, art, and games. The goal of the game was to collect different animals by growing different trees that would attract them. It would also use real-time weather data to recreate a scene that reflects the player’s current environment and render the animals accordingly to my stylistic choice (e.g. A cat on a sunny day would be slightly glowing to make it look like light). We were unable to recreate this fully, but built the foundation necessary to demonstrate basic functionality of the game.
On my side, I was responsible for all the rendering along with the game functionality. I used Unity 2018.4.9f1 along with the Lightweight Render Pipeline (LWRP) which was sufficient for my needs. More details will be explained in the sections below.

3  Rendering

This section deals with the various shaders I made either manually through vertex and fragment shaders or visually through the shader graph. There are many different ways to make shaders in Unity. The vertex and fragment shaders are the lower level method where all the setup and calculations must be done by hand. The vertex shader is a program that runs on each vertex of the object. It takes the vertex from its object space (where all the vertex are located relative to its center) to the clip space, which is how the player sees it from their screen/view. The fragment shader is a program that runs on every pixel in the screen to determine its color based on calculations done on the objects in the scene. [5]

Unity also has something called ”surface shaders” that are not really shaders, but higher level templates that automate the repetitive code and functions to handle lighting and shadows. It is then compiled into the standard vertex and fragment shaders. All surface shaders can be written as vertex and fragment shaders, but it is not necessarily the case the other way around. I did not look into writing surface shaders for my project.

The other method of writing shaders that concerns this project is the shader graph. This allows the developer to visually make shaders in real time by creating and connecting nodes in a graph network. It was released in Unity 2018.1 and works with the Scriptable Render Pipeline released at the same time. More can be learned about the Scriptable Render Pipeline at the resources below. [2]

3.1  Toon

The game is lighthearted, so the first shader I created was a toon shader (Figure 2).

\[ L = k_a I_a + k_d I_{max}(0, n \cdot l) + k_s I_{max}(0, n \cdot h) \]

[3]

\( k_a, k_d, k_s \) are the coefficients (essentially the color) for the ambient, diffuse, and specular model. \( I_a \) is the ambient light intensity while \( I \) is just intensity of the light source. \( n \) corresponds to the normal of the point being evaluated and \( l \) and \( h \) are the light direction and the half vector between the view and light direction, respectively.

Using the standard Lambertian, Blinn-Phong, and Ambient shading models above along with, I got a sphere rendered smoothing. But to achieve the toon look, more calculations had to be done in the fragment shader to set a threshold where the color would segment. I used a smoothstep function for that purpose.

Because custom shaders are no longer supported in the LWRP, the shader will actually not render in the source code. I included it anyways to show the code. Figure 2 above was taking in the built-in pipeline.
3.2 Weather Based

All four shaders below were created using Unity’s shader graph. It was extremely fun exploring the capabilities of the graph and seeing the material come to life right away. I decided to create shaders for four weather types: Sunny, cloudy, rain, and lightning due to either their high probability of being chosen or the capability of creating an interesting material.

![Figure 3](image1.png)

Figure 3: From left to right, the shader goes with a corresponding weather: Sunny, Coudy, Raining, Lightning storms

The sunny shader were the first one I worked on. I chose a more simple shader to start off with as I explored the Shader Graph. It uses emission to give off a orange/yellow glow combined with a sin function for time to fade in and out.

I worked on the dissolve function next. I used a sample black and white noise texture to use as the base for the shader. There is two parts to this: having the material “dissolve” as in become transparent and setting the edge of the material to glow. I set the noise texture as the alpha channel, which results in all pixels having a alpha value from range 0-1. Then I used another sin function that I remapped to interpolate from value 0.4 and 0.8. The reason being is that I then used the value output as the alpha clip threshold, which indicates a the minimum value necessary to be rendered. If say for example, the alpha threshold is 0.7, the first four pixels in the Figure 4 would be rendered, but the fifth pixel with an alpha value of 0.6 would not reach the threshold and be invisible in the scene. With that completed, I used the same noise material with a slight delay to add a glowing emission along the disappearing edges.

![Figure 4](image2.png)

Figure 4: The alpha map for an example image.[1]

The rain was by far the most extensive out of all the shaders. It took me a long time, but after searching around, I decided to use a render texture, a texture that can be rendered to, to emulate rain falling on objects. The first thing I did was set up particle systems that looked like rain falling, which include rain droplets fading in and out and trails of water sliding down. Once I had my particle systems completed, I then
added a camera into the scene and set the layer for both the particle system and camera to TransparentFX so the main camera would not capture it on screen during the actual game. I also had to change the culling mask on the main camera to avoid looking at the TransparentFX layer.

![Image of camera capturing rain](image)

**Figure 5:** The view from the rain capturing camera

I set the background of the camera view to be a solid color and set the target texture to be the render texture I created earlier. That means the texture reflects what the camera is currently seeing in the scene. With my texture established, I moved on to the shader graph and using a mixture of normal maps and smoothness, I was able to replicate the look of rain falling down on objects.

![Shader graph for rain texture](image)

**Figure 6:** Shader graph for the rain texture

After completing the rain shader, I went to work on the final cloud shader. I decided to try using new nodes and decided to use an actual cloud texture and move the material using tiling and offset. I combined a constantly moving uv map with the actual texture itself to have the material move in the game. I wanted to add more life to the object, so I tried out using normal maps, position and vertex displacement, but I decided to stick with a regular normal map for my need—although the final result looks more like a moon rather than a cloud like material.
4 Scripting

The rest of the project focused on scripting to add game functionality. This was done all in C#.

4.1 Weather Data

I wanted to make it an immersive game, so I immediately wanted to use real data information from the player’s location to render the objects accordingly and add ambiance to the scene. This was the first script I made and took me quite a bit of time. This script relied heavily on making many API requests, but many of the resources I turned towards were already deprecated and no longer applicable. The first step required the player’s IP address. I spent a long time trying to use Unity’s Network System to find the IP address, but there were problems accessing the IPv4, so I ended using ipify’s API to get the address. Using the IP, I found my longitude and latitude using geoplugin and then used that to make requests to OpenWeather’s API. I extracted the main weather from the request and saved it as a public variable for my other scripts to access. I had to import JSON Object from the Unity Asset Store to handle and parse the JSON I received from the requests. It was my first time working with API’s as well, so I got the chance to learn about authentication keys and requests.

4.2 Play Movement

This was one of the most time consuming part of the game, and required me to go through multiple tutorials and explanation to understand the physics engine in unity. I have a player object in the scene that interacts with the animals and collects them, but I have to be able to detect collisions and not pass through them. I initial moved my characters by moving their transform component (determines position, rotation, and scale) with the translate function and checked each animal in their collider component as a trigger. The issue with doing this is that using the translate function on the transform overrides any physics and does not work colliders, so my player object would walk through the animals. [4]

With that, I decided to use use the rigid body component to move and rotate my player. I used a rigid body on my animals as well and unchecked the trigger option from the collider component, so rather than check for a trigger, I looked for a collision instead. Any object with both a rigid body and a collider is a dynamic object, but anything with only a collider is a static object. This worked in the sense that I no longer ran through the animals, but the force of collision would knock both the player and animal object around in the scene. I realized the solution was to make both the animal and player into kinematic objects, which is an option under rigid body. A kinematic rigid body does not react to physics forces within the game and moves by using their transform. [6]

4.3 Snapshot Camera

Since the objective of the game is to collect animals, I wanted to have a feedback system that informed the player when they collected an animal and some storage capability.

I attached a secondary camera to the play object that was hidden on another layer much like what I did for the rain drop texture. Every time the play bumped into an animal, it would activate the snapshot script that turns on the secondary camera to create another instance of the animal object in the same layer as the secondary camera. Using the animal’s transform, I scaled and rotated it according so the secondary can capture it entirely. Using another render texture, I saved what the secondary camera saw as an image and stored it in a resource folder for later use. Once the image was saved, the newly instantiated object is then deleted and secondary camera turns off leaving the scene it is original state.

5 Conclusion

The end product is a very simple game, but laid down the foundation for further development. I took one of Julia’s tree script and incorporated into the game. In the demo video, the tree starts growing on play and once the tree finishes growing, it attracts an animal object (which is just a cube). At the time of the video,
it was raining in New Haven, so the object is being rendered with the rain drops shader. Bumping into the
animal saves an image of it locally.

6 Plans for the Future

There are a lot of functionality I want to add to the game. I spent a lot of time trying to get the photo
album to work, but was unsuccessful. I finished the setup for the photo album with the UI components such
as the button, the photo slots, and the grid layout, but ran into issues loading the images from the resource
folders. I want to finish developing the photo album and establish a saving method for the trees and animals.
I created and tested a basic storage mechanism for cubes, but that was a bit too generalized for my needs.

I also want to explore shaders more like I initially intended. The shader graph provides a simpler mean
to create shader, but there are still many nodes I have yet to use. I want to finish setting up the entire scene
with more weather components to add to the ambiance of the scene. I also want to go back deeper to the
vertex and fragment shaders. While it is nice that the shader graph is very high level, I think being able
to manually write vertex and fragment shaders in HLSL will provide me with a deeper understanding of
the graphics pipeline and how things are rendered. I have been exposed to the shading models before, but
writing the toon shader really solidified my understanding.

7 Acknowledgement

I want to thank Prof. Rushmeier for her guidance and support this semester. I also want to give thanks
to Bobby Berry who gave advice on Unity at the start of the semester. This helped greatly in preparing a
proposal and idea for the class.

References


