Computer Graphics Assignment 2 Report

Benjamin Le

Escape from Slenderman

I decided to base the theme of my OpenGL assignment around Slenderman, as it was the first idea that came to mind when reading the narrative of the specification sheet. The structure of my assignment came from “sample2” which was provided to us from the first practical, and so a lot of the key elements in the creation of my assignment came from ideas and features used within this sample code. Sample2 provided a good base to start off with and build upon, so I decided that the first step would be to create some trees and to start building the world.

The creation of the first tree was based around how the wooden table was created in sample2. The tree needs a scale and position array, containing vectors that make up the look of the tree and where that vector is located to form the shape of the tree. As seen in Figure 1, my tree contains two vectors to create the shape of the tree. One vector for the tree trunk and another vector for the tree leaves on top. The scale array is what I use to scale the tree trunk and tree leaves to form this shape, with the tree trunk being a thinner but tall box and the tree leaves being a cube-like shape, having the same dimensions in the X, Y and Z coordinates. The position array is what is being translated to get the tree leaves on top of the tree trunk.

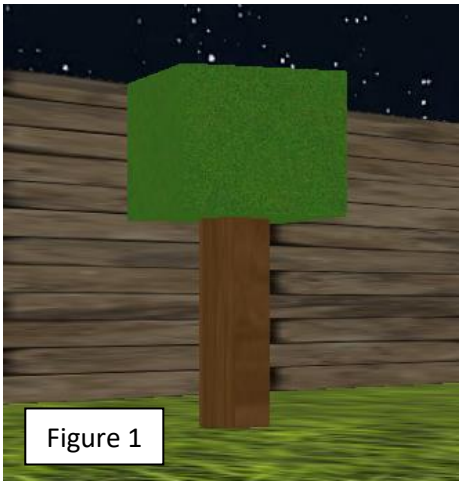

Figure 1

The idea behind making multiple trees was to create a new array, add in vectors with each vector being a different location on the map, and translating the array. I made an outer for loop which loops 50 times to create 50 trees and two inner for loops to create the tree trunk and the tree leaves. In order to render these, the tree models were created with a mat4 type model and to get the different textures to be added in, I had to bind the vertex array, assign a texture and to draw the arrays after.

I chose to hardcode the location of all 50 trees. At first it was just 20 trees and I found that hardcoding 20 trees wasn’t too bad, but eventually I decided to add more and more trees to make the world less empty. What I could have done for next time was perhaps create a random tree generator where I would get the computer to input random numbers in the X, Y and Z coordinates and so I wouldn’t have to think about

the location of the trees and play around with the numbers to try “evenly” spread out the trees on the map.

The concept of the wooden table being created was used for all composite objects, including the Slenderman. When the game starts, the Slenderman will move towards the position of the camera and if it gets to a certain distance within the current position of the camera, the player will lose the game and the camera will drop down to the floor and is prompted with the lose screen. I have a set spawn point (called slender_pos) where my Slenderman will always spawn at and I translate the position of it while the player is alive and has not won. The position of the Slenderman and the camera is constantly changing, and so in order to find that distance that is needed to make the player lose, a subtraction of the position of the camera and position of the Slenderman is sued to update the current position. This is then multiplied with a constant factor of 1.6 alongside delta time to make the Slenderman move. I simulated something close to getting the Slenderman to face the direction of the camera by subtracting the yaw value to make it 0 and rotating it along the Y axis. Yaw is a movement around the “yaw” axis which is pointing to the left or right of its motional direction. Figure 2 shows the Slenderman facing the camera, the textures used and the scale and position arrays being used to render the Slenderman and form the shape.

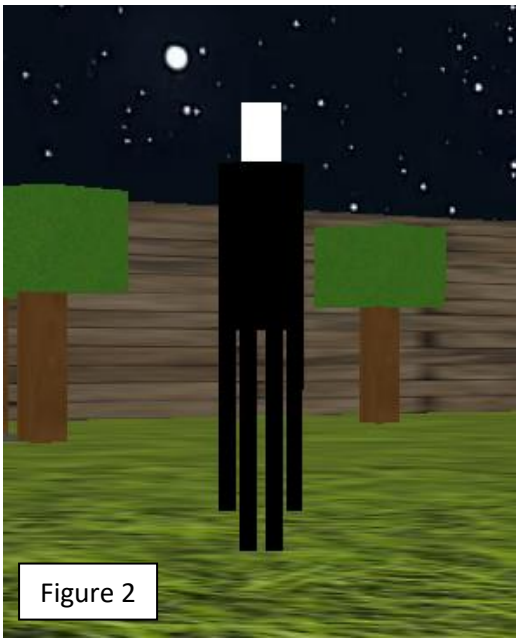

Figure 2

The lighting in the game starts off dark and can only be lit up if the player picks up the torch object (unless they turn on ambient brightness by pressing “O”). The torch object contains the light cube which is created using the lamp_shader vs/vf files. The original light cube starts off at a set position and is translated in front of the camera when picked up and acts as the players source of light. As shown in Figure 3, the light cube acts as the head of the torch and makes it easier for the player to find the torch while the world is dark. The concept of picking up the torch is based on sample2, where if the camera was close enough to the distance of the button, and a certain key was pressed, it would light up the area and activate an animation on the

Curtin University logo. With this idea in mind, I had the torch object at a set location and if the camera was within a certain distance of the torch, I would set a Boolean called TORCH_PICKEDUP to be true and if this Boolean is true (it is false by default), then I translate the torch object underneath the map to simulate that the object has “disappeared” and the light cube would then be translated in front of the camera. Figure 4 shows the torch object being translated underneath the map after being picked up.

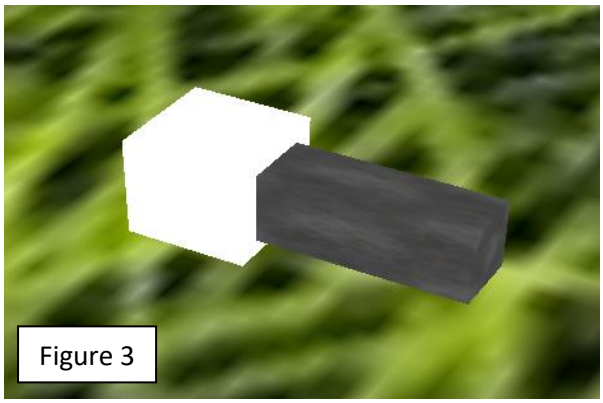

Figure 3

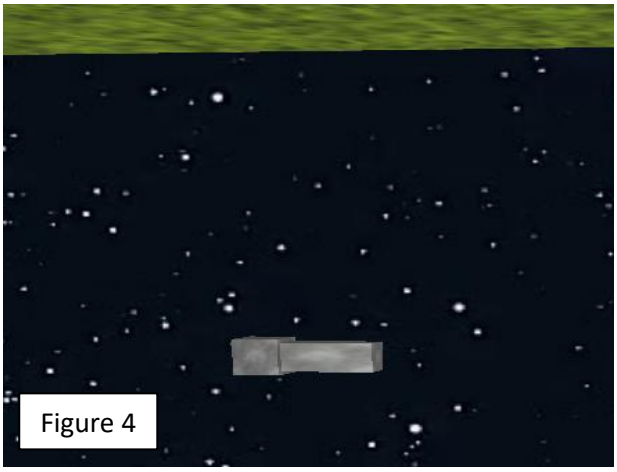

Figure 4

When the torch is picked up, the light cube follows the camera and emits a light resembling a “spotlight” effect, which only shines light in a circular shape with the outsides of the circle still being dark. This makes it realistic as if a real torch was being used in the real world. The torch also has different brightness levels by pressing the K and L buttons. K makes the light dimmer and L makes the light brighter. I chose to cap the brightness levels after 3 modes lighter and brighter of the original lighting settings. The values I used to change the linear and quadratic values of the lighting_shader was from a table found on LearnOpenGL. The algorithm to change the brightness of the torch using K and L was simply to have an integer variable called ATTENTUATION_NUM which decrements/increments depending on if the player pressed K or L. If the ATTENTUATION_NUM changed, it also changed the linear and quadratic values with a bunch of if and if else loops.

In order for the game to progress and eventually lead to a victory, the player must explore the world and find two hidden objects. The dagger (Figure 5) and the gun (Figure 6) are objects which also use the concept that the torch used when trying to pick up the objects. If the camera is close enough to the dagger or gun object, it is translated underneath the map to make it disappear from the game, and an integer

variable I made will be incremented. If this integer variable called ITEM_COUNT is equal to 2, a winning area that the player must go through will spawn in order to escape from the Slenderman.

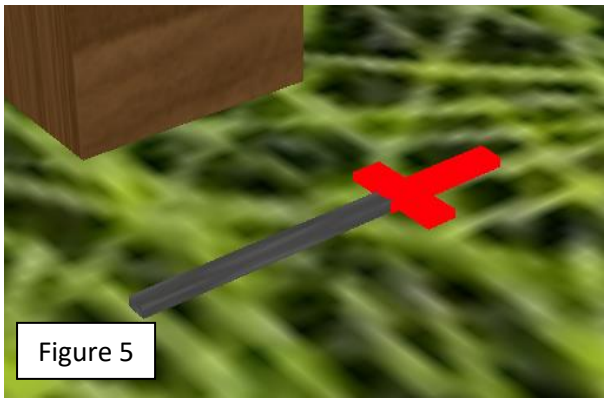
Figure 5

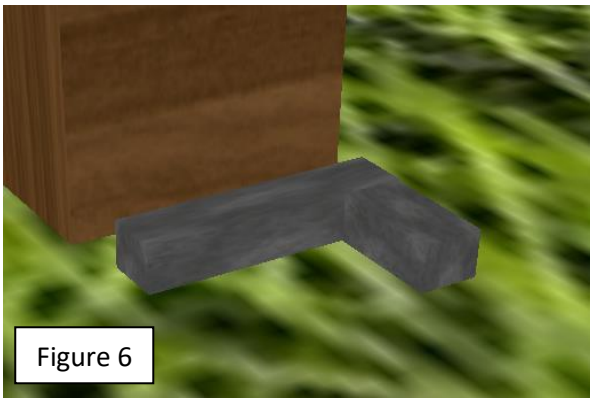
Figure 6

The game is won when the player has picked up both the dagger and the gun objects and walks through the area which only spawns when both objects have been retrieved. Inside this area, I decided to add in a “win cube” which uses the same logic as when a player picks up the torch, dagger and the gun. If the player is in range of the win cube, the camera will be translated outside of the map where the win screen will be shown. The win screen is simply a single vector which has a texture with words on it. This texture was obtained from a website called Textures4photoshop.com. When the player is inside the winning area, a Boolean variable is set to true, and if this Boolean is true, the Slenderman stops chasing the camera, the camera is translated to the win screen and the player cannot interact with the game anymore unless they press the “R” button to reset the game back to its original positions. Similarly, if the Slenderman catches the player, the camera drops to the floor and a lose screen is displayed in front of the camera. The player is unable to interact with the game unless they restart it.

In terms of animations, I just simply made the game start off with the tree leaves rotating on the Y axis and eventually once the player collects the two required items, the tree trunk will rotate on the Y axis but in the opposite way.

Creating boundaries so that the player doesn’t accidentally travel outside the bounds of the map was a simple concept to implement too. Originally I was going to use the same logic as the BUTTON_CLOSE_ENOUGH idea from sample2, where if the

camera was close enough to the button, then it could be interacted with. In this case, if the camera was close enough to the wall, the player cannot move past it by deactivating the movement keys at a certain range. An easier idea was to just limit the X and Z coordinates since the dimensions of the map was known. The size of my map is 60 in the X and 60 in the Z dimension, and so from the origin, it'll be 30 to the left, right, in front and behind the camera. The boundary object itself takes up a bit of space, so I made the boundaries for all sides of the map to be 29, and so if the camera in the X and the Z position reaches -29 or 29, then it will just set the X and Z position of the camera to be -29 or 29, meaning the camera cannot go past these bounds.

The majority of the composite objects I have modelled, use textures provided in the OpenGL folder, which contain specular textures. These specular textures are bound to the objects to give objects a realistic look. Specular simulates a bright spot of light that appears shiny objects and is visible when you go very close to an object. An example of where I used specular textures would any objects that use the wood texture.

Below are images of the rest of the models that I have designed.

- Graves. (Figure 7)
- Winning area. (Figure 8)

Figure 7

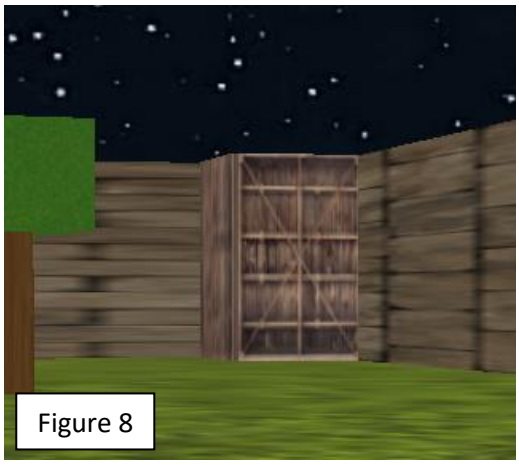

Figure 8

Below are the links used to create the OpenGL assignment and to create the textures with words used for the win and lose screen.

https://learnopengl.com/

http://www.textures4photoshop.com/textturizer/