CSU44052 – COMPUTER GRAPHICS

Megan Whelan Dalton | 16324322 | whelandm@tcd.ie

CONTENTS

CAMERA CONTROL & MOVEMENT

I chose to implement an FPS (first-person shooter) style camera, that allows the user to "walk" around the scene by using the mouse and WASD keys. This camera has a fixed value on the y-axis that never changes giving the illusion of the height for the user or player.

MOUSE TRACKING

I use the lookAt() function to create the view of the world that serves as a the camera. This function takes both the position of the camera, the target as defined by the mouse movement as well as an up vector. The target or the focus of the view is determined by the position of mouse, within the window through the glutPassiveMotionFunc(). In this callback the mouse function takes in both the x and y positions of the pointer within the window. The previous x and y values are saved for calculations of the direction, for this direction the previous mouse values are subtracted from the current values and based on its relation to 0 the target position on the respective axis is moved by a small increment.

```
if ((mouse_x - old_x) > 0)  // moved
                right
            target_x += 0.1f;
 else if ((mouse_x - old_x) < 0)    //
                moved left
            target_x -= 0.1f;
if ((mouse_y - old_y) > 0)  // moved up
            target_y += 0.1f;
 else if ((mouse_y - old_y) < 0)    //
                moved down
            target_y -= 0.1f;
```

I have also implemented a range check on the mouse positions so that if it gets out of the defined boundary (within 20 pixels of the edges of the window border) it will be warped back to the center of the window. This check called the glutWarpPointer function to do so.

```
            // range check
          if (mouse_y < 100) {
             mouse_y = 100;
 glutWarpPointer(mouse_x, height / 2);
                  }
     if (mouse_y > height - 100) {
        mouse_y = height - 100;
 glutWarpPointer(mouse_x, height / 2);
         }if (mouse_x < 20) {
             mouse_x = width / 2;
 glutWarpPointer(width / 2, mouse_y);
                  }
      if (mouse_x > width - 20) {
             mouse_x = width / 2;
 glutWarpPointer(width / 2, mouse_y);
                  }
```

KEYBOARD INPUT

Actions within the window are all controlled through keyboard input by the user. These inputs are tracked by a series of call-backs defined within the following functions:

- glutSpecialFunc(arrows): for special keys such as arrows
- glutKeyboardFunc(keypress): general keyboard input (ASCII)
- glutPassiveMotionFunc(mouse): mouse tracking

These exact controls will be discussed in more detail below, defining what keys do what and my reasoning for their implementation.

WASD INPUTS

The camera motion is controlled by the WASD keys; W to move forwards, A to move left, S to move backwards, and D to move right. To create this motion in the relation to the keys, I make use of a forward vector (vec3), the up vector (vec3), and a defined speed (float) of the motion. The forward vector is calculated by taking the camera target (which is calculated by the mouse position, as explained above) and subtracting the camera position to get the distance from where the player is to where I would like it to be.

```
forward_x = target_x - camera_x;
forward_y = target_y - camera_y;
forward_z = target_z - camera_z;
```

If a W keypress is logged the forward vector is simply multiplied by the speed and this value can be subtracted from the current camera position to "walk" towards the target the target, conversely for an S keypress the value is added.

```
if (key == 'w') {    // Move camera
              forwards
   camera_x += forward_x * speed;
   camera_z += forward_z * speed;
   target_x += forward_x * speed;
   target_z += forward_z * speed;
```

By also adding these values to the camera target we can allow for continuous motion in all directions. For sideways motion in relation to the target the cross product of a combination of the Up Vector and Forward Vector must be used. In the case of an A keypress, indicating the desire to move left, I used a Left vector that is the cross product of Up*Forward, whereas the right motion the cross product used is Forward*Up. The vector is the multiplied by the speed and as before added to the camera position and camera target.

```
if (key == 'a') {    // Move camera left
    vec3 left = cross(vec3(forward_x,
       forward_y, forward_z), up);
      camera_x -= left.v[0] * speed;
      camera_z -= left.v[2] * speed;
      target_x -= left.v[0] * speed;
      target_z -= left.v[2] * speed;
                  }
```

OTHER INPUTS

Camera translations are controlled by the WASD keys, as discussed in detail in the above. However, my object features some other inputs that are mainly used for debugging purposes and examining the model to make sure the animation hierarchy is working correctly.

Translations are implemented with the arrow keys which require the glutSpecialFunc() callback rather than the glutKeyboardFunc.

```
        if (key == GLUT_KEY_DOWN) {
           translation_y -= 0.1;
                  }
         if (key == GLUT_KEY_UP) {
           translation_y += 0.1;
                  }
        if (key == GLUT_KEY_LEFT) {
           translation_x -= 0.1;
                  }
      if (key == GLUT_KEY_RIGHT) {
           translation_x += 0.1;
                  }
```

Rotation of the objects is dictated by the ‘,’ and ‘.’ keys. These trigger a directional rotation on the model.

```
            if (key == ',') {
             model_rotation = 20.0f;
                  }
            if (key == '.') {
          model_rotation = -20.0f;
                  }
```

The space bar also serves as a “pause” function, stopping all current object animations.

```
          if (key == 32) {
      model_rotation = 0.0f;
       tail_rotation = 0.0f;
    leg_translation = 0.0f;
                  }
```

OTHER CHOICES

I disabled the mouse pointer from appearing so motion would feel more natural and the user would not be distracted by it.

THE MODEL

DESCRIPTION

For my model, I created a small Pomeranian style dog in Maya in a sitting position. I felt this pose was stationary enough to easily implement but also had enough options to create a one-to-many hierarchy and the animations required.

PROCESS

During the modelling process, I started with a cube and extruded the faces to create a basic blocked body shape. I then manipulated edges and vertices while alternating between the cubic view and the smoothed poly view whiling forming a more natural shape. While working on areas that were on both sides of the model (such as the hips and ears) I turned symmetry on, on the objects x-axis, to speed up the process as I only needed to shape one side. I then modelled paws, and tail as separate parts, followed same method. For the nose, eyes, eyelids, used spheres instead of cubes, less manipulation needed.

Once I had basic shape made, I increase subdivides (Smooth), which increases the number of polygons in the model, so it resembles the poly view. I used quite a lot of subdivides to allow me to create very small details during sculpting. Once I had enough polygons to create natural strokes, I lifted and pushed back the polygons to create facial features and stylised illusion of fur (though this will also be later improved in textures). I liberally used the smooth tool to disperse any harsh lines to more natural appearance rather than clinical smooth surfaces before sculpting.

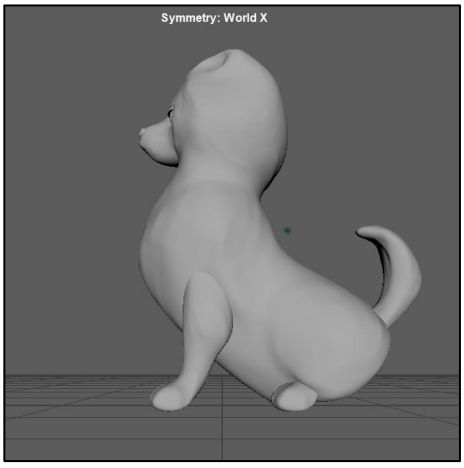
Once finished with sculpting I decreased the subdivides (Reduce), this ensured less vertices to be loaded and stored by the program. I repeated this process again for the limbs and tail.

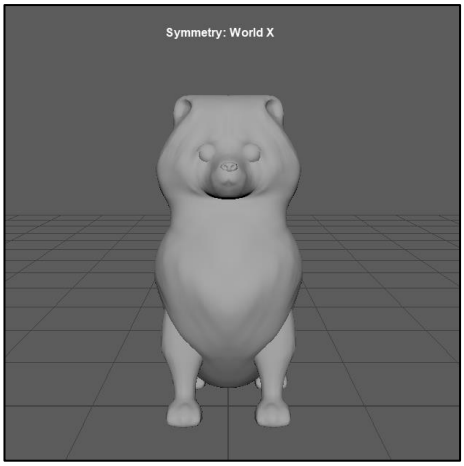
HIERARCHY

Before exporting the model to the '.dae' format, I separated it into a one to many hierarchy, and set each individual components' pivot to the origin to make transformations with OpenGL easier. The hierarchy for the model is as follows:

- Body (Root)
    - Tail
    - Paws (Front and Back)
    - Left Eye
    - Right Eye
    - Eyelids (Left and Right)

The child mesh was loaded into a separate VBO to the parent body, making use of ASSIMP to load the mesh data into a MeshData object comprised of a series of vertices. As both objects had their pivots set to the origin there was no need for further manipulation or translation in program.





THE ENVIRONMENT

DESCRIPTION

The scene I chose to construct was a simple, stylised back garden, with a kennel, picket fence and various dog-related items. I modelled each of the items individually following a similar, if not the same, process as that of the dog model but skipping the sculpting process. I could have added wood graining on the on the fence and kennel walls, however I implemented this effect with textures and bump mapping to

save time. Progressing from the midterm I added in added in more vegetation, in the form of small clusters of grass scattered around the scene and some large trees to add a greater sense of height and depth. The original environment and final environment are pictured to the right for comparison purposes.

HIERARCHY

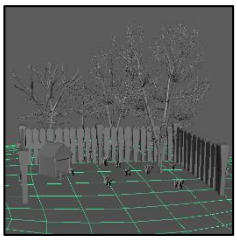
The hierarchy of the background environment is as follows, though there was only minimal animation to it (the trees) I wanted to use a hierarchy just for ease of repositioning it if was necessary upon importing into my program. These meshes were all exported and loaded

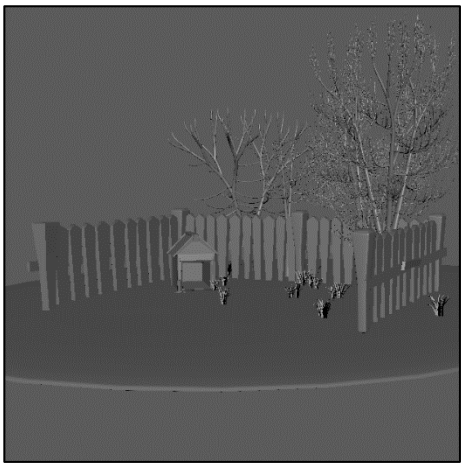
separately using ASSIMP, though tedious it had to be approached this way due to the fact they would all be using different textures. Some I did import all together (the collar, spikes, stake and chain are all within "collar.dae") as result of creating custom textures. The environmental hierarchy is as follows:

- Ground (Root)

- Picket Fence
- Kennel
- Stake
  - Chain Links
  - Spiked Collar
- Dog Bowl
- Trees (https://free3d.com/)
- Vegetation/Grass (https://free3d.com/)

Due to the grass and trees being last minute addition I go them from the sites references. However, all other objects were modelled by me.





ANIMATION

DOG

The model can be rotated on the x-axis using the ‘,’ and ‘.’ keys, the comma rotating it clockwise and the period rotating anti-clockwise. This is simply to illustrate the object hierarchy working, showing that both the dog rotates while the tail wags and paws patter. How these individual animations were implemented will be described in detail below.

The animation of the dog is linked to both proximity from (discussed in detail in advanced features) the camera but also a triggered upon a keypress from the ‘x’ key. It can also be stopped on the spacebar/pause keypress. This occurs in the glutKeyboardFunc(keypress) callback:

```
if (key == 'x') {
tail_rotation = 30.0f;
leg_translation = 0.01f;
}
```

The wagging of the tail is achieved by a rotation of 15 degrees from the axis on either side of the axis, once it hits this point it switches directions and begin to rotate 15 degrees in on the opposite side of the axis, giving the illusion of the tail wagging.

```
tail_z += tail_rotation * delta;
if (fmodf(tail_z, 15.0f) > 0)
tail_rotation = -30.0f;
if (tail_z/15.0f <= -1) tail_rotation =
30.0f;
```

The “tippy tap” movement of the front paws is result of a similar switching technique however instead of rotation, translation on the y-axis is used.

```
if (left_up) {
leg_l_y += leg_translation;
if (leg_r_y > 0.0f) leg_r_y -=
leg_translation;
if (leg_l_y >= 0.1f) {
left_up = false;
right_up = true;
}
}
```

The Booleans left_up and right_up enable animation to switch between whichever leg should be going up and which should be going down. The switch then swaps upon hitting or exceeding the defined limit. The translation speed increment is defined in the keypress function and is applied to both upwards and downwards transitions.

ENVIRONMENT

For the most part the environment is a static one, with most objects having no animation. I did end up added a very slight, almost unnoticeable animation on the trees. This animation is a very small rotation in either direction of the x-axis, achieved using the same method as the tail wag.

```
tree_z += tree_rotation * delta;
if (fmodf(tree_z, 1.0f) > 0)
tree_rotation = -0.1f;
```

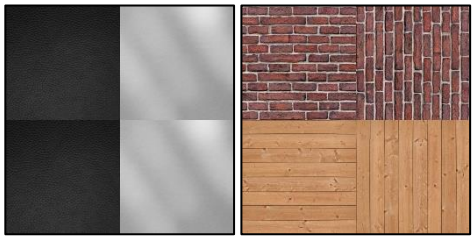
```
if (tree_z / 1.0f <= -1) tree_rotation
                = 0.1f;
```

Upon close inspection you can notice the motion among the branches, giving the impression of a slight breeze.

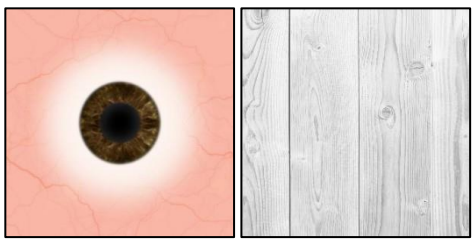
TEXTURES & UV MAPPING

Due to most of the models being of my own creation I had to UV map each of them to ensure their texture co-ordinate matched up properly when being read in program.

Texture mapping involves telling each vertex which section of the given texture it corresponds to through Texture Co-ordinates. These co-ordinates range from 0 to 1 on both axis and map the vertex to the appropriate area on the square texture .jpg. These co-ordinates are stored in the mesh data and stored during the loading through ASSIMP.



The texture files are square .jpg or .png files, contained within the textures/ subdirectory. To convert this image into something that can be read by OpenGl code, I loaded the using the STB Image library in the load_tex() function. This function takes the texture path and returns an array of unsigned integers which represent the image. This function also assigns the texture filtering and wrapping techniques with glTextParamteri(), as well as generating a texture buffer and binding the texture using glGenTextures() and glBindTextures() respectively.



The texture wrapping method that I chose to use was repeat. This is the default wrapping method which repeats the texture over and over, as most of my textures were either seamless or mapped to where repeating did not occur this was the ideal method.

```
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T, GL_REPEAT);
```

The texture filtering method I used was GL_Linear or bilinear filtering. This function create a more smooth and natural output as individual pixels are less visible. The effect is achieved as result of the interpolation of each texture pixel, based on the surrounding pixels and mixing the colours.

```
  glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

The textures and current vertex co-ordinate pair are passed into the shaders and assigned in the fragment shader, where the lighting and other effect can be applied before rendering.

However, before any of this can occur, UV mapping must occur to assign the vertexes of the custom models correctly. This is done in Maya before exporting the .dae file. The process is simply cutting up the models UV shell and arranging them on the texture correctly, most models were simple enough simple scaling the unfolded model to suit the square texture provided. For example, the kennel I separated it into the shells based on its faces, and translated, scaled, and rotated until the correct sections had the right material (wood or brick) and was oriented to an appropriate direction.

On the other hand, for natural models such as the dog I preferred to generate shells based on

the camera position and then mapped to an image of a dog. This was because this model has much more vertices and an unfolded or automatically generated UV make it difficult to distinguish what section is what. By doing this based on the camera position the UV shells are easily identified.

Texture JPG:

UV Mapping:

Result on Model:

PHONG ILLUMINATION

The lighting method I chose to implement through my Vertex and Fragment Shaders. Blinn-Phong is an expansion on regular Phong lighting which better handles the specular shine on flatter surfaces and speeds up computation. Instead of just working on the light directions it introduces a half-way direction which is used to calculate the location of the specular angle and shine. This halfway direction is a result of the normalization of the light direction (a fixed position) with the view direction (negative vertex position) added on. This calculation accounts for viewing from a shallow angle, as well as being faster due to the half-vector being less expensive than cosine calculations.

```
vec3 halfDir = normalize(lightDir +
            viewDir);
 float specAngle = max(dot(halfDir,
          normal), 0.0);
  specular = pow(specAngle, 16.0);
```

Aside from the calculation of the specular lighting, there also is a combination of ambient and diffuse (as exists in regular Phong illumination), that I define as constant values (vectors) in the fragment shader. Ambient is a global illumination model that applies a constant light value from no direct light source. Diffuse lighting dictates brightness based on the location of the light source and softening further from the source. The diffuse colour is multiplied by the Lambertian value calculated from the light direction and normal.

```
vec4 color = texture(ourTexture,
 TexCoord) * vec4(ambientColor +
lambertian * diffuseColor + specular *
          specColor, 1.0);
```

These lighting effects are then all applied to the texture to create the output colour of the current vertex which can be put into the gl_FragColor.

ADVANCED FEATURES

MODELLING

As previously discussed in The Model and The Environment sections above, all models in the scene were modelled and sculpted by me in

Maya, with the exception of the trees and grass strands. For further clarification here is a list of the objects I modelled:

- The Model
  - body.dae
  - l_eye.dae
  - r_eye.dae
  - eyelids.dae
  - tail.dae
  - l_legs.dae
  - r_legs.dae
  - nose.dae
- The Environment
  - collar.dae (includes stick, collar, spikes, and chain)
  - kennel.dae
  - fence.dae
- Not included in final scene
  - bowl.dae

BUMP-MAPPING

To further add depth to the textures that would otherwise appear flat, I implemented bump mapping to extend upon the Blinn-Phong illumination. This technique simulates the bumps and ridges within an image (despite the image being a 2D representation). This technique does not add anymore vertices to the object, saving time (from modelling the bumps) and space (storing additional vertices).

Rather than creating a separate bump map for each texture in photoshop (desaturating the texture and upping the contrast), I did this instead within the Fragment Shader. The bump map is generated by taking the texture and texture co-ordinate and creates and RGB mask by calling texture.rgb to get the bump map, these values are then clamped between -1 and 1 to create the map. This mapping is combined with the normal interpolation from the Vertex Shader and normalized.

```
vec3 bump = 2.0 * texture(ourTexture,
        TexCoord).rgb - 1.0;
vec3 normal = normalize(normalInterp +
                bump);
```

This normal value is then used in the lighting calculations, to calculate the lambert and specular lighting. The result is that the high points of the texture are affected by the lighting and the deeper ridges remain in shadow.

This technique worked best on the kennel where the wood panelling and bricks textures work very well with the bump mapping. It was not as effective on the dog fur mesh, but this was because of my choice of texture where the chest of the dog was in shadow resulting in it not receiving any light once the bump map was calculated, though it does give the effect of individual hairs on the face of the dog.

FOG

The calculation of the fog occurs in both the Vertex and Fragment Shaders. This effect is determined by an exponential function which takes into account the gradient, density and distance of an object from the camera and outputs a visibility value for the current vertex. Bot the gradient and density are static values I set based on the “thickness” and quickness of dispersion I wanted from the fog (play around with values until was happy with the result).

```
visibility = exp(-pow(distance*density,
                 gradient));
```

This value is outputted into the Fragment Shader for the fog colour to be applied to the colour/texture. The fog colour is the same as that of the sky (a moody gray colour) to ensure cohesion between the background.

The visibility variable is applied to the texture after all the lighting is calculated, this is done through a call to the mix function. The passed to the gl_FragColor.

```
gl_FragColor = mix(vec4(fogColor,1.0),
           color, visibility);
```

PROXIMITY TRIGGER

For the animations on the dog, I implemented a proximity trigger so that the animations would only occur when close to the dog's location. This is done in the updateScene() callback, where the animations are also contained.

```
    vec3 dist = vec3(translation_x,
    translation_y, translation_z) -
  vec3(camera_x, camera_y, camera_z);
     float x_distance = dist.v[0];
     float z_distance = dist.v[2];
```

The calculations shown above are how I define the distance, this is simply achieved by subtracting the camera location values from those of the model, producing a vector. However due to my height being stationary I only need to look at the x and z values, stored at index 0 and 2 respectively.

```
if (x_distance > -5 && x_distance < 5
 && z_distance > -5 && z_distance < 5
```

The If-Statement above contains the dog animations and the values are the range of proximity the camera must be in. The values were determined by me printing out the distances until I reached the radius that I felt suitable. Once this range is entered by the camera, the animation calculations can begin.

AMBIENT SOUND

Ambient sound was a final step I added before the glMainLoop(), to add atmosphere to the game. I simply made a call to the PlaySound() function and passing in a free ambient .wav file that I got from FreeSound.org.

```
PlaySound(TEXT(AMBIENT), NULL,
          SND_ASYNC);
```

However, this file is quite large being about 43MB so it adds a bit of a wait to the loading of my program.

LINKS

VIDEO DEMO

https://youtu.be/vcqUmqI1YKY