CSU44052 - Computer Graphics - 16318649 - Video Demo: https://www.youtube.com/watch?v=d_MEw3ADDi4

Mid-Term Report

Specifications

The aim of the CSU44052 lab assignments so far has been to assist with my development of the following specifications for my final project:

- 3-dimensional objects and views
- An environment
- User interaction and camera-control
- A well designed hierarchical animated object, creature or vehicle, a one-to-one hierarchy at the minimum

As part of my learning I followed the tutorials at https://learnopengl.com/ in order to help me implement key features such as VAOs, camera, etc.

Hello Triangle

The first couple of weeks involved an introduction to OpenGL and the graphics pipeline. We learnt how to work with shaders, vertex buffer objects and vertex array objects in order to display two simple triangles in a window. I was able to play around with colour and positioning as shown in Fig 1.0.

This result was produced by having three global arrays - one for the VBOs, another for the VAOs, and the last for the shader program IDs. I used two sets of Vertex and Fragment shaders, one set per triangle, in order to colour and position them differently. The triangles were then drawn using their corresponding VBOs and VAOs.

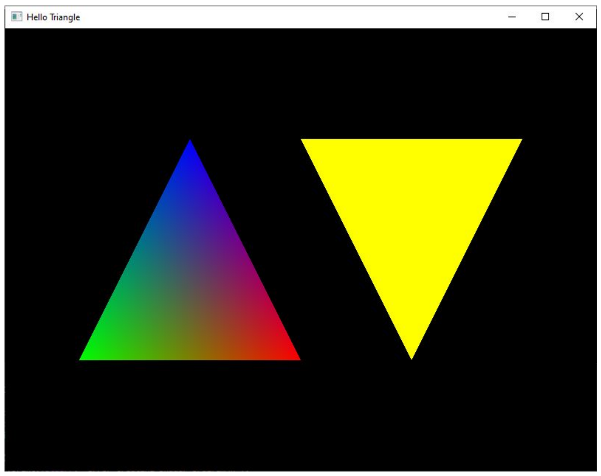


Figure 1.0 - Triangles drawn with VAOs

The respective shader IDs were passed into `linkCurrentBuffertoShader(shaderProgramID)` which allowed me to pick what shader programs would be linked to the currently bound VBO. This process allowed the triangles to be drawn simply by calling `glBindVertexArray(VAOs[i])`before each draw call.

Projection

After playing around with the Lab 3 sample code that included the monkey head parent and child models, I imported a model of a spider that I found on https://free3d.com. The mesh had to be triangulated in Maya and converted to a DAE file before I was able to import it using the Lab 3 code provided. I decided to use the maths_funcs library that was also supplied rather than GLM. However I noticed there was no function for calculating an orthographic projection matrice so I added the code below as I followed this tutorial http://in2gpu.com/2015/05/23/enter-the-matrix-and-projection.

```
mat4 ortho(float width, float height, float nearZ, float farZ) {
      mat4 m = zero_mat4();
      m.m[0] = 12.0f / width; //changed 2 to 12 to increase scale of model
      m.m[5] = 12.0f / height;
      m.m[10] = 1.0f / (nearZ - farZ);
      m.m[14] = -nearZ / (nearZ - farZ);
      m.m[15] = 1.0f;
      return m; }
```

The tutorial gave this matrix for orthographic projection which takes aspect ratio in account.

For perspective projection I used the function that was included in the maths_funcs library :

`mat4 perspective (float fovy, float aspect, float near, float far)`

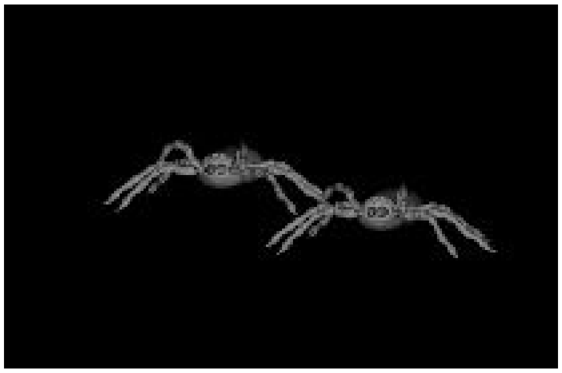
$$\begin{pmatrix} \frac{2}{screen\_width} & 0 & 0 & 0 \\ 0 & \frac{2}{screen\_height} & 0 & 0 \\ 0 & 0 & \frac{1}{far-near} & \frac{-near}{far-near} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

1.1 Orthographic projection matrix

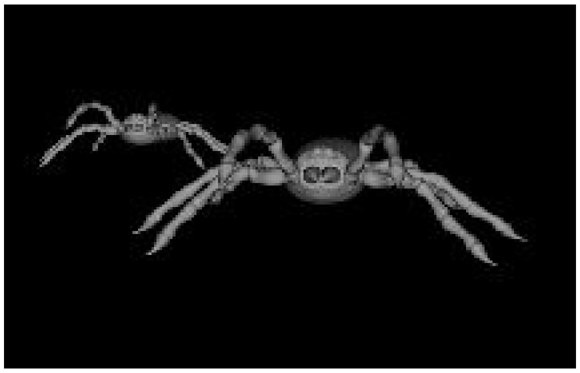
I then wrote this function that returns a projection matrix depending on what key is pressed.

```
mat4 projection() {
      mat4 projection;
      if (test_ortho) { //global boolean set to true when ‘o’ key is pressed
            projection = ortho(width, height, 1.0f, 1000.0f);
      } else { //test_ortho is set to false when ‘p’ is pressed
            projection = perspective(55.0f, (float)width / (float)height, 1.0f,
1000.0f);
      }
      return(projection); }
```

This function gave the results shown in Fig 1.1 and 1.2. The orthographic projection makes the spiders look as if they are positioned at the same z-coordinate even though the spider on the right is actually in front of the one on the left.



1.2 - Orthographic projection



1.3 - Perspective projection

Camera

The next step was to implement a camera. Before this, I simulated the movement of a camera by translating the models in the opposite direction, i.e. if A was pressed to move the 'camera' left, the models would be translated to the right in the x-axis. In order to create an actual camera, I used the `mat4 look_at (const vec3& cam_pos, vec3 targ_pos, const vec3& up)` function in maths_funcs. The view matrix returned from this is used to update the corresponding shader uniforms. I kept three global variables (for the x, y and z position of the camera) and passed this into the function - this allows the camera to move with W,S,A,D key presses.

However a static camera that moves only with keys is not often used without mouse movement also affecting where it is pointed. The style of camera I wanted was one to resemble the style of most FPS games. The `glutPassiveMotionFunc(void mouseCallback)` function allows behaviour to be defined in the mouseCallback function when the mouse moves. I also used `glutMotionFunc(void mouseCallback)`so the same behaviour happens if the mouse moves while clicking.

The following code is executed on the movement of the mouse.

```
void mouseCallback(int xpos, int ypos) {
	GLfloat xoffset = xpos - lastX;
	GLfloat yoffset = lastY - ypos; //reversed since y-coords range from bottom 
to top
	lastX = xpos; //global vars keep track of previous cursor coordinates
	lastY = ypos;
	GLfloat sensitivity = 0.05f;
	xoffset *= sensitivity;
	yoffset *= sensitivity;
	yaw += xoffset; //increase global yaw & pitch values by calculated offsets
	pitch += yoffset;
	//limit the pitch so the camera's degrees of freedom resembles a person looking down or up
	if(pitch > 89.0f)
		pitch = 89.0f;
	if (pitch < -89.0f)
		pitch = -89.0f;
	updateCamera();
}
```

After the new offsets for the pitch and yaw values are calculated and added, the camera must be updated. This is done with the `updateCamera()` function. This recalculates the front vector which determines the camera target and then calls the display function so the view matrix is updated.

```
void updateCamera() {
	GLfloat front_x = cos(ONE_DEG_IN_RAD*yaw) * cos(ONE_DEG_IN_RAD*pitch);
	GLfloat front_y = sin(ONE_DEG_IN_RAD*pitch);
	GLfloat front_z = sin(ONE_DEG_IN_RAD*(yaw)) * cos(ONE_DEG_IN_RAD*pitch);
	cameraFront = vec3(front_x, front_y, front_z);
	// Normalize vectors as the movement will slow as the length nears zero when the camera looks up/down
	cameraFront = normalise(cameraFront);
	cameraRight = normalise(cross(cameraFront, worldUp));
	cameraUp = normalise(cross(cameraRight, cameraFront));
	glutPostRedisplay();
}
```

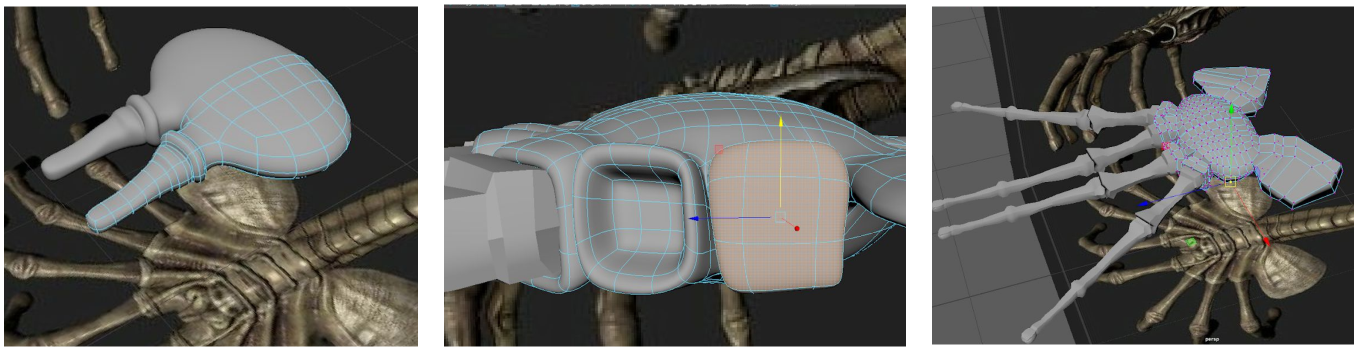
This successfully implemented a camera which looked around and could move around the models. However I found the degrees of freedom was limited by the window size since the cursor was able to leave the window. I realised the cursor needs to be pinned to the centre of the screen to give that FPS effect of being able to look around fully. This was done with the simple function `centerCursor()` which uses `glutWarpPointer(int x, int y)` to keep the cursor in the centre of the screen. This function is called at the start of the mouseCallback function.

```
void centerCursor() {
	glutWarpPointer(width / 2, height / 2);
	lastX = width / 2;
	lastY = height / 2;
}
```

I then added `glutSetCursor(GLUT_CURSOR_NONE)` to my main function in order to hide the cursor. Since the curser cannot leave the center of the screen, I added a keypress handler to allow the user to exit the program when the 'ESC' key is pressed.

Modelling

I wanted to make a start on learning how to model in Maya before the second half of the term in order to prevent time constraints from impacting my final project. For my final project I wanted to create a scene inspired by Ridley Scott's *Alien* franchise, with the main animated model being what's called a 'facehugger' that featured in the more recent films. I used reference images I found on ZbrushCentral.com in order to help me with the proportions of the model.



1.4 - Photos of modelling process

Creating the model myself gave me a greater understanding of how hierarchies worked for animation and how I wanted my model to move and deform. For the animation part of the project, I decided to start with a simple rotation movement of the tail. Initially I created the tail as part of the body but then decided to separate it into two segments - the base and the main tail - with the body parenting these two objects. This is shown in Fig 1.5. The base acts as the parent of the main tail, so it will move the main tail along with it when it rotates. I also separated out the legs into a proper hierarchy with three objects, the upper part of the leg parenting the middle and lower parts, then the middle parenting the lower part. However I then exported the leg objects together as one model to display in my program as I was not planning on animating them yet.

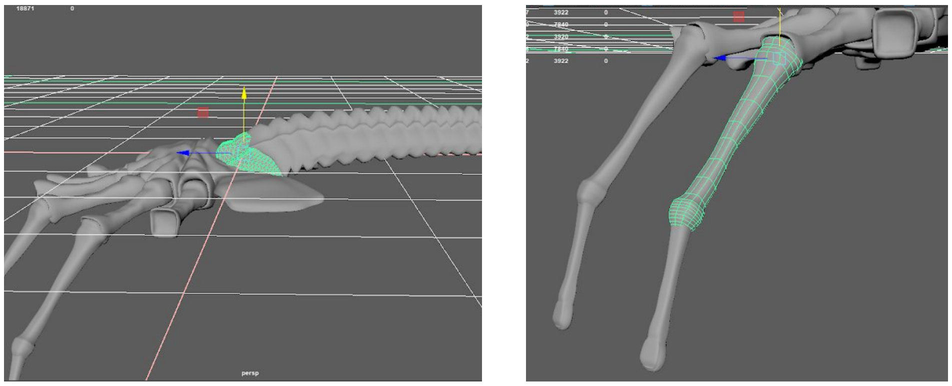


Fig 1.5 - Separation into hierarchy

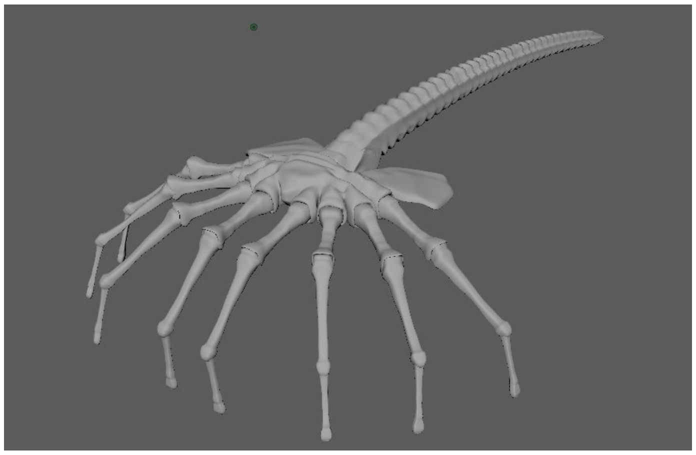


Fig 1.6 - Finished Facehugger model

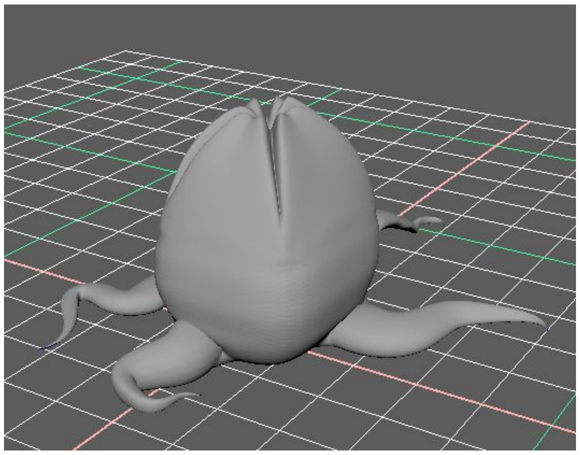


Fig 1.7 - Alien egg model

I wanted to place this model in a cave scene, and this was also developed in Maya. I created this out of a cylinder primitive that I scaled up and deformed as shown in Fig 1.8. I also flipped the normals of the vertices so the lighting would be reflected on the inside of the model as this is where my camera would be placed. I then created a model of an alien egg to place in my scene for decoration and possibly animation later, as in the films the facehugger is known to hide inside the eggs and jump out.

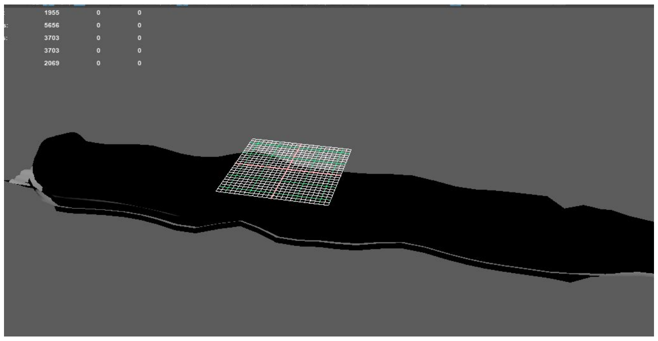

```
1955    0    0
5656    0    0
3703    0    0
3703    0    0
2069    0    0
```

Fig 1.8 - Entire cave model

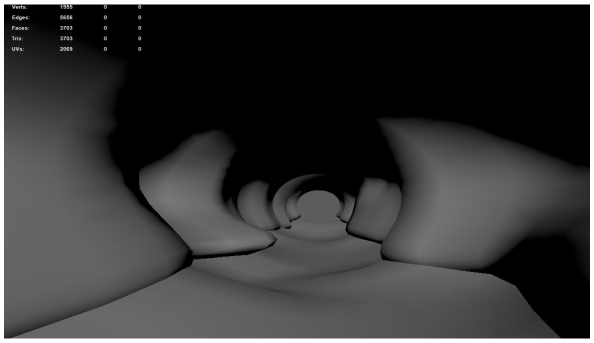

```
Verts:    1955    0    0
Edges:    5656    0    0
Faces:    3703    0    0
Tris:     3703    0    0
UVs:      2069    0    0
```

Fig 1.9 - Interior of cave model

Creating the Scene

I continued on from the Lab 3 code in which I had implemented my camera and orthographic projection in. I adapted some of the functions that were provided, such as for generating the buffers, in order to display multiple objects on the screen. I kept the global arrays of VAOs, VBOs and shaderIds and also added an array of attribute locations which keeps the locations of the vertex positions, normals (and later textures) of each object. This means each model has a VAO, 3 VBOs (one VBO for position, one for normals and one for textures) and 3 location attributes.

Textures will be implemented in the second half of the semester so only 2 of the VBOs and location attributes are being used at the moment for each object. I adapted the generateObjectBufferMesh function to take the following parameters.

```
void generateObjectBufferMesh(const char* mesh_name, ModelData  &mesh_data, GLuint
shaderProgramID, int vao, int vp_vbo, int loc)
```

These are the parameters that are used for each model my program draws. The mesh_name variable hold the name of the model's file and mesh_data holds the number of points the model has. A shaderProgramID is passed which allows the user to choose what shader will be used to draw the model. The final three parameters are the indexes to the global VAO, VBO and attribute location arrays.

Rather than copying and pasting all the code for updating the view, model and projection uniforms in the display() function, I moved all this code into a separate function that would be called for each object to draw.

```
mat4 displayObject(int vaoOffset,vec3 posVec, vec3 rotVec,vec3 scaleVec, ModelData &mesh_data,
int shaderProgramID, bool orthoEnabled)
```

This function takes the index of the corresponding VAO for the mesh data, and also takes in vector-types that determine the position, rotation and scale of the object being drawn. The shaderProgramID is passed into `glUserProgram(shaderProgramID)` at the start of the function. I also added the local boolean orthoEnabled to allow me to pick what objects can be viewed in orthographic projection. This means I am able to see the difference between objects in ortho and objects in perspective simultaneously when 'o' is pressed as shown in Fig 2.0. The function returns the model matrix of the object being drawn to allow me to draw a child object later.

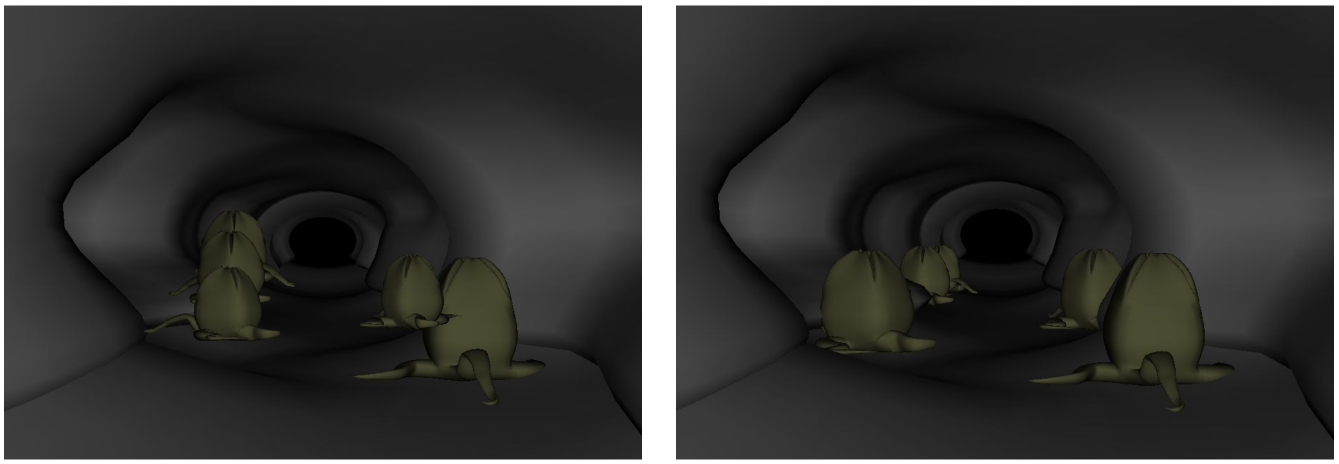


Fig 2.0 - Perspective vs Orthographic projection

In the images above, there are five egg objects. The right image represents the initial perspective projection of the scene. The left image shows the cave and closest egg in perspective and the other four eggs in orthographic.

Animation

Lastly I aimed to create a simple one-to-one animated hierarchy represented by the alien's tail. In order to do this, I split the model in Maya into the body, the legs, the tail base, and the main tail and exported them as separate objects. This was possible as the vertex positioning is kept intact. Because I wanted to have the tail rotate around it's base without worrying about translating back and forth between the origin and it's specific position, I translated the entire alien model so that the base was at the origin in Maya. This means that when translating all the parts of the model in OpenGL, I can translate them by the same factor and they will remain in the correct position relative to each other.

I created a function `displayChild()` which takes the same parameters as `displayObject()`, as this is called in order to draw the parent first. The model matrix of the parent is returned and multiplied by the child's model matrix in order to get the child object to move with the parents movement. For this animation I just needed rotation, so I passed global variables as the x, y and z rotation coordinates for both the parent and the child. This is because I wanted the main tail to move with the base but also to move slightly further than it for an exaggerated look. This is a feature that I could continue all down the tail by separating it into more parent-child relationships to get it to move in a realistic manner.

```
void displayChild(int parentVaoOffset, vec3 posVec, vec3 rotVec, vec3 scaleVec,
ModelData &mesh_data, int shaderProgramID, bool orthoEnabled) {
	mat4 parent_model = displayObject(parentVaoOffset, posVec, rotVec, scaleVec,
mesh_data, shaderProgramID, orthoEnabled);
	int matrix_location = glGetUniformLocation(shaderProgramID, "model");
	mat4 child_model = identity_mat4();
	child_model = rotate_x_deg(child_model, rotVec.v[0]);
	child_model = rotate_y_deg(child_model, rotVec.v[1]);
	child_model = rotate_z_deg(child_model, rotVec.v[2]);
	// Apply the root matrix to the child matrix
	child_model = parent_model * child_model;
	glUniformMatrix4fv(matrix_location, 1, GL_FALSE, child_model.m);
	int childVaoOffset = parentVaoOffset+1;
	glBindVertexArray(VAOs[childVaoOffset]);
	glDrawArrays(GL_TRIANGLES, 0, mesh_data.mPointCount);
}
```

The following function is called when the user presses a key that initiates rotation of the tail. That is, when x is pressed, the tail rotates in the x axis and the same for 'y', 'z' and their axes.

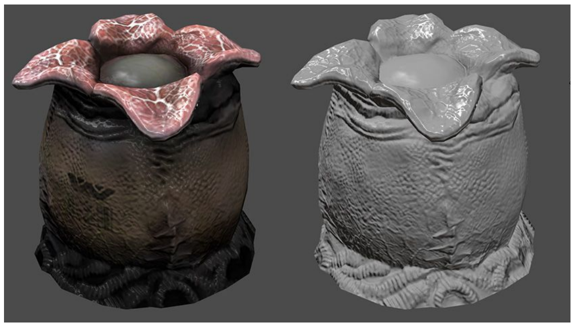
```
void updateTail(GLfloat *rotate_axis_tail) {
	if (*rotate_axis_tail <= -10.0f) tailMove = true; //degrees of freedom
	 else if (*rotate_axis_tail >= 10.0f) tailMove = false;
	 // rotate the base at a slower rate than the main tail
	if (tailMove) *rotate_axis_tail += 0.5f;
	else *rotate_axis_tail -= 0.5f;
	glutPostRedisplay();
}
```

Future Features

I plan to implement the following features for the final version of my project.

Textures

At the moment my models are very flat and have just been coloured through the shaders. I plan to develop my cave scene further by creating a more rocky texture to the walls and floor. I could do this through a simple image texture, however, to get a better and more realistic look I've seen that this can be done with displacement maps in Maya. A tutorial I found at https://www.youtube.com/watch?v=Z67Akgy5Gjo gives a way of creating a rocky surface by converting a still image to a displacement map using Arnold. This displacement map can then be converted into polygons. This would speed up the process of making the model look more real as opposed to sculpting or manually moving edges and faces.



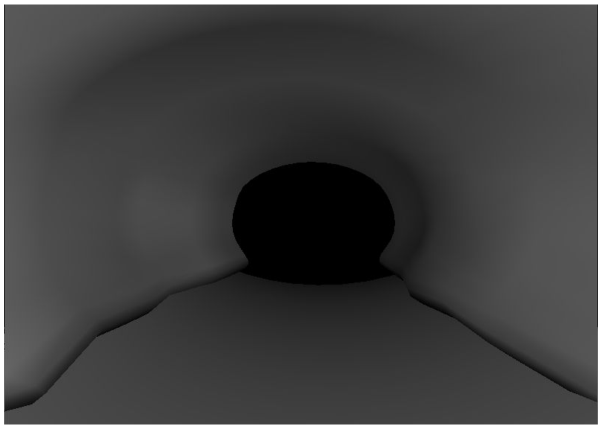
I also plan to either texture the alien model through painting in Mudbox. I'll aim for a more bumpy texture to the alien egg models such as in Fig 2.1. This could be done with bump maps or displacement maps.

Figure 2.1 - Textured Alien Egg from https://polycount.com/

If I have time I could also create a version of the egg model that opens up, to add variation to the scene.

Lighting

Fig 2.2 - Current lighting



Although we have yet to learn about lighting, I have noticed some ways in which this aspect of my project could be improved. My cave model stretches a distance away from the camera and then is cut off with a hole at the very end. In order to make this look like the cave was very dark at the end of it, I made the background of the OpenGL window black. However this is problematic as you near the end of the tunnel, the circle remains dark but the lighting around it becomes lighter. This gives a very flat looking effect. I would like to find a way to keep the lighting around the end of the tunnel dark once we cover lighting, so it looks as if there is a large dark room at the end. I'd also like to experiment with some fog moving along the floor to give a more *Alien* effect to the scene.

Advanced Animation

I plan to improve the current animation I have implemented as it is currently very basic and not akin to the movements of the creature that inspired my model. I would like to introduce some animation into the legs and have the alien move around the cave. This would require reworking the current hierarchy I have to include the body and the legs along with the tail base and the tail. The hierarchy would likely be done as:

- Alien body
  - Tail Base
    - Tail Middle Joint
      - Tail Lower Joint
  - Upper Leg
    - Middle Leg
      - Lower Leg

Where Alien Body is the parent of the Tail Base and Upper Leg and so on. I would also like to extend the current animation I have for the tail to include more children in order to get a more "slithering" effect of the tail.

Collision Detection

I feel my scene would benefit from implementation of collision detection. Since my scene is a model that encircles the camera, the camera is currently able to leave the scene by moving through it. This is seen in my demonstration video. I would like to address this through collision detection, which would prevent this undesirable effect.