Sarah Phillips, 16318649
Youtube Link:

CSU44052 Computer Graphics

Final Report

Specifications

The final state of my project should have the following functionality at a minimum:

- 3-dimensional objects and views
- A consistent scene environment to place the objects in
- User interaction and camera-control
- A well designed hierarchical animated creature (one-to-one hierarchy at the minimum)
- Texture-mapping of models
- An implemented Phong illumination model

In addition to basic functionality, the project should have advanced features included. Extra functionality I implemented includes the following:

- 3D models created by myself in Maya (the alien, cave, and egg models)
- Keyframe animation
- Normal and specular mapping
- Implementation of material-dependant lighting

As part of my learning I followed the tutorials at https://learnopengl.com/ in order to help me implement features such as texturing and lighting.

Contents

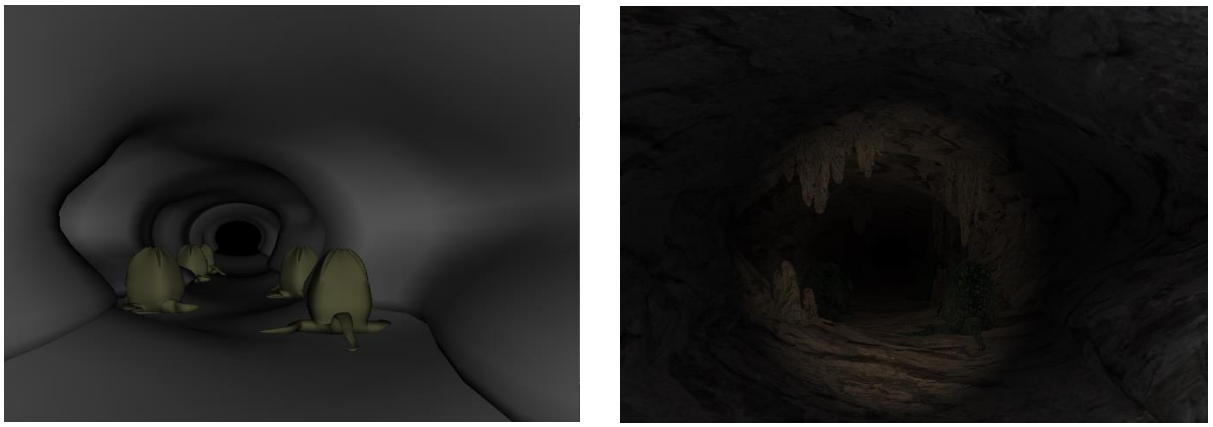


Fig 1.0 – Midterm Scene vs Final Scene

3D Objects and Scene

For my final project I wanted to create a scene inspired by Ridley Scott's *Alien* franchise, with the main animated model based on by H.R. Giger's 'Facehugger' that features in the films. I used reference images I found on ZbrushCentral.com in order to help me with the proportions of the model.

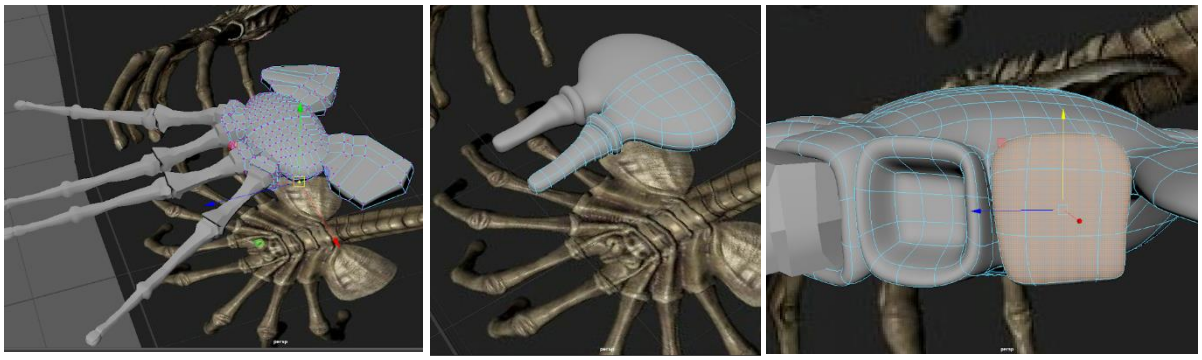


Fig 1.1 - Photos of modelling process

Creating the models myself in Maya 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 do 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 1.2. The base acts as the parent of the main tail, so it will move the main tail along with it when it rotates.

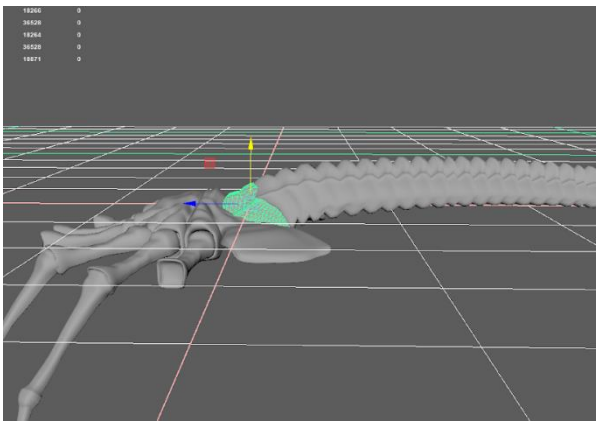


Fig 1.2 - Separation of tail

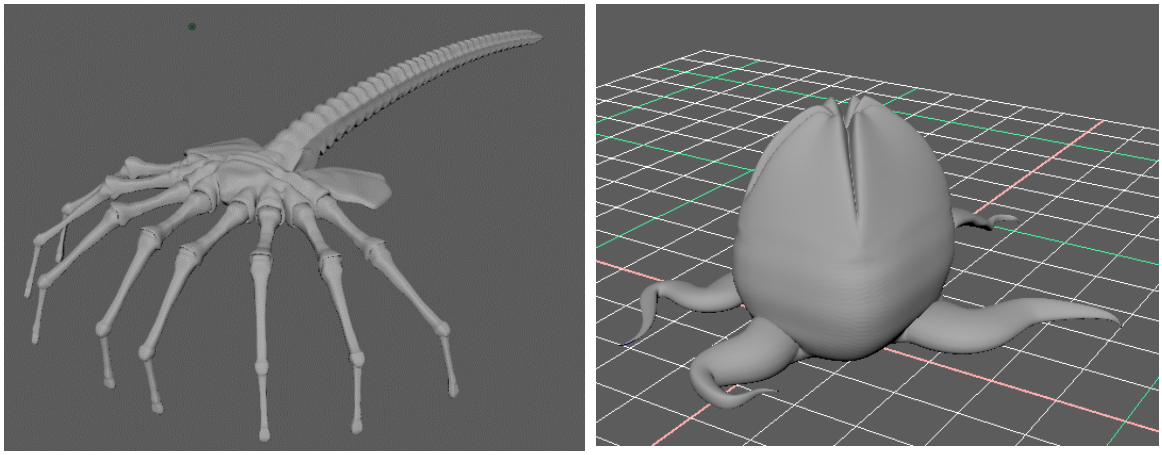


Fig 1.3 - Finished Facehugger and egg models

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 1.4. 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, as in the films the facehugger is known to jump out of these.

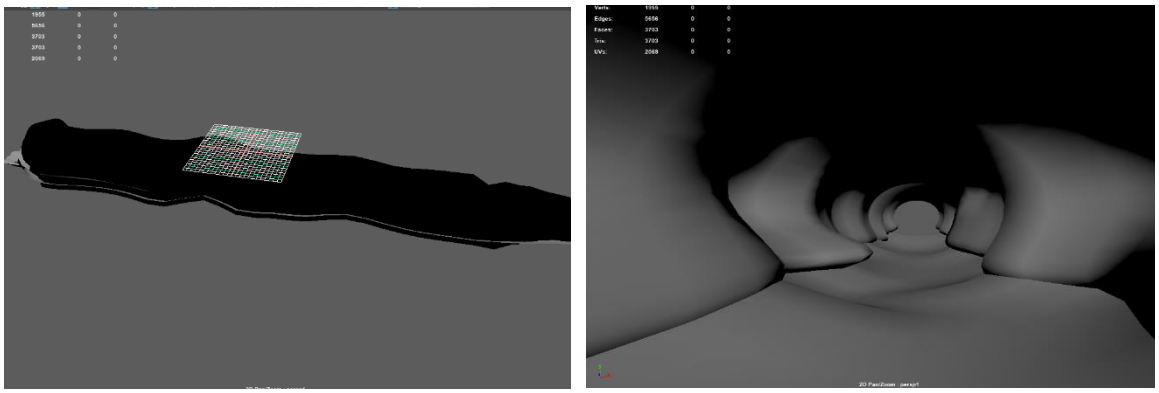


Fig 1.4 - Entire cave model (left) and its interior (right)

In addition to the models I created, I also included some models of stalactites and stalagmites in my cave last minute. These models I found on Turbosquid and are shown in Figure 1.5. I added these to decorate the cave scene further.

Fig 1.5 – Stalagmites model

User Interaction and Camera Control

The camera moves around the scene using W, S, A, D key presses and this was done by updating global variables corresponding to the camera's x, y and z position whenever the keys were pressed. The `mat4 look_at (const vec3& cam_pos, vec3 targ_pos, const vec3& up)` function was used to implement the camera. The view matrix returned from this is used to update the corresponding shader uniforms.

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. The alien model can also be scaled up or down by pressing the '+' and '-' keys. The camera only can move up or down in the Y-axis by the user if E or Q is pressed, this is so the camera doesn't move up and down when WSAD are pressed in order to simulate a person moving and looking around. There is a user control for initiating the keyframe animation of the alien also, which is done by pressing I. Every time the I key is pressed, a new keyframe of the alien is drawn. This will be expanded upon later in the report.

Lighting

Since my scene is in a cave, I wanted the lighting to be dark in order to invoke a sense of realism. I did however want to be able to show off the textures and models so I decided to implement a spotlight, as this would resemble a flashlight. Most of the implementation for the lighting was done in the fragment shader, including the Phong illumination model consisting of ambient, diffuse and specular lighting. I only required the use of one shader program for my entire scene.

The vertex shader takes in the position, normal and texture coordinate of each vertex, and contains uniforms that are updated with the view, projection and model matrices of each model. Here, it calculates the fragment position, which is the location of the fragment in window space, as well as the vertex position in clip space. The transpose of the inverse of the current model matrix is also multiplied by the vertex normal in order to preserve the normals when the model matrix contains a non-uniform scale. The texture coordinates of the vertex as passed along to the fragment shader.

```
void main(){
  FragPos = vec3(model* vec4(vertex_position,1.0));
 //multiply normal by the transpose of the inverse of the model matrix   //to preserve normals on a non-uniform scale
  Normal = mat3(transpose(inverse(model)))*vertex_normal;
  // Convert position to clip coordinates and pass along
  gl_Position = proj * view * model * vec4(vertex_position,1.0);
  Texcoord = vertex_texture;
}
```

In the fragment shader, calculations for the ambient, diffuse and specular lighting are carried out. This is also where the spotlight and attenuation are implemented. I implemented a uniform struct called Material so I could determine how the lighting on the different models would be affected by their corresponding texture, specular and normal maps, which are passed in from the main program to the shader via this struct. Since the position and direction of the camera determines the position and direction of the spotlight, I also had another uniform struct called Light, which allows values to be passed regarding the direction, position and cut off of the spotlight.

Figure 1.6 – Alien with specular lighting vector set to 1 (left) then to light orange (right)

I liked the effect the lighting gave on the alien model without textures as it looked very smooth and shiny. So, instead of texturing the alien like I did the egg and cave models, I added in if statements and vectors in the fragment shader to separately affect how the lighting showed on the alien model. This was done as the lighting on the rest of the models is determined by their texture, specular and normal maps. I gave the ambient and diffuse lighting zeroed out values as I wanted the alien to appear black. Then, I set the vector that determines the colour of the alien's specular lighting to a light orange, as this is the colour that tends to be reflected when a warm light is shone on a black creature. In order to let the fragment shader know when a fragment corresponds to the alien model, I pass in a float that acts as a Boolean named isAlien. For the alien model's object, this float is set to 1.0 and for all other objects this is set to 0.0.

```
//alien lighting
if(material.isAlien > 0.0) {
   norm = normalize(Normal);
}
//calculate specular lighting
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);

if(material.isAlien > 0.0) {
   specular = lightAmbient* spec*alienSpecular;
   diffuse = alienAmbient;
   ambient = alienAmbient;
}
```

The spotlight cutOff (the radius of the spotlight) is calculated by getting the dot product between the vector pointing from the fragment to the light source, and the direction the light is pointing towards. Then, an outer radius is also calculated in order to give a gradual decrease in lighting outside the spotlight. This is so the flashlight looks more realistic, otherwise the spotlight would be as if we were looking through a hole into the scene.

```
//flashlight calculations
 float innerRadius = dot(lightDir, normalize(-light.direction));
 float outerRadius = (light.cutOff - light.outerCutOff);
 float intensity = clamp((innerRadius - light.outerCutOff)/outerRadius,0.0, 1.0);
 diffuse  = diffuse*intensity;
 specular = specular*intensity;
```

I also added attenuation in the lighting model. This adds the effect of the light reducing in intensity as it travels further back into the cave. This way, objects that are further away from my flashlight appear darker, which is more realistic. Each lighting vector is multiplied by the attenuation float in order to see this effect across the ambient, diffuse and specular lighting.

Figure 1.7 – Attenuation turned off (left) vs turned on (right)

Textures

Continuing from my progress at mid-term, I made a few changes to my models so I could texture them. I had to teach myself how to UV map each of the models I had created so texture images would be placed on them the way I wanted. This was done in Maya. In order to get repeated textures, I simply scaled up the UV shell to get the scale I wanted, rather than doing it manually in OpenGL. Although this gave me the results I wanted for my application, I realise it is a convention to keep the UV shell of models normalised between 0 and 1.

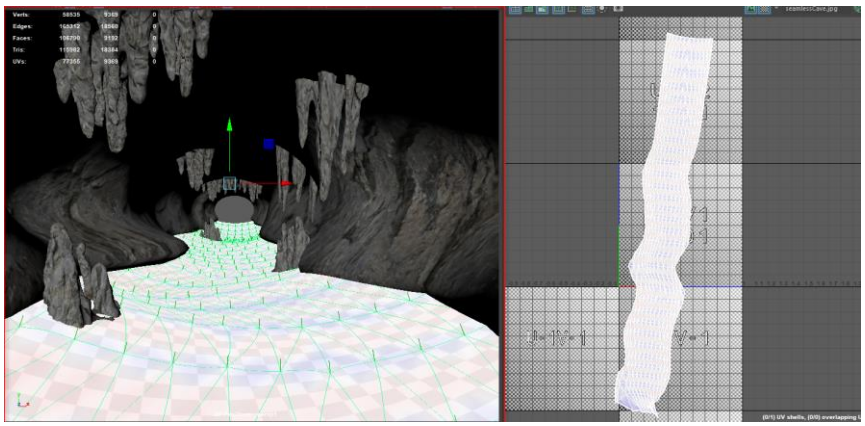


Figure 1.8 - UV mapping of Cave model

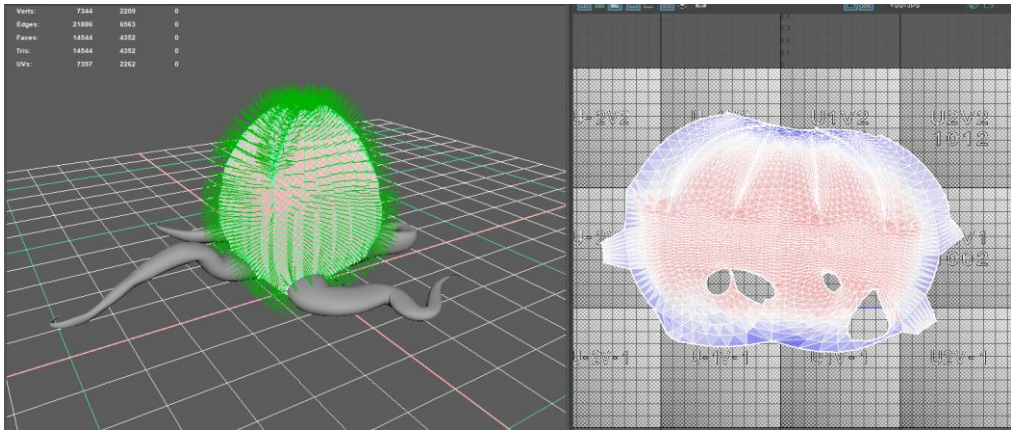


Figure 1.9 - UV mapping of Alien Egg Model

My models that make up the scene are textured with seamless image textures, as well as specular and normal maps. I used GIMP to convert the texture images to specular and normal maps. GIMP has a plugin which allows for easy conversion of an image to a normal map, and then the specular image was created by converting the texture to a greyscale image and adjusting the contrast. Two different textures were used. For the Alien eggs, the images and maps in Figure 2.0 were used and for the cave, stalagmites and stalactites the textures in Figure 2.1 were used.

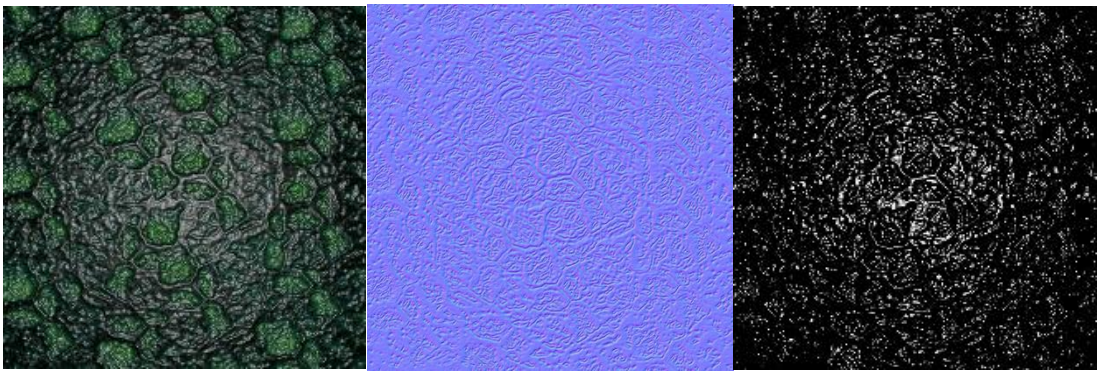


Figure 2.0 – Alien egg textures, original (left), normal map (centre) and specular map (right)

Figure 2.1 – Cave textures, original (left), normal map (centre) and specular map (right)

I created the following function `load_texture` to load in each of the textures. They are loaded in, before the integer associated with the texture is passed into the corresponding Object constructors. The Object class also contains methods where a specular and normal map can be associated with the Object, passing them into the fragment shader when the Object is drawn.

```
unsigned int load_texture(const char* file_name) {
	int width, height,numComponents;
	unsigned char* data = stbi_load(file_name,&width, &height, &numComponents, 3);
	if (data == NULL)
	  std::cerr << "Unable to load texture: " << file_name << std::endl;
	unsigned int m_texture;
	glGenTextures(1, &m_texture);
	glBindTexture(GL_TEXTURE_2D, m_texture);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
	  GL_UNSIGNED_BYTE, data);
	glGenerateMipmap(GL_TEXTURE_2D);
	stbi_image_free(data);
	return m_texture;
}
```

Hierarchical Model and Animation

Hierarchical Animation

The hierarchical model I implemented was the Alien model, consisting of a simple one-to-one relationship between the body, the base of the tail and the main body of the tail. I reworked my code for generating and drawing objects completely, creating an Object class for every model I wanted to load in. I did this in order to reduce the number of parameters required for the generateObjectBuffers method which took in the following parameters before:

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

The displayObject function used to draw each object took in a similar number of parameters:

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

The Object class I created gets around the issue of passing loads of parameters for every object every time I want to generate buffers or draw the object, which is messy. It allows me to pass in corresponding texture IDs, meshes and shader program IDs (in the event I want to use different shaders for different objects, although I ended up just using the same shader program for each object).

```
Object(const char* mesh_name, unsigned int textureId, GLuint shaderProgram) {
              mesh_data = load_mesh(mesh_name); //load the mesh
              textureID = textureId;
              shaderProgramID = shaderProgram;
              GenerateObjectBuffers();
              model = identity_mat4();
       }

void GenerateObjectBuffers();
void Display(vec3 position, vec3 rotation, vec3 scale);
void DisplayChild(int childOffset, vec3 posVec, vec3 rotVec, vec3 scaleVec);
Object createChildObject(const char* mesh_name);
void setSpecular(unsigned int specularTextureID, GLfloat newVal);
void setNormalMap(unsigned int normalMapID);
void setModel(ModelData mesh_data);
```

An Object is created for each model that I want drawn, and then drawing numerous instances of one object is possible just by calling Object.Display and passing in vectors that determine the position, rotation and scale of the instance. Forming my hierarchy is easy to do with this class. Calling Object.createChildObject and passing in the name of the mesh corresponding to the child creates a new Object for the child and pushes this to an array of child Objects for the parent Object. Then when drawing the hierarchy, I simply call Object.Display then Object.DisplayChild.

```
void Object::DisplayChild(int childOffset,vec3 posVec, vec3 rotVec, vec3 scaleVec) {
       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 parent matrix to the child matrix
       child_model = model * child_model;
       glUniformMatrix4fv(matrix_location, 1, GL_FALSE, child_model.m);
       Object child = childObjects[childOffset];
       glBindVertexArray(child.vao);
       glDrawArrays(GL_TRIANGLES, 0, child.mesh_data.mPointCount);
}
```

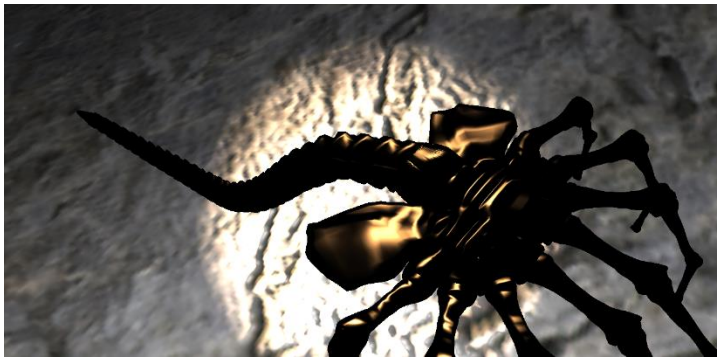


Fig 2.2 – Drawn tail hierarchy

The hierarchy moves in the corresponding axes whenever the user presses X, Y or Z. The updateTail function is called which takes in the corresponding coordinate of the vector that determines the rotation of the hierarchy.

```
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();
}
```

Key Frame Animation

Another change I made to my models is adjusting the position of the Alien model I previously created. Originally, I had positioned the tail in a very straight position which made it look quite rigid. In order to adjust its positioning, I decided to implement a (very basic) rig of joints. This allowed me to move it to a more natural looking position as shown in Figure 2.3 below:

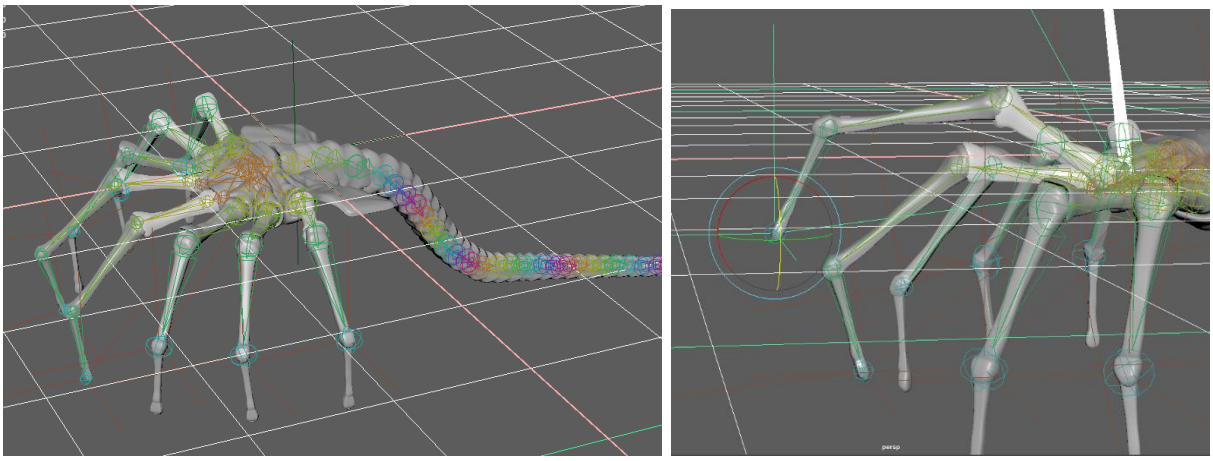
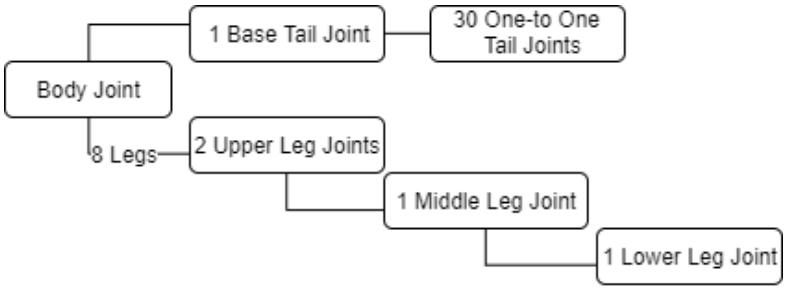


Figure 2.3 - Rigged Alien model (left) and IK handles on legs (right)

The joints are laid out in the following one-to-many hierarchy:



I also created IK handles for each of the legs to allow for easy positioning. Since I had a fully rigged model, I thought it would be interesting to experiment with keyframe animation to see the difference in style, and if it would actually work in OpenGL.

I implemented key frame animation on the body of the alien by positioning it in Maya and exporting each key frame as a separate .obj file. Then, I loaded each of them into my program and created a vector to push the models into in the order of which they would be drawn in. When the user presses I, this increases or decreases the index into the vector of keyframes and draws whatever model (keyframe) is contained at the index.

Figure 2.4 – Two of the keyframes obj files in 3D viewer

I also added a small decorative animation for the eggs, which consisted of a simple scaling up and down every time a new frame is drawn. This gave a pulsing effect to make them look more realistic. In the draw function the following function is called to update the eggs scale factor:

```
bool eggGrowing = true;
void updateEgg() {

	float delta = 0.0003f;
	if (egg_x > 5.2f) {
		eggGrowing = false;
	}
	else if (egg_x < 5.0f) {
		eggGrowing = true;
	}
	if (eggGrowing) {
		egg_x += delta;
		egg_y += delta;
		egg_y += delta;
	//have some eggs shrink and increase at a different rate
		egg2_x -= delta;
		egg2_y -= delta;
		egg2_z -= delta;
	}
	else {
		egg_x -= delta;
		egg_y -= delta;
		egg_y -= delta;
		egg2_x += delta;
		egg2_y += delta;
		egg2_z += delta;
	}
}
```