CSU44052 Computer Graphics Mid - Term Report

Name: Jack Joseph Gilbride

Student Number: 17340868

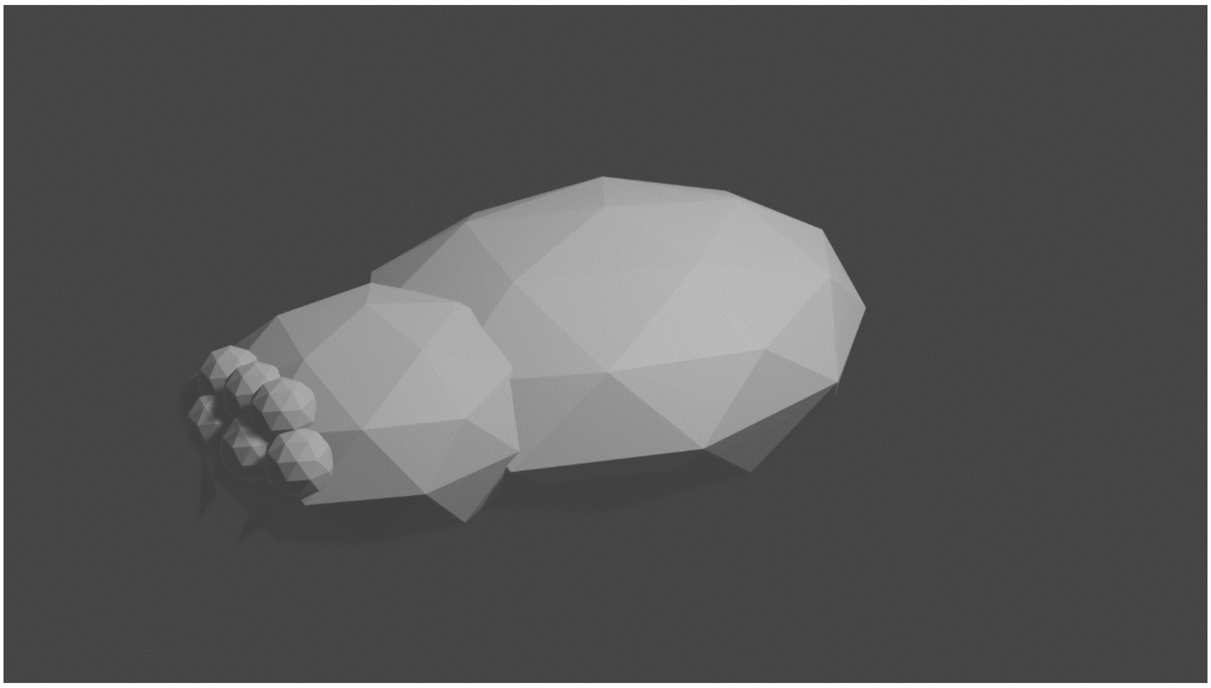
Introduction

The CSU44052 Computer Graphics Project is to create a "spooky" scene with animated 3D insects/creatures. This report is intended as a halfway point to the final project; the aim is to create a hierarchically structured insect that can be used in the final scene, and a camera to navigate the scene.

Creating the Models in Blender

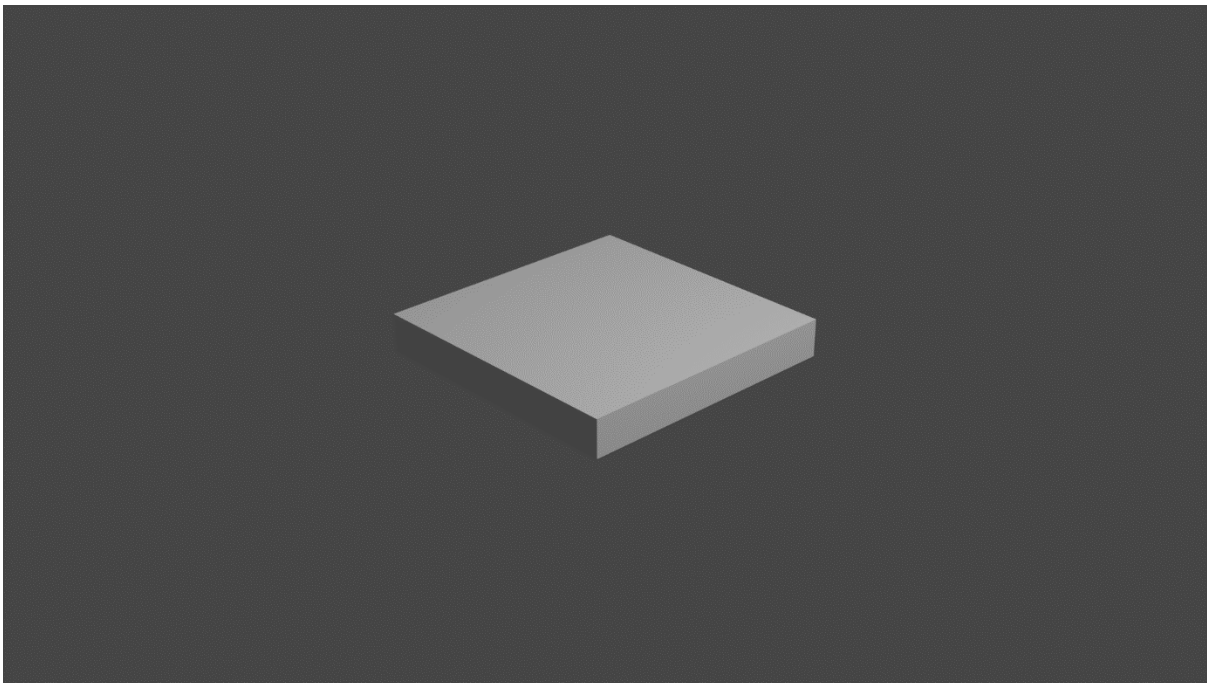
I created the models in Blender before importing them to OpenGL. When creating the models, it was important for me to have an idea of the hierarchy I wanted to implement. I decided that I wanted each of the spider's legs to be children of the main spider model. This was the most intuitive hierarchy to me, as each leg could have its own transformations (e.g. in a walking animation) while still being connected to the rest of the spider.

First, I created the main spider object. This consisted of 3D shapes representing the body, head, eyes and fangs of the spider. I gave it a large head with large eyes to give it an intimidating look up-close. I went with a low polygon look to add some character to the simple design beyond simple shapes, especially as there are no textures on the models at this stage of the project. The low polygon look also helps the spider to look intimidating close up. It has the bonus of being easier to render in real time for less powerful machines.



Next, I created the leg object. This sticks with the low polygon look and is made of a few simple shapes to fit in with the body. I created this as one model rather than a multi-part hierarchical model because I felt that I could get satisfying animation moving this entire leg in relation to the body, and the effect of moving individual joints on the leg would be negligible on the animation for the amount of effort involved. I only needed to create this model once as I could duplicate it in OpenGL.

Finally, a floor was required for the scene. I created this in Blender as it meant I could re-use code that I would need to use to create the spider and its legs. I created a simple tile, knowing that I could scale it in OpenGL to occupy the area needed for a floor.



Having created the spider, leg and tile, I was finished in Blender and it was time to implement the project in OpenGL.

Writing the Model Class

The next step was to import the model into OpenGL. This functionality could be intuitively grouped together, so I decided to put it into a class. The "Model" class would contain the functions and variables required to create and access the model's representation in the program.

Firstly, a data structure was needed to hold the model's data. The ModelData struct holds the number of points, its vertices, normal and texture coordinates. Texture coordinates are not used yet but will be useful in a later stage of the project.

```
struct ModelData
{
    size_t mPointCount = 0;
    std::vector<vec3> mVertices;
    std::vector<vec3> mNormals;
    std::vector<vec2> mTextureCoords;
};
```

Now that this struct exists, the Model needs five pointers, to:

- An instance of the ModelData struct, containing the vertices, normal and texture coordinates.
- A Vertex Array Object (VAO) to hold the object's representation on the graphics card.
- A constructor.
- A function to create the instance of the ModelData struct.
- A function to, given a shader, generate Vertex Buffer Objects (VBOs) for the models, and store it in the VAO.

These can be seen below:

```
class Model
{
public:
    unsigned int vao;
    ModelData mesh_data;
    Model(const char* file_name);
    void generateObjectBufferMesh(GLuint shaderProgramID);

private:
    ModelData load_mesh(const char* file_name);
};
```

The constructor for Model simply loads the mesh. We do this in a private function called load_mesh to improve readability of the constructor to understand the high level functionality at a glance. load_mesh takes the name of the Model's file created in Blender, and converts it to its ModelData representation. The assimp library has built-in functions to get the vertices, normal and texture coordinates from the file, so load_mesh uses these functions and loads the respective results into mesh_data in the appropriate form. So when the constructor exits, the model is now contained in mesh_data.

generateObjectBufferMesh requires a shader program to generate the model's VBOs. As a result it is not called from the constructor, instead it is available as a public function. The main program must call this after the Shaders have been compiled, and before drawing commences, as the Shaders are used to generate the VBOs, and the VBOs are used to draw the model.

Firstly, the function generates space for the VAO. Then, it binds it on the graphics card, ensuring that all of the following OpenGL functions are done in the context of the correct VAO.

```
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

Now that we are in context of the VAO of the model, we must generate VBOs for the model's data. The function creates three VBOs; one for the vertices, one for the normals and one for the textures. The code is the same for the three VBOs and can be done in sequence.

As an example, look at the VBO for the model's vertices. First it initialized, generated and binded for the following OpenGL functions. The VBO is then populated with the normals from mesh_data. Next, we populate the vertex attribute pointer with the location of the vertices attribute in the Shader.

```
unsigned int vp_vbo = 0;
unsigned int vp_vbo = 0;
glGenBuffers(1, &vp_vbo);
glBindBuffer(GL_ARRAY_BUFFER, vp_vbo);
glBufferData(GL_ARRAY_BUFFER, mesh_data.mPointCount * sizeof(vec3), &mesh_data
.mVertices[0], GL_STATIC_DRAW);
GLuint vertex_position_location_in_shader = glGetAttribLocation(
      shaderProgramID, "vertex_position");
glEnableVertexAttribArray(vertex_position_location_in_shader);
glVertexAttribPointer(vertex_position_location_in_shader, 3, GL_FLOAT,
      GL_FALSE, 0, NULL);
```

The appropriate data for the normals has been loaded into the VBO and vertex attribute array. The same is done for both the vertex normals and vertex textures. This was all done in the context of the model's VAO, so to draw the model in the main program, we will just need to bind the appropriate VAO and call the draw function. In essence setting up the VBOs and VAO at the start of the program saves a lot of work in the display function later on.

Compiling the Shaders

As mentioned, to render the models, the Shaders need to be compiled. We were given a simple fragment shader and simple texture shader to work with, and functions to generate them.

All of this seemed like functionality which could be segmented from the program, so I created a class called Shaders with this specific functionality. All of this functionality originally returned one single int, the Shader Program ID, so it did not make much sense to instantiate an object for it. Instead I built it as static functionality into the Shader.h file. This way it was segmented from the program without the overhead of creating an object.

The file contains one public function called CompileShaders, which generates a shader program and attaches the simple vertex shader and simple fragment shader to it. To attach the shaders, CompileShaders calls a private function called AddShader, which compiles, binds and attaches a shader to a shader program using OpenGL library functions. AddShader itself makes use of another private function, readShaderSource, which implements file reading functionality to convert the file to a string.

The same shader program is used for each of the models in the program, so CompileShaders is only used once. The ID of the Shader is kept in a variable called ShaderProgramID in the main program,

Moving the Camera

At this point the models can be instantiated and linked with shaders, so we need to think about how they will be drawn in the world. As described later in the report, this will involve applying transformations to the different objects to scale, translate and rotate them appropriately. This will initialise the models in the correct position with respect to each other, but will also be used to move the spider in the world space.

As well as moving the model in the world space, we also wish to move the user's view. The user is looking into the virtual world via a viewport, and if we apply transformations to this viewport, the user's view changes. This is analogous to moving a camera when filming something.

Keeping track of the transformations to apply to each model object as well as the camera in the main program can get confusing, so these can be put into a Camera class. The Camera class contains four vectors. One represents its position in the world. The other three represent its orientation – its forward direction, up direction and right direction. It also contains a constructor, a function to move and a function to rotate.

```
Class Camera
{
public:
    vec3 up;
    vec3 direction;
    vec3 position;
    vec3 right;
    Camera(vec3 position, vec3 direction);
    void move(GLfloat forward, GLfloat horizontal, GLfloat vertical);
    void rotate(GLfloat x, GLfloat y, GLfloat z);
};
```

The constructor simply initialises these vectors, while the functions manipulate them. The rotate function takes three floats, x, y and z, to rotate the camera about. It uses these to update the direction vector, up vector and right vector, which affects the pitch, yaw and roll of the camera. The move function allows the camera to move along these vectors by a factor of the float values passed to it.

Handling User Input

Much like the Models and the Shaders, an instance of the Camera is initialized in the mainline to utilize its functions and variables. It makes sense to map the Camera movements to keyboard and mouse input, that way the user can control their view of the environment. There are two functions to handle these respectively, keypress and mouseMoved, which are linked to glut, the library used for I/O, as below:

```
glutKeyboardFunc(keypress);
glutPassiveMotionFunc(mouseMoved);
```

The exact mapping of user input to Camera transformations is a design choice. I chose to simply let the user "walk" along the ground of the scene. This restricts their movement to the ground, giving a "spookier" feel as the spider will tower over them in the scene.

To restrict movement to the ground of the scene, the camera is only rotated based on the mouse x. This stops the camera "flying" in the scene, but can easily be changed in future to rotate based on the y to better manoeuvre the scene. Glut uses a function to bind the pointer to the centre of the window; this makes it invisible and makes full 360 rotation possible, making the feature more immersive. The user must be able to disable this (so they can, for example, navigate outside the window to exit it), so this mouse functionality depends on a variable called mouseInput which can be toggled by the user.

```
void mouseMoved(int newMouseX, int newMouseY) {
    if (mouseInput) {
        camera.rotate((GLfloat)newMouseX - width / 2, 0, 0);
        glutWarpPointer(width / 2, height / 2);
    }
}
```

The keypress function is a switch statement which carries out different functionality based on the key pressed. The user can "walk" around the scene with WASD. The user can also apply a series of transformations to the spider with the number keys at the top of the keyboard. Each transformation has an inverse, which is performed by pressing SHIFT along with the key. On top of this there are keybindings to toggle mouse input, toggle animation and quit the program.

This is the full list of keybindings:

- W – walk forward
- A – walk left
- S – walk backward
- D – walk right
- M – toggle mouse input
- E – toggle animation
- Q – quit program
- P – change perspective fov
- ` – reset spider
- 1, ! – Rotate spider on its x-axis
- 2, " – Rotate spider on its y-axis
- 3, £ - Rotate spider on it's z-axis
- 4, $ - Move spider along its x-axis
- 5, % - Move spider along its y-axis
- 6, ^ - Move spider along its z-axis
- 7, & - Scale spider along its x-axis
- 8, * - Scale spider along its y-axis
- 9, (- Scale spider along its z-axis
- 0,) – Scale spider uniformly

Much like the WASD movement changes variables in the camera object, the spider transformation operations change global variables which affect the transformations made to the spider in the draw loop.

Transforming the Model & View

The spider is transformed by utilising functions in the maths_funcs file that was provided for this assignment. The "model" matrix represents the transformations applied to the model, and begins as an identity matrix. Each of the keybindings update different global variables, so in the draw function, the matrix is scaled by, rotated by, and translated by the relevant global variables.

There are also transformations to initially position the correct place in the world. These transformations are fixed in every iteration, so that the transformations applied by the user also happen in the context of them.

```
model = identity_mat4();
model = scale(model, vec3(scale_x, scale_y, scale_z));
model = rotate_x_deg(model, rotate_x);
model = rotate_y_deg(model, 180);
model = rotate_y_deg(model, rotate_y);
model = rotate_z_deg(model, rotate_z);
model = translate(model, vec3(0.0f, 1.4f, 10.0f));
model = translate(model, vec3(translate_x, translate_y, translate_z));
```

Now that the model is positioned correctly in the world, the camera also needs to be positioned. The maths_funcs file also contains functionality to "look at" a point from another point, this can simply be passed the vectors from the camera:

```
view = look_at(
    camera.position,
    camera.position + camera.direction,
    camera.up
);
```

Now that the matrices are transformed appropriately, it is time to draw the spider. The spider's vao in bound, the matrices are linked to their respective locations in the shaders, and the spider is drawn.

Drawing the Legs – Animation & Hierarchy

Much like the spider body, each leg will need to be drawn individually. The logic is the same for each; start with an identity matrix, apply the relevant transformations and draw. I realised that there would be some overlap in the functionality, so I put the matrices into an array, which could be iterated over for any repeated functions.

```
glBindVertexArray(leg.vao);
mat4 legArray[8 * sizeof(mat4)];
```

I did not map the movement of any legs to user input, instead I decided to animate them. Every time the scene was updated, I calculated a delta since the last time. I passed this delta to a function to rotate the spider's legs. This delta is calculated based on time, rather than frames, to ensure that the legs move smoothly, and at the same speed, no matter what the frame rate is at the moment. This delta is passed to the function twice, for two leg sets. The initial leg rotations for both sets are different, meaning both sets move out of sync in such a way to make it look like the spider is walking.

```
void updateScene() {
    static DWORD last_time = 0;
    DWORD curr_time = timeGetTime();
    if (last_time == 0)
        last_time = curr_time;
    float delta = (curr_time - last_time) * 0.001f;
    last_time = curr_time;

    if (animation) {
        update_leg_rotation(leg_set_1_rotate_x, leg_set_1_rotate_x_increasing,
 delta);
        update_leg_rotation(leg_set_2_rotate_x, leg_set_2_rotate_x_increasing,
 delta);
    }
    // Draw the next frame
    glutPostRedisplay();
}
```

The update_leg_rotation simply increase the leg rotation until it gets to 5, then decreases it until it gets to -5, and so on. So the leg is rotated back and forward 10 degrees in its own x axis.

Now that there is functionality to animate the legs, we can apply transformations to all 8 of them.

```
legArray[0] = scale(identity_matrix, leg_scaling_factor);
legArray[0] = rotate_x_deg(legArray[0], leg_set_1_rotate_x);
legArray[0] = translate(legArray[0], vec3(0.0f, 0.0f, -1.5f));

legArray[1] = scale(identity_matrix, leg_scaling_factor);
legArray[1] = rotate_y_deg(legArray[1], 180);
legArray[1] = rotate_x_deg(legArray[1], leg_set_2_rotate_x);
legArray[1] = translate(legArray[1], vec3(0.0f, 0.0f, -1.5f));
```

Each leg is scaled down slightly to fit better with the spider model. The spider should have four left legs and four right legs, so every second leg is rotated by 180 degrees on its y-axis. The leg is also rotated in x depending on which animation set it belongs to, and moved in its z-axis depending on which pair it belongs to, so that each pair is spaced out along the spider's body.

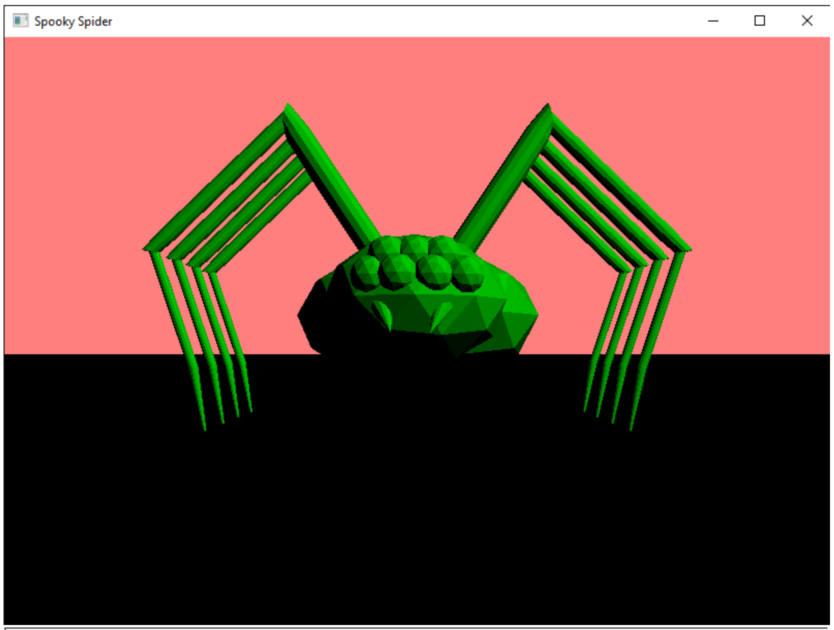
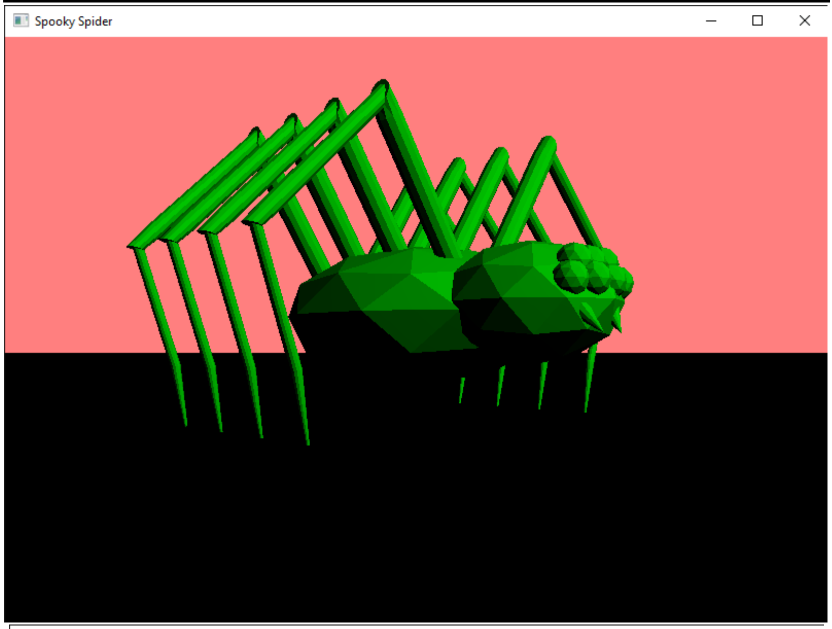
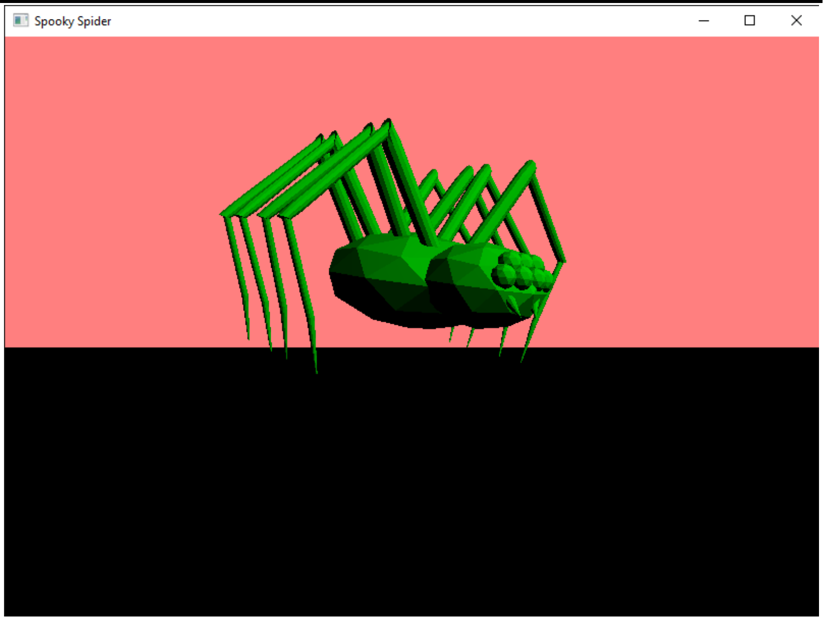
These legs form a hierarchy with the spider. Intuitively they are part of the spider, and any transformations to the spider should apply to the legs too. I.e. when the spider moves forward, the legs should move forward too, to maintain their relative position to the spider ("attached" to it). To ensure that this happens, multiply the model's transformation matrix by the leg's transformation matrix. The leg's transformations are now a result of both transformation sets.

```
for (int i = 0; i < 8; i++) {
    legArray[i] = model * legArray[i];
    glUniformMatrix4fv(matrix_location, 1, GL_FALSE, legArray[i].m);
    glDrawArrays(GL_TRIANGLES, 0, leg.mesh_data.mPointCount);
}
```

The last thing to be drawn is the floor tile. The only transformation to apply is to scale it in the x and z axes to occupy a large space in the scene. Again, its VAO is bound and it is drawn.

Result

A video of the progress made so far can be seen here: https://youtu.be/9pl7gN1Kon0. The source code for the project so far can be seen here: jackgilbride999/Computer-Graphics (github.com). Here are some screenshots of the hierarchical spider as it is currently:


Spooky Spider

Spooky Spider

Spooky Spider

Sources:

Here is a list of online resources I used to help me with this stage of the project:

- I used this video to help me model the spider in Blender: https://youtu.be/6DdqndkKkGw
- I used the following sources to help me with the Camera class:
    - https://youtu.be/HhXzKOMkA1k
    - https://youtu.be/v6RZRPo0O3k
    - https://youtu.be/7oNLw9Bct1k
    - cpp-opengl/Camera.cpp
    - LearnOpenGL - Camera
- I used the Lab04 sample project provided in class to help with the shaders, vao/vbo functionality and transformations.
- CSU44052 Weekly Tutorials.
- Discussion with classmates at various stages.