Assignment #4: Object Hierarchy

Ben Colwell – 15325781

Note: Have not written about these in the order specified in the lab instructions – but rather what makes more chronological sense for what I did.

Part 01

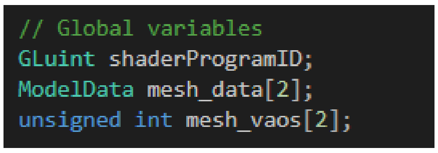
(f) Loading your own models (~20%)

Much like what I did in lab 03, the very first thing I did was to swap out all of Anton's methods for GLM – I just find it easier to use and there's much more information/tutorials online for GLM if I get stuck on anything.

For finding a model to load I used the website Turbosquid. After testing numerous models, I found that the program seemed to handle .dae files best, but even the they wouldn't always behave as expected. For several of the models I tested they caused distorted and odd shapes to be generated. In the end I settled on some seasonally appropriate Halloween pumpkin and skull models which behaved very well in my program.

Loading these models into the program involved a few steps. Firstly I adapted the provided '`generateObjectBufferMesh()`' program to take in parameters so that it could be used for multiple different models rather than hard coded to use the smooth monkey head model provided.

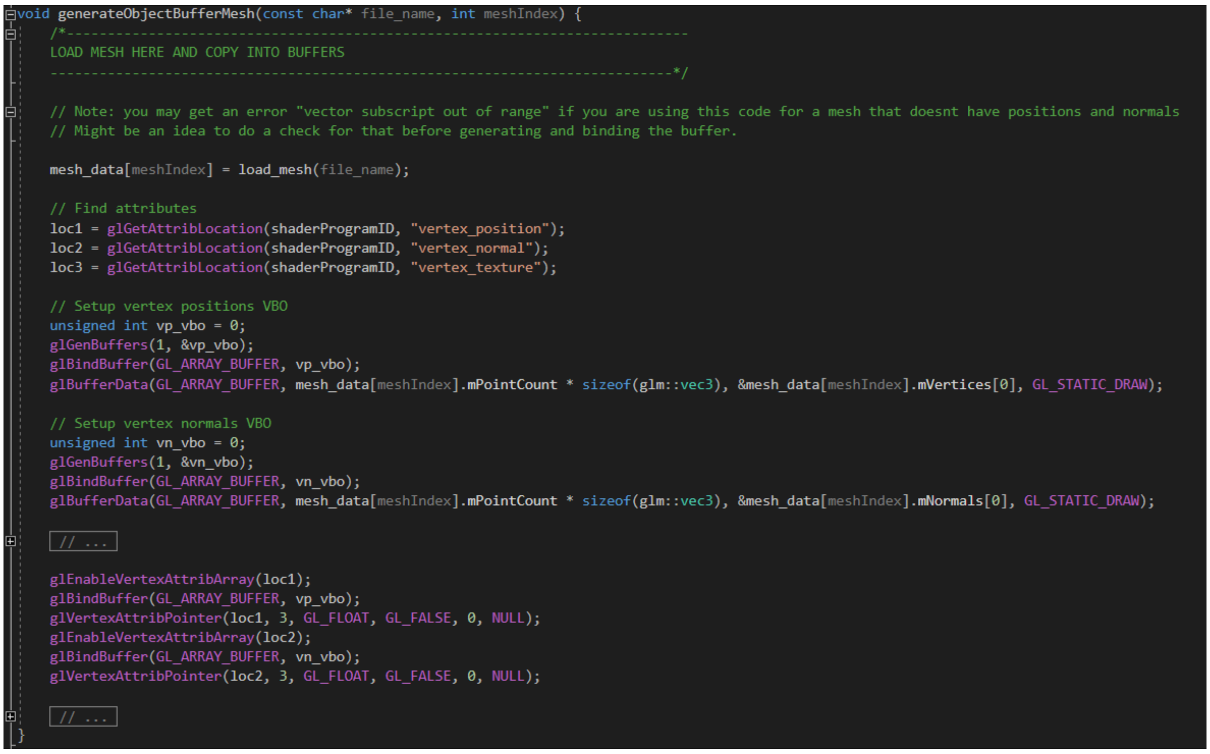
The way I went about this was to have multiple global arrays for the variables required for each mesh:



```
// Global variables
GLuint shaderProgramID;
ModelData mesh_data[2];
unsigned int mesh_vaos[2];
```

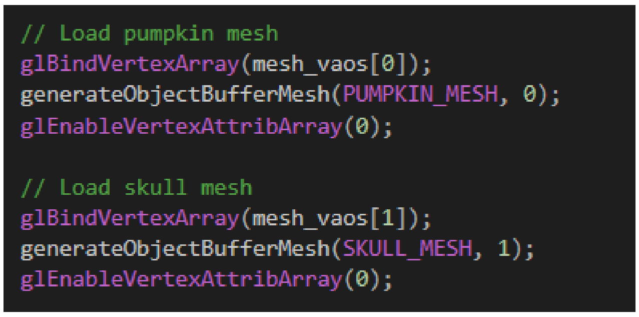
And then assign a mesh index to each mesh – for example: my pumpkin mesh was mesh 1, therefore all its variables were in index 0 of each array, and so on.

With this in mind, I could write my generation function as so:



```
void generateObjectBufferMesh(const char* file_name, int meshIndex) {
    /*----------------------------------------------------------------------------
    LOAD MESH HERE AND COPY INTO BUFFERS
    ----------------------------------------------------------------------------*/

    // Note: you may get an error "vector subscript out of range" if you are using this code for a mesh that doesnt have positions and normals
    // Might be an idea to do a check for that before generating and binding the buffer.

    mesh_data[meshIndex] = load_mesh(file_name);

    // Find attributes
    loc1 = glGetAttribLocation(shaderProgramID, "vertex_position");
    loc2 = glGetAttribLocation(shaderProgramID, "vertex_normal");
    loc3 = glGetAttribLocation(shaderProgramID, "vertex_texture");

    // Setup vertex positions VBO
    unsigned int vp_vbo = 0;
    glGenBuffers(1, &vp_vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vp_vbo);
    glBufferData(GL_ARRAY_BUFFER, mesh_data[meshIndex].mPointCount * sizeof(glm::vec3), &mesh_data[meshIndex].mVertices[0], GL_STATIC_DRAW);

    // Setup vertex normals VBO
    unsigned int vn_vbo = 0;
    glGenBuffers(1, &vn_vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vn_vbo);
    glBufferData(GL_ARRAY_BUFFER, mesh_data[meshIndex].mPointCount * sizeof(glm::vec3), &mesh_data[meshIndex].mNormals[0], GL_STATIC_DRAW);

    // ...

    glEnableVertexAttribArray(loc1);
    glBindBuffer(GL_ARRAY_BUFFER, vp_vbo);
    glVertexAttribPointer(loc1, 3, GL_FLOAT, GL_FALSE, 0, NULL);
    glEnableVertexAttribArray(loc2);
    glBindBuffer(GL_ARRAY_BUFFER, vn_vbo);
    glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0, NULL);

    // ...
}
```

My models were then loaded and assigned to VAO's within init(). I did have issues for a while because I hadn't properly setup my VAO's – forgetting that one is enabled by default but when two or more are needed you must create them yourself.



```
// Load pumpkin mesh
glBindVertexArray(mesh_vaos[0]);
generateObjectBufferMesh(PUMPKIN_MESH, 0);
glEnableVertexAttribArray(0);

// Load skull mesh
glBindVertexArray(mesh_vaos[1]);
generateObjectBufferMesh(SKULL_MESH, 1);
glEnableVertexAttribArray(0);
```

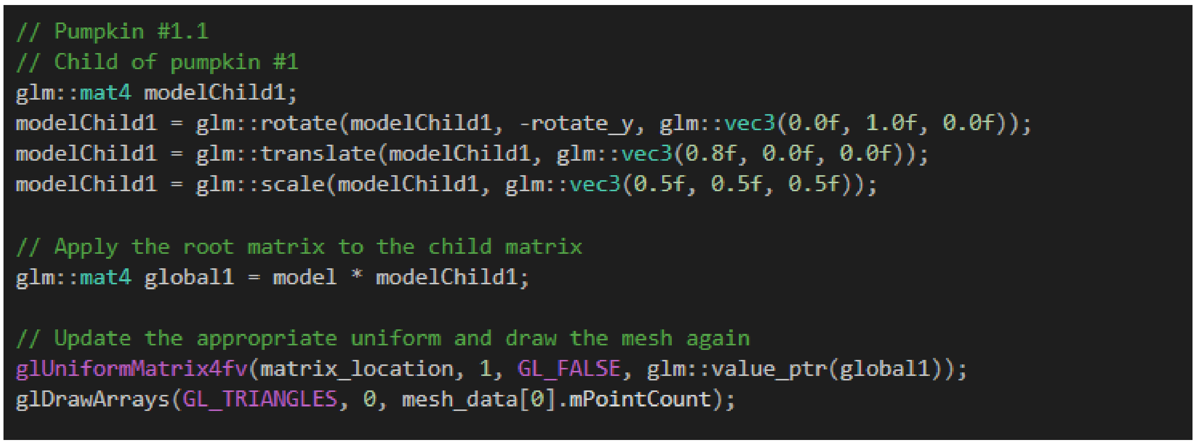
With this all setup drawing either of my models was as simple as binding the right VAO before calling a draw.

Part 02

(c.) Show a one-to-many relationship (~20%)

For my one-to-many relationship I decided to have 4 child pumpkins rotating around the root pumpkin and one sitting on top that spun.

The root model was loaded as normal, just like in the supplied code, and then for each of the child pumpkins I have the following:



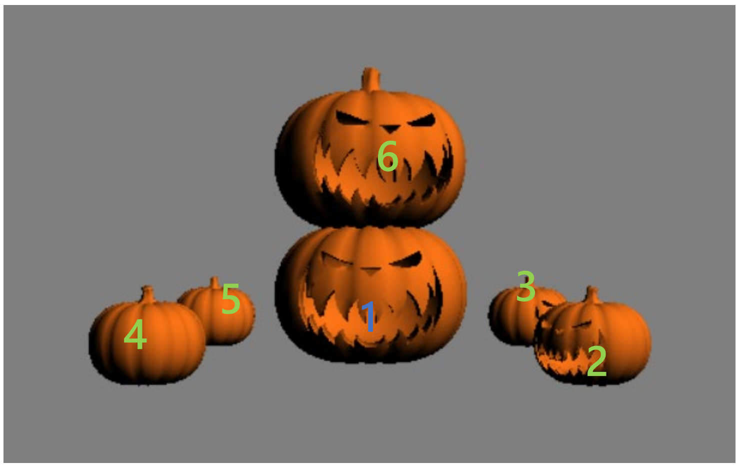
```
// Pumpkin #1.1
// Child of pumpkin #1
glm::mat4 modelChild1;
modelChild1 = glm::rotate(modelChild1, -rotate_y, glm::vec3(0.0f, 1.0f, 0.0f));
modelChild1 = glm::translate(modelChild1, glm::vec3(0.8f, 0.0f, 0.0f));
modelChild1 = glm::scale(modelChild1, glm::vec3(0.5f, 0.5f, 0.5f));

// Apply the root matrix to the child matrix
glm::mat4 global1 = model * modelChild1;

// Update the appropriate uniform and draw the mesh again
glUniformMatrix4fv(matrix_location, 1, GL_FALSE, glm::value_ptr(global1));
glDrawArrays(GL_TRIANGLES, 0, mesh_data[0].mPointCount);
```

The first chunk of code simply positions the model where I wanted it, 0.8f along the X-axis and half size, rotate_y is constantly updated in updateScene() causing the pumpkin to move. Each of the other three small pumpkins were positioned at 90/180/270 degrees around the Y axis before translating out so that they would be evenly spread.

Multiplying the translation matrix of this model (modelChild1) by the root model creates the parent-child relationship between them and causes this pumpkin to rotate around the root pumpkin rather than just spin on the spot.

The pumpkin is then drawn and this whole process is repeated (with minor changes) for each of the child pumpkins to give the following, where pumpkin 1 is the root, and pumpkins 2-6 are its children in the one-to-many relationship:

Output:



Part 03

(b.) Show a one-to-one relationship (~10%)

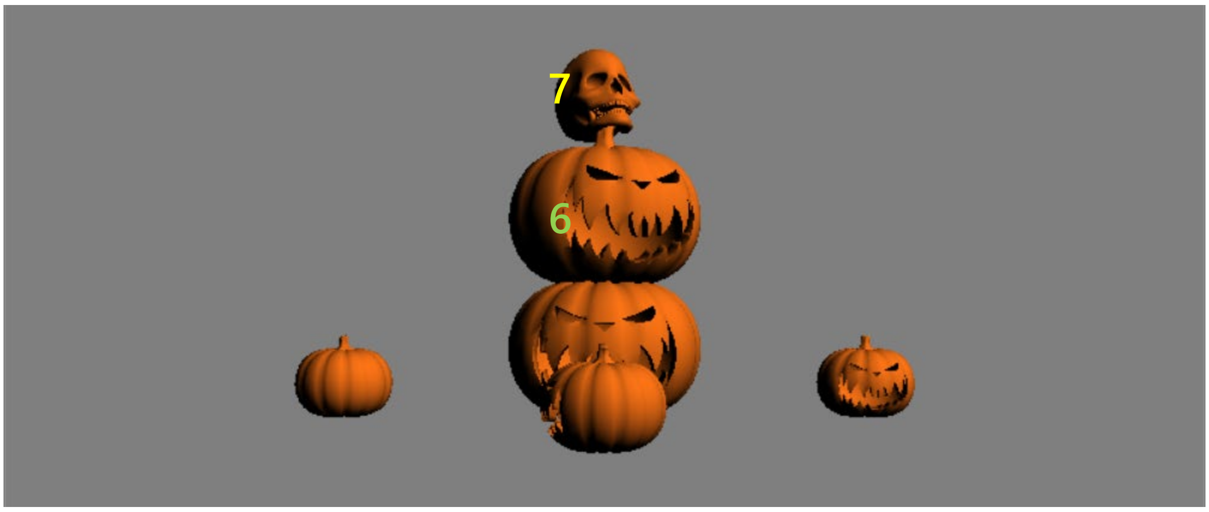
To do this I decided to add a child 'skull' model to the top pumpkin, making it the grandchild of the root model.

```
// Bind skull mesh vao
glBindVertexArray(mesh_vaos[1]);

// Skull #1.5.1
// Child of pumpkin #1.5 - grandchild of pumpkin #1
glm::mat4 modelChild6;
modelChild6 = glm::translate(modelChild6, glm::vec3(0.0f, 0.5f, 0.0f));

// Apply the root matrix and the parent matrix to the child matrix
glm::mat4 global6 = model * modelChild5 * modelChild6;

// Update the appropriate uniform and draw the mesh again
glUniformMatrix4fv(matrix_location, 1, GL_FALSE, glm::value_ptr(global6));
glDrawArrays(GL_TRIANGLES, 0, mesh_data[1].mPointCount);
```

Firstly, the VAO that contained the skull information had to be bound, otherwise we would just draw another pumpkin. Making it a child of pumpkin #1.5 (the top one) rather than another child of the root meant we had to multiply its translation matrix by not only the root translation matrix but also, it's parent (pumpkin 1.5). I figured out that it's important here to multiply the current model's matrix by the parents' model matrix only, not by the global matrix that is created for drawing pumpkin 1.5 – otherwise we would be multiplying by the root models' matrix twice.

This creates a one-to-one relationship between the top pumpkin and a skull on top of it, you can see from the code that there is no rotation happening, and yet because its parent model is rotating it will rotate with it (as shown). In the picture pumpkin 6 forms a one-to-one relationship with the skull (7).

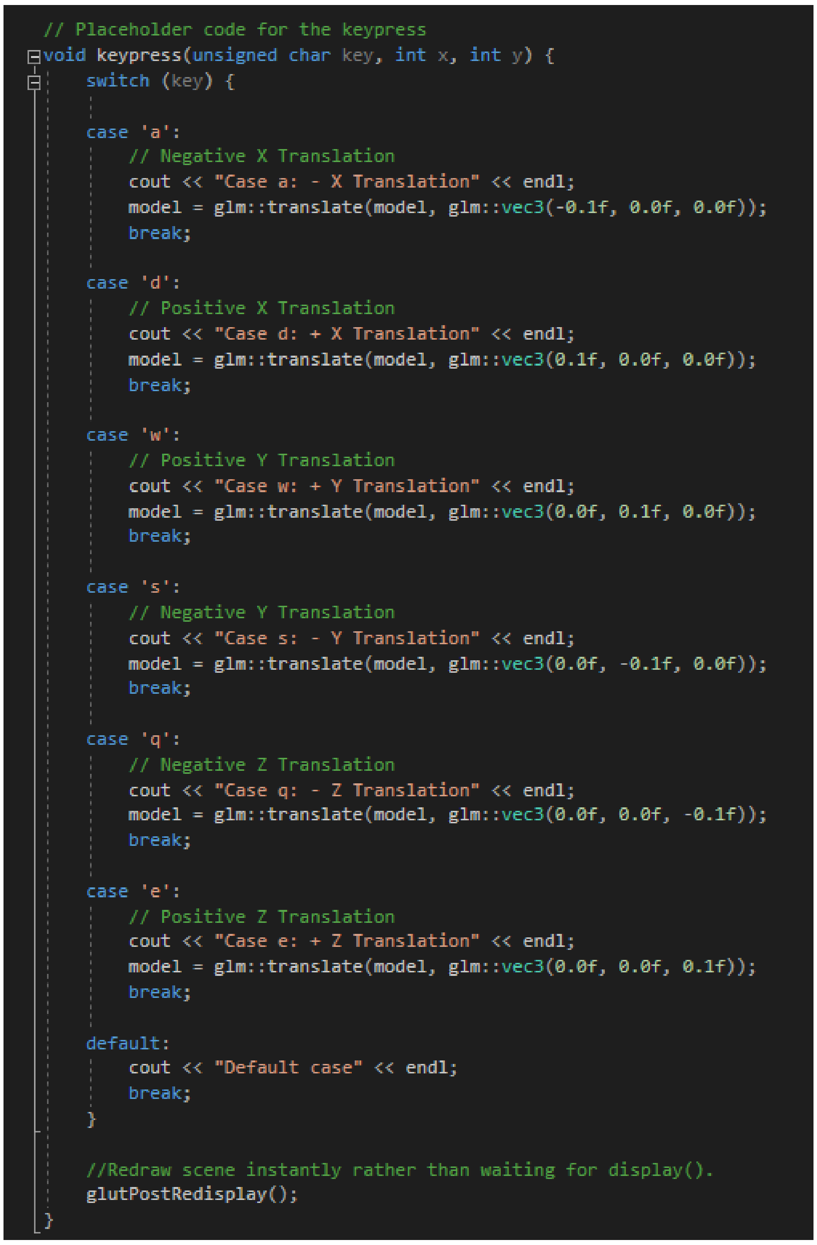
Output:



Part 04

(d.) Keyboard control of the translation of the root object (~10%)

This was very similar to keyboard control of a triangle completed in previous assignments. Using gluts function '`glutKeyboardFunc`' I call the following function whenever a key is pressed:

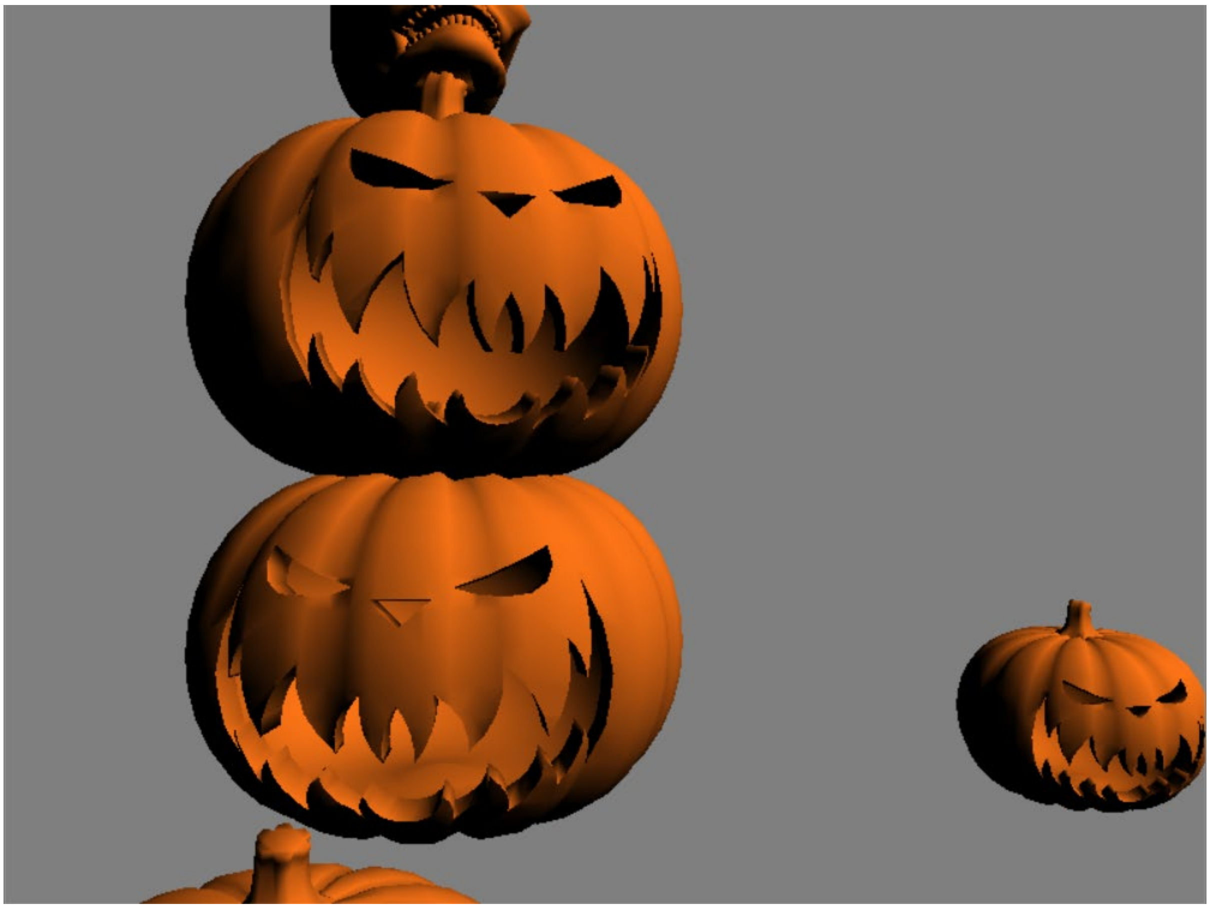


```
// Placeholder code for the keypress
void keypress(unsigned char key, int x, int y) {
    switch (key) {

    case 'a':
        // Negative X Translation
        cout << "Case a: - X Translation" << endl;
        model = glm::translate(model, glm::vec3(-0.1f, 0.0f, 0.0f));
        break;

    case 'd':
        // Positive X Translation
        cout << "Case d: + X Translation" << endl;
        model = glm::translate(model, glm::vec3(0.1f, 0.0f, 0.0f));
        break;

    case 'w':
        // Positive Y Translation
        cout << "Case w: + Y Translation" << endl;
        model = glm::translate(model, glm::vec3(0.0f, 0.1f, 0.0f));
        break;

    case 's':
        // Negative Y Translation
        cout << "Case s: - Y Translation" << endl;
        model = glm::translate(model, glm::vec3(0.0f, -0.1f, 0.0f));
        break;

    case 'q':
        // Negative Z Translation
        cout << "Case q: - Z Translation" << endl;
        model = glm::translate(model, glm::vec3(0.0f, 0.0f, -0.1f));
        break;

    case 'e':
        // Positive Z Translation
        cout << "Case e: + Z Translation" << endl;
        model = glm::translate(model, glm::vec3(0.0f, 0.0f, 0.1f));
        break;

    default:
        cout << "Default case" << endl;
        break;
    }

    //Redraw scene instantly rather than waiting for display().
    glutPostRedisplay();
}
```

This, quite straightforwardly, modifies the translation matrix of the root model based on key input. The only difficulty I ran into was initially I was setting up my root models' translation matrix within display. The problem with doing that is that every time display is run (many times a second), it reset the matrix back to its initial setup because it re-ran my setup code for it.

To get around this I simply made the rood models' translation matrix global and established it within init() rather than display. With this fixed keyboard control of the root model was functioning – with all its children & grandchildren following its translations.

Sample Output:

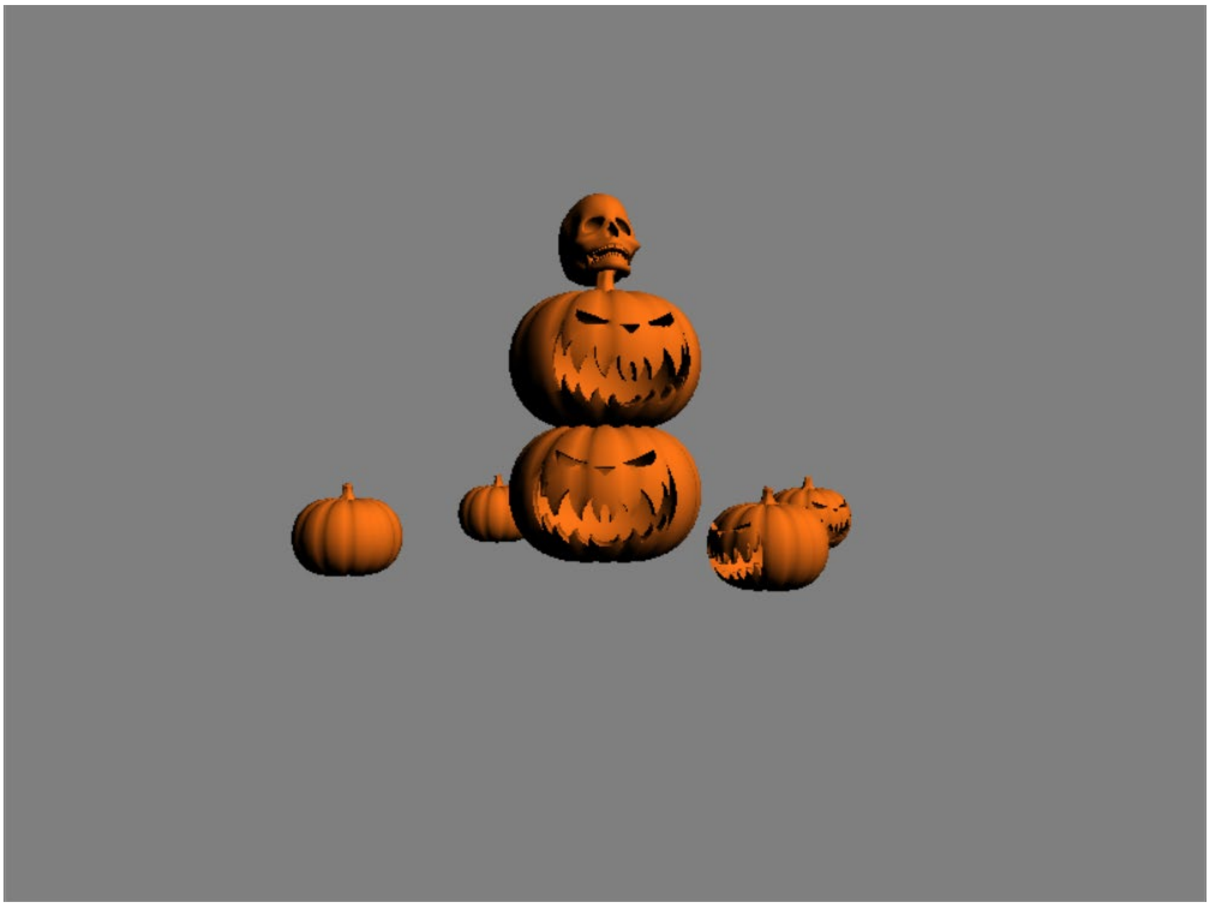


Conclusion/Notes

The following parts were implicitly completed over the previous parts:

- *a. The hierarchy should be constructed from at least 5 objects (~20%)*
- *e. Interesting/Inventive/Unusual Structure (~20%)*

Overall my final output looked as follows:



Happy Halloween!