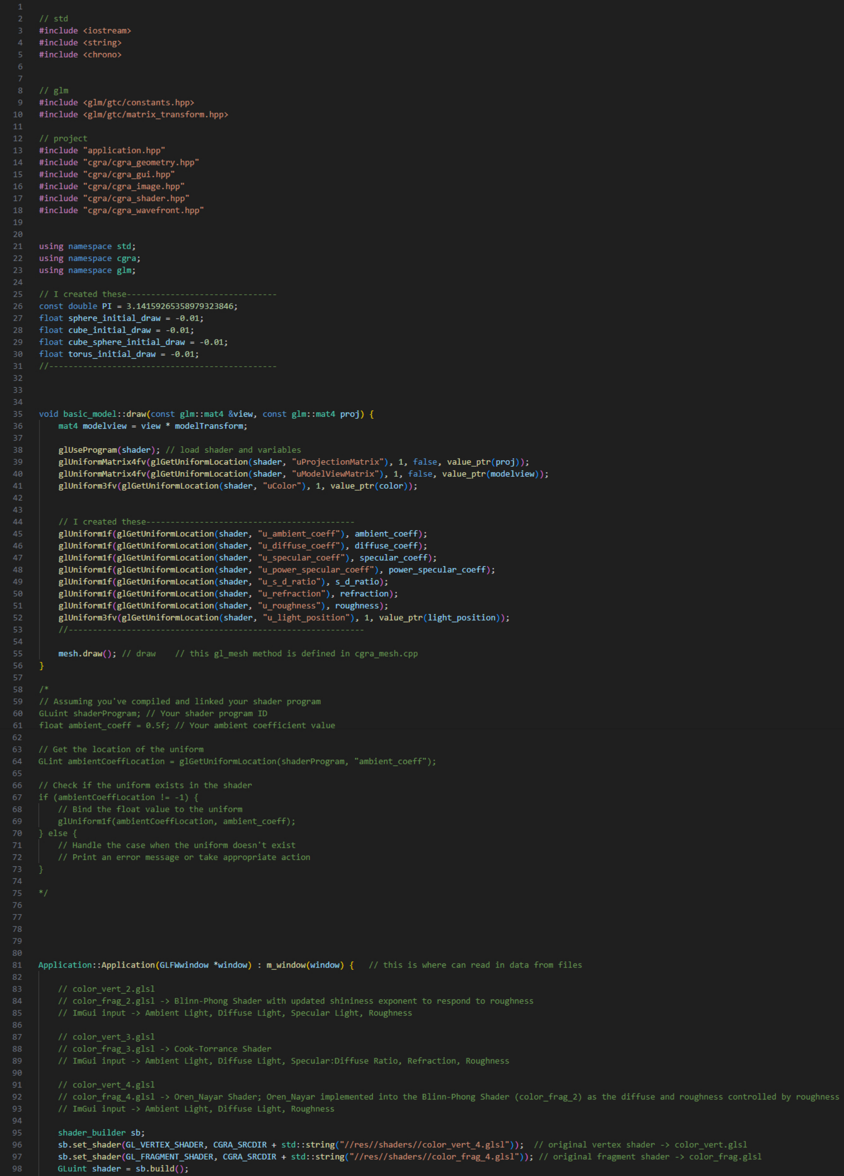
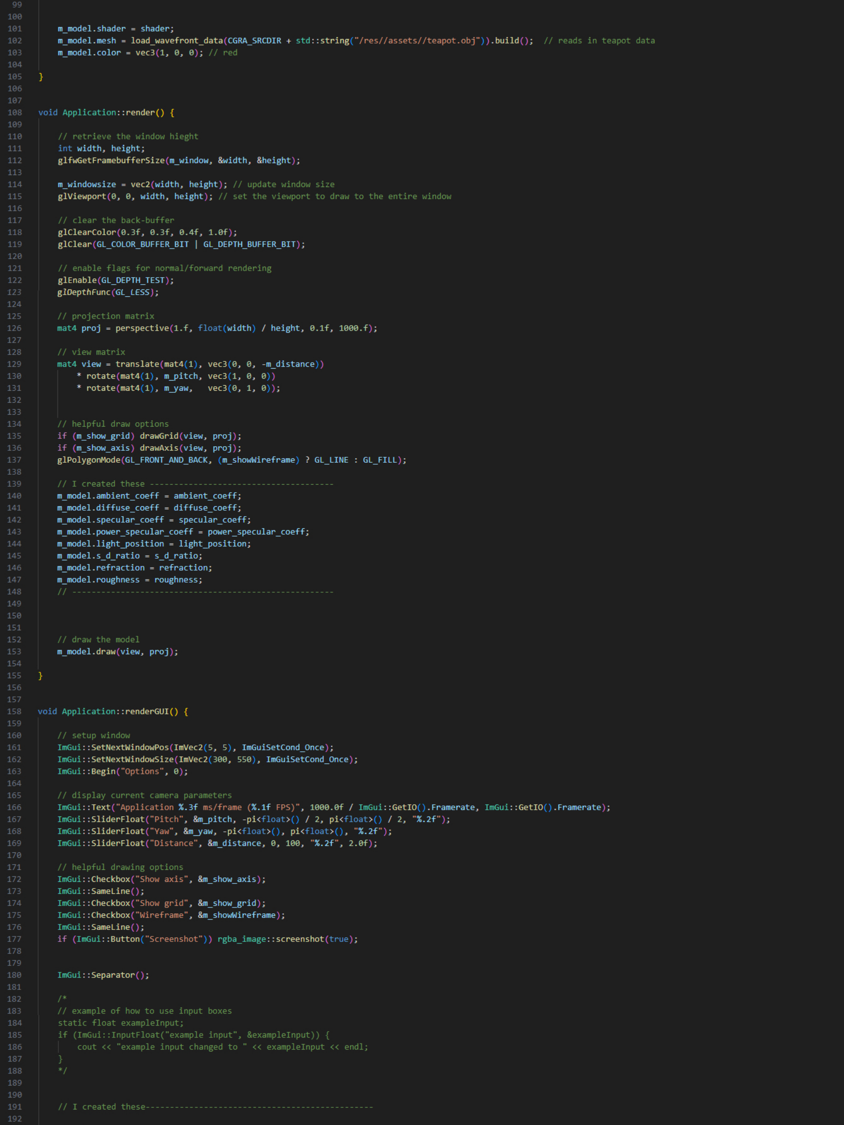
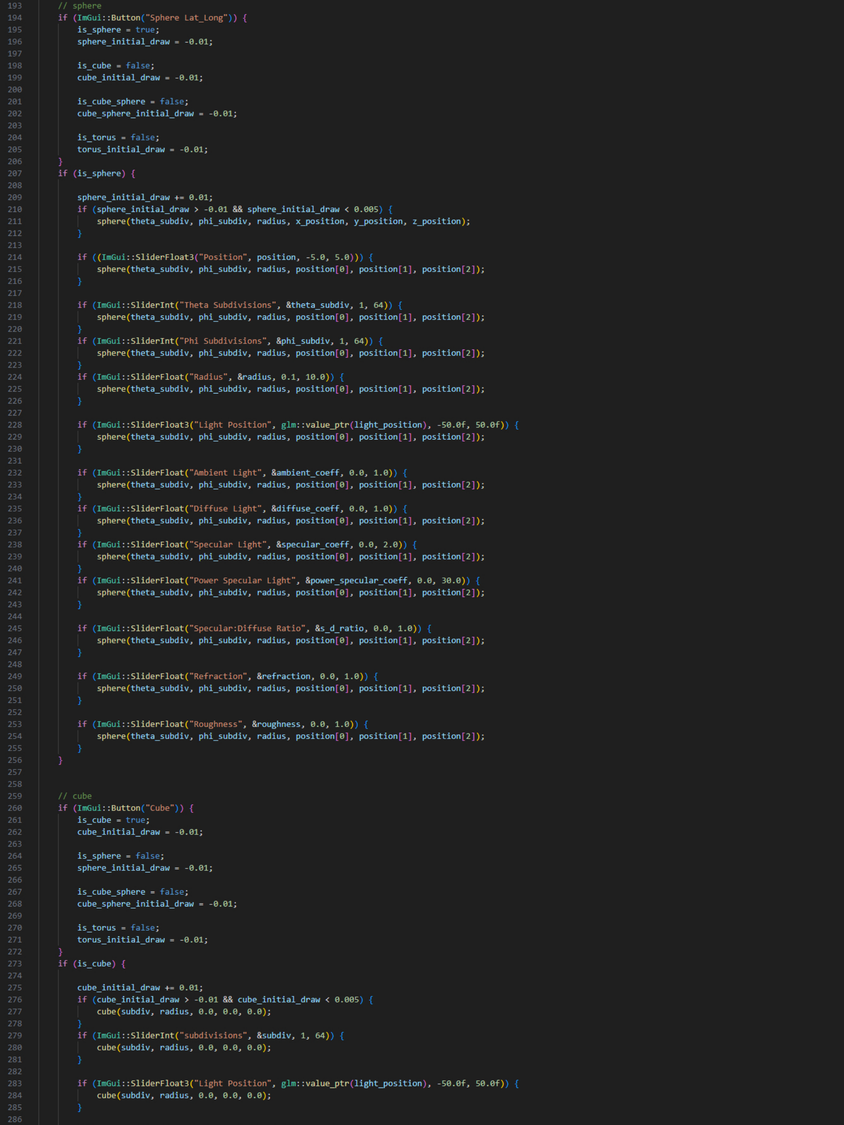


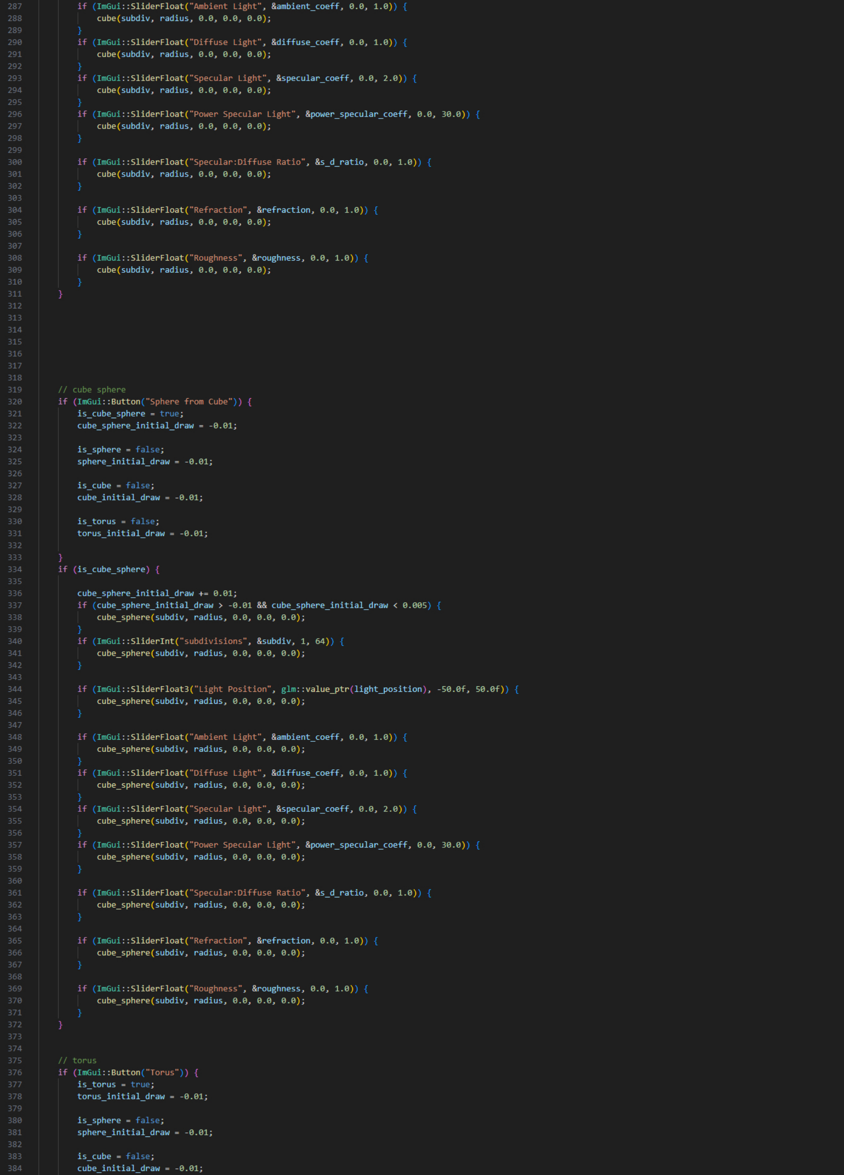
```

// std
#include <iostream>
#include <string>
#include <chrono>


// glm
#include <glm/gtc/constants.hpp>
#include <glm/gtc/matrix_transform.hpp>

// project
#include "application.hpp"
#include "cgra/cgra_geometry.hpp"
#include "cgra/cgra_gui.hpp"
#include "cgra/cgra_image.hpp"
#include "cgra/cgra_shader.hpp"
#include "cgra/cgra_wavefront.hpp"


using namespace std;
using namespace cgra;
using namespace glm;

// I created these------------------------------
const double PI = 3.14159265358979323846;
float sphere_initial_draw = -0.01;
float cube_initial_draw = -0.01;
float cube_sphere_initial_draw = -0.01;
float torus_initial_draw = -0.01;
//----------------------------------------------



void basic_model::draw(const glm::mat4 &view, const glm::mat4 proj) {
    mat4 modelview = view * modelTransform;

    glUseProgram(shader); // load shader and variables
    glUniformMatrix4fv(glGetUniformLocation(shader, "uProjectionMatrix"), 1, false, value_ptr(proj));
    glUniformMatrix4fv(glGetUniformLocation(shader, "uModelViewMatrix"), 1, false, value_ptr(modelview));
    glUniform3fv(glGetUniformLocation(shader, "uColor"), 1, value_ptr(color));


    // I created these-------------------------------------------
    glUniform1f(glGetUniformLocation(shader, "u_ambient_coeff"), ambient_coeff);
    glUniform1f(glGetUniformLocation(shader, "u_diffuse_coeff"), diffuse_coeff);
    glUniform1f(glGetUniformLocation(shader, "u_specular_coeff"), specular_coeff);
    glUniform1f(glGetUniformLocation(shader, "u_power_specular_coeff"), power_specular_coeff);
    glUniform1f(glGetUniformLocation(shader, "u_s_d_ratio"), s_d_ratio);
    glUniform1f(glGetUniformLocation(shader, "u_refraction"), refraction);
    glUniform1f(glGetUniformLocation(shader, "u_roughness"), roughness);
    glUniform3fv(glGetUniformLocation(shader, "u_light_position"), 1, value_ptr(light_position));
    //-------------------------------------------------------------

    mesh.draw(); // draw    // this gl_mesh method is defined in cgra_mesh.cpp
}

/*
// Assuming you've compiled and linked your shader program
GLuint shaderProgram; // Your shader program ID
float ambient_coeff = 0.5f; // Your ambient coefficient value

// Get the location of the uniform
GLint ambientCoeffLocation = glGetUniformLocation(shaderProgram, "ambient_coeff");

// Check if the uniform exists in the shader
if (ambientCoeffLocation != -1) {
    // Bind the float value to the uniform
    glUniform1f(ambientCoeffLocation, ambient_coeff);
} else {
    // Handle the case when the uniform doesn't exist
    // Print an error message or take appropriate action
}

*/





Application::Application(GLFWwindow *window) : m_window(window) {   // this is where can read in data from files

    // color_vert_2.glsl
    // color_frag_2.glsl -> Blinn-Phong Shader with updated shininess exponent to respond to roughness
    // ImGui input -> Ambient Light, Diffuse Light, Specular Light, Roughness

    // color_vert_3.glsl
    // color_frag_3.glsl -> Cook-Torrance Shader
    // ImGui input -> Ambient Light, Diffuse Light, Specular:Diffuse Ratio, Refraction, Roughness

    // color_vert_4.glsl
    // color_frag_4.glsl -> Oren_Nayar Shader; Oren_Nayar implemented into the Blinn-Phong Shader (color_frag_2) as the diffuse and roughness controlled by roughness
    // ImGui input -> Ambient Light, Diffuse Light, Roughness

    shader_builder sb;
    sb.set_shader(GL_VERTEX_SHADER, CGRA_SRCDIR + std::string("//res//shaders//color_vert_4.glsl"));  // original vertex shader -> color_vert.glsl
    sb.set_shader(GL_FRAGMENT_SHADER, CGRA_SRCDIR + std::string("//res//shaders//color_frag_4.glsl")); // original fragment shader -> color_frag.glsl
    GLuint shader = sb.build();
```



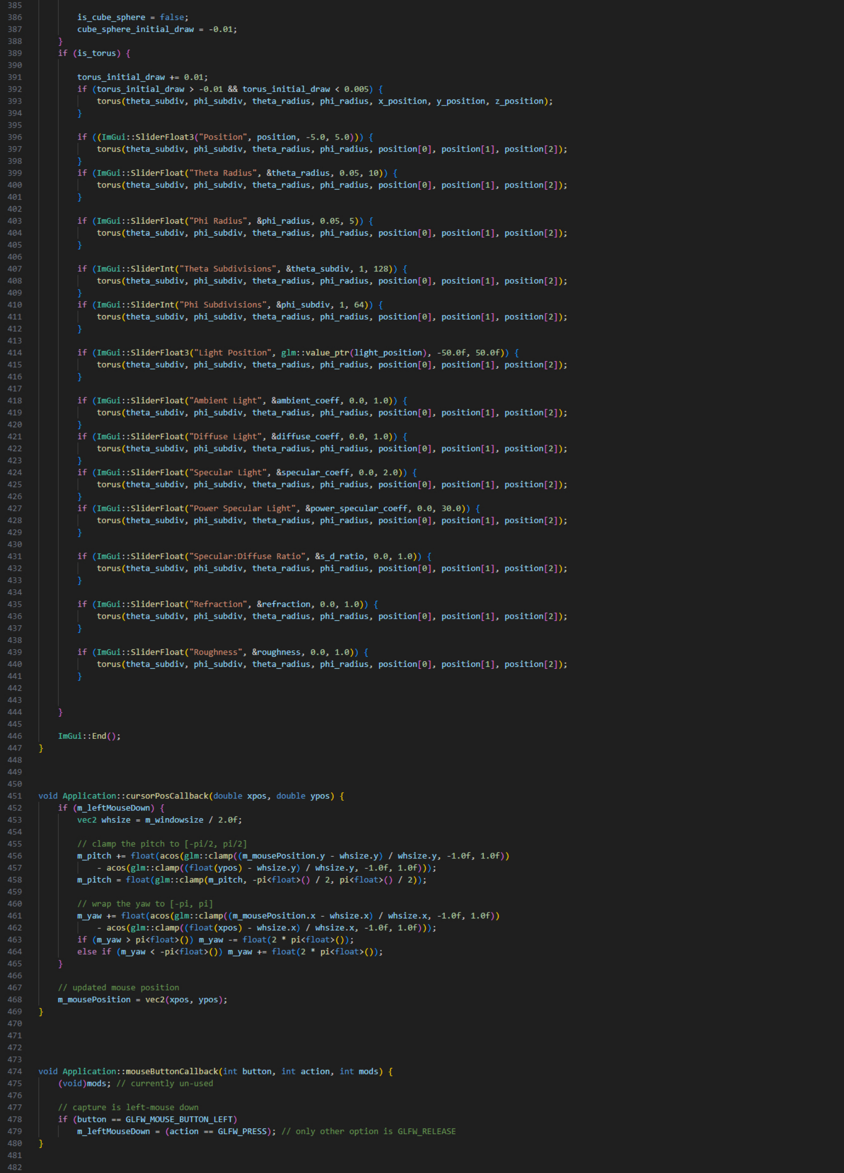
```cpp


    m_model.shader = shader;
    m_model.mesh = load_wavefront_data(CGRA_SRCDIR + std::string("/res//assets//teapot.obj")).build();  // reads in teapot data
    m_model.color = vec3(1, 0, 0); // red

}


void Application::render() {

    // retrieve the window hieght
    int width, height;
    glfwGetFramebufferSize(m_window, &width, &height);

    m_windowsize = vec2(width, height); // update window size
    glViewport(0, 0, width, height); // set the viewport to draw to the entire window

    // clear the back-buffer
    glClearColor(0.3f, 0.3f, 0.4f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // enable flags for normal/forward rendering
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

    // projection matrix
    mat4 proj = perspective(1.f, float(width) / height, 0.1f, 1000.f);

    // view matrix
    mat4 view = translate(mat4(1), vec3(0, 0, -m_distance))
        * rotate(mat4(1), m_pitch, vec3(1, 0, 0))
        * rotate(mat4(1), m_yaw,   vec3(0, 1, 0));


    // helpful draw options
    if (m_show_grid) drawGrid(view, proj);
    if (m_show_axis) drawAxis(view, proj);
    glPolygonMode(GL_FRONT_AND_BACK, (m_showWireframe) ? GL_LINE : GL_FILL);

    // I created these -------------------------------------
    m_model.ambient_coeff = ambient_coeff;
    m_model.diffuse_coeff = diffuse_coeff;
    m_model.specular_coeff = specular_coeff;
    m_model.power_specular_coeff = power_specular_coeff;
    m_model.light_position = light_position;
    m_model.s_d_ratio = s_d_ratio;
    m_model.refraction = refraction;
    m_model.roughness = roughness;
    // -----------------------------------------------------



    // draw the model
    m_model.draw(view, proj);

}


void Application::renderGUI() {

    // setup window
    ImGui::SetNextWindowPos(ImVec2(5, 5), ImGuiSetCond_Once);
    ImGui::SetNextWindowSize(ImVec2(300, 550), ImGuiSetCond_Once);
    ImGui::Begin("Options", 0);

    // display current camera parameters
    ImGui::Text("Application %.3f ms/frame (%.1f FPS)", 1000.0f / ImGui::GetIO().Framerate, ImGui::GetIO().Framerate);
    ImGui::SliderFloat("Pitch", &m_pitch, -pi<float>() / 2, pi<float>() / 2, "%.2f");
    ImGui::SliderFloat("Yaw", &m_yaw, -pi<float>(), pi<float>(), "%.2f");
    ImGui::SliderFloat("Distance", &m_distance, 0, 100, "%.2f", 2.0f);

    // helpful drawing options
    ImGui::Checkbox("Show axis", &m_show_axis);
    ImGui::SameLine();
    ImGui::Checkbox("Show grid", &m_show_grid);
    ImGui::Checkbox("Wireframe", &m_showWireframe);
    ImGui::SameLine();
    if (ImGui::Button("Screenshot")) rgba_image::screenshot(true);


    ImGui::Separator();

    /*
    // example of how to use input boxes
    static float exampleInput;
    if (ImGui::InputFloat("example input", &exampleInput)) {
        cout << "example input changed to " << exampleInput << endl;
    }
    */


    // I created these----------------------------------------------

```



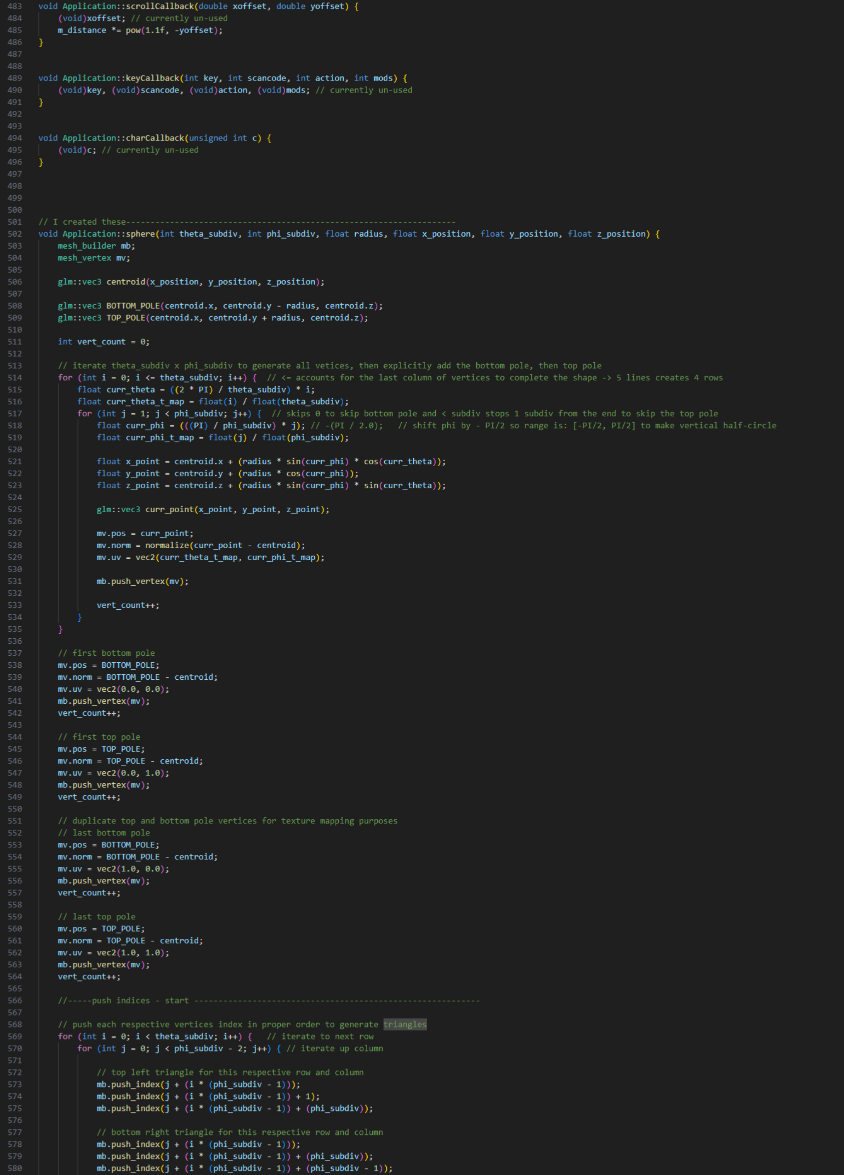
```
    // sphere
    if (ImGui::Button("Sphere Lat_Long")) {
        is_sphere = true;
        sphere_initial_draw = -0.01;

        is_cube = false;
        cube_initial_draw = -0.01;

        is_cube_sphere = false;
        cube_sphere_initial_draw = -0.01;

        is_torus = false;
        torus_initial_draw = -0.01;
    }
    if (is_sphere) {

        sphere_initial_draw += 0.01;
        if (sphere_initial_draw > -0.01 && sphere_initial_draw < 0.005) {
            sphere(theta_subdiv, phi_subdiv, radius, x_position, y_position, z_position);
        }

        if ((ImGui::SliderFloat3("Position", position, -5.0, 5.0))) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderInt("Theta Subdivisions", &theta_subdiv, 1, 64)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderInt("Phi Subdivisions", &phi_subdiv, 1, 64)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderFloat("Radius", &radius, 0.1, 10.0)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat3("Light Position", glm::value_ptr(light_position), -50.0f, 50.0f)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat("Ambient Light", &ambient_coeff, 0.0, 1.0)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderFloat("Diffuse Light", &diffuse_coeff, 0.0, 1.0)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderFloat("Specular Light", &specular_coeff, 0.0, 2.0)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderFloat("Power Specular Light", &power_specular_coeff, 0.0, 30.0)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat("Specular:Diffuse Ratio", &s_d_ratio, 0.0, 1.0)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat("Refraction", &refraction, 0.0, 1.0)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat("Roughness", &roughness, 0.0, 1.0)) {
            sphere(theta_subdiv, phi_subdiv, radius, position[0], position[1], position[2]);
        }
    }


    // cube
    if (ImGui::Button("Cube")) {
        is_cube = true;
        cube_initial_draw = -0.01;

        is_sphere = false;
        sphere_initial_draw = -0.01;

        is_cube_sphere = false;
        cube_sphere_initial_draw = -0.01;

        is_torus = false;
        torus_initial_draw = -0.01;
    }
    if (is_cube) {

        cube_initial_draw += 0.01;
        if (cube_initial_draw > -0.01 && cube_initial_draw < 0.005) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }
        if (ImGui::SliderInt("subdivisions", &subdiv, 1, 64)) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }

        if (ImGui::SliderFloat3("Light Position", glm::value_ptr(light_position), -50.0f, 50.0f)) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }

```



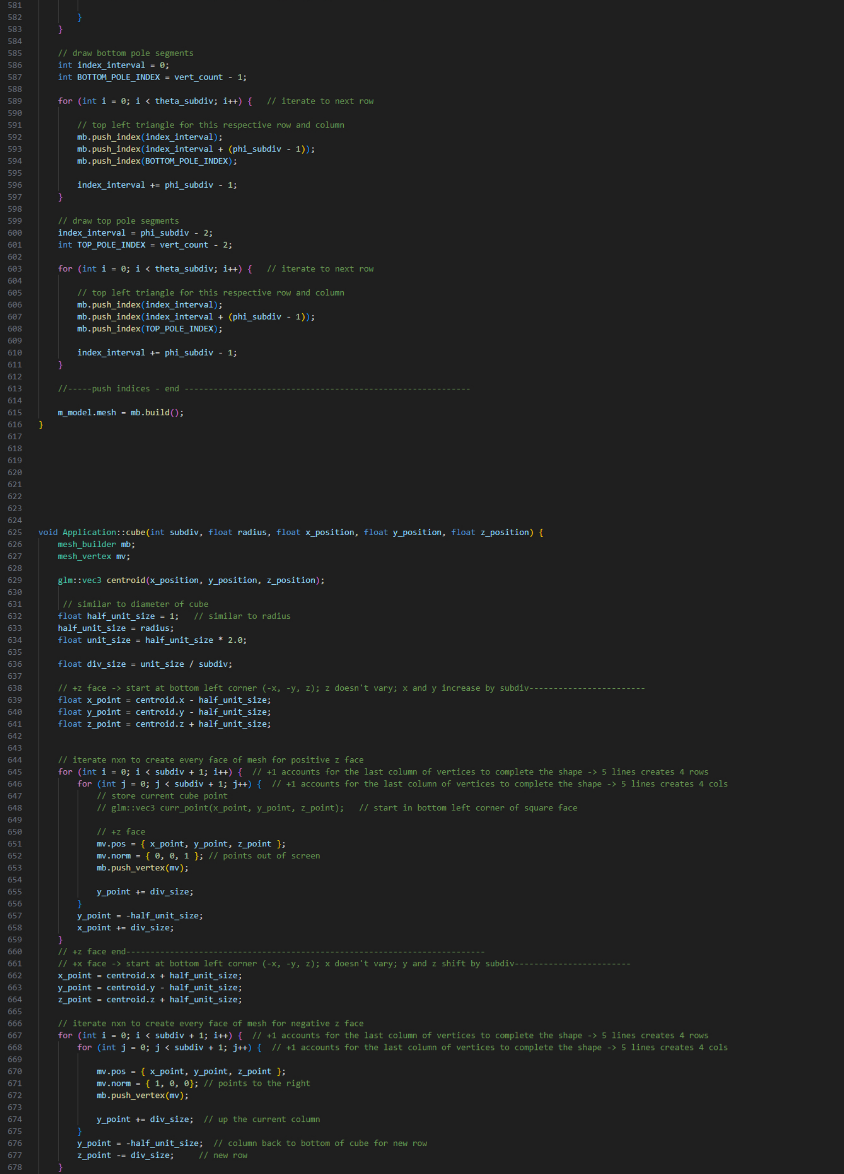
```
        if (ImGui::SliderFloat("Ambient Light", &ambient_coeff, 0.0, 1.0)) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }
        if (ImGui::SliderFloat("Diffuse Light", &diffuse_coeff, 0.0, 1.0)) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }
        if (ImGui::SliderFloat("Specular Light", &specular_coeff, 0.0, 2.0)) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }
        if (ImGui::SliderFloat("Power Specular Light", &power_specular_coeff, 0.0, 30.0)) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }

        if (ImGui::SliderFloat("Specular:Diffuse Ratio", &s_d_ratio, 0.0, 1.0)) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }

        if (ImGui::SliderFloat("Refraction", &refraction, 0.0, 1.0)) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }

        if (ImGui::SliderFloat("Roughness", &roughness, 0.0, 1.0)) {
            cube(subdiv, radius, 0.0, 0.0, 0.0);
        }
    }







    // cube sphere
    if (ImGui::Button("Sphere from Cube")) {
        is_cube_sphere = true;
        cube_sphere_initial_draw = -0.01;

        is_sphere = false;
        sphere_initial_draw = -0.01;

        is_cube = false;
        cube_initial_draw = -0.01;

        is_torus = false;
        torus_initial_draw = -0.01;

    }
    if (is_cube_sphere) {

        cube_sphere_initial_draw += 0.01;
        if (cube_sphere_initial_draw > -0.01 && cube_sphere_initial_draw < 0.005) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }
        if (ImGui::SliderInt("subdivisions", &subdiv, 1, 64)) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }

        if (ImGui::SliderFloat3("Light Position", glm::value_ptr(light_position), -50.0f, 50.0f)) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }

        if (ImGui::SliderFloat("Ambient Light", &ambient_coeff, 0.0, 1.0)) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }
        if (ImGui::SliderFloat("Diffuse Light", &diffuse_coeff, 0.0, 1.0)) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }
        if (ImGui::SliderFloat("Specular Light", &specular_coeff, 0.0, 2.0)) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }
        if (ImGui::SliderFloat("Power Specular Light", &power_specular_coeff, 0.0, 30.0)) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }

        if (ImGui::SliderFloat("Specular:Diffuse Ratio", &s_d_ratio, 0.0, 1.0)) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }

        if (ImGui::SliderFloat("Refraction", &refraction, 0.0, 1.0)) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }

        if (ImGui::SliderFloat("Roughness", &roughness, 0.0, 1.0)) {
            cube_sphere(subdiv, radius, 0.0, 0.0, 0.0);
        }
    }


    // torus
    if (ImGui::Button("Torus")) {
        is_torus = true;
        torus_initial_draw = -0.01;

        is_sphere = false;
        sphere_initial_draw = -0.01;

        is_cube = false;
        cube_initial_draw = -0.01;
```



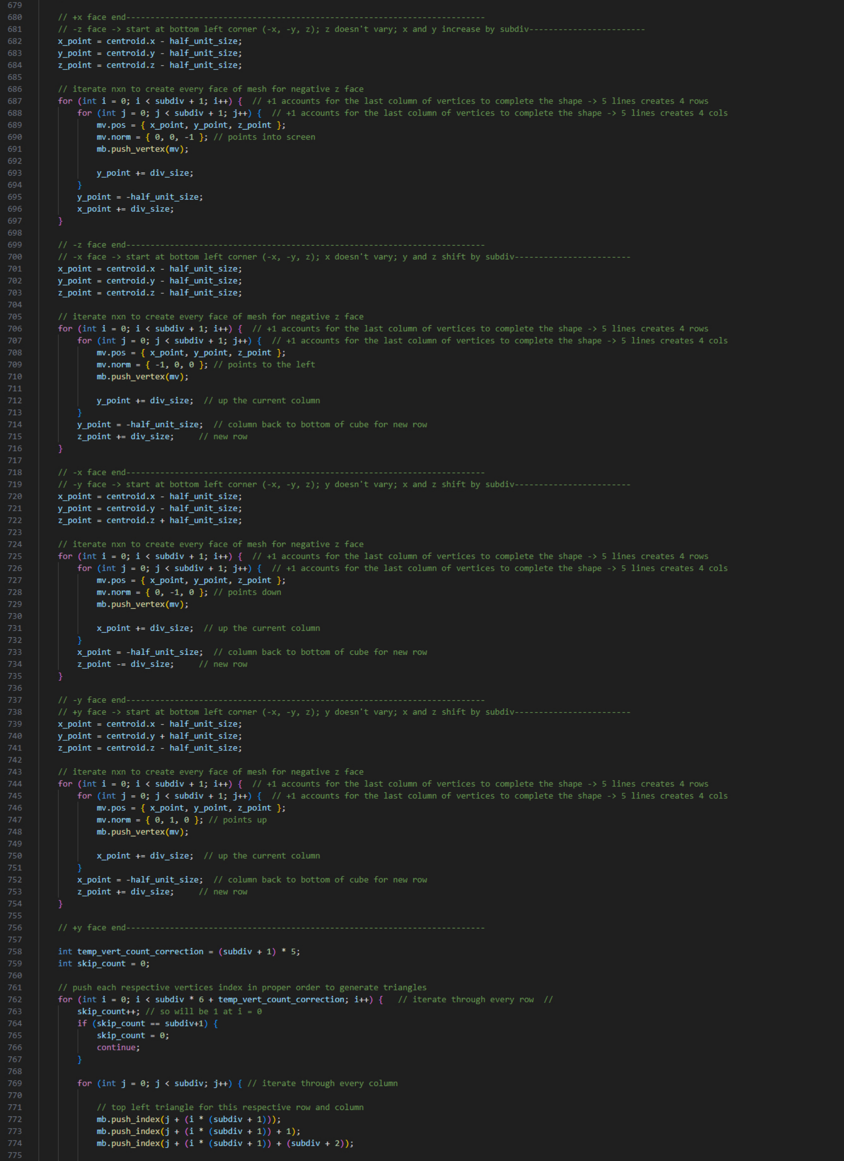
```

        is_cube_sphere = false;
        cube_sphere_initial_draw = -0.01;
    }
    if (is_torus) {

        torus_initial_draw += 0.01;
        if (torus_initial_draw > -0.01 && torus_initial_draw < 0.005) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, x_position, y_position, z_position);
        }

        if ((ImGui::SliderFloat3("Position", position, -5.0, 5.0))) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderFloat("Theta Radius", &theta_radius, 0.05, 10)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat("Phi Radius", &phi_radius, 0.05, 5)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderInt("Theta Subdivisions", &theta_subdiv, 1, 128)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderInt("Phi Subdivisions", &phi_subdiv, 1, 64)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat3("Light Position", glm::value_ptr(light_position), -50.0f, 50.0f)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat("Ambient Light", &ambient_coeff, 0.0, 1.0)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderFloat("Diffuse Light", &diffuse_coeff, 0.0, 1.0)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderFloat("Specular Light", &specular_coeff, 0.0, 2.0)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }
        if (ImGui::SliderFloat("Power Specular Light", &power_specular_coeff, 0.0, 30.0)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat("Specular:Diffuse Ratio", &s_d_ratio, 0.0, 1.0)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat("Refraction", &refraction, 0.0, 1.0)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }

        if (ImGui::SliderFloat("Roughness", &roughness, 0.0, 1.0)) {
            torus(theta_subdiv, phi_subdiv, theta_radius, phi_radius, position[0], position[1], position[2]);
        }


    }

    ImGui::End();
}



void Application::cursorPosCallback(double xpos, double ypos) {
    if (m_leftMouseDown) {
        vec2 whsize = m_windowsize / 2.0f;

        // clamp the pitch to [-pi/2, pi/2]
        m_pitch += float(acos(glm::clamp((m_mousePosition.y - whsize.y) / whsize.y, -1.0f, 1.0f))
            - acos(glm::clamp((float(ypos) - whsize.y) / whsize.y, -1.0f, 1.0f)));
        m_pitch = float(glm::clamp(m_pitch, -pi<float>() / 2, pi<float>() / 2));

        // wrap the yaw to [-pi, pi]
        m_yaw += float(acos(glm::clamp((m_mousePosition.x - whsize.x) / whsize.x, -1.0f, 1.0f))
            - acos(glm::clamp((float(xpos) - whsize.x) / whsize.x, -1.0f, 1.0f)));
        if (m_yaw > pi<float>()) m_yaw -= float(2 * pi<float>());
        else if (m_yaw < -pi<float>()) m_yaw += float(2 * pi<float>());
    }

    // updated mouse position
    m_mousePosition = vec2(xpos, ypos);
}




void Application::mouseButtonCallback(int button, int action, int mods) {
    (void)mods; // currently un-used

    // capture is left-mouse down
    if (button == GLFW_MOUSE_BUTTON_LEFT)
        m_leftMouseDown = (action == GLFW_PRESS); // only other option is GLFW_RELEASE
}

```



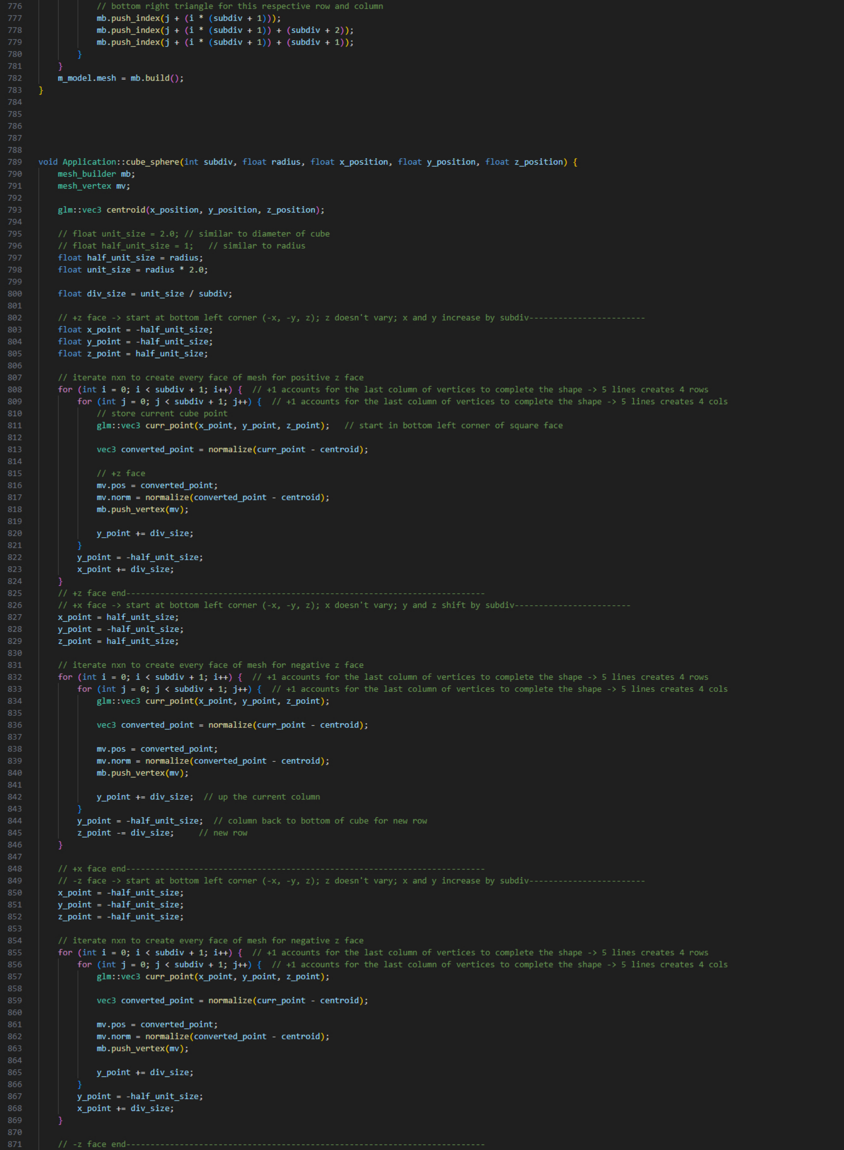
```cpp
void Application::scrollCallback(double xoffset, double yoffset) {
    (void)xoffset; // currently un-used
    m_distance *= pow(1.1f, -yoffset);
}


void Application::keyCallback(int key, int scancode, int action, int mods) {
    (void)key, (void)scancode, (void)action, (void)mods; // currently un-used
}


void Application::charCallback(unsigned int c) {
    (void)c; // currently un-used
}




// I created these--------------------------------------------------------------------
void Application::sphere(int theta_subdiv, int phi_subdiv, float radius, float x_position, float y_position, float z_position) {
    mesh_builder mb;
    mesh_vertex mv;

    glm::vec3 centroid(x_position, y_position, z_position);

    glm::vec3 BOTTOM_POLE(centroid.x, centroid.y - radius, centroid.z);
    glm::vec3 TOP_POLE(centroid.x, centroid.y + radius, centroid.z);

    int vert_count = 0;

    // iterate theta_subdiv x phi_subdiv to generate all vetices, then explicitly add the bottom pole, then top pole
    for (int i = 0; i <= theta_subdiv; i++) {  // <= accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        float curr_theta = ((2 * PI) / theta_subdiv) * i;
        float curr_theta_t_map = float(i) / float(theta_subdiv);
        for (int j = 1; j < phi_subdiv; j++) {  // skips 0 to skip bottom pole and < subdiv stops 1 subdiv from the end to skip the top pole
            float curr_phi = (((PI) / phi_subdiv) * j); // -(PI / 2.0);    // shift phi by - PI/2 so range is: [-PI/2, PI/2] to make vertical half-circle
            float curr_phi_t_map = float(j) / float(phi_subdiv);

            float x_point = centroid.x + (radius * sin(curr_phi) * cos(curr_theta));
            float y_point = centroid.y + (radius * cos(curr_phi));
            float z_point = centroid.z + (radius * sin(curr_phi) * sin(curr_theta));

            glm::vec3 curr_point(x_point, y_point, z_point);

            mv.pos = curr_point;
            mv.norm = normalize(curr_point - centroid);
            mv.uv = vec2(curr_theta_t_map, curr_phi_t_map);

            mb.push_vertex(mv);

            vert_count++;
        }
    }

    // first bottom pole
    mv.pos = BOTTOM_POLE;
    mv.norm = BOTTOM_POLE - centroid;
    mv.uv = vec2(0.0, 0.0);
    mb.push_vertex(mv);
    vert_count++;

    // first top pole
    mv.pos = TOP_POLE;
    mv.norm = TOP_POLE - centroid;
    mv.uv = vec2(0.0, 1.0);
    mb.push_vertex(mv);
    vert_count++;

    // duplicate top and bottom pole vertices for texture mapping purposes
    // last bottom pole
    mv.pos = BOTTOM_POLE;
    mv.norm = BOTTOM_POLE - centroid;
    mv.uv = vec2(1.0, 0.0);
    mb.push_vertex(mv);
    vert_count++;

    // last top pole
    mv.pos = TOP_POLE;
    mv.norm = TOP_POLE - centroid;
    mv.uv = vec2(1.0, 1.0);
    mb.push_vertex(mv);
    vert_count++;

    //-----push indices - start -----------------------------------------------------------

    // push each respective vertices index in proper order to generate triangles
    for (int i = 0; i < theta_subdiv; i++) {   // iterate to next row
        for (int j = 0; j < phi_subdiv - 2; j++) { // iterate up column

            // top left triangle for this respective row and column
            mb.push_index(j + (i * (phi_subdiv - 1)));
            mb.push_index(j + (i * (phi_subdiv - 1)) + 1);
            mb.push_index(j + (i * (phi_subdiv - 1)) + (phi_subdiv));

            // bottom right triangle for this respective row and column
            mb.push_index(j + (i * (phi_subdiv - 1)));
            mb.push_index(j + (i * (phi_subdiv - 1)) + (phi_subdiv));
            mb.push_index(j + (i * (phi_subdiv - 1)) + (phi_subdiv - 1));
```



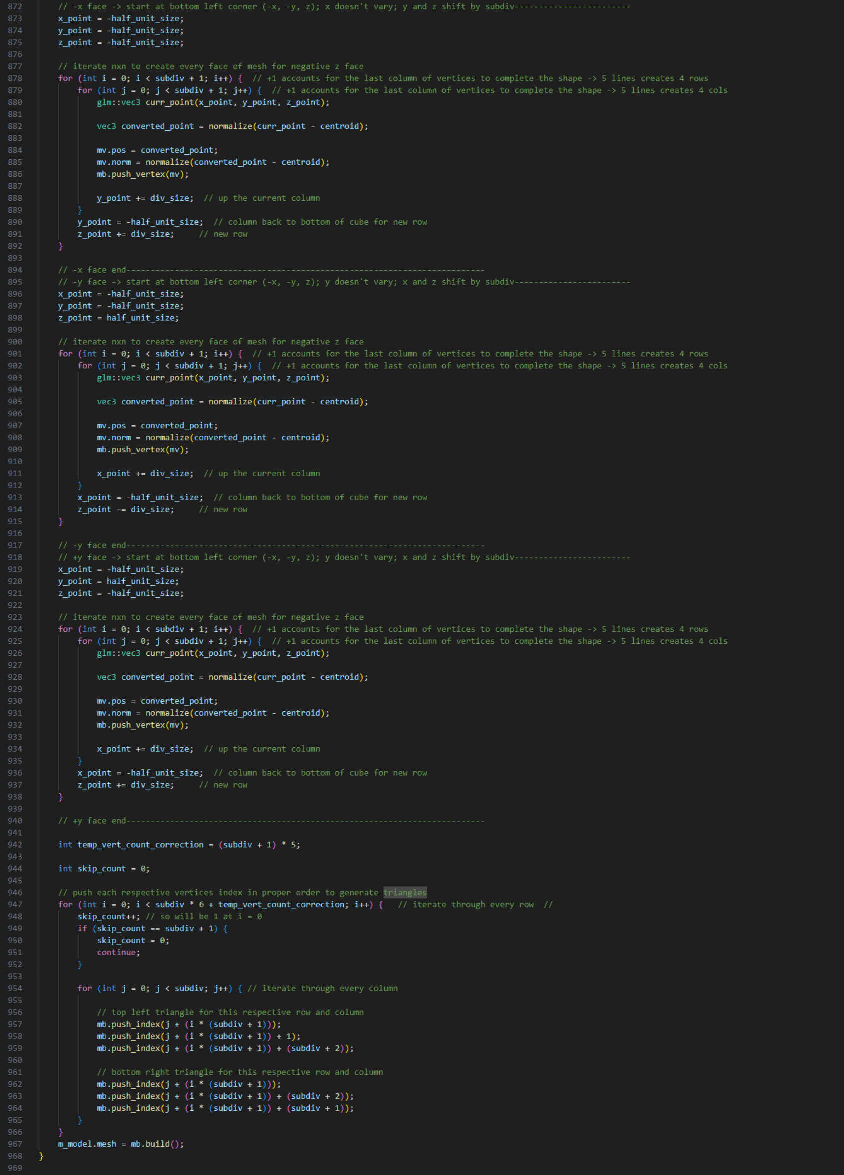
```cpp

        }
    }

    // draw bottom pole segments
    int index_interval = 0;
    int BOTTOM_POLE_INDEX = vert_count - 1;

    for (int i = 0; i < theta_subdiv; i++) {   // iterate to next row

        // top left triangle for this respective row and column
        mb.push_index(index_interval);
        mb.push_index(index_interval + (phi_subdiv - 1));
        mb.push_index(BOTTOM_POLE_INDEX);

        index_interval += phi_subdiv - 1;
    }

    // draw top pole segments
    index_interval = phi_subdiv - 2;
    int TOP_POLE_INDEX = vert_count - 2;

    for (int i = 0; i < theta_subdiv; i++) {   // iterate to next row

        // top left triangle for this respective row and column
        mb.push_index(index_interval);
        mb.push_index(index_interval + (phi_subdiv - 1));
        mb.push_index(TOP_POLE_INDEX);

        index_interval += phi_subdiv - 1;
    }

    //-----push indices - end ----------------------------------------------------------

    m_model.mesh = mb.build();
}









void Application::cube(int subdiv, float radius, float x_position, float y_position, float z_position) {
    mesh_builder mb;
    mesh_vertex mv;

    glm::vec3 centroid(x_position, y_position, z_position);

     // similar to diameter of cube
    float half_unit_size = 1;   // similar to radius
    half_unit_size = radius;
    float unit_size = half_unit_size * 2.0;

    float div_size = unit_size / subdiv;

    // +z face -> start at bottom left corner (-x, -y, z); z doesn't vary; x and y increase by subdiv-----------------------
    float x_point = centroid.x - half_unit_size;
    float y_point = centroid.y - half_unit_size;
    float z_point = centroid.z + half_unit_size;


    // iterate nxn to create every face of mesh for positive z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            // store current cube point
            // glm::vec3 curr_point(x_point, y_point, z_point);   // start in bottom left corner of square face

            // +z face
            mv.pos = { x_point, y_point, z_point };
            mv.norm = { 0, 0, 1 }; // points out of screen
            mb.push_vertex(mv);

            y_point += div_size;
        }
        y_point = -half_unit_size;
        x_point += div_size;
    }
    // +z face end--------------------------------------------------------------------------
    // +x face -> start at bottom left corner (-x, -y, z); x doesn't vary; y and z shift by subdiv-----------------------
    x_point = centroid.x + half_unit_size;
    y_point = centroid.y - half_unit_size;
    z_point = centroid.z + half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols

            mv.pos = { x_point, y_point, z_point };
            mv.norm = { 1, 0, 0}; // points to the right
            mb.push_vertex(mv);

            y_point += div_size;  // up the current column
        }
        y_point = -half_unit_size;  // column back to bottom of cube for new row
        z_point -= div_size;     // new row
    }
```



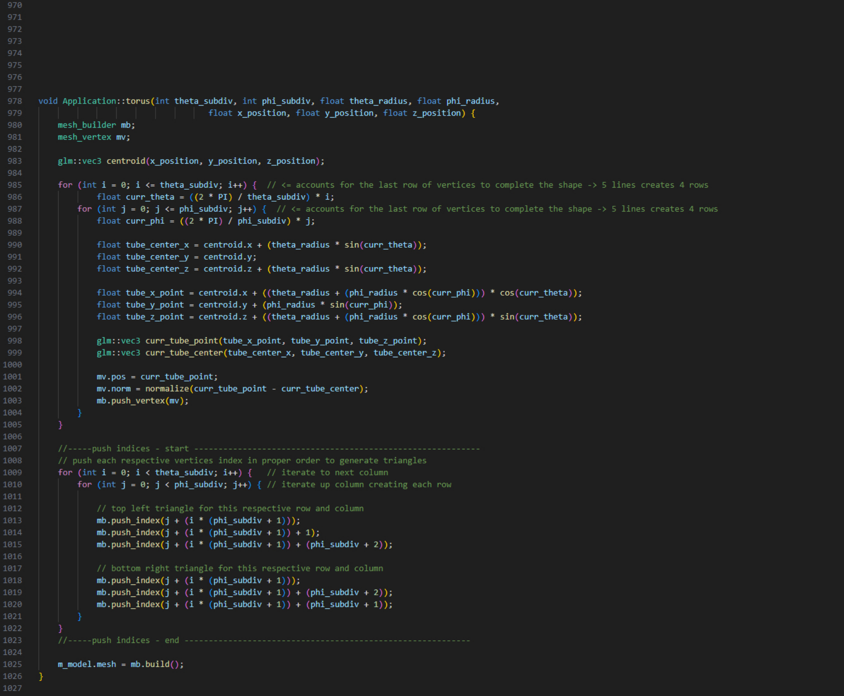
```

    // +x face end--------------------------------------------------------------------------
    // -z face -> start at bottom left corner (-x, -y, z); z doesn't vary; x and y increase by subdiv-----------------------
    x_point = centroid.x - half_unit_size;
    y_point = centroid.y - half_unit_size;
    z_point = centroid.z - half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            mv.pos = { x_point, y_point, z_point };
            mv.norm = { 0, 0, -1 }; // points into screen
            mb.push_vertex(mv);

            y_point += div_size;
        }
        y_point = -half_unit_size;
        x_point += div_size;
    }

    // -z face end--------------------------------------------------------------------------
    // -x face -> start at bottom left corner (-x, -y, z); x doesn't vary; y and z shift by subdiv-----------------------
    x_point = centroid.x - half_unit_size;
    y_point = centroid.y - half_unit_size;
    z_point = centroid.z - half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            mv.pos = { x_point, y_point, z_point };
            mv.norm = { -1, 0, 0 }; // points to the left
            mb.push_vertex(mv);

            y_point += div_size;  // up the current column
        }
        y_point = -half_unit_size;  // column back to bottom of cube for new row
        z_point += div_size;     // new row
    }

    // -x face end--------------------------------------------------------------------------
    // -y face -> start at bottom left corner (-x, -y, z); y doesn't vary; x and z shift by subdiv-----------------------
    x_point = centroid.x - half_unit_size;
    y_point = centroid.y - half_unit_size;
    z_point = centroid.z + half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            mv.pos = { x_point, y_point, z_point };
            mv.norm = { 0, -1, 0 }; // points down
            mb.push_vertex(mv);

            x_point += div_size;  // up the current column
        }
        x_point = -half_unit_size;  // column back to bottom of cube for new row
        z_point -= div_size;     // new row
    }

    // -y face end--------------------------------------------------------------------------
    // +y face -> start at bottom left corner (-x, -y, z); y doesn't vary; x and z shift by subdiv-----------------------
    x_point = centroid.x - half_unit_size;
    y_point = centroid.y + half_unit_size;
    z_point = centroid.z - half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            mv.pos = { x_point, y_point, z_point };
            mv.norm = { 0, 1, 0 }; // points up
            mb.push_vertex(mv);

            x_point += div_size;  // up the current column
        }
        x_point = -half_unit_size;  // column back to bottom of cube for new row
        z_point += div_size;     // new row
    }

    // +y face end--------------------------------------------------------------------------

    int temp_vert_count_correction = (subdiv + 1) * 5;
    int skip_count = 0;

    // push each respective vertices index in proper order to generate triangles
    for (int i = 0; i < subdiv * 6 + temp_vert_count_correction; i++) {   // iterate through every row  //
        skip_count++; // so will be 1 at i = 0
        if (skip_count == subdiv+1) {
            skip_count = 0;
            continue;
        }

        for (int j = 0; j < subdiv; j++) { // iterate through every column

            // top left triangle for this respective row and column
            mb.push_index(j + (i * (subdiv + 1)));
            mb.push_index(j + (i * (subdiv + 1)) + 1);
            mb.push_index(j + (i * (subdiv + 1)) + (subdiv + 2));

```



```
            // bottom right triangle for this respective row and column
            mb.push_index(j + (i * (subdiv + 1)));
            mb.push_index(j + (i * (subdiv + 1)) + (subdiv + 2));
            mb.push_index(j + (i * (subdiv + 1)) + (subdiv + 1));
        }
    }
    m_model.mesh = mb.build();
}





void Application::cube_sphere(int subdiv, float radius, float x_position, float y_position, float z_position) {
    mesh_builder mb;
    mesh_vertex mv;

    glm::vec3 centroid(x_position, y_position, z_position);

    // float unit_size = 2.0; // similar to diameter of cube
    // float half_unit_size = 1;   // similar to radius
    float half_unit_size = radius;
    float unit_size = radius * 2.0;

    float div_size = unit_size / subdiv;

    // +z face -> start at bottom left corner (-x, -y, z); z doesn't vary; x and y increase by subdiv------------------------
    float x_point = -half_unit_size;
    float y_point = -half_unit_size;
    float z_point = half_unit_size;

    // iterate nxn to create every face of mesh for positive z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            // store current cube point
            glm::vec3 curr_point(x_point, y_point, z_point);   // start in bottom left corner of square face

            vec3 converted_point = normalize(curr_point - centroid);

            // +z face
            mv.pos = converted_point;
            mv.norm = normalize(converted_point - centroid);
            mb.push_vertex(mv);

            y_point += div_size;
        }
        y_point = -half_unit_size;
        x_point += div_size;
    }
    // +z face end---------------------------------------------------------------------------
    // +x face -> start at bottom left corner (-x, -y, z); x doesn't vary; y and z shift by subdiv------------------------
    x_point = half_unit_size;
    y_point = -half_unit_size;
    z_point = half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            glm::vec3 curr_point(x_point, y_point, z_point);

            vec3 converted_point = normalize(curr_point - centroid);

            mv.pos = converted_point;
            mv.norm = normalize(converted_point - centroid);
            mb.push_vertex(mv);

            y_point += div_size;  // up the current column
        }
        y_point = -half_unit_size;  // column back to bottom of cube for new row
        z_point -= div_size;     // new row
    }

    // +x face end---------------------------------------------------------------------------
    // -z face -> start at bottom left corner (-x, -y, z); z doesn't vary; x and y increase by subdiv------------------------
    x_point = -half_unit_size;
    y_point = -half_unit_size;
    z_point = -half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            glm::vec3 curr_point(x_point, y_point, z_point);

            vec3 converted_point = normalize(curr_point - centroid);

            mv.pos = converted_point;
            mv.norm = normalize(converted_point - centroid);
            mb.push_vertex(mv);

            y_point += div_size;
        }
        y_point = -half_unit_size;
        x_point += div_size;
    }

    // -z face end---------------------------------------------------------------------------
```



```
    // -x face -> start at bottom left corner (-x, -y, z); x doesn't vary; y and z shift by subdiv-----------------------
    x_point = -half_unit_size;
    y_point = -half_unit_size;
    z_point = -half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            glm::vec3 curr_point(x_point, y_point, z_point);

            vec3 converted_point = normalize(curr_point - centroid);

            mv.pos = converted_point;
            mv.norm = normalize(converted_point - centroid);
            mb.push_vertex(mv);

            y_point += div_size;  // up the current column
        }
        y_point = -half_unit_size;  // column back to bottom of cube for new row
        z_point += div_size;     // new row
    }

    // -x face end-------------------------------------------------------------------------
    // -y face -> start at bottom left corner (-x, -y, z); y doesn't vary; x and z shift by subdiv-----------------------
    x_point = -half_unit_size;
    y_point = -half_unit_size;
    z_point = half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            glm::vec3 curr_point(x_point, y_point, z_point);

            vec3 converted_point = normalize(curr_point - centroid);

            mv.pos = converted_point;
            mv.norm = normalize(converted_point - centroid);
            mb.push_vertex(mv);

            x_point += div_size;  // up the current column
        }
        x_point = -half_unit_size;  // column back to bottom of cube for new row
        z_point -= div_size;     // new row
    }

    // -y face end-------------------------------------------------------------------------
    // +y face -> start at bottom left corner (-x, -y, z); y doesn't vary; x and z shift by subdiv-----------------------
    x_point = -half_unit_size;
    y_point = half_unit_size;
    z_point = -half_unit_size;

    // iterate nxn to create every face of mesh for negative z face
    for (int i = 0; i < subdiv + 1; i++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 rows
        for (int j = 0; j < subdiv + 1; j++) {  // +1 accounts for the last column of vertices to complete the shape -> 5 lines creates 4 cols
            glm::vec3 curr_point(x_point, y_point, z_point);

            vec3 converted_point = normalize(curr_point - centroid);

            mv.pos = converted_point;
            mv.norm = normalize(converted_point - centroid);
            mb.push_vertex(mv);

            x_point += div_size;  // up the current column
        }
        x_point = -half_unit_size;  // column back to bottom of cube for new row
        z_point += div_size;     // new row
    }

    // +y face end-------------------------------------------------------------------------

    int temp_vert_count_correction = (subdiv + 1) * 5;

    int skip_count = 0;

    // push each respective vertices index in proper order to generate triangles
    for (int i = 0; i < subdiv * 6 + temp_vert_count_correction; i++) {   // iterate through every row  //
        skip_count++; // so will be 1 at i = 0
        if (skip_count == subdiv + 1) {
            skip_count = 0;
            continue;
        }

        for (int j = 0; j < subdiv; j++) { // iterate through every column

            // top left triangle for this respective row and column
            mb.push_index(j + (i * (subdiv + 1)));
            mb.push_index(j + (i * (subdiv + 1)) + 1);
            mb.push_index(j + (i * (subdiv + 1)) + (subdiv + 2));

            // bottom right triangle for this respective row and column
            mb.push_index(j + (i * (subdiv + 1)));
            mb.push_index(j + (i * (subdiv + 1)) + (subdiv + 2));
            mb.push_index(j + (i * (subdiv + 1)) + (subdiv + 1));
        }
    }
    m_model.mesh = mb.build();
}
```



```cpp
void Application::torus(int theta_subdiv, int phi_subdiv, float theta_radius, float phi_radius,
                                   float x_position, float y_position, float z_position) {
    mesh_builder mb;
    mesh_vertex mv;

    glm::vec3 centroid(x_position, y_position, z_position);

    for (int i = 0; i <= theta_subdiv; i++) {  // <= accounts for the last row of vertices to complete the shape -> 5 lines creates 4 rows
        float curr_theta = ((2 * PI) / theta_subdiv) * i;
    for (int j = 0; j <= phi_subdiv; j++) {  // <= accounts for the last row of vertices to complete the shape -> 5 lines creates 4 rows
        float curr_phi = ((2 * PI) / phi_subdiv) * j;

        float tube_center_x = centroid.x + (theta_radius * sin(curr_theta));
        float tube_center_y = centroid.y;
        float tube_center_z = centroid.z + (theta_radius * sin(curr_theta));

        float tube_x_point = centroid.x + ((theta_radius + (phi_radius * cos(curr_phi))) * cos(curr_theta));
        float tube_y_point = centroid.y + (phi_radius * sin(curr_phi));
        float tube_z_point = centroid.z + ((theta_radius + (phi_radius * cos(curr_phi))) * sin(curr_theta));

        glm::vec3 curr_tube_point(tube_x_point, tube_y_point, tube_z_point);
        glm::vec3 curr_tube_center(tube_center_x, tube_center_y, tube_center_z);

        mv.pos = curr_tube_point;
        mv.norm = normalize(curr_tube_point - curr_tube_center);
        mb.push_vertex(mv);
        }
    }

    //-----push indices - start -----------------------------------------------------------
    // push each respective vertices index in proper order to generate triangles
    for (int i = 0; i < theta_subdiv; i++) {   // iterate to next column
        for (int j = 0; j < phi_subdiv; j++) { // iterate up column creating each row

            // top left triangle for this respective row and column
            mb.push_index(j + (i * (phi_subdiv + 1)));
            mb.push_index(j + (i * (phi_subdiv + 1)) + 1);
            mb.push_index(j + (i * (phi_subdiv + 1)) + (phi_subdiv + 2));

            // bottom right triangle for this respective row and column
            mb.push_index(j + (i * (phi_subdiv + 1)));
            mb.push_index(j + (i * (phi_subdiv + 1)) + (phi_subdiv + 2));
            mb.push_index(j + (i * (phi_subdiv + 1)) + (phi_subdiv + 1));
        }
    }
    //-----push indices - end -----------------------------------------------------------

    m_model.mesh = mb.build();
}
```