CSE 4208: Computer Graphics Laboratory

3D MODEL CITY USING MODERN OPENGL

By

Ehsanul Karim
Roll: 1907039

Submitted To:

Dr. Sk. Md. Masudul Ahsan
Professor
Dept. of Computer Science and Engineering
Khulna University of Engineering & Technology, KUET

Md. Badiuzzaman Shuvo
Lecturer
Dept. of Computer Science and Engineering
Khulna University of Engineering & Technology, KUET

Department of Computer Science and Engineering
Khulna University of Engineering & Technology
Khulna 9203, Bangladesh
26th January 2024

Contents

1 Introduction

The "3D Model City using modern OpenGL" project is a creative investigation into computer graphics that uses OpenGL to produce an aesthetically complex and realistic urban setting. With the addition of user-controlled items like cars and pedestrians, this simulation features a variety of city features, such as streets, buildings, playgrounds, and automobiles. Through the use of sophisticated OpenGL capabilities like fractals, dynamic lighting, and texture mapping, the project highlights the potential of 3D modeling in visualization and design.

1.1 Background

OpenGL, a widely used graphics library. Real-time rendering of 3D environments is made possible by its robust tools. The project models city structures efficiently while maintaining visual richness. It uses geometric primitives like spheres, cones, and cubes. Dynamic lighting systems give the scene depth and ambience to its environment. Features such as texture mapping enhance the realism by simulating intricate surface details. Additionally, the integration of external 3D models, such as a static helicopter from Blender, demonstrates the versatility of OpenGL in incorporating diverse assets into interactive scenes. The blending of different shader was also a major contribution of OpenGL and this project also adds this interactive feature.

1.2 Objectives

The project's major purpose is given below:

- To design and develop a realistic 3D city simulation using OpenGL.
- To implement dynamic interactions, including user-controlled movements of vehicles and pedestrians.
- To enhance scene realistic vibe through advanced techniques like lighting, texturing, and fractals.
- To draw curvy objects with Bezier curve.
- To applying blending shader effect in the scene.

2 Project Implementation

This section outlines the steps taken to develop the 3D city simulation, focusing on dynamic features, lighting controls, texturing, and advanced object integration. Each component contributes to creating an immersive and interactive environment. Figure 2.1 contains a snapshot of the overall project in birds eye point of view.

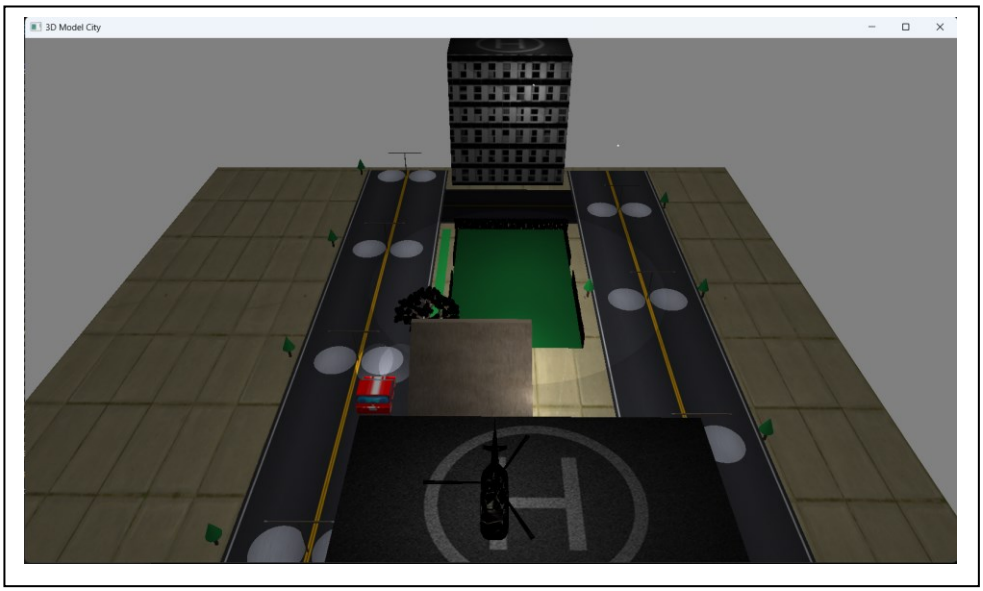

Figure 2.1: Bird's Eye View of the entire project

2.1 Physical Objects

This section describes the various physical objects created for the 3D city simulation. Each object was modeled programmatically using algorithms for vertex generation, ensuring geometric precision and visual consistency.

2.1.1 Cube

The cube is a simple geometric object composed of six faces, each a quadrilateral. Vertex positions were generated by defining eight corner points in 3D space and connecting them to form faces. The eight vertices are defined inside the program. It's used for modeling houses, walls. Also, I use cube for modeling many complex objects.
Below are some codes responsible for cube.

```
void cube_generateData()
{
	float half_x = this->cube_xlen;
	float half_y = this->cube_ylen;
	float half_z = this->cube_zlen;

	unsigned int idx = 0;

	// back
	addVertex(-half_x, half_y, -half_z, 0.0f, 0.0f, -1.0f, 0.0f, 1.0f);
	addVertex(half_x, half_y, -half_z, 0.0f, 0.0f, -1.0f, 1.0f, 1.0f);
	addVertex(-half_x, -half_y, -half_z, 0.0f, 0.0f, -1.0f, 0.0f, 0.0f);
	addVertex(half_x, -half_y, -half_z, 0.0f, 0.0f, -1.0f, 1.0f, 0.0f);

	indices.insert(indices.end(), { idx + 0, idx + 2, idx + 3 });
	indices.insert(indices.end(), { idx + 3, idx + 1, idx + 0 });

	idx += 4;
	textureCount0 = indices.size();
	///… similarly for other faces

}
```

Below figure 2.2 shows some snapshots of Cube used in many fields in the project.

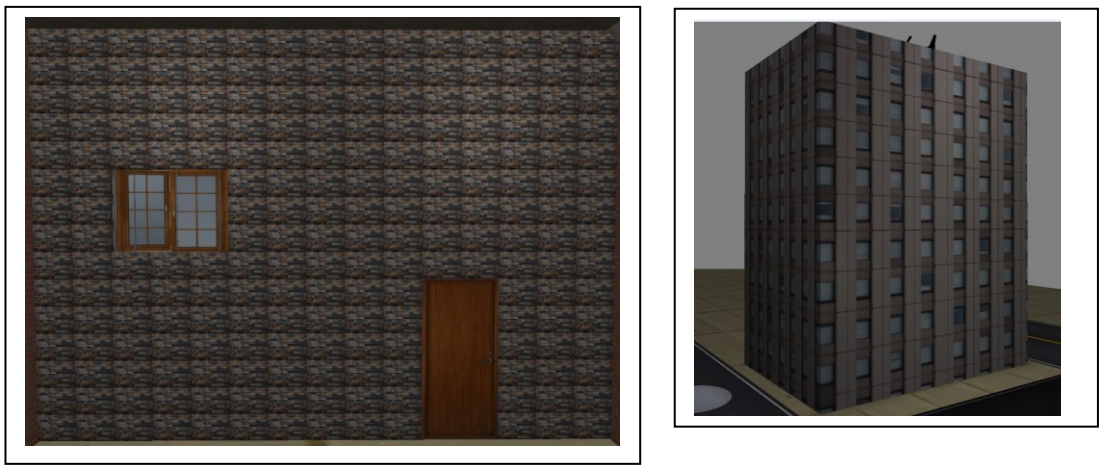

Figure 2.2: Object made with cube

2.1.2 Cylinder

A cylinder has traditionally been a three-dimensional solid, one of the most basic of curvilinear geometric shapes. In this project, I used it while creating of complex object. It is used as the car wheel. Cylinders were modeled using circular bases and a curved surface connecting the two. Vertices are generated for the top and bottom circles by iterating over angles in a loop (using trigonometric functions for x and z coordinates). Also, the normal for each vertex is calculated for smooth shading. It's used to model car wheel. A cone can be generated with this same class, by defining the top radius or bottom radius as zero.
Below code show how cylinder vertices were generated in the project.

```
void cy_generateData()
{
	vector<float> topCircle, baseCircle;
	for (int i = 0; i <= this->cy_stackCount; ++i)
	{
		float y = (this->cy_height / 2.0f) - (float)i / this->cy_stackCount * this->cy_height;
		float radius = this->cy_topRadius + (float)i / this->cy_stackCount * (this->cy_baseRadius -
this->cy_topRadius);
		float ny = 0.0f;
		float v = 1 - (float)i / this->cy_stackCount;
		for (int j = 0; j <= this->cy_sectorCount; ++j)
		{
			float sectorAngle = (float)j / this->cy_sectorCount * 2 * PI;
			float x = radius * cosf(sectorAngle);
			float z = radius * sinf(sectorAngle);
			float nx = cosf(sectorAngle);
			float nz = sinf(sectorAngle);
			float u = (float)j / this->cy_sectorCount;
			addVertex(x, y, z, nx, ny, nz, u, v);
			if (i == 0)
			{
				float u = nx / 2.0f + 0.5f;
				float v = nz / 2.0f + 0.5f;
				topCircle.push_back(x), topCircle.push_back(y), topCircle.push_back(z);
				topCircle.push_back(0.0f), topCircle.push_back(1.0f), topCir-
cle.push_back(0.0f);
				topCircle.push_back(u), topCircle.push_back(v);
			}
			else if (i == this->cy_stackCount)
			{
				float u = nx / 2.0f + 0.5f;
				float v = nz / 2.0f + 0.5f;
				baseCircle.push_back(x), baseCircle.push_back(y), baseCircle.push_back(z);
				baseCircle.push_back(0.0f), baseCircle.push_back(-1.0f), baseCir-
cle.push_back(0.0f);
				baseCircle.push_back(u), baseCircle.push_back(v);
			}
		}
	}
	for (auto a : topCircle)
	{	vertices.push_back(a);}
	for (auto a : baseCircle)
	{	vertices.push_back(a);}
	addVertex(0.0f, this->cy_height / 2.0f, 0.0f, 0.0f, 1.0f, 0.f, 0.5f, 0.5f);
	addVertex(0.0f, -(this->cy_height / 2.0f), 0.0f, 0.0f, -1.0f, 0.f, 0.5f, 0.5f);
	for (int i = 0; i < this->cy_stackCount; ++i)
	{
		int k1 = i * (this->cy_sectorCount + 1), k2 = k1 + (this->cy_sectorCount + 1);
		for (int j = 0; j < this->cy_sectorCount; ++j, ++k1, ++k2)
		{
			indices.push_back(k1), indices.push_back(k2), indices.push_back(k2 + 1);
			indices.push_back(k2 + 1), indices.push_back(k1 + 1), indices.push_back(k1);
			if (i != 0)
			{	updateNormal(k1, k2, k2 + 1);
				updateNormal(k2 + 1, k1 + 1, k1);} }
	}
	textureCount0 = indices.size();
	// top Circle
	int k = (this->cy_stackCount + 1) * (this->cy_sectorCount + 1);
	int center = (this->cy_stackCount + 3) * (this->cy_sectorCount + 1);

	for (int i = 0; i < this->cy_sectorCount; ++i, ++k)
	{ indices.push_back(center), indices.push_back(k), indices.push_back(k + 1);}
	textureCount1 = indices.size();
	// base Circle
	k = (this->cy_stackCount + 2) * (this->cy_sectorCount + 1);
	center = center + 1;
	for (int i = 0; i < this->cy_sectorCount; ++i, ++k)
	{ indices.push_back(center), indices.push_back(k), indices.push_back(k + 1);   }
	textureCount2 = indices.size();
}
```

Below figure 2.3 has some snapshots of the claimed cylinder.

Figure 2.3: Cylindrical Object and conic object

2.1.3 Sphere

A sphere is the set of points that are all at the same distance `r` from a given point in three-dimensional space. In this project, Sphere is used to draw the pedestrian head, hand any in making of more complex object. UV mapping aligns textures seamlessly across the spherical surface.

Below code show how sphere is integrated in the project.

```
void buildCoordinatesAndIndices() {
    float x, y, z, xy;                              // vertex position
    float nx, ny, nz, lengthInv = 1.0f / radius;    // normalized vertex normal
    float s, t;                                     // vertex texture coordinate
    float sectorStep = 2 * PI / sectorCount;
    float stackStep = PI / stackCount;
    float sectorAngle, stackAngle;
    for (int i = 0; i <= stackCount; ++i) {
        stackAngle = PI / 2 - i * stackStep; // starting from pi/2 to -pi/2
        xy = radius * cosf(stackAngle);      // r * cos(u)
        z = radius * sinf(stackAngle);       // r * sin(u)
        for (int j = 0; j <= sectorCount; ++j) {
            sectorAngle = j * sectorStep;    // starting from 0 to 2pi
            // vertex position (x, y, z)
            x = xy * cosf(sectorAngle);     // r * cos(u) * cos(v)
            y = xy * sinf(sectorAngle);     // r * cos(u) * sin(v)
            coordinates.push_back(x);
            coordinates.push_back(y);
            coordinates.push_back(z);
            // normalized vertex normal (nx, ny, nz)
            nx = x * lengthInv;
            ny = y * lengthInv;
            nz = z * lengthInv;
            normals.push_back(nx);
            normals.push_back(ny);
            normals.push_back(nz);
            // vertex texture coordinate (s, t)
            s = (float)j / sectorCount;
            t = (float)i / stackCount;
            texCoords.push_back(s);
            texCoords.push_back(t);
        }
    }
    // indices
    int k1, k2;
    for (int i = 0; i < stackCount; ++i) {
        k1 = i * (sectorCount + 1); // beginning of current stack
        k2 = k1 + sectorCount + 1;  // beginning of next stack
        for (int j = 0; j < sectorCount; ++j, ++k1, ++k2) {
            // two triangles per sector
            indices.push_back(k1); indices.push_back(k2); indices.push_back(k1 + 1);
             indices.push_back(k1 + 1); indices.push_back(k2); indices.push_back(k2+ 1);
        }
    }
}
```

Below figure 2.4 demonstrates the claimed sphere that is used in the project.

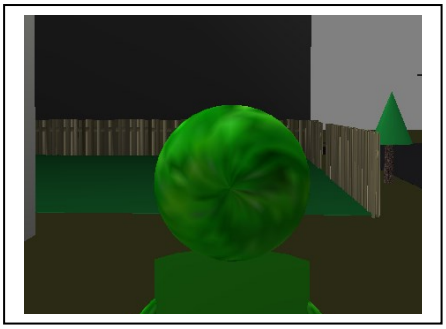

Figure 2.4: Spherical Object used in the project

2.1.4 Fractal

Fractal geometry is the geometry of self-similarity in which an object appears to look similar at different scales. Fractal trees were created by recursive branching, starting from a trunk and dividing into smaller branches. In this project, fractal is used for creating natural, visually appealing trees providing an organic look with adjustable complexity. A base cylinder with repetitive calling of itself with different scaling and positioning created the fractal tree.

Below code show how fractal object is created using simple recursion.

```
void drawFractalTree(Shader& shader, const glm::mat4& model, int depth, Shape& CubeGreen, const
glm::mat4& a_translate_mat, Bush& Leaf_Obj, const glm::mat4& model2) {
    if (depth == 0) return;
    // Draw the current cube
    CubeGreen.drawTexture(shader, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, a_trans-
late_mat * model);
    if (depth == 1)
    {
        Leaf_Obj.drawWithTexture(shader, a_translate_mat * model2);
    }
    // Scaling and translation for the branches
    scaleMatrix = glm::scale(glm::mat4(1.0f), glm::vec3(0.5f, 0.75f, 0.5f));
    float deg = 0;
    rotateY = rotate(glm::mat4(1.0f), glm::radians(deg), glm::vec3(0.0f, 1.0f, 0.0f));
    // Left branch 1
    translateLeft = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 2.0f, 0.05f));
    rotateLeft = glm::rotate(glm::mat4(1.0f), glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.0f));
    leftBranchModel = translateLeft * rotateY * rotateLeft * scaleMatrix * model;
    leftBranchforLeaf = translateLeft * rotateY * rotateLeft * model2;
    drawFractalTree(shader, leftBranchModel, depth - 1, CubeGreen, a_translate_mat, Leaf_Obj,
leftBranchforLeaf);
    // Right branch 1
    translateRight = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 2.0f, 0.05f));
    rotateRight = glm::rotate(glm::mat4(1.0f), glm::radians(-45.0f), glm::vec3(0.0f, 0.0f,
1.0f));
    rightBranchModel = translateRight * rotateY * rotateRight * scaleMatrix * model;
    rightBranchforLeaf = translateRight * rotateY * rotateRight *  model2;
    drawFractalTree(shader, rightBranchModel, depth - 1, CubeGreen, a_translate_mat, Leaf_Obj,
rightBranchforLeaf);
}
```

Below figure 2.5 demonstrates the claimed fractal tree inside the project.

Figure 2.5: Fractal Tree used in the project

2.1.5 External Blender Objects

Blender is a free and open-source 3D creation suite used for modeling, animation, rendering, and more. It is a collection of vertices, edges, and faces that define the shape and structure of the object. In this project, some blender objects including tree brush, helicopter which were imported as physical object to make the scene more realistic. The models were downloaded from https://poly.pizza/ website.

Below python snippet show how blender object vertices are extracted (i.e Scene Coordinate to World Coordinate transformation) to use it inside our project.

```
def parse_obj_file(file_path):
    vertices = []
    normals = []
    textures = []
    faces = []

    with open(file_path, "r") as file:
        for line in file:
            parts = line.strip().split()
            if not parts:
                continue
            if parts[0] == "v":
                vertices.append([float(x) for x in parts[1:]])
            elif parts[0] == "vn":
                normals.append([float(x) for x in parts[1:]])
            elif parts[0] == "vt":
                textures.append([float(x) for x in parts[1:]])
            elif parts[0] == "f":
                # Parse face indices (vertex/texture/normal)
                face = [
                    [int(idx) - 1 if idx else -1 for idx in vert.split("/")]
                    for vert in parts[1:]
                ]
                faces.append(face)

    # Automatically calculate texture coordinates if missing
    if not textures:
        textures = calculate_texture_coordinates(vertices)

    return vertices, normals, textures, faces
```

Below figure 2.6 shows the blender objects imported in this project.

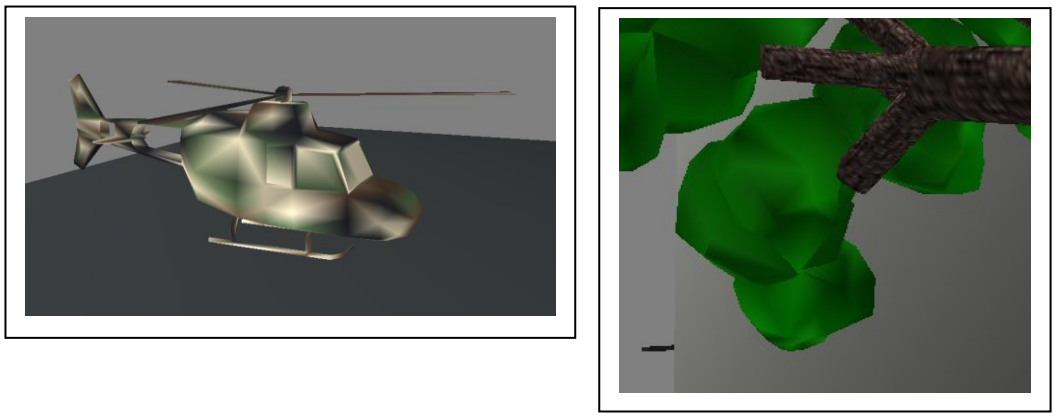

Figure 2.6: 3D model object from blender

2.2 Lighting Controls

Lighting enhances the realism of the environment by simulating how objects interact with light sources. Three types of lights were implemented: point lights, spotlights, and directional lights. In this section, what kinds of lighting was implemented in the project will be described.

2.2.1 Point Light

A point light emits light in all directions from a specific position. In this project, total 4 Point-lights were added to simulate building roof lighting and home lighting. Each light's on/off state is toggled using specific keys. Table-1 below shows which keys toggles which point light in the scene.

Table-1: Keyboard activity triggering point light control

Key	Activity
Num1, Num2, Num3, Num4	Turn on point light 1, 2, 3 and 4 respectively.

Point lights were implemented using OpenGL's shader programs. Their position and properties (diffuse, ambient, and specular) were passed to the fragment shader to illuminate objects within their range.

Below are some codes responsible for point light.

```
vec3 CalcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    // attenuation
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic *
(distance * distance));
    // combine results
    vec3 ambient = light.ambient * vec3(texture(material.diffuseMap, TexCoords)) * material.am-
bient;
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuseMap, TexCoords)) * mate-
rial.diffuse;
    vec3 specular = light.specular * spec * vec3(texture(material.diffuseMap, TexCoords)) * ma-
terial.specular;
    ambient *= attenuation;
    diffuse *= attenuation;
    specular *= attenuation;
    return (ambient + diffuse + specular);
}
```

Below figure 2.7 shows snapshots of the claimed point light functionality.

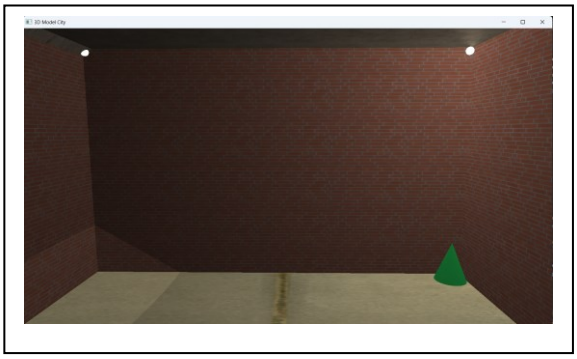

Figure 2.7: Only Point Light 2 is activated

2.2.2 Spotlight

Spotlights emit light in a specific direction, creating a focused beam. This project includes a spotlight at the playground for enhanced realism. In this project, total 13 spotlights were added to simulate car headlights and road lampposts. Any class lights can be toggled with used intervention. Table-2 below shows which keys toggles which spot light in the scene.

Table-2: Keyboard activity triggering spotlight control

Key	Activity
Key E	Toggling spotlight on the field.
Key Comma	Toggling road spot light.
Key 8	Toggling car headlight.

Spotlights were added by defining a cone-shaped area of illumination using the fragment shader. Below code show how spotlight integrated in the project.

```
vec3 CalcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 viewDir)
{
    vec3 lightDir = normalize(light.position - fragPos);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
    // attenuation
    float distance = length(light.position - fragPos);
    float attenuation = 1.0 / (light.constant + light.linear * distance + light.quadratic *
(distance * distance));
    // spotlight intensity
    float theta = dot(lightDir, normalize(-light.direction));
    float epsilon = light.cutOff - light.outerCutOff;
    float intensity = clamp((theta - light.outerCutOff) / epsilon, 0.0, 1.0);
    // combine results
    vec3 ambient = light.ambient * vec3(texture(material.diffuseMap, TexCoords)) * material.am-
bient;
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuseMap, TexCoords)) * mate-
rial.diffuse;
    vec3 specular = light.specular * spec * vec3(texture(material.diffuseMap, TexCoords)) * ma-
terial.specular;
    ambient *= attenuation * intensity;
    diffuse *= attenuation * intensity;
    specular *= attenuation * intensity;
    return (ambient + diffuse + specular);
}
```

Below figure 2.8 has some snapshots of the claimed spotlight functionality.

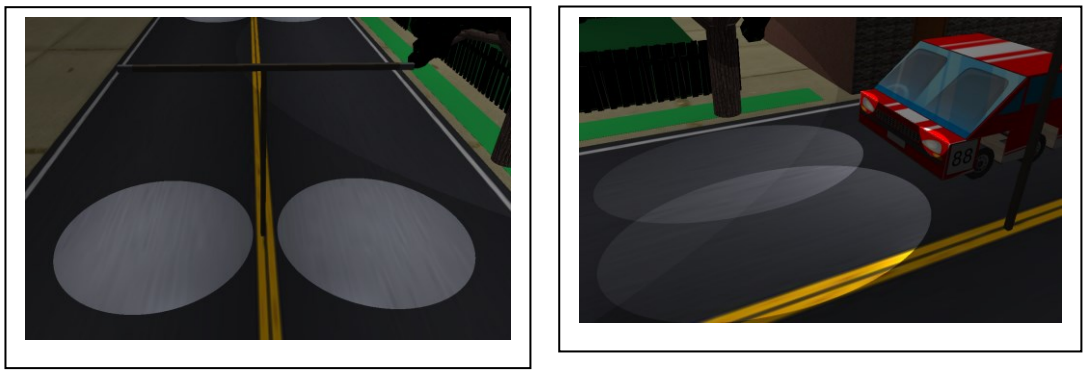

Figure 2.8: Spotlight on road lamppost and car headlight.

2.2.3 Directional Light

Directional lights simulate sunlight by providing uniform lighting across the scene from a fixed direction. In this project, one directional light added to simulate sun light. This light can be toggled with used intervention making day and night. Table-3 below shows which keys toggles the directional light.

Table-3: Keyboard activity triggering directional light control

Key	Activity
Key Q	Toggling direction light

Directional lights were implemented by specifying a light direction vector in the shaders. Below code show how directional light integrated in the project.

```
vec3 CalcDirLight(DirLight light, vec3 normal, vec3 viewDir)
{
    vec3 lightDir = normalize(-light.direction);
    // diffuse shading
    float diff = max(dot(normal, lightDir), 0.0);
    // specular shading
    vec3 reflectDir = reflect(-lightDir, normal);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    // combine results
    vec3 ambient = light.ambient * vec3(texture(material.diffuseMap, TexCoords)) * material.am-
bient;
    vec3 diffuse = light.diffuse * diff * vec3(texture(material.diffuseMap, TexCoords)) * mate-
rial.diffuse;
    vec3 specular = light.specular * spec * vec3(texture(material.diffuseMap, TexCoords)) * ma-
terial.specular;
    return (ambient + diffuse + specular);
}
```

Below figure 2.9 shows some snapshots of the claimed directional light functionality.

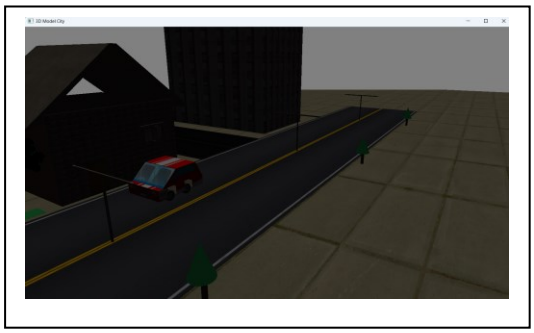

Figure 2.9: Only directional light is on.

2.3 Texturing

Textures enhance visual realism by adding surface details to objects. The texture coordinates are auto generated. In this section, how texture mapping is applied in the project is discussed thoroughly.

2.3.1 On Cube

Texture coordinates for cubes were assigned to each face using mappings. Each face texture must be passed when creating an object of the Cube class. The objects material property is also initialized at that time.

2.3.2 On Sphere

Spherical objects required UV mapping, where latitude and longitude lines were used to assign texture coordinates based on angular calculations. This project includes a spherical object.

2.3.3 On Cylinder

Cylinders were textured by wrapping the texture around the curved surface and aligning it with the circular caps. In this project, while creating a cylinder object, three texture parameter must be passed. Below figure 2.10 shows some textures that are used in the project.

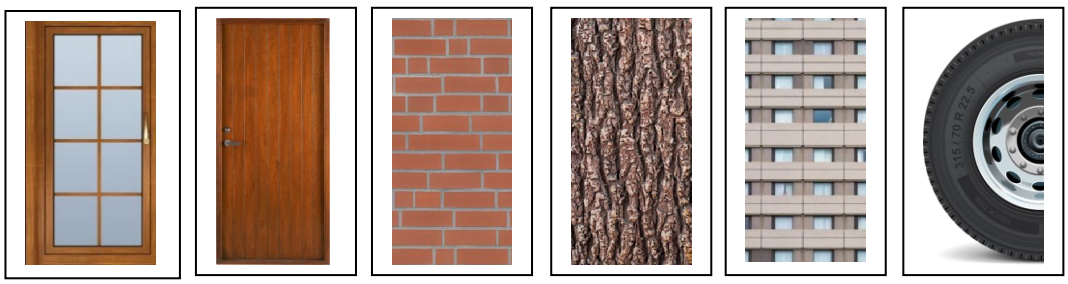

Figure 2.10: Textures used in the project

2.4 Blending Multiple Shader

A methodology applied in this project, where the pixel intensity is calculated by blending both lighting and texture shaders. First the lighting shader calculates the lighting intensity at each pixel. Then texture shader samples the color from the texture at the corresponding texture coordinates. Finally, blending of the lighting intensity and the texture color is performed to get the final pixel color (i.e. just multiplying the texture color with the lighting intensity). Table-4 shows the keys that toggles the blending shader.

Table-4: Keyboard activity triggering spotlight control

Key	Activity
Key F, G	Toggling between no light shader and its blending effect
Key K, L	Toggling between Gouraud shader and its blending effect
Key O, P	Toggling between Phong shader and its blending effect

Below figure 2.11, 2.12 and 2.13 some snapshots of the claimed Multiple shader blending effect used in the project.

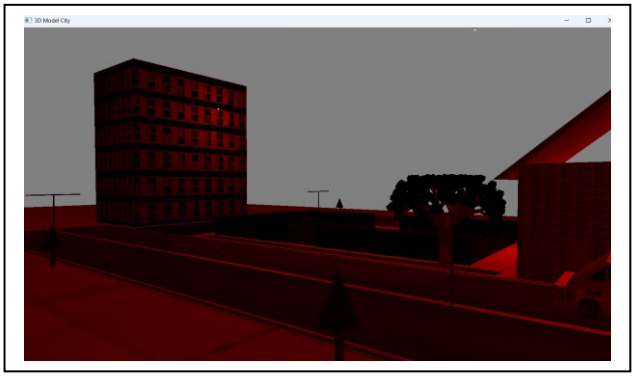

Figure 2.11: Blending with Phong shading

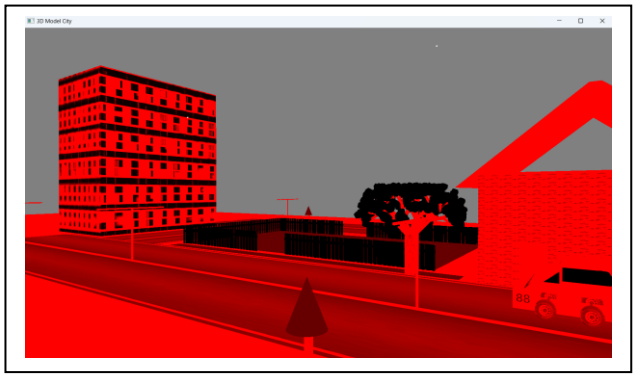

Figure 2.12: Blending with Gouraud shading

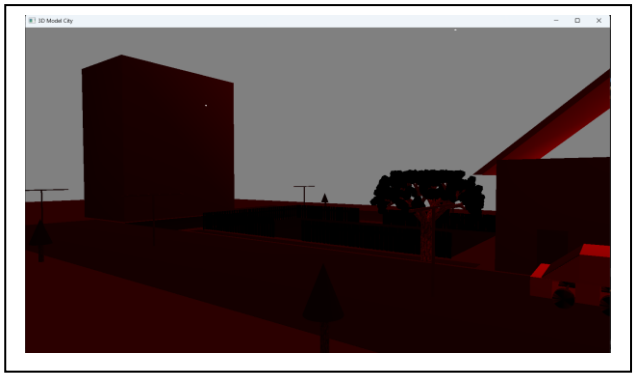

Figure 2.13: Blending with No light shading

2.5 Curvy Object Integration

Bezier curves are used in computer graphics to draw shapes, for CSS animation and in many other places. A set of discrete "control points" defines a smooth, continuous curve by means of a formula. Bezier curves was tried to implement in this project to create a ferris wheel. The integration of it wasn't possible, the idea of generating a curvy object using Bezier curve was studied thoroughly.

Below is the claimed Bezier curve integration related code.

```
void BezierCurve(double t, float xy[2], GLfloat ctrlpoints[], int L)
{
    double y = 0;
    double x = 0;
    t = t > 1.0 ? 1.0 : t;
    for (int i = 0; i < L + 1; i++)
    {
        long long ncr = nCr(L, i);
        double oneMinusTpow = pow(1 - t, double(L - i));
        double tPow = pow(t, double(i));
        double coef = oneMinusTpow * tPow * ncr;
        x += coef * ctrlpoints[i * 3];
        y += coef * ctrlpoints[(i * 3) + 1];
    }
    xy[0] = float(x);
    xy[1] = float(y);
}

const float dtheta = 2 * 3.1416 / ntheta;        //angular step size
float t = 0;
float dt = 1.0 / nt;
float xy[2];
for (i = 0; i <= nt; ++i)              //step through y
{
    BezierCurve(t, xy, ctrlpoints, L);
    r = xy[0];
    y = xy[1];
    theta = 0;
    t += dt;
    lengthInv = 1.0 / r;
    for (j = 0; j <= ntheta; ++j)
    {
        double cosa = cos(theta);
        double sina = sin(theta);
        z = r * cosa;
        x = r * sina;
        coordinates.push_back(x);
        coordinates.push_back(y);
        coordinates.push_back(z);
        // normalized vertex normal (nx, ny, nz)
        // center point of the circle (0,y,0)
        nx = (x - 0) * lengthInv;
        ny = (y - y) * lengthInv;
        nz = (z - 0) * lengthInv;
        normals.push_back(nx);
        normals.push_back(ny);
        normals.push_back(nz);
        theta += dtheta;
    }
}
```

2.6 Dynamic Interactions (Keyboard Functionalities)

To make the scene dynamic and interactive, various keyboard inputs were implemented for controlling objects and the camera. These controls allow users to navigate the environment and interact with elements such as cars, pedestrians, and lighting. Table-5 below summarizes the key mappings and their respective activities. These functionalities are handled using OpenGL's event processing, providing real-time feedback for a smooth user experience.

Table-5: Keyboard activity triggering dynamic interaction

Key	Activity
W, A, S, D	Move the camera forward, left, backward, right
Up/Down Arrow	Move the pedestrian along the X/Z axis

Num 8, Num 2	Move the car forward and backward along the X-axis
Comma (,)	Open/close the window of a house

Below Figure 2.14 and 2.15 shows some snapshots of the claimed functionalities. The snapshot could be uncleared, so it's recommendable to visit the video demonstration to observe the functionalities thoroughly.

Figure 2.14: Car moving in the scene and window open

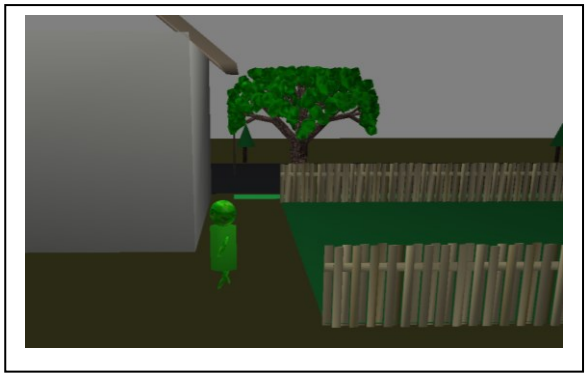

Figure 2.15: Pedestrian walking in the field.

3 Project Demonstration

This project contains realistic 3D modelingm This section contains several screen shots of the entire project from different angle view. Fig 3.1 contains a scene with car, buildin1, house and roads. Fig 3.2 contains a scene with fields and house. Fig 3.3 contains a scene with building1 and road lamposts. Fig 3.4 shows the helicopter with helipad at top of building2.

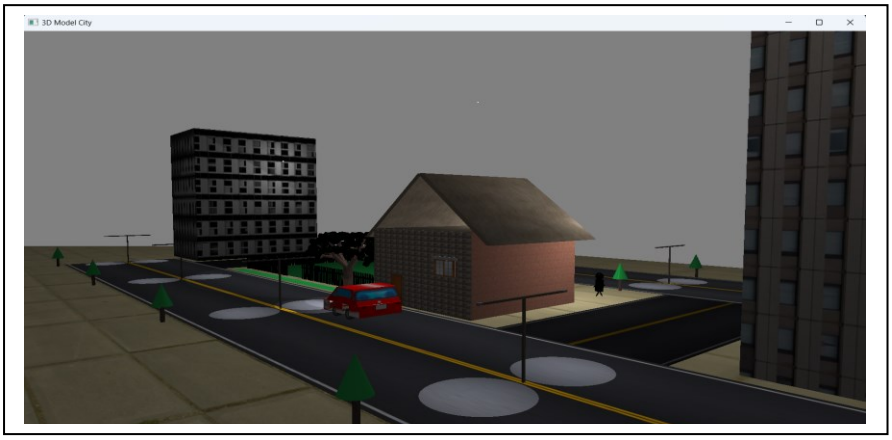

Figure 3.1: Scene 1 of the 3D model city

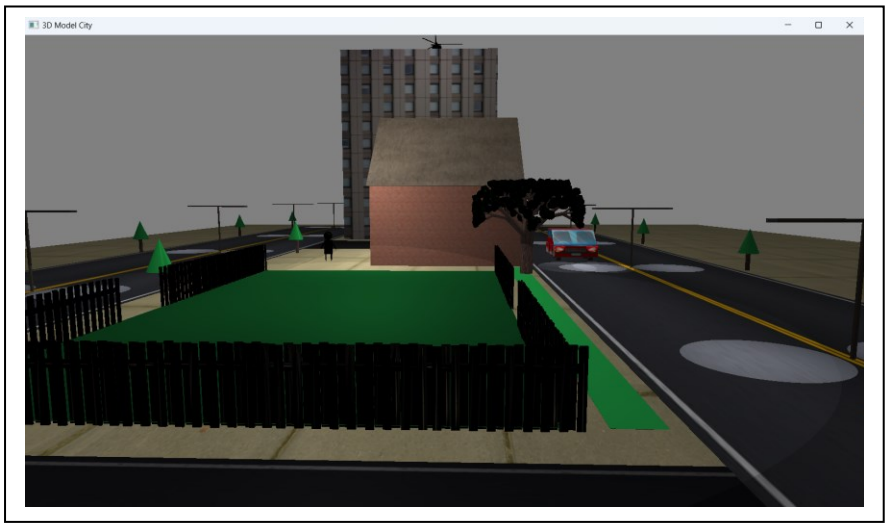

Figure 3.2: Scene 2 of the 3D model city

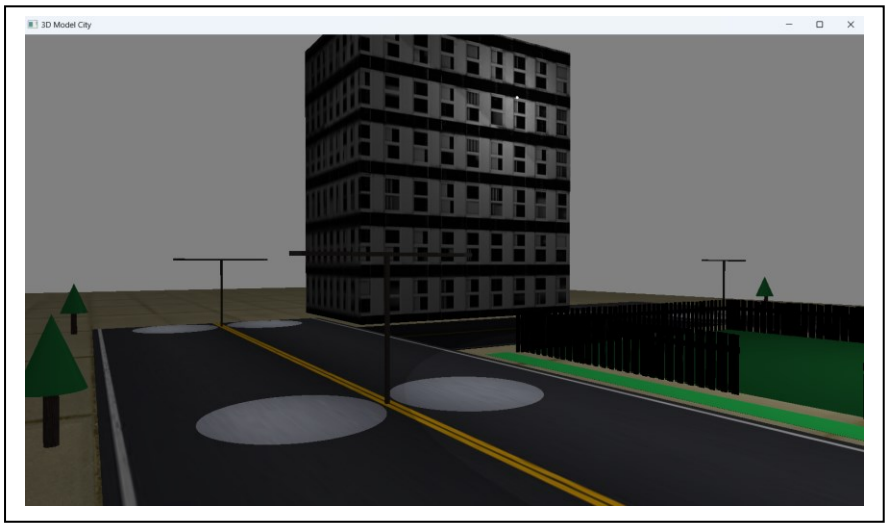

Figure 3.3: Scene 3 of the 3D model city

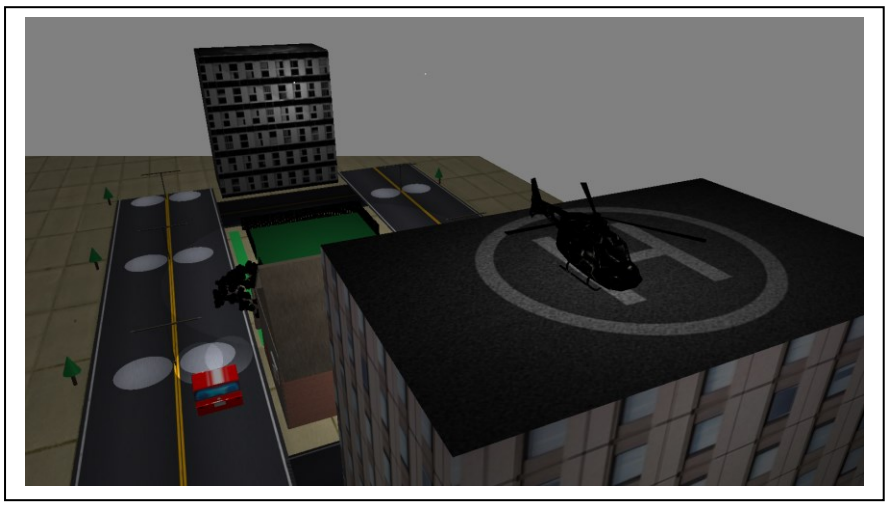

Figure 3.4: Scene 4 of the 3D model city

4 Conclusion

This project effectively demonstrates the capability of OpenGL to create a dynamic and interactive 3D simulation of a city. Through the use of texture mapping, lighting, geometric modeling, and interactive features, the project delivers a visually engaging and realistic urban environment. Static and dynamic elements, such as curvy objects and user-controlled vehicles, enhance the scene's depth. Future improvements could involve optimizing performance, adding more advanced features, and incorporating higher levels of detail to make the simulation more immersive and versatile.

4.1 Conclusion and challenges faced

The development of project presented several challenges that required iterative refinement and problem-solving. Performance optimization was a key hurdle, as frame rates and ensuring smooth rendering in a complex scene with multiple dynamic objects proved difficult. Lighting, such as balancing the intensity and positioning of point lights and spotlights, were necessary to achieve realistic without overexposure. Implementing Bezier curves and fractals required a deep understanding of mathematical, while texture mapping on irregular shapes like spheres and cones demanded meticulous alignment. Integrating external Blender models into the OpenGL environment without losing detail was another significant challenge. Additionally, designing responsive keyboard inputs and maintaining the realism of dynamic interactions further tested the robustness of the implementation. Overcoming these provided invaluable insights and practical skills.

5 References

[1] Demko, Stephen, Laurie Hodges, and Bruce Naylor. "Construction of fractal objects with iterated function systems." Proceedings of the 12th annual conference on Computer graphics and interactive techniques. 1985.

[2] Farin, Gerald. "Class a Bézier curves." Computer aided geometric design 23.7 (2006): 573-581.

[3] Shreiner, Dave, et al. "OpenGL Programming Guide: The Official Guide to Learning OpenGL." Addison-Wesley Professional, 2013.