OpenGL 3D PaperBoy Project

Compulsory OpenGL elementsdww

Displaying 3D Polygon Mesh:

In OpenGL I created a class for models and for meshes so that each model imported cllould be composed of several meshes and each mesh could be later manipulated relative to the overall model.
To load 3D models into the game I used Assimp, which is an Open Asset Import Library that allows you to import and export various 3d-model-formats. I used the .obj format for all my models and prepared them all for export in Blender.
The objects I modelled and included in my scene are: a bicycle, 4 houses with post boxes, a street, and some hedges.

For models that move such as the paperboy and newspaper I created their own class which contained variables such as position, orientation etc.

The paperboy class contained other variables and functions which relate to gameplay, such as “remaining_papers” and “throwPapers()”.

Interactive Manipulation

I designed the controls for the game to be the same as the original with WASD used for steering the bike and Q and E used to throw the papers left and right.

The controls that need to be held down for movement (WASD) are handled in the keyDown and keyUp function which are called on every loop of main().
The keypress function is also called on every loop of main() and is used to trigger once off keypresses, in this case throwing the papers left and right.

The other interactive manipulation I have implemented is to use the keys IJKL to move the camera around the scene. It's the same camera controls that's used to navigate the camera in Unreal Engine and Unity
spawn a newspaper and project it along a parabolic curve when the throw button is pressed.

Complex Object with a Hierarchical Structure

I chose to make the bike the complex object and its wheels spinning being the hierarchical structure.
To do this I first needed to ensure the wheels were separate meshes in Blender. I then needed to export the .obj with the moving parts of the mesh translated to the origin of the world. This meant when I translated the mesh in the game it would rotate about its centre point as opposed to the centre of the bicycle model.
In OpenGL

In the renderPaperBoy() function I initialise a 4x4 identity matrix for the bicycle. I then translate that matrix using the position and rotation variables contained by the bicycle class. I then render all the non moving meshes at that position.
For the wheels, a 4x4 identity matrix is initialised, it is rotated to a certain degree that was passed in by the PaperBoy class, and then translated to the position it should be if the bike was at the centre of the world. That matrix is then multiplied by the matrix of the bicycle so that it is moved to the same position of the bike in the world. This is done to both wheels before passing them into the RenderMesh() function.

```
void PaperBoy::renderPaperBoy(mat4 pos, int shaderID) {

    //Transform the PaperBoy
    mat4 paperboy_pos = identity_mat4();
    paperboy_pos = translate(paperboy_pos, vec3(0.0f, 0.0f, 0.0f));
    paperboy_pos = rotate_y_deg(paperboy_pos, Yaw);
    paperboy_pos = translate(paperboy_pos, Pos);

    int world_loc = glGetUniformLocation(shaderID, "world");
    glUniformMatrix4fv(world_loc, 1, GL_FALSE, paperboy_pos.m);


    for (int i = 0; i < paperboy.getNumberMeshes(); i++) {
        if (i != 3 && i != 14) {
            paperboy.RenderMesh(paperboy_pos, shaderID, i);
        }
    }
    mat4 backwheelMat = identity_mat4();
    backwheelMat = rotate_z_deg(backwheelMat, -rotate_y_sin);
    backwheelMat = translate(backwheelMat, vec3(-0.358359f, 0.256741f, -0.001052f));
    backwheelMat = paperboy_pos * backwheelMat;
    glUniformMatrix4fv(world_loc, 1, GL_FALSE, backwheelMat.m);
    paperboy.RenderMesh(backwheelMat, shaderID, 3);

    mat4 frontwheelMat = identity_mat4();
    frontwheelMat = rotate_z_deg(frontwheelMat, -rotate_y_sin);
    frontwheelMat = translate(frontwheelMat, vec3(0.408732f, 0.25413, -0.000986));
    frontwheelMat = paperboy_pos * frontwheelMat;
    glUniformMatrix4fv(world_loc, 1, GL_FALSE, frontwheelMat.m);
    paperboy.RenderMesh(frontwheelMat, shaderID,14);

}
```

The Scene must be Lit and Shaded, including diffuse and specular objects

The lighting for the game is all calculated in the fragment shader.

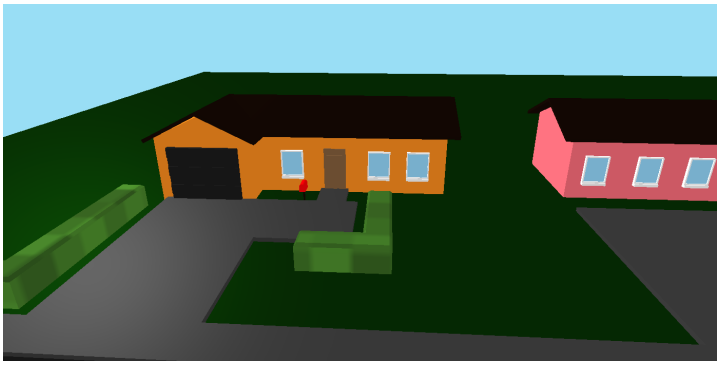
To calculate the lighting I calculated separately the diffuse, specular and ambient light values for each pixel and then used the sum of these values to determine the final pixel colour.

The Diffuse lighting is the primary source of light in a scene and is responsible for the general level of illumination. I calculated this using the dot product between the surface normal and the light direction, which results in a scalar value that represents the intensity of the light at that point.

Specular lighting is the bright highlights that appear on shiny surfaces and is used to simulate the reflection of light off of these surfaces. I calculated this using the dot product between the surface normal and the reflected light direction, as well as the dot product between the view direction and the reflected light direction.

Ambient lighting is a general, uniform light that fills the entire scene and is used to provide a base level of illumination. It is not dependent on the position or direction of any specific light source, but rather is determined by the overall ambient lighting of the scene. I kept the ambient at a value of 1, so that the game stayed bright and cartoonish looking like the original.

In Figure X, you can see the diffuse lighting on the side of the pink building. The specular highlights from the directional sun light can be seen on the driveway leading up to the house and the ambient light is what is used to light the areas that would otherwise be in shade such as the front of the two houses.

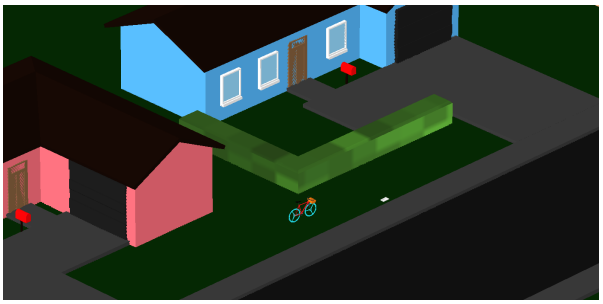
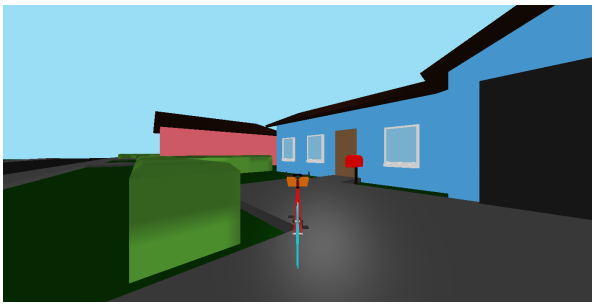


Camera Viewpoints

I have created three types of cameras that can be used in the game.
The first camera I implemented was the fly around camera. The mouse controls the pitch and yaw of the camera and changes the “Front” vector of the camera. The keys IJKL then move the camera left right forward and back relative to the front vector of the camera.

The second camera I implemented is the one that closely resembles the camera in the game. It follows the position of the bike but doesn’t rotate when the bike rotates. It has an isometric perspective which I faked by reducing the angle of view to a very small degree and positioning the camera far away.

The third camera I implemented is from a third person perspective of the bike. It is set at a fixed distance behind the bike and follows its position and direction as it moves.

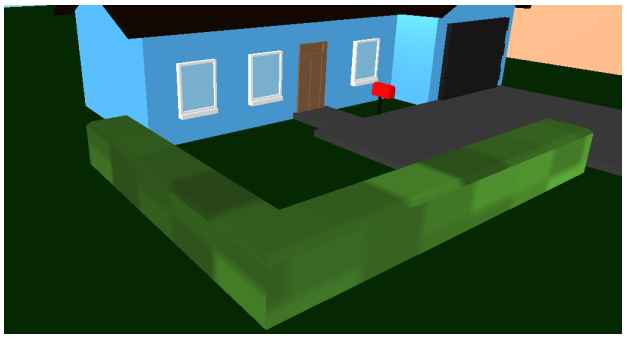



Other Advanced OpenGL Features

Texturing

I added a texture and a material class to handle the loading of .mtl files and their corresponding textures. To demo this I added a texture to the hedges in the game. I was unable to scale the UVs and adjust the scale of the texture but it still gives an interesting effect. (I could scale down but not up, likely due to memory constraints.)

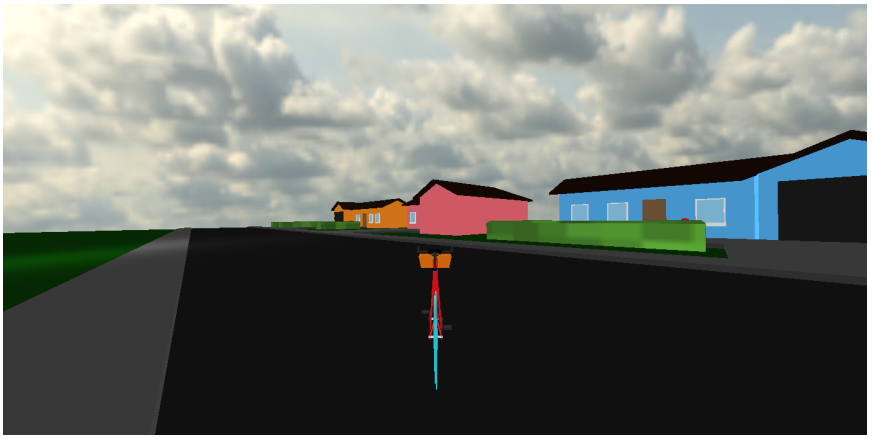
The variables of the texture class include the filename, the texture target (2D etc) and the texture name.
I created a load function that first loaded the texture image from storage into an array using stbi_load from the stb_image public domain library. It first uses glGenTextures to generate a texture name. It then uses glBindTexture to bind the new texture name to a texture target. It then defines the texture using glTexImage2D. This loads the image data into a texture object. It then uses glTexParameter to assign various parameters to the texture such as which wrapping and minification and magnification techniques to use. I also created a bind function that is called at least once for the specific texture unit. It first sets the current TextureUnit as the active texture using glActiveTexture and subsequently binds the texture name to the texture target.



SkySphere

I created a sphere model in blender and texture mapped it with a HDR image of a sky.
I imported the object into the application and set up the projection and model-view matrices so that the sky sphere was always centred around the camera and appears to be infinitely far away. I set up a separate shader so that just the diffuse texture would appear.

```
if (skyMap == true) {
        skyShader.use();
        world = scale(world, vec3(2.0f, 2.0f, 2.0f));
        sphere.RenderModel(world * view, skyShader.ID);
}
```



Fog

I implemented the fog in the fragment shader. I used a mix function that blends between the background colour and the original fragment colour for each pixel.
The weight given to the fog is determined by a variable called visibility calculated in the vertex shader. The visibility variable is calculated by using the distance from the pixel to the camera and a gradient as an exponent. (See calculation)

```
//Vertex Shader
vec4 worldPosition = world * vec4(vertex_position,1.0);
vec4 positionRelativeToCam = view * worldPosition;
float distance = length(positionRelativeToCam.xyz);
visibility = exp(-pow((distance*density),gradient));
visibility = clamp(visibility, 0,1);

//Fragment Shader
gl_FragColor = mix(backgroundColour,gl_FragColor, visibility);
```

Video Demonstration Link

https://youtu.be/JIYxIeUEjpk

GitHub Link

https://github.com/cormacmadden/Computer_Graphics_2022