Computer Graphics

implementation Plan

3D Polygon Mesh:

Model Basic Bike and character **with Hierarchy**, see (Lab3 screenshot)
Model of basic building
3D Map with footpaths and driveways
Newspaper pickup
Hedges

Interactive Manipulation

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

Complex Object with a Hierarchical Structure

The bike with the spinning wheels and the human with moving legs is the goal, but initially I will work on the spinning wheels, then the steering and the legs if

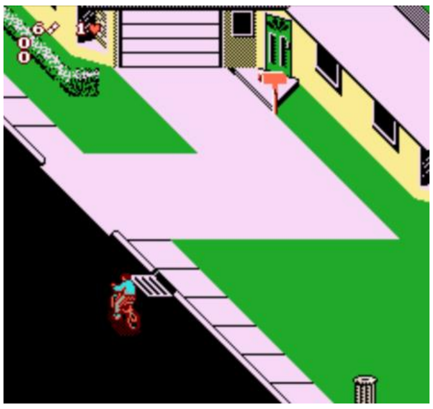
Lit and Shaded, including diffuse and specular objects

The plan is to create diffuse, and specular objects but to shade the buildings as flat and to have the newspapers be highly specular and shiney.

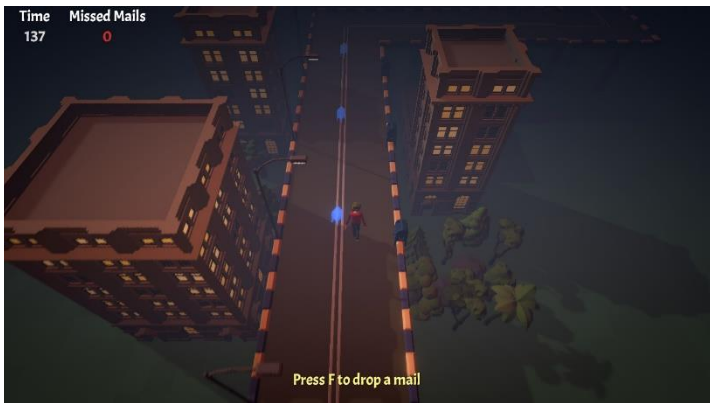
Camera Viewpoints

I'm going to create an isometric viewpoint that is the same as the original game.

I'm also going to create a perspective camera from the same viewpoint and a top-down perspective tracked to the postman.



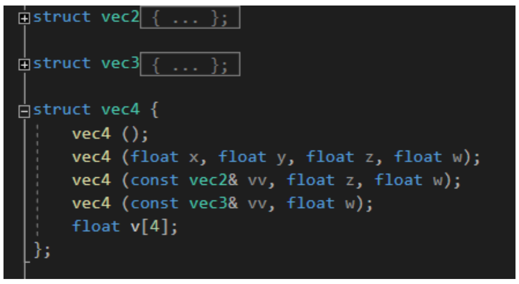
(isometric view from original game)



(top-down perspective view from different game)

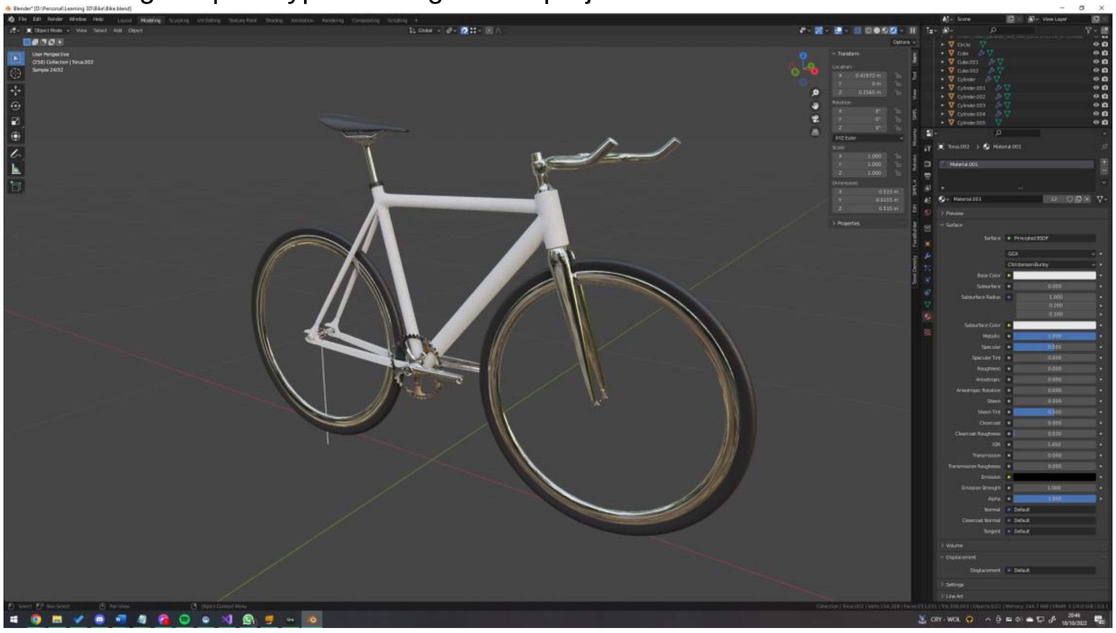
Lab 3:

1. Include Maths Class
   (using custom file with math functions) “math_funcs.h”

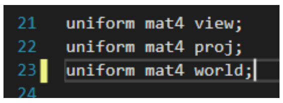


```
struct vec2 { ... };

struct vec3 { ... };

struct vec4 {
    vec4 ();
    vec4 (float x, float y, float z, float w);
    vec4 (const vec2& vv, float z, float w);
    vec4 (const vec3& vv, float w);
    float v[4];
};
```

2. 3D Models
   I have started working on a bike model in blender. I’ll probably just use a low poly version to get a prototype working for the project.

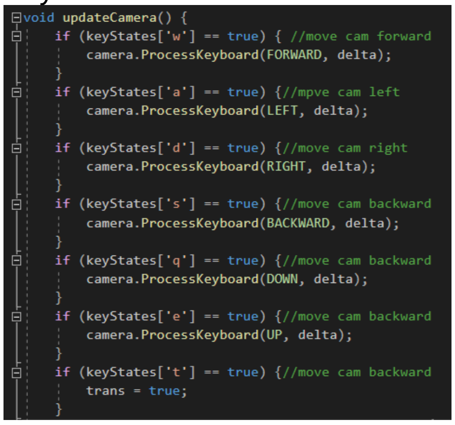


3. Multiple VBOs
   (see 7)
4. Uniform Variables



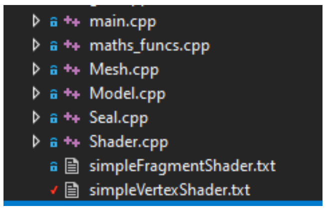
```
uniform mat4 view;
uniform mat4 proj;
uniform mat4 world;
```

5. Keyboard and Mouse handler



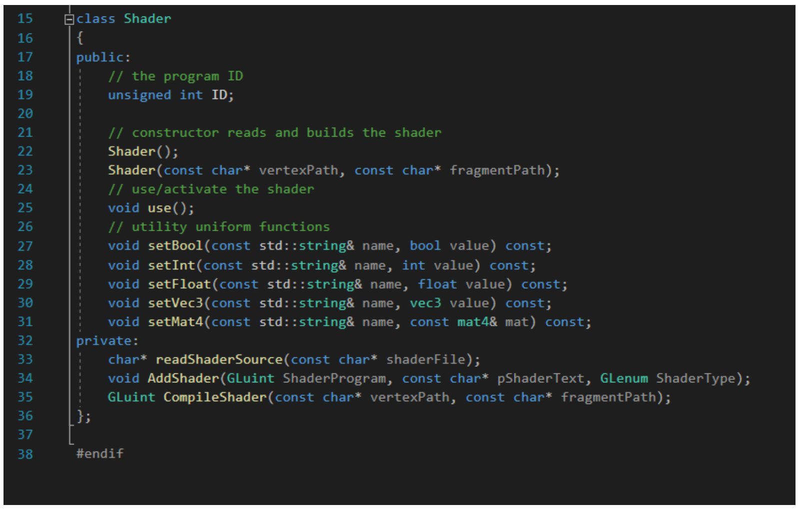
```
void updateCamera() {
    if (keyStates['w'] == true) { //move cam forward
        camera.ProcessKeyboard(FORWARD, delta);
    }
    if (keyStates['a'] == true) {//mpve cam left
        camera.ProcessKeyboard(LEFT, delta);
    }
    if (keyStates['d'] == true) {//move cam right
        camera.ProcessKeyboard(RIGHT, delta);
    }
    if (keyStates['s'] == true) {//move cam backward
        camera.ProcessKeyboard(BACKWARD, delta);
    }
    if (keyStates['q'] == true) {//move cam backward
        camera.ProcessKeyboard(DOWN, delta);
    }
    if (keyStates['e'] == true) {//move cam backward
        camera.ProcessKeyboard(UP, delta);
    }
    if (keyStates['t'] == true) {//move cam backward
        trans = true;
    }
```

6. External Shaders



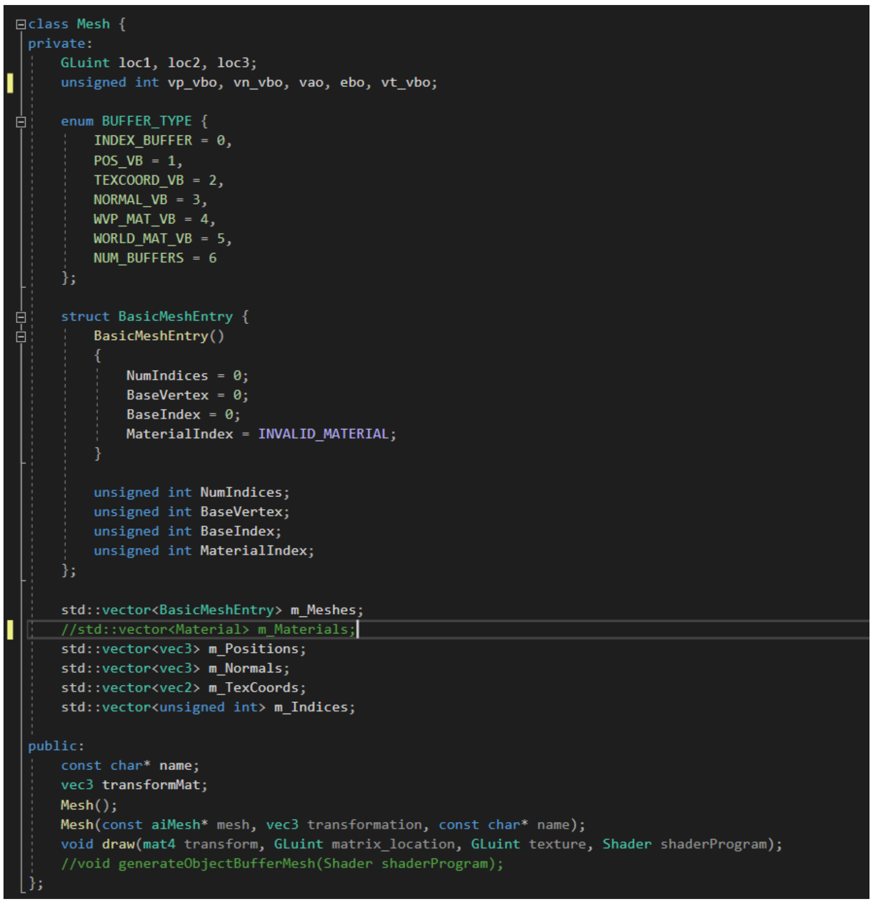
```
main.cpp
maths_funcs.cpp
Mesh.cpp
Model.cpp
Seal.cpp
Shader.cpp
simpleFragmentShader.txt
simpleVertexShader.txt
```

7. Multiple Shaders



```
class Shader
{
public:
    // the program ID
    unsigned int ID;

    // constructor reads and builds the shader
    Shader();
    Shader(const char* vertexPath, const char* fragmentPath);
    // use/activate the shader
    void use();
    // utility uniform functions
    void setBool(const std::string& name, bool value) const;
    void setInt(const std::string& name, int value) const;
    void setFloat(const std::string& name, float value) const;
    void setVec3(const std::string& name, vec3 value) const;
    void setMat4(const std::string& name, const mat4& mat) const;
private:
    char* readShaderSource(const char* shaderFile);
    void AddShader(GLuint ShaderProgram, const char* pShaderText, GLenum ShaderType);
    GLuint CompileShader(const char* vertexPath, const char* fragmentPath);
};

#endif
```

8. Mesh Class



```
class Mesh {
private:
    GLuint loc1, loc2, loc3;
    unsigned int vp_vbo, vn_vbo, vao, ebo, vt_vbo;

    enum BUFFER_TYPE {
        INDEX_BUFFER = 0,
        POS_VB = 1,
        TEXCOORD_VB = 2,
        NORMAL_VB = 3,
        WVP_MAT_VB = 4,
        WORLD_MAT_VB = 5,
        NUM_BUFFERS = 6
    };

    struct BasicMeshEntry {
        BasicMeshEntry()
        {
            NumIndices = 0;
            BaseVertex = 0;
            BaseIndex = 0;
            MaterialIndex = INVALID_MATERIAL;
        }

        unsigned int NumIndices;
        unsigned int BaseVertex;
        unsigned int BaseIndex;
        unsigned int MaterialIndex;
    };

    std::vector<BasicMeshEntry> m_Meshes;
    //std::vector<Material> m_Materials;
    std::vector<vec3> m_Positions;
    std::vector<vec3> m_Normals;
    std::vector<vec2> m_TexCoords;
    std::vector<unsigned int> m_Indices;

public:
    const char* name;
    vec3 transformMat;
    Mesh();
    Mesh(const aiMesh* mesh, vec3 transformation, const char* name);
    void draw(mat4 transform, GLuint matrix_location, GLuint texture, Shader shaderProgram);
    //void generateObjectBufferMesh(Shader shaderProgram);
};
```