Graphics Programming
Coursework

Eugén Cowie (S1714780)
BSc Computer Games (Software Development)

ecowie202@caledonian.ac.uk

I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.

– Eugén Cowie

Table of Contents

1 Game Framework 3

1.1 Game class 3

1.2 Game class constructor 3

1.3 Run method 4

1.4 Update and render methods 4

2 Models 5

2.1 Model class 5

2.2 Model constructor 5

3 Shaders 6

3.1 Colourful shader 6

3.2 Phong lighting shader 7

3.3 Toon lighting shader 13

4 Camera 15

4.1 Camera class 15

5 Shader class 16

5.1 Shader class 16

5.2 Program class 16

6 Shader implementation methods 17

6.1 Uniform methods 17

6.2 Vertex attribute methods 17

1 Game Framework

1.1 Game class

The game class (Figure 1) is responsible for initialising and running the game. It owns and manages the window and input manager, viewport and camera, shader programs, light sources, game objects and game state.

```
class Game
{
public:
    Game();
    void run();

private:
    void update(int elapsedTime);
    void render(int elapsedTime);

    Window m_window;
    Input m_input;
    Viewport m_viewport;
    Camera m_camera;

    vector<ProgramPtr> m_shaders;
    vector<ILightPtr> m_lights;
    vector<GameObjectPtr> m_objects;
};
```

Figure 1. Game class (game.hpp).

1.2 Game class constructor

The game class constructor (Figure 2) initialises the window, sets the viewport and camera to their initial values, loads the shader programs, creates the initial light sources, creates and loads the game objects and sets the initial game state.

```
Game::Game() :
    m_window("GFX Coursework", {1280, 720}),
    m_viewport(m_window.size()),
    m_camera({-15, 3, 5}, {-5, 3, -5}),
    m_shaders({
        Program("res/shaders/toon"),
        // etc...
    }),
    m_lights({
        DirectionalLight({0, -1, 0}, vec3(0), {0, 0, 0.2f}),
        // etc...
    }),
    m_objects({
        GameObject(
            Model(shaders[0], "res/models/street/street.obj"),
            Transform({}, {-0.5f, 0.5f, 0.5f}, {{270}}),
            m_viewport, m_camera
        ),
        // etc...
    }),
    m_currentShader((int)m_shaders.size())
{
}
```

Figure 2. Game constructor (game.cpp).

1.3 Run method

The run method (Figure 3) contains the main game loop, which will run for as long as the window should remain open. It measures the time elapsed since the previous frame and passes it to the update and render methods as a parameter.

```
void Game::run()
{
    int prevTime = m_window.ticks();

    while (!m_window.shouldClose())
    {
        int currentTime = m_window.ticks();
        int elapsedTime = currentTime - prevTime;

        update(elapsedTime);
        render(elapsedTime);

        prevTime = currentTime;
    }
}
```

Figure 3. Run method (game.cpp).

1.4 Update and render methods

The update method (Figure 4) processes window and input events, triggers actions based on user input, updates game objects and switches to the next shader every 2 seconds. The render method clears the screen, draws all game objects and then swaps the front and back buffers.

```
void Game::update(int elapsedTime)
{
    m_window.processEvents();
    m_input.processEvents(m_window.events());

    if (m_input.keyJustReleased(SDLK_LEFT)) {
        m_paused = true;
        prevShader();
    }

    if (m_input.keyJustReleased(SDLK_RIGHT)) {
        m_paused = true;
        nextShader();
    }

    for (GameObjectPtr object : m_objects)
        object->update(elapsedTime);
}

void Game::render(int elapsedTime)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    for (GameObjectPtr object : m_objects)
        object->draw(m_lights);

    m_window.swapBuffers();
}
```

Figure 4. Update method (game.cpp).

2 Models

2.1 Model class

The model class (Figure 5) is responsible for loading and managing 3D models from the filesystem.

```
class Model
{
public:
    Model(ProgramPtr shader, string& path, bool flipUVs);
    void draw(mat4& model, mat4& view, mat4& projection, vector<ILightPtr>& lights);
    void shader(ProgramPtr shader);

private:
    static vector<Vertex> extractVertices(
        const attrib_t& attrib,
        const vector<index_t>& attribIds,
        vector<size_t>* indices = nullptr);
    static vector<uvec3> transformIndices(vector<size_t> indices);
    static vector<Material> extractMaterials(
        const vector<material_t>& materials,
        const vector<int>& materialIds,
        const string& baseDir = "",
        bool flipUVs = true);

    vector<shared_ptr<Mesh>> m_meshes;
};
```

Figure 5. Model class (model.hpp).

2.2 Model constructor

The model constructor (Figure 6) loads and processes a 3D model file. It uses TinyObjLoader to load OBJ files, which can contain multiple individual objects, so it processes each object as a separate mesh and stores it in the list of meshes.

```
Model::Model(ProgramPtr shader, const string& path, bool flipUVs)
{
    string dir = path.substr(0, path.find_last_of('/')) + '/';

    attrib_t attrib;
    vector<shape_t> shapes;
    vector<material_t> materials;
    string err;

    bool obj = LoadObj(&attrib, &shapes, &materials, &err, path.c_str(), dir.c_str());
    if (!obj) util::panic("Failed to load model: " + path, err);

    for (const shape_t& shape : shapes)
    {
        vector<size_t> idx;

        vector<Vertex> vertices = extractVertices(attrib, shape.mesh.indices, &idx);
        vector<uvec3> indices = transformIndices(idx);
        vector<Material> mats =
            extractMaterials(materials, shape.mesh.material_ids, dir, flipUVs);

        m_meshes.push_back(make_shared<Mesh>(shader, vertices, indices, mats));
    }
}
```

Figure 6. Model constructor (model.cpp).

3 Shaders

3.1 Colourful shader

This shader uses the vertex normal vector to colourise a mesh based on the direction of each individual vertex. Vertices which face along the +X or-X axis appear red, vertices which face along the +Y or -Y axis appear green and vertices which face along the +Z or -Z axis appear blue. This produces a colourful effect which can be seen in Figure 7.

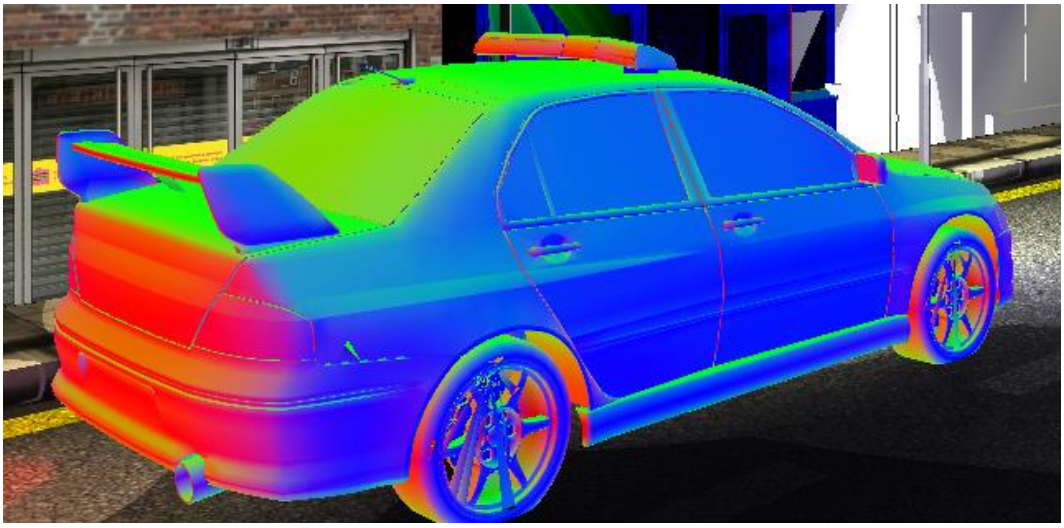

Figure 7. Example of the colourful shader in action.

To achieve this effect, the vertex shader takes the vertex normal vector and applies the abs function to it to get a vector containing the absolute values from the normal vector. When applied to a vector, the abs function is applied to each of the components individually and a new vector is returned, as shown by the equation in Figure 8.

$$abs \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} abs\ x \\ abs\ y \\ abs\ z \end{bmatrix}$$

Figure 8. Equation for applying abs to a vector.

The absolute values are needed because the normal vector can contain negative values but colour values must be positive. These values are then passed on to the fragment shader. Applying the model-view-projection matrix to the vertex position to transform it from model space to clip space is also carried out in the vertex shader. The implementation of the vertex shader can be seen in Figure 9.

```
layout (location = 0) in vec3 v_Position;
layout (location = 1) in vec4 v_Normal;

out vec4 f_Color;

uniform mat4 modelViewProjection;

void main()
{
    gl_Position = modelViewProjection * vec4(v_Position, 1);
    f_Color = abs(v_Normal);
}
```

Figure 9. Colourful vertex shader (colored.vert).

The fragment shader (Figure 10) is a passthrough shader. For efficiency, all calculations are performed on the vertex shader and the result is given to the fragment shader as an input. The fragment shader simply passes on the result from the vertex shader.

```
in vec4 f_Color;
out vec4 p_Color;

void main()
{
    p_Color = f_Color;
}
```

Figure 10. Colourful fragment shader (colored.frag).

3.2 Phong lighting shader

This shader implements Phong lighting, which calculates the contribution of ambient, diffuse and specular lighting in a scene to provide an approximation of realistic lighting, as seen in Figure 11.

Figure 11. Example of Phong lighting shader in action.

The primary function of the vertex shader is to transform the vertex position and normal vectors from model space to view space, which is the coordinate space used in the fragment shader to perform the lighting calculations. In model space, coordinates are relative to the 3D model whereas in view space, coordinates are relative to the position of the camera, as seen in Figure 12. This makes the lighting calculations more efficient and simpler to implement.

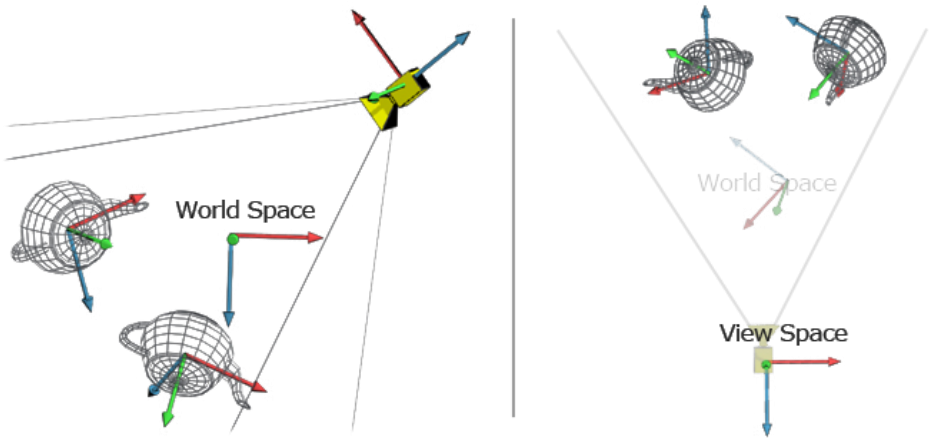

Figure 12. The difference between world space and view space. Source: Marco Alamia.

To transform the position from model space to view space, the model-view matrix is applied to it and the result is passed to the fragment shader. The model-view matrix is the result of combining the model and view matrices by multiplying them together. The model matrix transforms coordinates from model space to world space and the view matrix transforms coordinates from world space to view space, as seen in Figure 13. The combination of the two therefore transforms coordinates from model to view space.

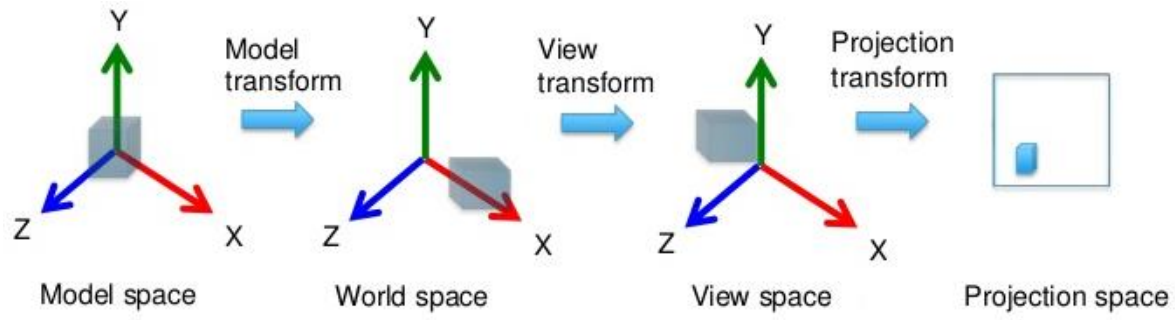

Figure 13. Transformation in 3D space. Source: Takao Wada.

To transform the vertex normal vector from model space to view space is not quite as simple. It is not enough to simply apply the model-view matrix to the normal because normal vectors only represent a direction and not a position. Applying the model-view matrix as-is could perform a non-uniform scale as seen in Figure 14.

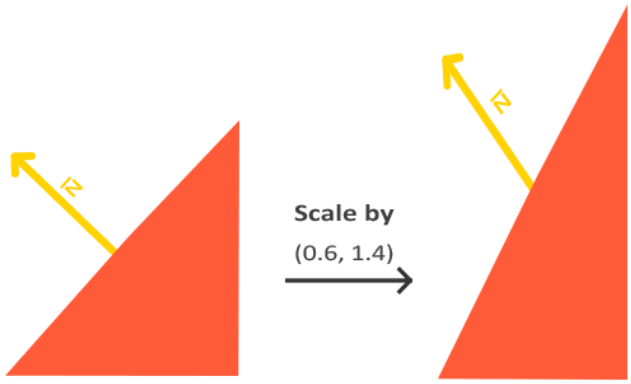

Figure 14. Effect of a non-uniform scale on a normal vector. Source: Joey de Vries.

Instead, a special matrix called the normal matrix is needed. The normal matrix is defined as “the transpose of the inverse of the upper-left corner of the model-view matrix”. Calculating this matrix is a computationally expensive operation so it is only performed once per mesh on the CPU in our application code, as seen in Figure 15.

```
void Mesh::draw(mat4& model, mat4& view, mat4& projection, vector<ILightPtr>& lights)
{
    //...

    mat4 normal = transpose(inverse(modelView));

    m_shader->uniform("normal", normal);

    //...
}
```

Figure 15. Calculating the normal matrix (mesh.cpp).

The normal matrix can be accessed from the vertex shader (Figure 16) as a uniform variable. The normal matrix is applied to the normal vector and the result is passed to the fragment shader.

```
layout (location = 0) in vec3 v_Position;
layout (location = 1) in vec3 v_Normal;

out vec3 f_Position;
out vec3 f_Normal;

uniform mat4 modelViewProjection;
uniform mat4 modelView;
uniform mat4 normal;

void main()
{
    gl_Position = modelViewProjection * vec4(v_Position, 1);
    f_Position = vec3(modelView * vec4(v_Position, 1));
    f_Normal = vec3(normal * vec4(v_Normal, 1));
}
```

Figure 16. Phong vertex shader (lit.vert).

The Phong lighting calculations are performed in the fragment shader, which defines several data structures to manage the values required to perform the lighting calculations. The material data structure (Figure 17) contains a shininess value which determines the level of specular highlight.

```
struct Material {
    float shininess;
};
```

Figure 17. Material structure (lit.frag).

The directional light data structure (Figure 18) contains values which represent a simple directional light. The direction represents the direction that the light is pointing. The ambient, diffuse and specular values control the colour and intensity of the ambient, diffuse and specular component contributions to the final output colour.

```
struct DirectionalLight {
    vec3 direction;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
```

Figure 18. Directional light structure (lit.frag).

The point light data structure (Figure 19) contains values which represent an omni-directional point light. The position vector is the position of the light source in world space. The linear, quadratic and constant values are used to calculate the attenuation of the light. Attenuation is the way that the intensity of the light reduces over distance.

```
struct PointLight {
    vec3 position;
    float linear;
    float quadratic;
    float constant;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
```

Figure 19. Point light structure (lit.frag).

The spot light data structure (Figure 20) contains values which represent a directional point light or spotlight. The cut-off and outer cut-off values determine the size of the inner and outer cones of light respectively. The cut-off is the angle at which the light no longer illuminates the surface. This provides a hard edge which is not realistic, so the outer cut-off is a larger angle which is used to interpolate between the illuminated inner cone and unilluminated outer cone, providing a soft edge to the light.

```
struct SpotLight {
    vec3 position;
    vec3 direction;
    float cutOff;
    float outerCutOff;
    float constant;
    float linear;
    float quadratic;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};
```

Figure 20. Spot light structure (lit.frag).

The fragment shader takes the view-space vertex position and normal from the vertex shader and outputs a colour value. It requires access to the view and normal matrices and the material of the mesh. It applies a global ambient light before performing the Phong lighting calculations, which can support multiple directional, point and spot lights. These inputs, outputs and uniforms can be seen in Figure 21.

```
in vec3 f_Position;
in vec3 f_Normal;

out vec4 p_Color;

uniform mat4 view;
uniform mat4 normal;
uniform Material material;

uniform vec3 ambientLight;
uniform DirectionalLight directionalLights[MAX_DIR_LIGHTS];
uniform PointLight pointLights[MAX_POINT_LIGHTS];
uniform SpotLight spotLights[MAX_SPOT_LIGHTS];

uniform int numDirectionalLights;
uniform int numPointLights;
uniform int numSpotLights;
```

Figure 21. Inputs, outputs and uniforms (lit.frag).

The implementations of the directional, point and spot light calculations are contained in separate functions to keep the body of the main function short and readable. Details of their implementations will follow.

The main method of the fragment shader stores the normalised view-space normal vector as well as the direction from the vertex position to the camera position. Because we are using view space, the vertex position vector points from the camera to the vertex, so we can simply take its negative to get the direction from the vertex position to the camera position.

To combine the results of the various lighting calculations, the fragment shader starts by assigning the global ambient light value to an initial result. It then iterates through all directional, point and spot

lights and adds the output of each directional, point or spot light calculation to the result, which is then output by the fragment shader. The main method implementation can be seen in Figure 22.

```
void main()
{
    vec3 norm = normalize(f_Normal);
    vec3 viewDir = normalize(-f_Position);

    vec3 result = ambientLight;

    for (int i = 0; i < numDirectionalLights; i++)
        result += calculateDirectionalLight(directionalLights[i], norm, viewDir);

    for (int i = 0; i < numPointLights; i++)
        result += calculatePointLight(pointLights[i], norm, viewDir);

    for (int i = 0; i < numSpotLights; i++)
        result += calculateSpotLight(spotLights[i], norm, viewDir);

    p_Color = vec4(result, 1);
}
```

Figure 22. Main method (lit.frag).

When calculating the diffuse contribution of a directional light, the direction of the light needs to be transformed into view space by applying the normal matrix to it. The direction of the perfect reflection of the light from the vertex normal is needed when calculating the specular contribution.

The strength of the diffuse light depends on the angle (dot product) between the surface normal and the light direction, as shown in Figure 23. The strength of the specular light depends on the angle (dot product) between the view direction and perfect reflector, as shown in Figure 24. It is highly concentrated along the perfect reflector, which is achieved by raising it to the power of the shininess property. The implementation of the directional light calculation can be seen in Figure 25.

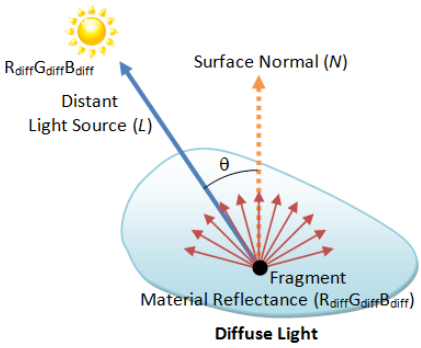

Figure 23. Diffuse light illustration. Source: GCU.

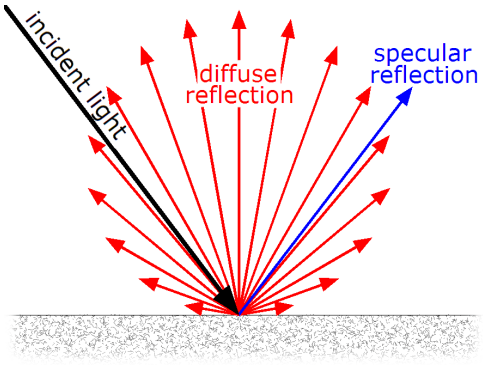

Figure 24. Specular light illustration. Source: GCU.

```
vec3 calculateDirectionalLight(DirectionalLight light, vec3 surfNormal, vec3 viewDir)
{
    vec3 lightDir = normalize(-vec3(normal * vec4(light.direction, 1)));
    vec3 reflectDir = reflect(-lightDir, surfNormal);

    vec3 ambient = light.ambient;
    vec3 diffuse = light.diffuse * max(dot(surfNormal, lightDir), 0);
    vec3 specular =
        light.specular * pow(max(dot(viewDir, reflectDir), 0), material.shininess);

    return (ambient + diffuse + specular);
}
```

Figure 25. Calculate directional light method (lit.frag).

The point light calculation differs from the directional light calculation in that the ambient, diffuse and specular light is multiplied by an attenuation value which causes the intensity of the light to drop off as the distance from the light increases. The equation to calculate the attenuation can be seen in Figure 26, where K_c is the constant factor, K_l is the linear factor, K_q is the quadratic factor and d is the distance.

$$F_{att} = \frac{1}{K_c + K_l * d + K_q * d^2}$$

Figure 26. Equation for attenuation of a point light.

The distance is the length of the difference between the vertex surface position and the light position in view space. The light position is transformed from world space to view space by multiplying it with the view matrix. The light angle can then be obtained by subtracting the surface position from the light position. The rest of the code is the same as the directional light calculation. The implementation of the point light calculation can be seen in Figure 27.

```
vec3 calculatePointLight(PointLight light, vec3 surfNormal, vec3 viewDir)
{
    vec3 lightPos = vec3(view * vec4(light.position, 1));
    vec3 lightAngle = normalize(lightPos - f_Position);
    vec3 reflectDir = reflect(-lightAngle, surfNormal);

    float dist = length(lightPos - f_Position);
    float attenuation =
        1 / (light.constant + light.linear * dist + light.quadratic * (dist * dist));

    vec3 ambient = light.ambient * attenuation;
    vec3 diffuse = light.diffuse * max(dot(surfNormal, lightAngle), 0) * attenuation;
    vec3 specular = light.specular *
        pow(max(dot(viewDir, reflectDir), 0), material.shininess) * attenuation;

    return (ambient + diffuse + specular);
}
```

Figure 27. Calculate point light method (lit.frag).

The spot light calculation differs from the point light calculation in that the ambient, diffuse and specular light is multiplied by an intensity value which constrains the light to a circular cone with a soft edge. The equation to calculate the intensity value is shown in Figure 28, where theta θ is the angle (dot product) between the light direction and light angle, γ is the outer cut-off angle and ϵ is the difference between the inner and outer cut-off angles.

$$I = \frac{\theta - \gamma}{\epsilon}$$

Figure 28. Equation for intensity of a spot light.

This value is then clamped to the 0.0 - 1.0 range. The rest of the code is the same as the point light calculation. The implementation of the spot light calculation can be seen in Figure 29.

```
vec3 calculateSpotLight(SpotLight light, vec3 surfNormal, vec3 viewDir)
{
    vec3 lightDir = normalize(-vec3(normal * vec4(light.direction, 1)));
    vec3 lightPos = vec3(view * vec4(light.position, 1));
    vec3 lightAngle = normalize(lightPos - f_Position);
    vec3 reflectDir = reflect(-lightAngle, surfNormal);

    float dist = length(lightPos - f_Position);
    float attenuation =
        1 / (light.constant + light.linear * dist + light.quadratic * (dist * dist));

    float theta = dot(lightAngle, lightDir);
    float epsilon = light.cutOff - light.outerCutOff;
    float intensity = clamp((theta - light.outerCutOff) / epsilon, 0, 1);

    vec3 ambient = light.ambient * attenuation * intensity;
    vec3 diffuse =
        light.diffuse * max(dot(surfNormal, lightAngle), 0) * attenuation * intensity;
    vec3 specular =
        light.specular *
        pow(max(dot(viewDir, reflectDir), 0), material.shininess) *
        attenuation * intensity;

    return (ambient + diffuse + specular);
}
```

Figure 29. Calculate spot light method (lit.frag).

3.3 Toon lighting shader

This shader extends the result of the lighting shader by applying toon shading, which uses a lookup table which assigns solid colours to ranges of values to provide a non-photorealistic effect, as shown in Figure 30.

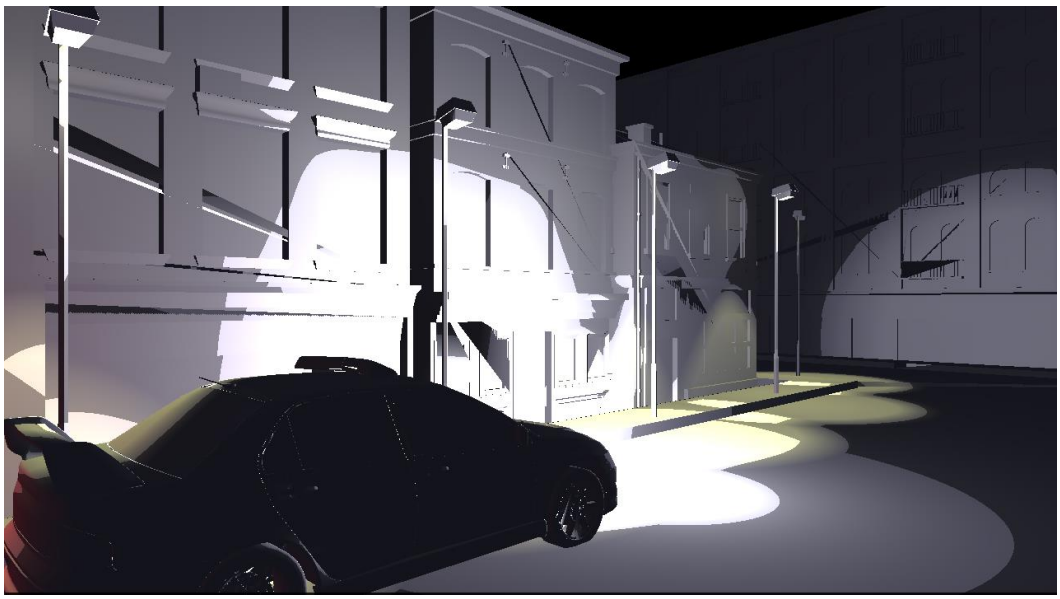

Figure 30. Example of toon lighting shader in action.

As the calculations in the fragment shader are also performed in view space, the vertex shader for this effect is identical to the Phong lighting vertex shader above (Figure 16, pp 9). It simply transforms the vertex position and normal vectors from model space to view space and passes them on to the fragment shader.

The fragment shader requires the same inputs, outputs and uniforms as the Phong lighting fragment shader above (Figure 21, pp 10) and the methods to calculate the directional, point and spot light contribution remain the same as well.

To achieve the toon lighting effect, the fragment shader averages the results of the lighting calculation to determine the overall intensity of the light. It then checks which range of values the intensity falls in and multiplies the original result by a constant factor associated with that range of values to get the final output. The final implementation of the main method of the fragment shader can be seen in Figure 31.

```
void main()
{
    vec3 norm = normalize(f_Normal);
    vec3 viewDir = normalize(-f_Position);

    vec3 result = ambientLight;

    for (int i = 0; i < numDirectionalLights; i++)
        result += calculateDirectionalLight(directionalLights[i], norm, viewDir);

    for (int i = 0; i < numPointLights; i++)
        result += calculatePointLight(pointLights[i], norm, viewDir);

    for (int i = 0; i < numSpotLights; i++)
        result += calculateSpotLight(spotLights[i], norm, viewDir);

    float intensity = (result.x + result.y + result.z) / 3;
    if      (intensity > 0.95) p_Color = vec4(1.0, 1.0, 1.0, 1) * vec4(result, 1);
    else if (intensity > 0.5)  p_Color = vec4(0.6, 0.6, 0.6, 1) * vec4(result, 1);
    else if (intensity > 0.25) p_Color = vec4(0.3, 0.3, 0.3, 1) * vec4(result, 1);
    else                       p_Color = vec4(0.2, 0.2, 0.2, 1) * vec4(result, 1);
}
```

Figure 31. Main method (toon.frag).

4 Camera

4.1 Camera class

The camera class (Figure 32) manages the view matrix. It contains the three vectors used to generate the view matrix: the position vector, the target vector and the up vector.

```
class Camera
{
public:
    Camera(vec3& position, vec3& target = {0,0,0}, vec3& up = {0,1,0})
        : m_position(position), m_target(target), m_up(up)
    {
    }

    mat4 view() const
    {
        return lookAt(m_position, m_target, m_up);
    }

private:
    vec3 m_position;
    vec3 m_target;
    vec3 m_up;
};
```

Figure 32. Camera class (camera.inl).

5 Shader class

5.1 Shader class

The shader class (Figure 33) is a lightweight RAII wrapper around an OpenGL shader. The constructor creates an OpenGL shader of the desired type and the destructor deletes it. Passthrough functions are implemented for sending source and compiling. A secondary constructor allows the user to also provide some source code to be uploaded to the shader and compiled.

```
class Shader
{
public:
    Shader(GLenum type) { m_shader = glCreateShader(type); }
    ~Shader() { glDeleteShader(m_shader); }
    Shader(GLenum type, const string& code) : Shader(type) {
        source(code);
        compile();
    }
    void source(const string& source) {
        const GLchar* data = source.c_str();
        glShaderSource(m_shader, 1, &data, nullptr);
    }
    void compile() { glCompileShader(m_shader); }
private:
    GLuint m_shader;
};
```

Figure 33. Shader class (shader.inl).

5.2 Program class

The program class (Figure 34) is a lightweight RAII wrapper around an OpenGL shader program. The constructor creates an OpenGL shader program and the destructor deletes it. Passthrough functions are implemented for attaching/detaching shaders, linking, validating and binding/unbinding. A secondary constructor allows the user to provide a base path to a set of shaders to be loaded and attached to the program before linking and validating the program and finally detaching the shaders.

```
class Program
{
public:
    Program() { m_program = glCreateProgram(); }
    ~Program() { glDeleteShader(m_program); }
    Program(const string& basePath) : Program() {
        Shader vertexShader(GL_VERTEX_SHADER, util::readFile(basePath + ".vert"));
        Shader fragmentShader(GL_FRAGMENT_SHADER, util::readFile(basePath + ".frag"));
        attach(vertexShader, fragmentShader);
        link();
        validate();
        detach(vertexShader, fragmentShader);
    }
    void attach(const Shader& shader) { glAttachShader(m_program, shader.handle()); }
    void detach(const Shader& shader) { glDetachShader(m_program, shader.handle()); }
    void link() { glLinkProgram(m_program); }
    void validate() { glValidateProgram(m_program); }
    void bind() { glUseProgram(m_program); }
    void unbind() { glUseProgram(0); }
private:
    GLuint m_program;
};
```

Figure 34. Program class (program.inl).

6 Shader implementation methods

6.1 Uniform methods

The uniform methods (Figure 35) provide a way of passing values to shaders. They take as a parameter the name of the uniform to update and the value to update it to. They query OpenGL with the uniform name to get the uniform location, which they then use to tell OpenGL to set the value at that location.

```
class Program
{
public:
    void uniform(const string& name, int value) {
        glUniform1i(glGetUniformLocation(m_program, name.c_str()), value);
    }
    void uniform(const string& name, float value) {
        glUniform1f(glGetUniformLocation(m_program, name.c_str()), value);
    }
    void uniform(const string& name, vec3 v) {
        glUniform3fv(glGetUniformLocation(m_program, name.c_str()), 1, value_ptr(v));
    }
    void uniform(const string& name, mat4 value, bool transpose = false) {
        GLuint location = glGetUniformLocation(m_program, name.c_str());
        glUniformMatrix4fv(location, 1, transpose, value_ptr(value));
    }
private:
    GLuint m_program;
};
```

Figure 35. Uniform methods (program.inl).

6.2 Vertex attribute methods

The vertex attribute methods (Figure 36) provide a way of passing vertex data to shaders. They take as a parameter either the location or the name (used to get the location) of the vertex attribute, the number of components per value (either 1, 2, 3 or 4), the data type of the values (GL_FLOAT et. al.), whether to normalise the values, the offset between consecutive vertex attributes and the offset of the first component of the first vertex attribute in the currently bound buffer.

```
class Program
{
public:
    void vertexAttribPointer(GLuint location, GLint size, GLenum type,
        GLboolean normalised, GLsizei stride, const void* pointer)
    {
        glVertexAttribPointer(location, size, type, normalised, stride, pointer);
        glEnableVertexAttribArray(location);
    }
    void vertexAttribPointer(const string& name, GLint size, GLenum type,
        GLboolean normalised, GLsizei stride, const void* pointer)
    {
        GLuint location = glGetAttribLocation(m_program, name.c_str());
        vertexAttribPointer(location, size, type, normalised, stride, pointer);
    }
private:
    GLuint m_program;
};
```

Figure 36. Vertex attribute methods (program.inl).