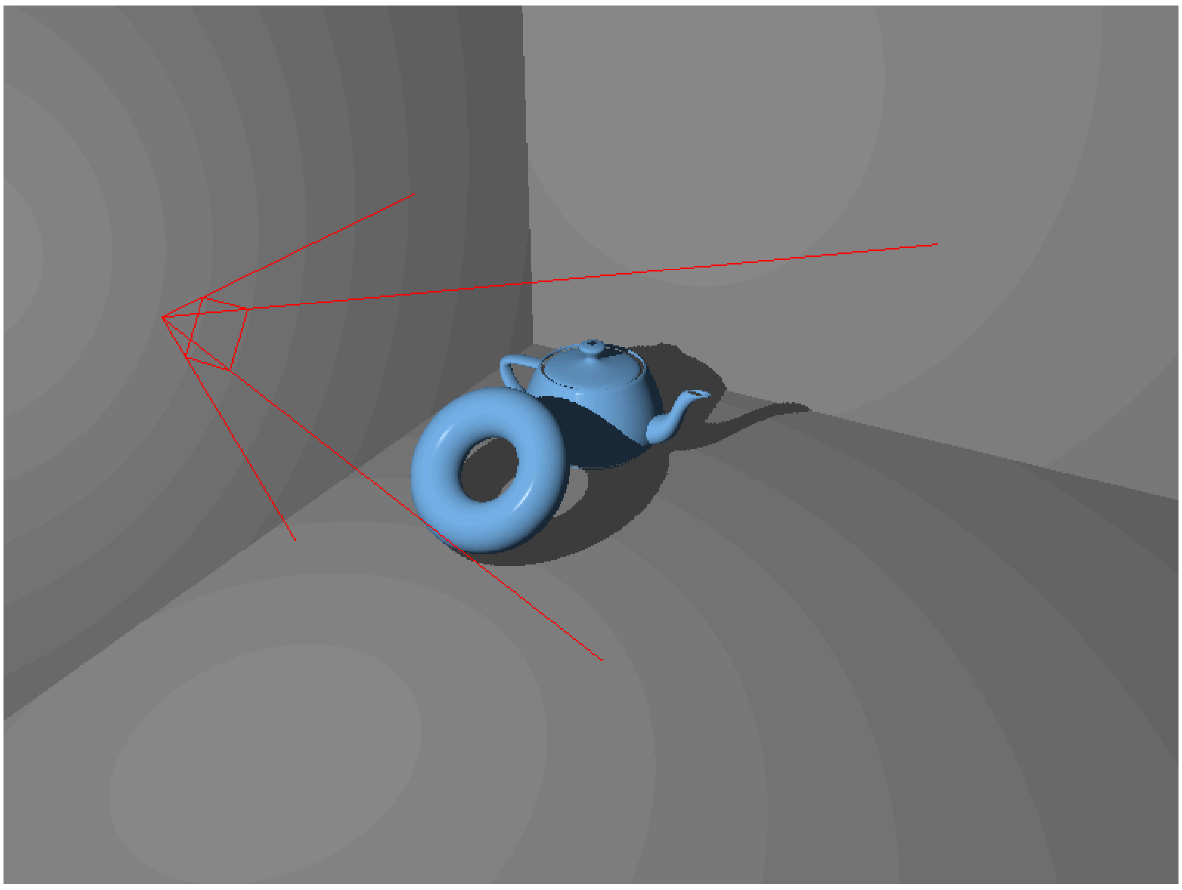
Shadows

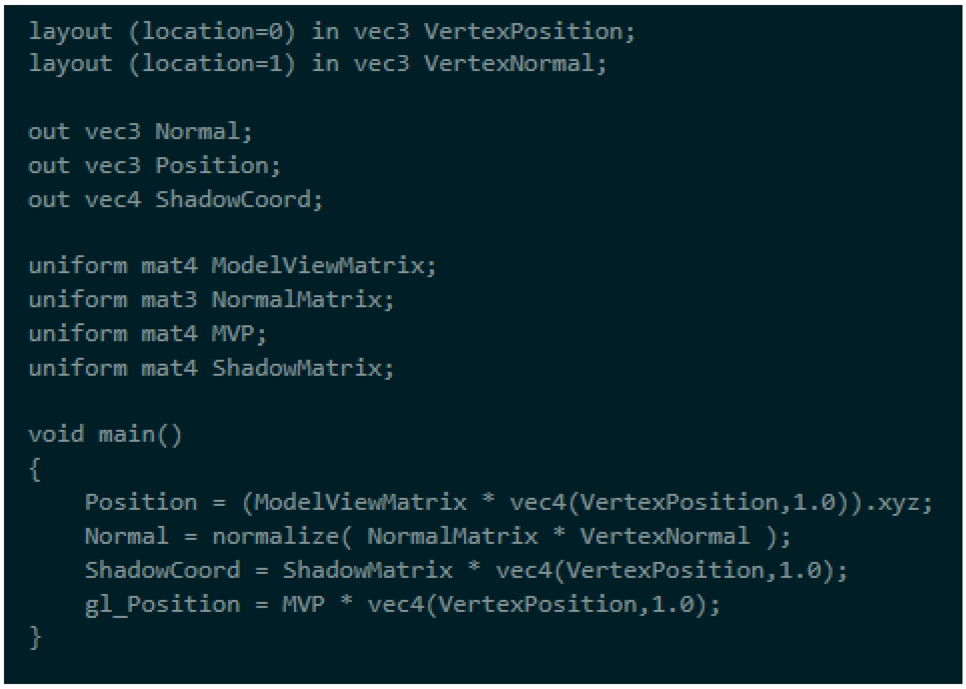
COMP3015 Lab 8

Shadow Maps

Result



Vertex Shader



```
layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexNormal;

out vec3 Normal;
out vec3 Position;
out vec4 ShadowCoord;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 MVP;
uniform mat4 ShadowMatrix;

void main()
{
    Position = (ModelViewMatrix * vec4(VertexPosition,1.0)).xyz;
    Normal = normalize( NormalMatrix * VertexNormal );
    ShadowCoord = ShadowMatrix * vec4(VertexPosition,1.0);
    gl_Position = MVP * vec4(VertexPosition,1.0);
}
```

Fragment Shader

Overview

Each fragment is shaded based on the result of a depth test that makes use of a shadow map. The fragment positions are first converted into the coordinate system of the light source & projected using the light source’s projection matrix. The result is intentionally biased in order to acquire valid texture coordinates & tested a second time against the shadow map.

Globals

Create a texture named ‘shadow map’ (or simply ‘depth map’) for storing the depth buffer’s information. The shadow map roughly constitutes a set of distances from the given light source to various surface locations.

```
uniform struct LightInfo {
    vec4 Position;
    vec3 Intensity;
} Light;

uniform struct MaterialInfo {
    vec3 Ka;
    vec3 Kd;
    vec3 Ks;
    float Shininess;
} Material;

uniform sampler2DShadow ShadowMap;

in vec3 Position;
in vec3 Normal;
in vec4 ShadowCoord;

layout (location = 0) out vec4 FragColor;
```

Diffuse & Specular Phong Model

```
vec3 phongModelDiffAndSpec()
{
    vec3 n = Normal;
    vec3 s = normalize(vec3(Light.Position) - Position);
    vec3 v = normalize(-Position.xyz);
    vec3 r = reflect( -s, n );
    float sDotN = max( dot(s,n), 0.0 );
    vec3 diffuse = Light.Intensity * Material.Kd * sDotN;
    vec3 spec = vec3(0.0);
    if( sDotN > 0.0 )
        spec = Light.Intensity * Material.Ks *
            pow( max( dot(r,v), 0.0 ), Material.Shininess );

    return diffuse + spec;
}
```

Subroutine Declarations

```
subroutine void RenderPassType();
subroutine uniform RenderPassType RenderPass;
```

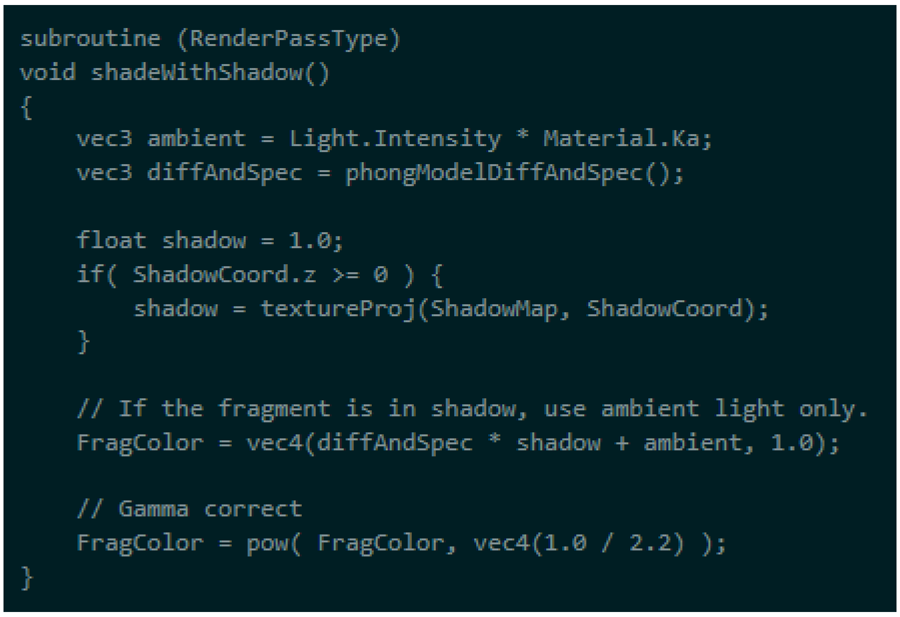
Shading with Shadow

Depth Test

If the fragment's depth is greater than the depth stored in the shadow map, this determines that a surface is present between the fragment & the light source. The result is that the fragment is shadowed & is consequently shaded using ambient lighting only. In the case where the fragment is not shadowed, it is shaded normally.

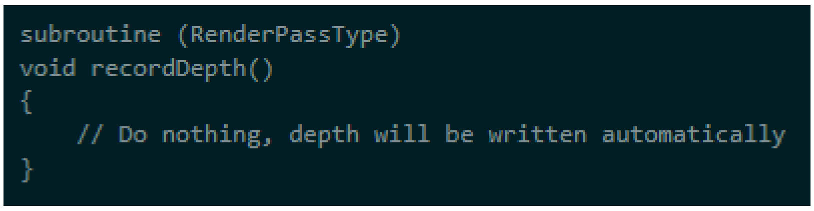
Coordinates

The x & y components of the position in clip coordinates are roughly what are required to access the shadow map. The z coordinate contains the depth information necessary for comparisons against the shadow map. However, these values cannot be utilised for these purposes immediately. First, they must be biased so that they range from 0 to 1, instead of between -1 to 1. Secondly, perspective division must be applied.



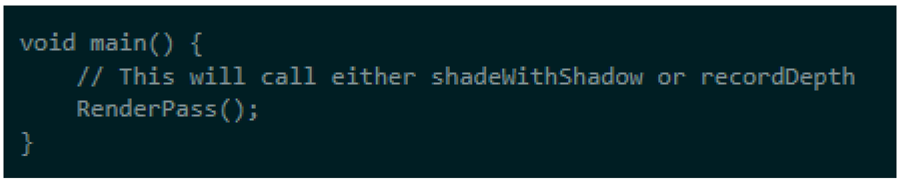
```
subroutine (RenderPassType)
void shadeWithShadow()
{
    vec3 ambient = Light.Intensity * Material.Ka;
    vec3 diffAndSpec = phongModelDiffAndSpec();

    float shadow = 1.0;
    if( ShadowCoord.z >= 0 ) {
        shadow = textureProj(ShadowMap, ShadowCoord);
    }

    // If the fragment is in shadow, use ambient light only.
    FragColor = vec4(diffAndSpec * shadow + ambient, 1.0);

    // Gamma correct
    FragColor = pow( FragColor, vec4(1.0 / 2.2) );
}
```

Depth Recording



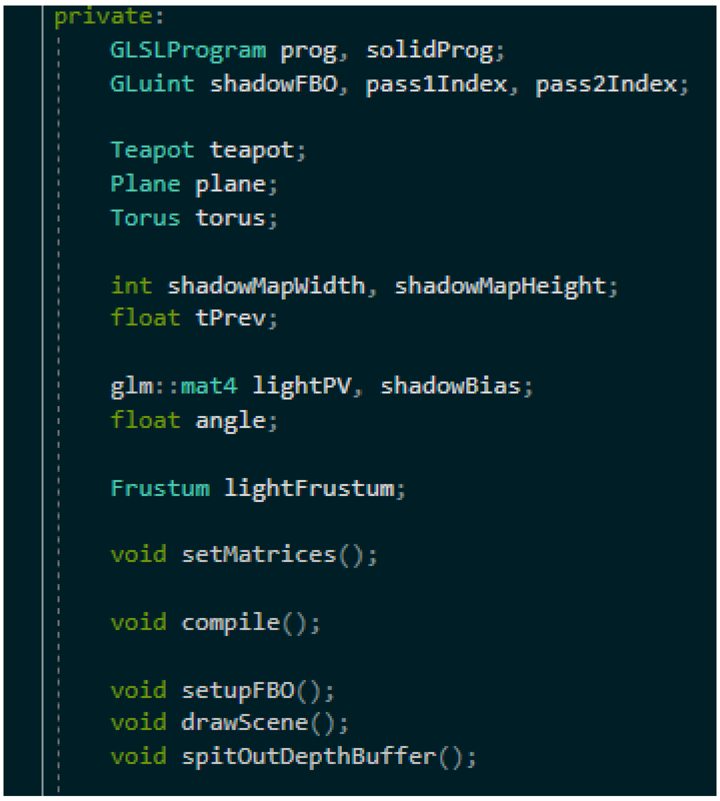
```
subroutine (RenderPassType)
void recordDepth()
{
    // Do nothing, depth will be written automatically
}
```

Main Function



```
void main() {
    // This will call either shadeWithShadow or recordDepth
    RenderPass();
}
```

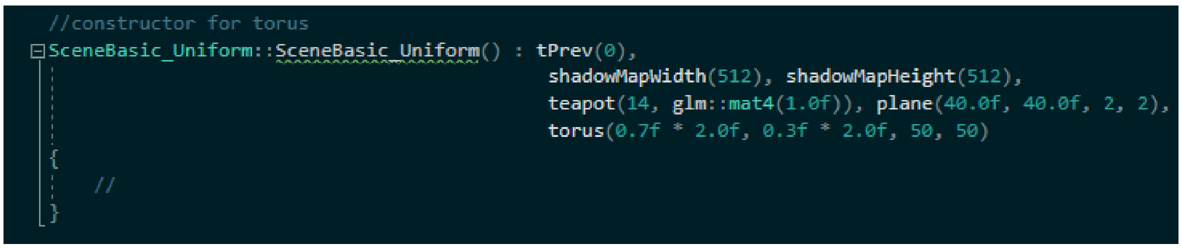
Scenebasic Uniform Header



```
private:
    GLSLProgram prog, solidProg;
    GLuint shadowFBO, pass1Index, pass2Index;

    Teapot teapot;
    Plane plane;
    Torus torus;

    int shadowMapWidth, shadowMapHeight;
    float tPrev;

    glm::mat4 lightPV, shadowBias;
    float angle;

    Frustum lightFrustum;

    void setMatrices();

    void compile();

    void setupFBO();
    void drawScene();
    void spitOutDepthBuffer();
```

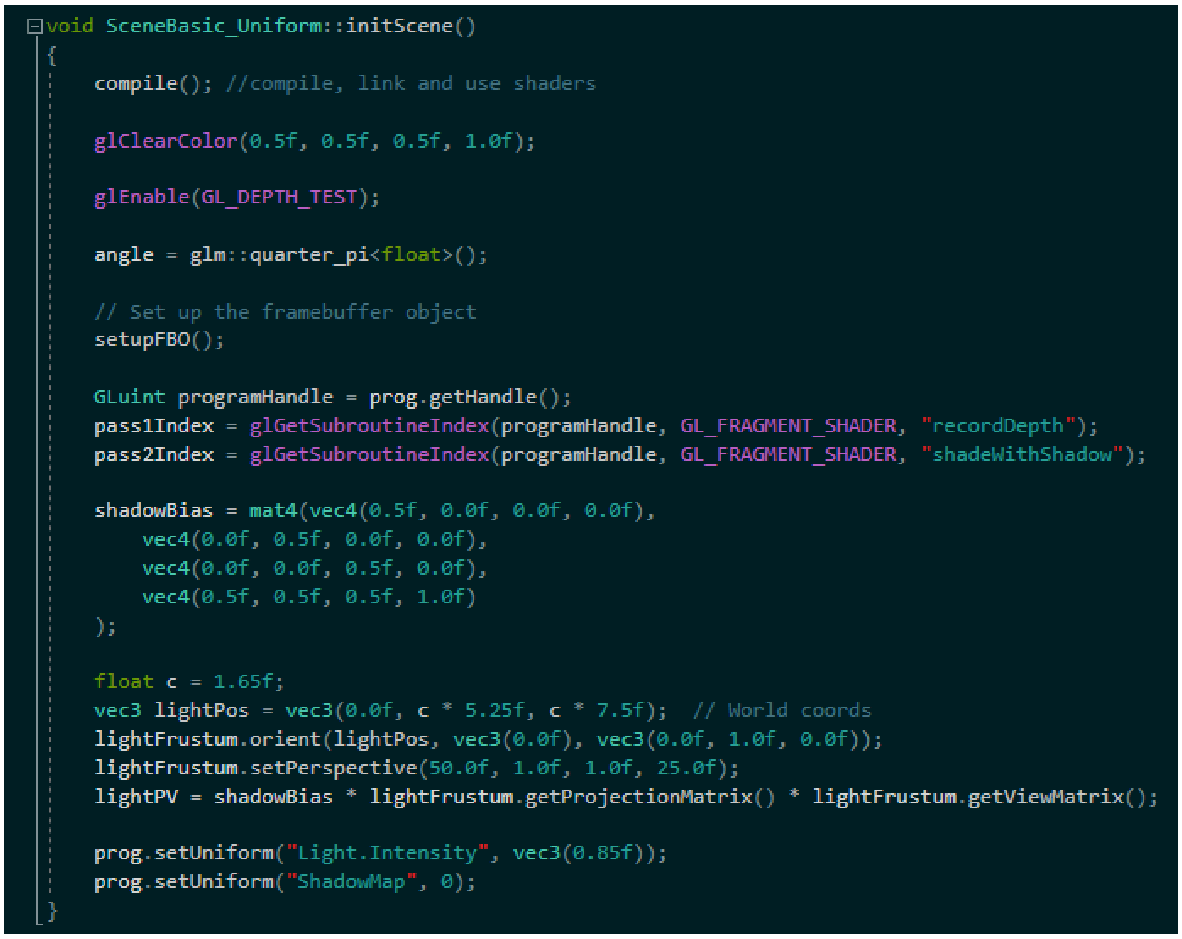
Scenebasic Uniform CPP

Constructor



```
//constructor for torus
SceneBasic_Uniform::SceneBasic_Uniform() : tPrev(0),
                                            shadowMapWidth(512), shadowMapHeight(512),
                                            teapot(14, glm::mat4(1.0f)), plane(40.0f, 40.0f, 2, 2),
                                            torus(0.7f * 2.0f, 0.3f * 2.0f, 50, 50)
{
    //
}
```

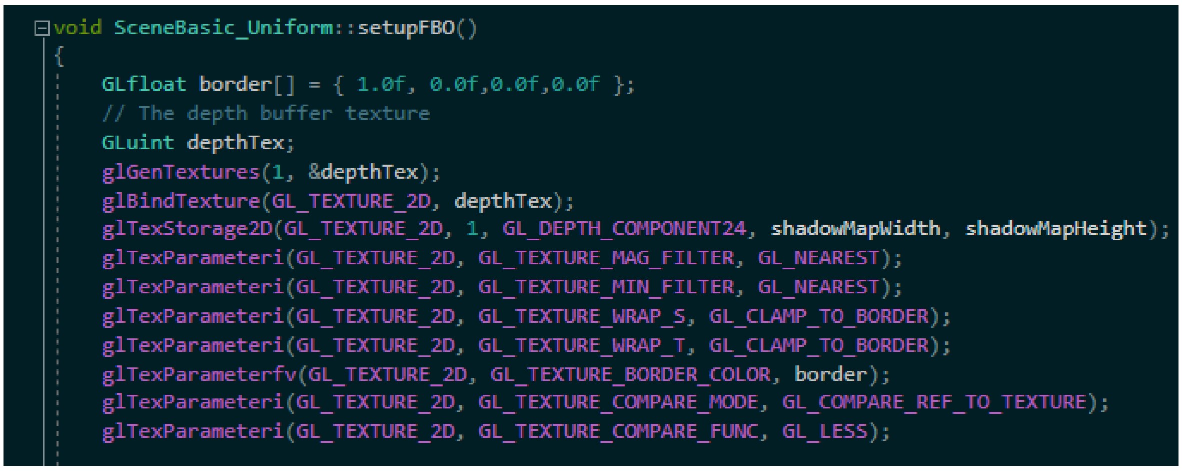
Scene Initialisation



```
void SceneBasic_Uniform::initScene()
{
    compile(); //compile, link and use shaders

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

    glEnable(GL_DEPTH_TEST);

    angle = glm::quarter_pi<float>();

    // Set up the framebuffer object
    setupFBO();

    GLuint programHandle = prog.getHandle();
    pass1Index = glGetSubroutineIndex(programHandle, GL_FRAGMENT_SHADER, "recordDepth");
    pass2Index = glGetSubroutineIndex(programHandle, GL_FRAGMENT_SHADER, "shadeWithShadow");

    shadowBias = mat4(vec4(0.5f, 0.0f, 0.0f, 0.0f),
        vec4(0.0f, 0.5f, 0.0f, 0.0f),
        vec4(0.0f, 0.0f, 0.5f, 0.0f),
        vec4(0.5f, 0.5f, 0.5f, 1.0f)
    );

    float c = 1.65f;
    vec3 lightPos = vec3(0.0f, c * 5.25f, c * 7.5f);  // World coords
    lightFrustum.orient(lightPos, vec3(0.0f), vec3(0.0f, 1.0f, 0.0f));
    lightFrustum.setPerspective(50.0f, 1.0f, 1.0f, 25.0f);
    lightPV = shadowBias * lightFrustum.getProjectionMatrix() * lightFrustum.getViewMatrix();

    prog.setUniform("Light.Intensity", vec3(0.85f));
    prog.setUniform("ShadowMap", 0);
}
```

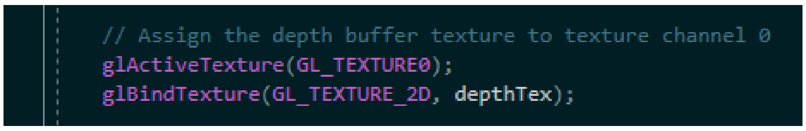
FBO Setup

Depth Buffer Texture



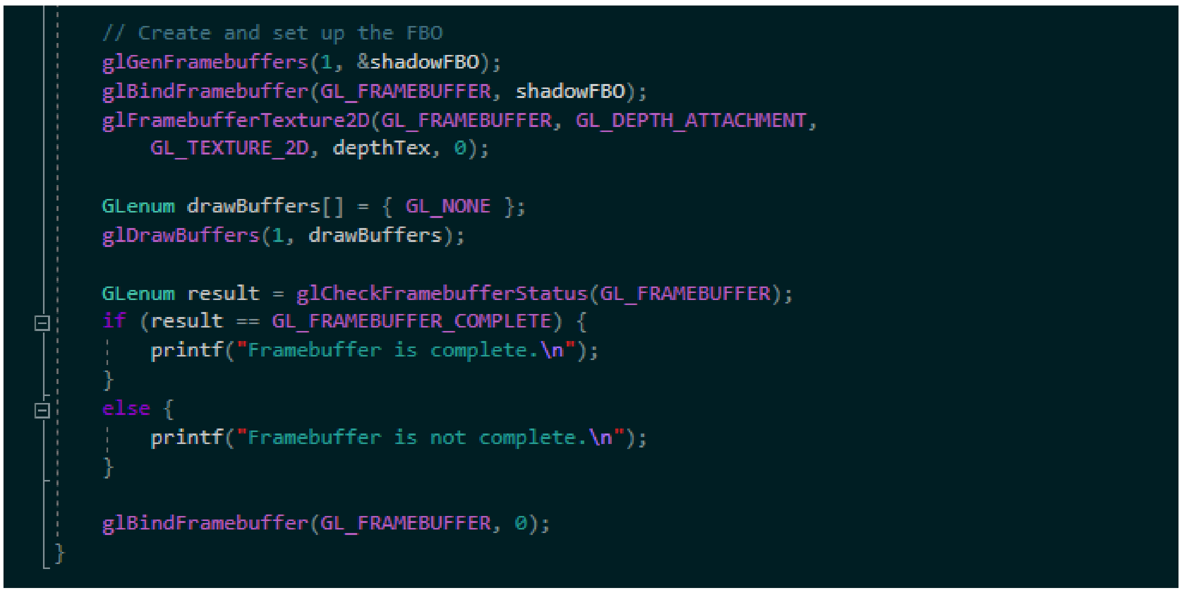
```
void SceneBasic_Uniform::setupFBO()
{
    GLfloat border[] = { 1.0f, 0.0f,0.0f,0.0f };
    // The depth buffer texture
    GLuint depthTex;
    glGenTextures(1, &depthTex);
    glBindTexture(GL_TEXTURE_2D, depthTex);
    glTexStorage2D(GL_TEXTURE_2D, 1, GL_DEPTH_COMPONENT24, shadowMapWidth, shadowMapHeight);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
    glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, border);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE, GL_COMPARE_REF_TO_TEXTURE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LESS);
```

Depth Buffer Assignment to Texture Channel



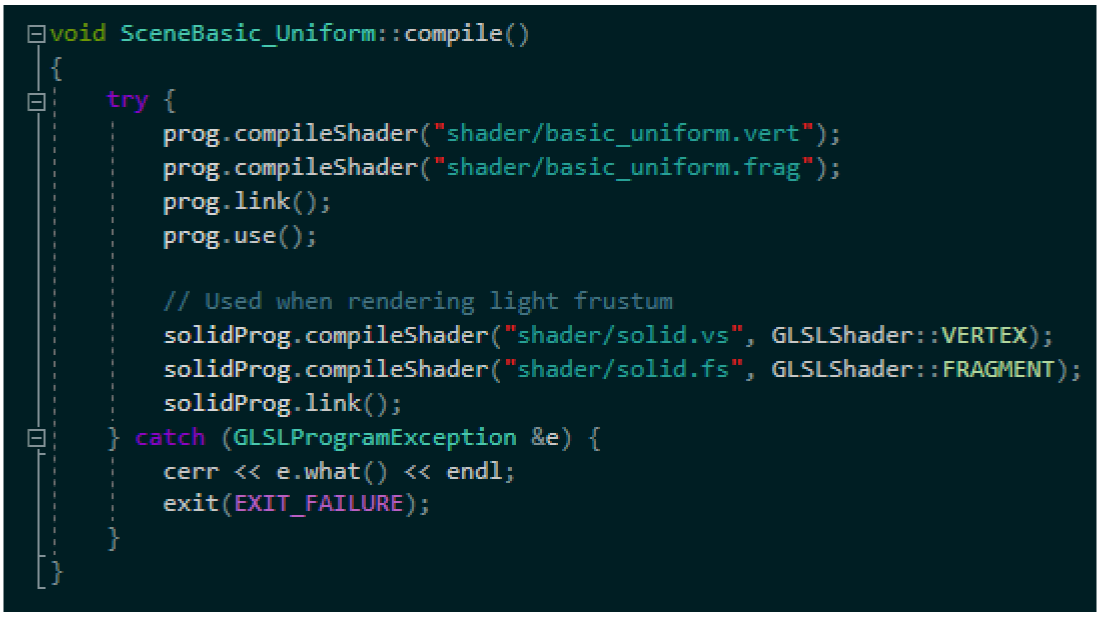
```
// Assign the depth buffer texture to texture channel 0
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, depthTex);
```

FBO Initialisation



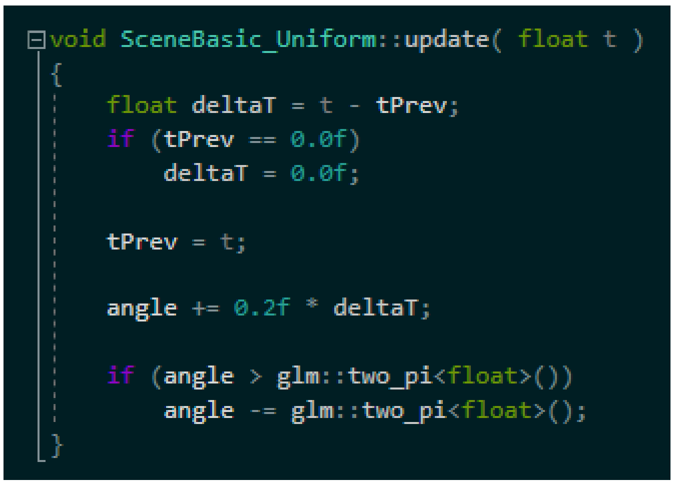
```
    // Create and set up the FBO
    glGenFramebuffers(1, &shadowFBO);
    glBindFramebuffer(GL_FRAMEBUFFER, shadowFBO);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
        GL_TEXTURE_2D, depthTex, 0);

    GLenum drawBuffers[] = { GL_NONE };
    glDrawBuffers(1, drawBuffers);

    GLenum result = glCheckFramebufferStatus(GL_FRAMEBUFFER);
    if (result == GL_FRAMEBUFFER_COMPLETE) {
        printf("Framebuffer is complete.\n");
    }
    else {
        printf("Framebuffer is not complete.\n");
    }

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Shader Compilation



```
void SceneBasic_Uniform::compile()
{
    try {
        prog.compileShader("shader/basic_uniform.vert");
        prog.compileShader("shader/basic_uniform.frag");
        prog.link();
        prog.use();

        // Used when rendering light frustum
        solidProg.compileShader("shader/solid.vs", GLSLShader::VERTEX);
        solidProg.compileShader("shader/solid.fs", GLSLShader::FRAGMENT);
        solidProg.link();
    } catch (GLSLProgramException &e) {
        cerr << e.what() << endl;
        exit(EXIT_FAILURE);
    }
}
```

Updating



```
void SceneBasic_Uniform::update( float t )
{
    float deltaT = t - tPrev;
    if (tPrev == 0.0f)
        deltaT = 0.0f;

    tPrev = t;

    angle += 0.2f * deltaT;

    if (angle > glm::two_pi<float>())
        angle -= glm::two_pi<float>();
}
```

Rendering

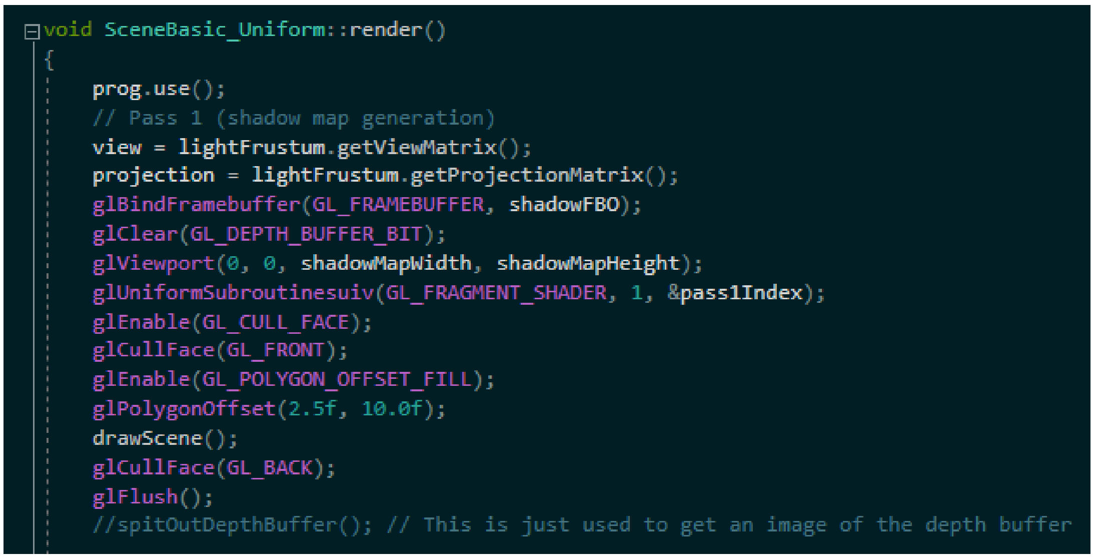
Shadow Map Generation Pass (1)

View & Projection Matrices

Setup the view matrix to be rendering the scene as if the camera is located at the position of the light source & is oriented towards the shadow-casting objects. Additionally, setup the projection matrix so the view frustum encloses all objects that may cast shadows & the area where the shadows will appear.

Shadow FBO, Buffer & Recording, & Culling & Drawing

Bind to the framebuffer named ‘shadowFBO’ that contains the shadow map. Clear the depth buffer, select the subroutine recordDepth() function, enable front-face culling & lastly draw the scene.



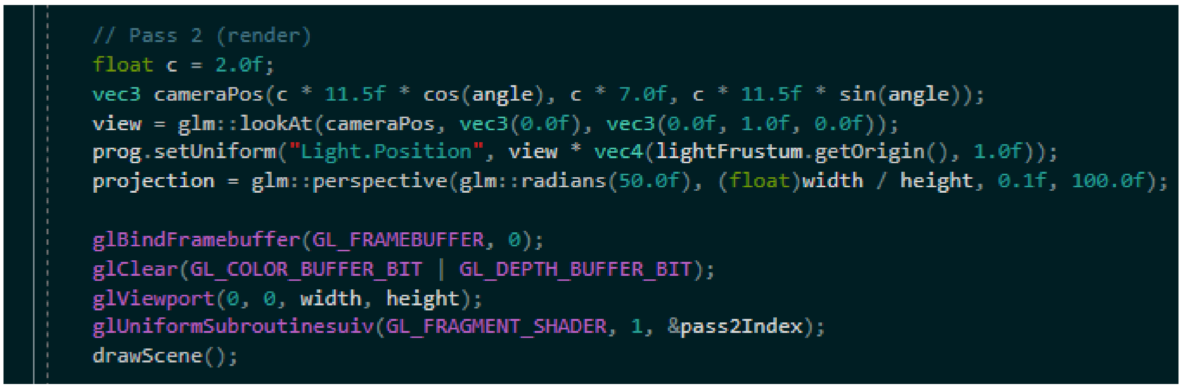
```
void SceneBasic_Uniform::render()
{
    prog.use();
    // Pass 1 (shadow map generation)
    view = lightFrustum.getViewMatrix();
    projection = lightFrustum.getProjectionMatrix();
    glBindFramebuffer(GL_FRAMEBUFFER, shadowFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    glViewport(0, 0, shadowMapWidth, shadowMapHeight);
    glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, &pass1Index);
    glEnable(GL_CULL_FACE);
    glCullFace(GL_FRONT);
    glEnable(GL_POLYGON_OFFSET_FILL);
    glPolygonOffset(2.5f, 10.0f);
    drawScene();
    glCullFace(GL_BACK);
    glFlush();
    //spitOutDepthBuffer(); // This is just used to get an image of the depth buffer
```

Rendering Pass (2)

View & Projection Matrices

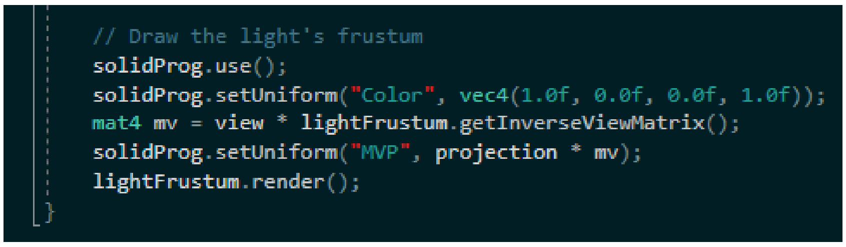
Framebuffer Binding, Culling, Shadow Shading & Drawing

Select the viewport, view & projection matrices appropriate for the scene. Bind to the default framebuffer, either disable culling or switch to back-face culling, select the subroutine function shadeWithShadow() function & draw the scene. Lastly, render the scene again from the point of view of the camera.



```
// Pass 2 (render)
float c = 2.0f;
vec3 cameraPos(c * 11.5f * cos(angle), c * 7.0f, c * 11.5f * sin(angle));
view = glm::lookAt(cameraPos, vec3(0.0f), vec3(0.0f, 1.0f, 0.0f));
prog.setUniform("Light.Position", view * vec4(lightFrustum.getOrigin(), 1.0f));
projection = glm::perspective(glm::radians(50.0f), (float)width / height, 0.1f, 100.0f);

glBindFramebuffer(GL_FRAMEBUFFER, 0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glViewport(0, 0, width, height);
glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, &pass2Index);
drawScene();
```

Drawing of Light’s Frustum



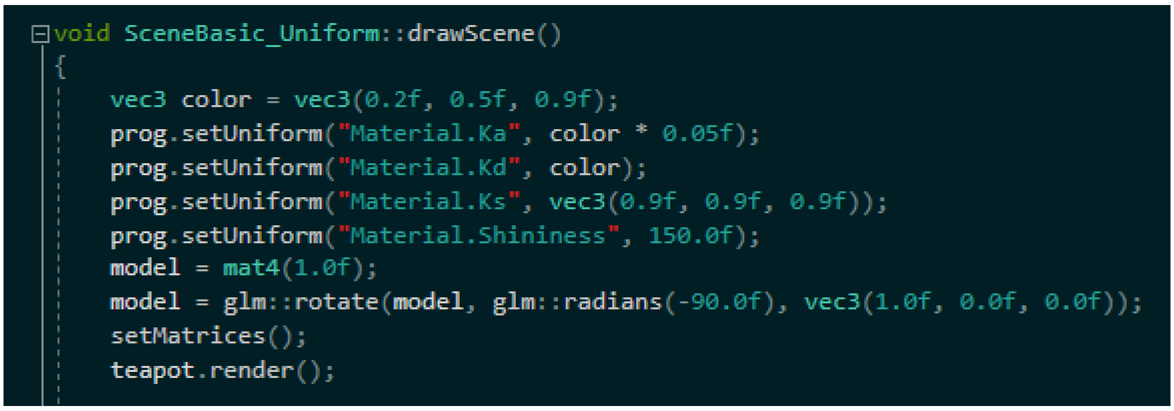
```
    // Draw the light's frustum
    solidProg.use();
    solidProg.setUniform("Color", vec4(1.0f, 0.0f, 0.0f, 1.0f));
    mat4 mv = view * lightFrustum.getInverseViewMatrix();
    solidProg.setUniform("MVP", projection * mv);
    lightFrustum.render();
}
```

Scene Drawing

Overview

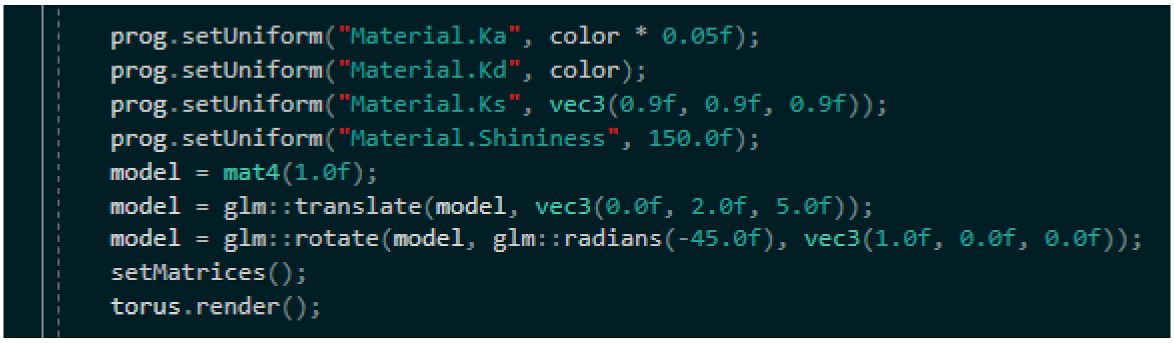
Render the scene as normal.

Teapot



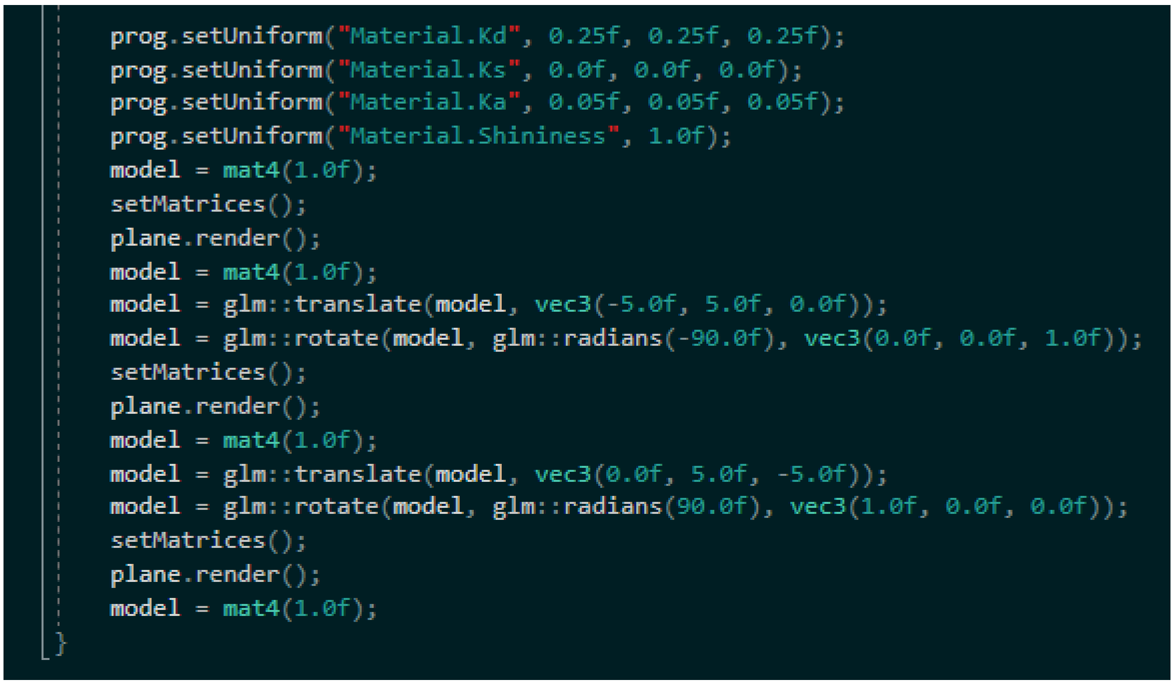
```
void SceneBasic_Uniform::drawScene()
{
    vec3 color = vec3(0.2f, 0.5f, 0.9f);
    prog.setUniform("Material.Ka", color * 0.05f);
    prog.setUniform("Material.Kd", color);
    prog.setUniform("Material.Ks", vec3(0.9f, 0.9f, 0.9f));
    prog.setUniform("Material.Shininess", 150.0f);
    model = mat4(1.0f);
    model = glm::rotate(model, glm::radians(-90.0f), vec3(1.0f, 0.0f, 0.0f));
    setMatrices();
    teapot.render();
```

Torus



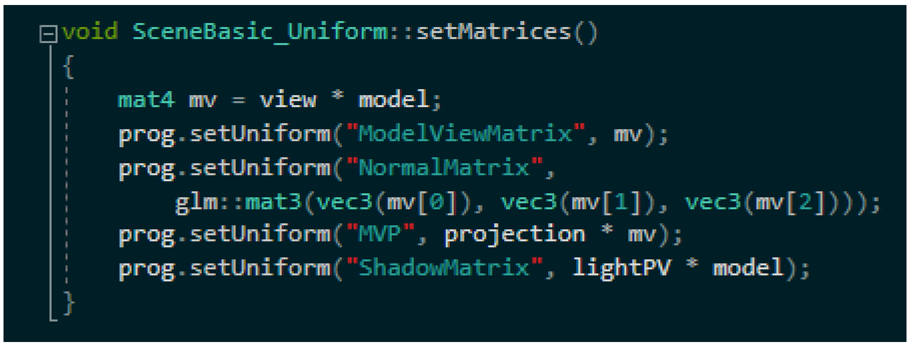
```
    prog.setUniform("Material.Ka", color * 0.05f);
    prog.setUniform("Material.Kd", color);
    prog.setUniform("Material.Ks", vec3(0.9f, 0.9f, 0.9f));
    prog.setUniform("Material.Shininess", 150.0f);
    model = mat4(1.0f);
    model = glm::translate(model, vec3(0.0f, 2.0f, 5.0f));
    model = glm::rotate(model, glm::radians(-45.0f), vec3(1.0f, 0.0f, 0.0f));
    setMatrices();
    torus.render();
```

Planes



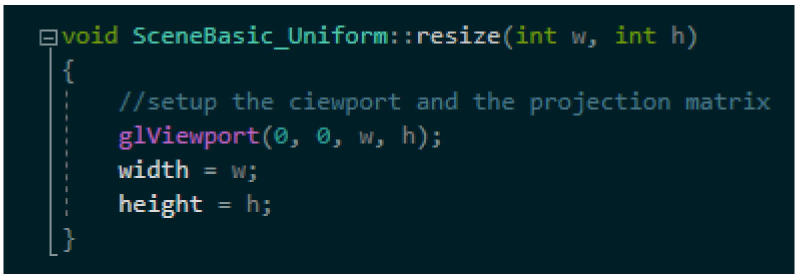
```
    prog.setUniform("Material.Kd", 0.25f, 0.25f, 0.25f);
    prog.setUniform("Material.Ks", 0.0f, 0.0f, 0.0f);
    prog.setUniform("Material.Ka", 0.05f, 0.05f, 0.05f);
    prog.setUniform("Material.Shininess", 1.0f);
    model = mat4(1.0f);
    setMatrices();
    plane.render();
    model = mat4(1.0f);
    model = glm::translate(model, vec3(-5.0f, 5.0f, 0.0f));
    model = glm::rotate(model, glm::radians(-90.0f), vec3(0.0f, 0.0f, 1.0f));
    setMatrices();
    plane.render();
    model = mat4(1.0f);
    model = glm::translate(model, vec3(0.0f, 5.0f, -5.0f));
    model = glm::rotate(model, glm::radians(90.0f), vec3(1.0f, 0.0f, 0.0f));
    setMatrices();
    plane.render();
    model = mat4(1.0f);
}
```

Setting of Matrices



```
void SceneBasic_Uniform::setMatrices()
{
    mat4 mv = view * model;
    prog.setUniform("ModelViewMatrix", mv);
    prog.setUniform("NormalMatrix",
        glm::mat3(vec3(mv[0]), vec3(mv[1]), vec3(mv[2])));
    prog.setUniform("MVP", projection * mv);
    prog.setUniform("ShadowMatrix", lightPV * model);
}
```

Resizing



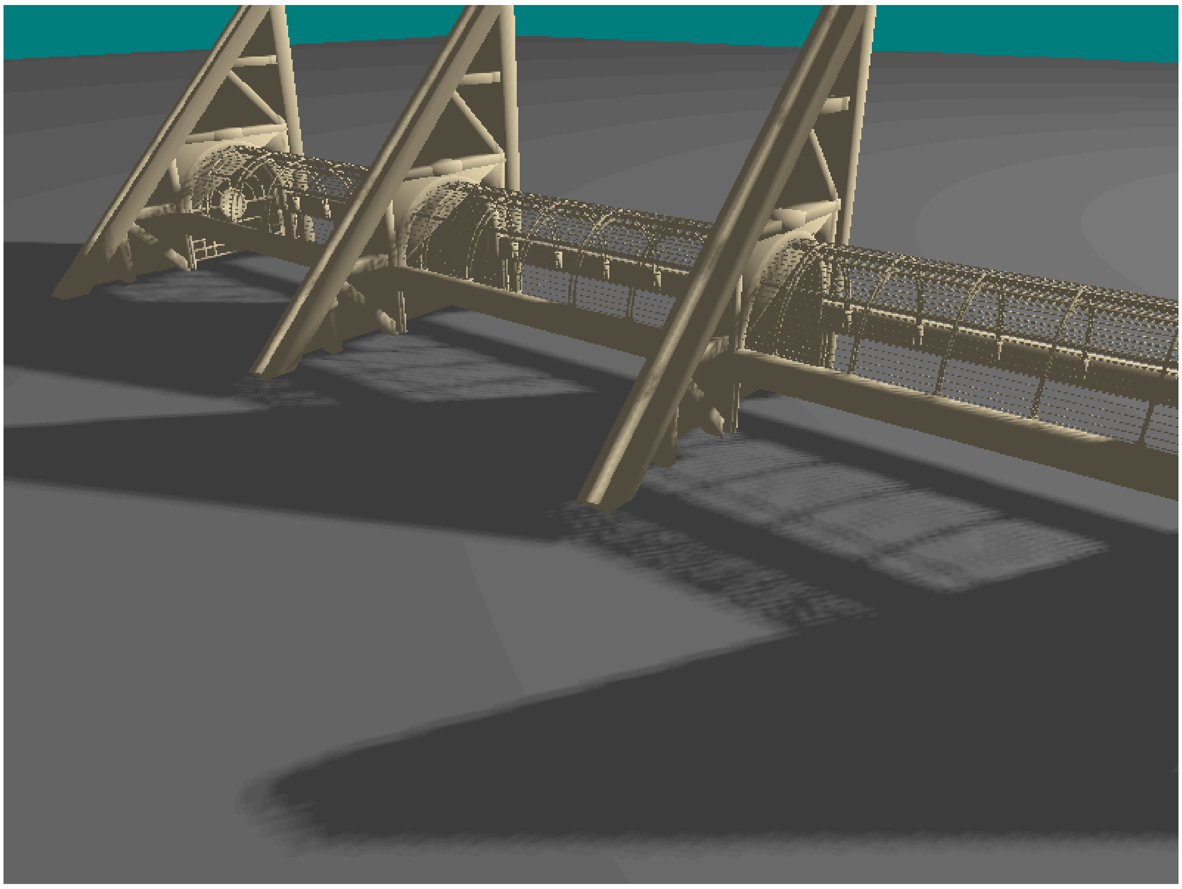
```
void SceneBasic_Uniform::resize(int w, int h)
{
    //setup the ciewport and the projection matrix
    glViewport(0, 0, w, h);
    width = w;
    height = h;
}
```

Anti-Aliasing Shadow Edges with Percentage-Closer Filtering (PCF)

Overview

This implementation is very similar to the standard shadow maps implementation.

Result

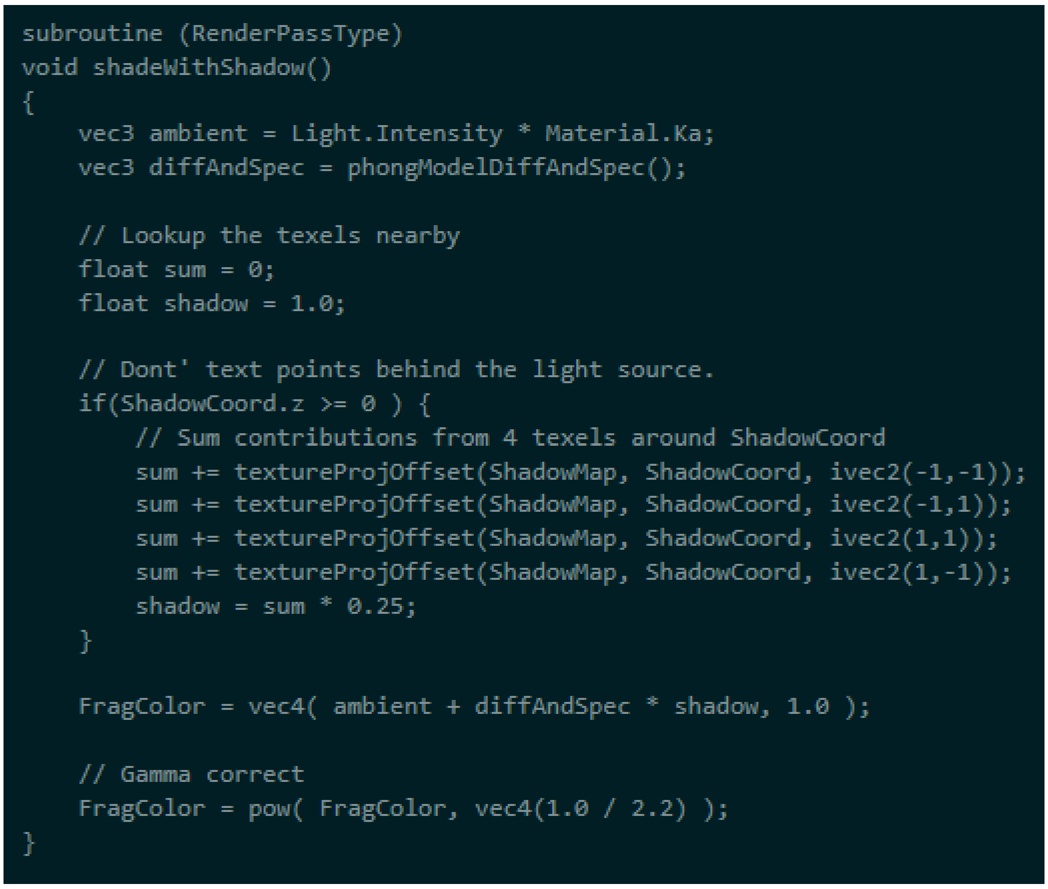


Vertex Shader

Use the vertex shader from the standard shadow map implementation.

Fragment Shader

Use the fragment shader from the standard shadow map implementation. However, slightly modify the shadeWithShadow() function:



```
subroutine (RenderPassType)
void shadeWithShadow()
{
    vec3 ambient = Light.Intensity * Material.Ka;
    vec3 diffAndSpec = phongModelDiffAndSpec();

    // Lookup the texels nearby
    float sum = 0;
    float shadow = 1.0;

    // Dont' text points behind the light source.
    if(ShadowCoord.z >= 0 ) {
        // Sum contributions from 4 texels around ShadowCoord
        sum += textureProjOffset(ShadowMap, ShadowCoord, ivec2(-1,-1));
        sum += textureProjOffset(ShadowMap, ShadowCoord, ivec2(-1,1));
        sum += textureProjOffset(ShadowMap, ShadowCoord, ivec2(1,1));
        sum += textureProjOffset(ShadowMap, ShadowCoord, ivec2(1,-1));
        shadow = sum * 0.25;
    }

    FragColor = vec4( ambient + diffAndSpec * shadow, 1.0 );

    // Gamma correct
    FragColor = pow( FragColor, vec4(1.0 / 2.2) );
}
```

Scenebasic Uniform Header

Similar to the header from the standard shadow map implementation.

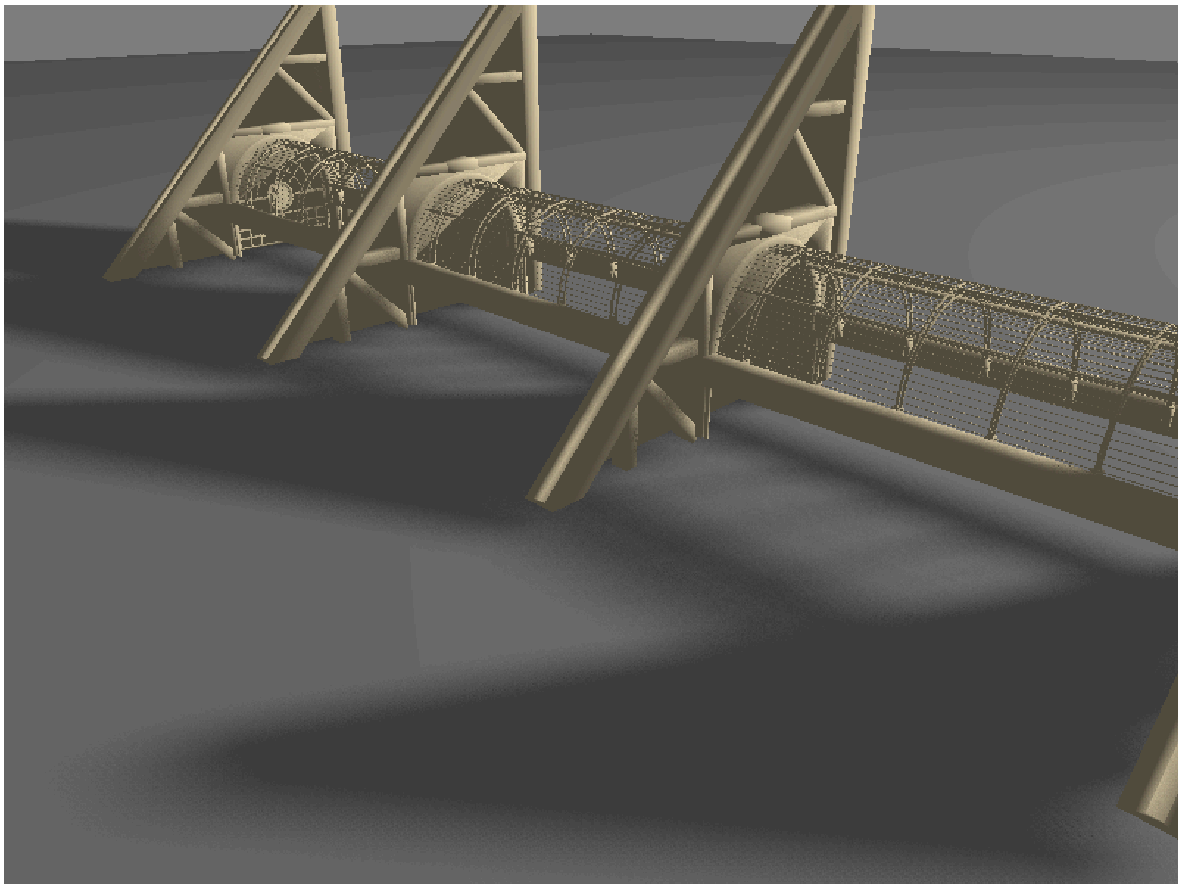
Scenebasic Uniform CPP

In the setupFBO() function, ensure that linear filtering is used on the depth texture, as opposed to nearest. Also ensure that the building model is used. No teapot or torus are required. Also make sure to adjust the camera so that the building model is visible.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Shadow Maps with Soft Edges

Result



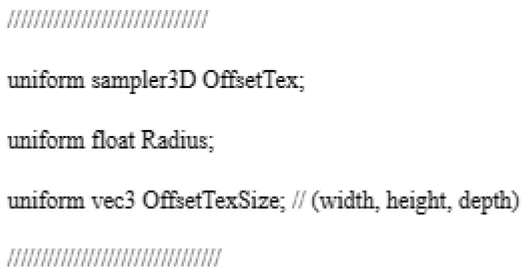
Vertex Shader

Use the vertex shader from the standard shadow map implementation.

Fragment Shader

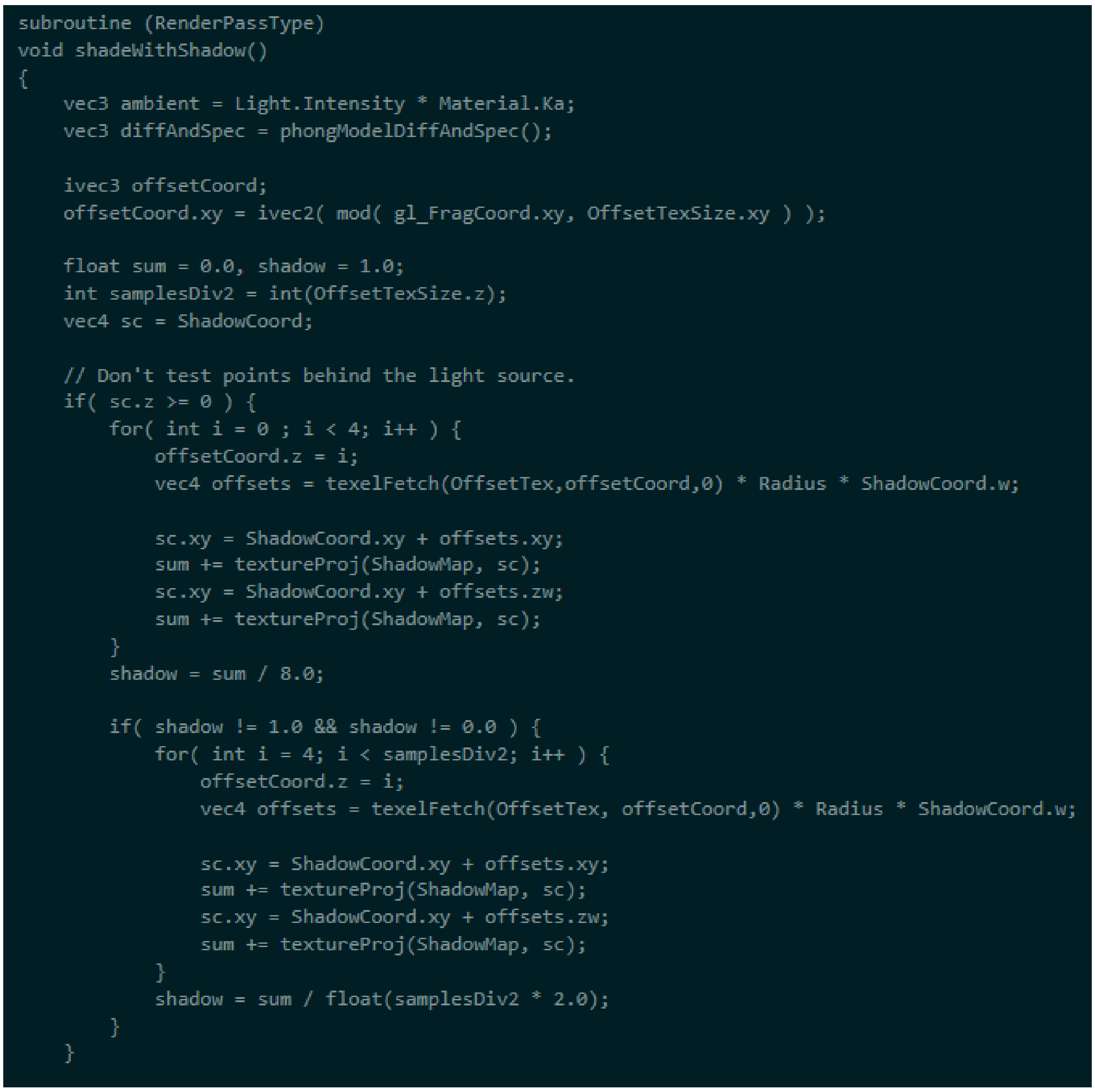
Globals

Use the fragment shader from the standard shadow map implementation. However, add the following 3 extra variables:



```
////////////////////////////////

uniform sampler3D OffsetTex;

uniform float Radius;

uniform vec3 OffsetTexSize; // (width, height, depth)

//////////////////////////////////
```

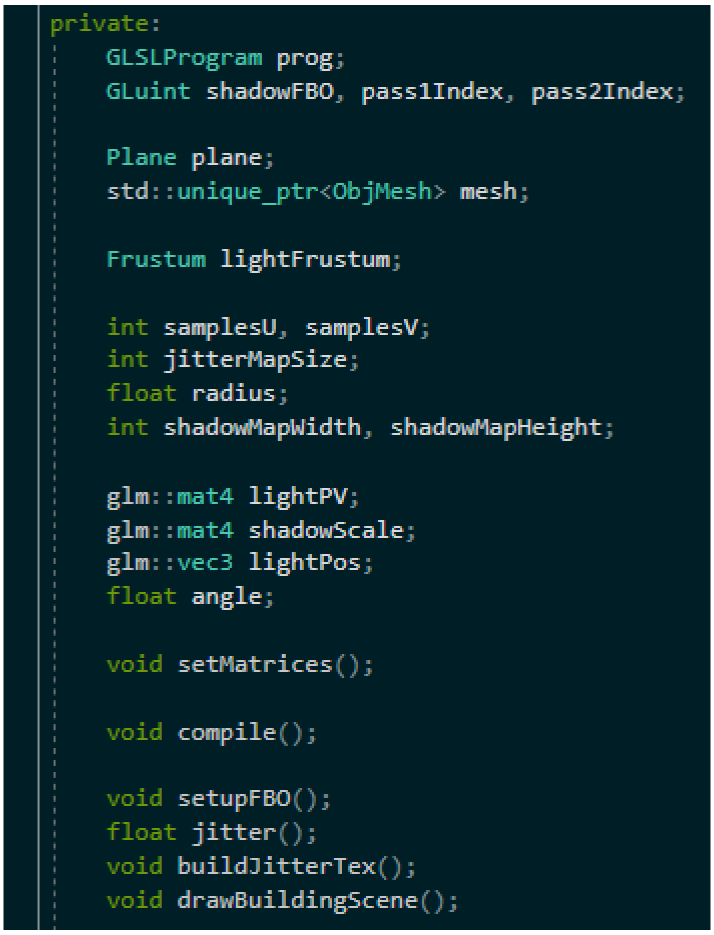
Shading with Shadow



```
subroutine (RenderPassType)
void shadeWithShadow()
{
    vec3 ambient = Light.Intensity * Material.Ka;
    vec3 diffAndSpec = phongModelDiffAndSpec();

    ivec3 offsetCoord;
    offsetCoord.xy = ivec2( mod( gl_FragCoord.xy, OffsetTexSize.xy ) );

    float sum = 0.0, shadow = 1.0;
    int samplesDiv2 = int(OffsetTexSize.z);
    vec4 sc = ShadowCoord;

    // Don't test points behind the light source.
    if( sc.z >= 0 ) {
        for( int i = 0 ; i < 4; i++ ) {
            offsetCoord.z = i;
            vec4 offsets = texelFetch(OffsetTex,offsetCoord,0) * Radius * ShadowCoord.w;

            sc.xy = ShadowCoord.xy + offsets.xy;
            sum += textureProj(ShadowMap, sc);
            sc.xy = ShadowCoord.xy + offsets.zw;
            sum += textureProj(ShadowMap, sc);
        }
        shadow = sum / 8.0;

        if( shadow != 1.0 && shadow != 0.0 ) {
            for( int i = 4; i < samplesDiv2; i++ ) {
                offsetCoord.z = i;
                vec4 offsets = texelFetch(OffsetTex, offsetCoord,0) * Radius * ShadowCoord.w;

                sc.xy = ShadowCoord.xy + offsets.xy;
                sum += textureProj(ShadowMap, sc);
                sc.xy = ShadowCoord.xy + offsets.zw;
                sum += textureProj(ShadowMap, sc);
            }
            shadow = sum / float(samplesDiv2 * 2.0);
        }
    }
```

Scenebasic Uniform Header

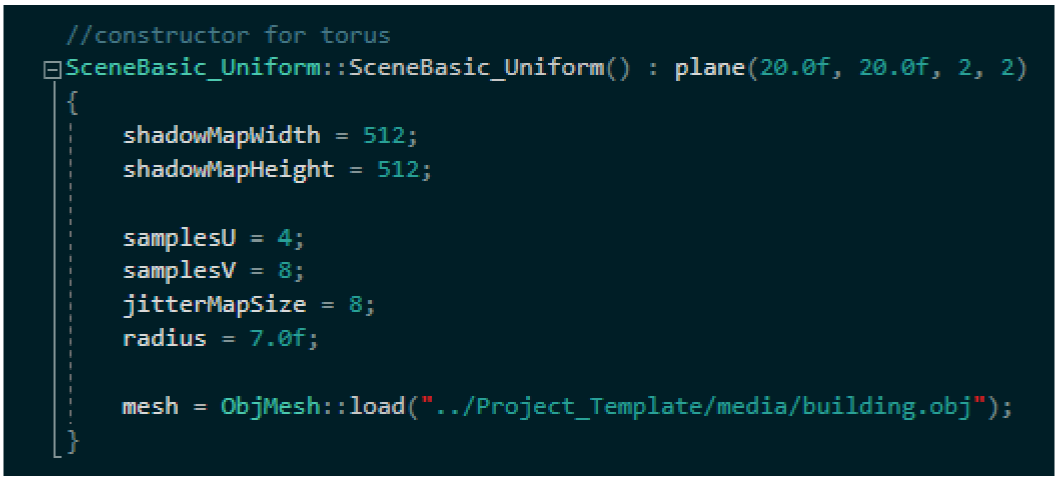
The header is similar to the anti-aliasing shadow edges with percentage-closer filtering implementation. Add 2 new variables & methods for creating the jitter effect:



```
private:
    GLSLProgram prog;
    GLuint shadowFBO, pass1Index, pass2Index;

    Plane plane;
    std::unique_ptr<ObjMesh> mesh;

    Frustum lightFrustum;

    int samplesU, samplesV;
    int jitterMapSize;
    float radius;
    int shadowMapWidth, shadowMapHeight;

    glm::mat4 lightPV;
    glm::mat4 shadowScale;
    glm::vec3 lightPos;
    float angle;

    void setMatrices();

    void compile();

    void setupFBO();
    float jitter();
    void buildJitterTex();
    void drawBuildingScene();
```

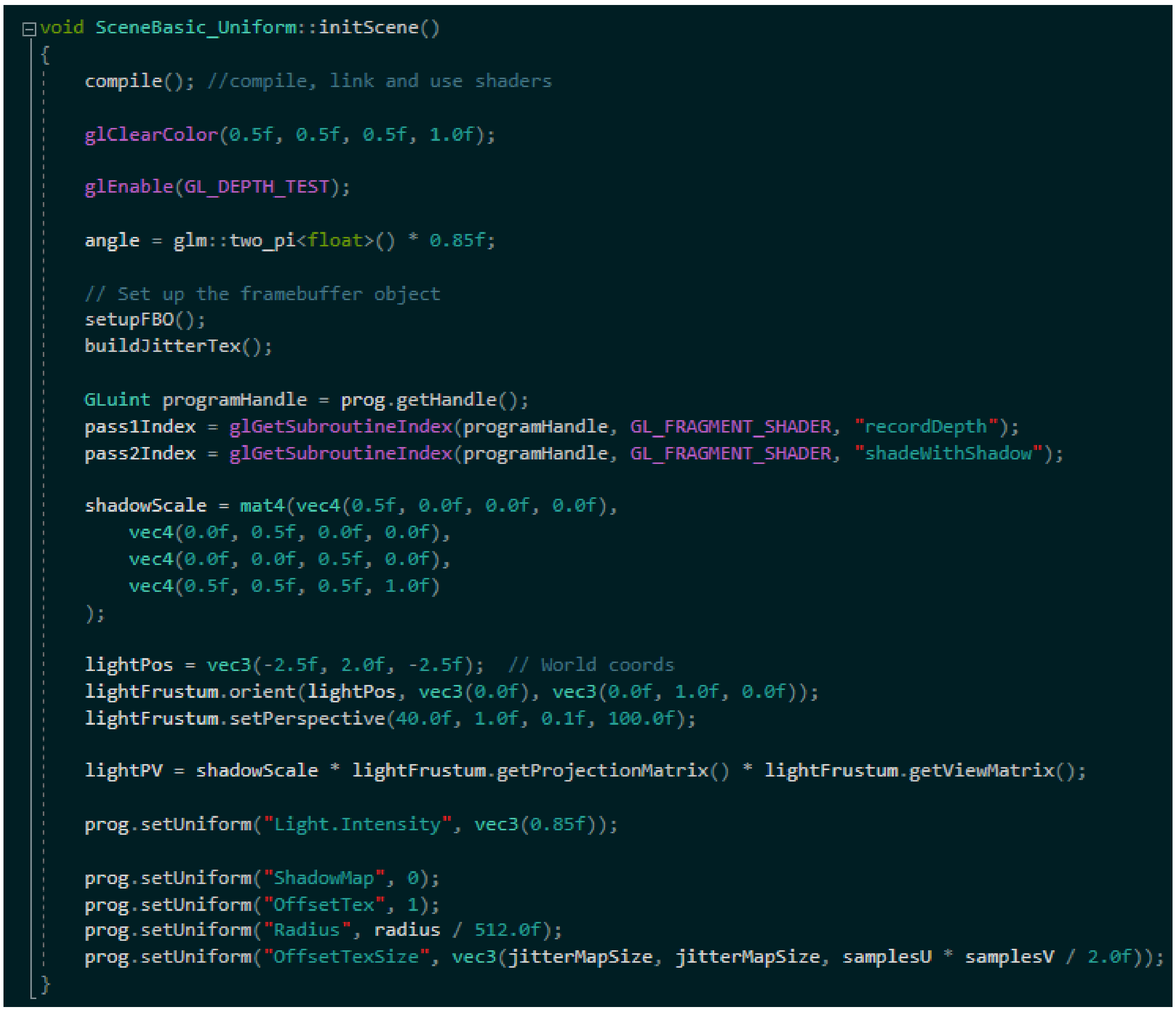
Scenebasic Uniform CPP

Constructor



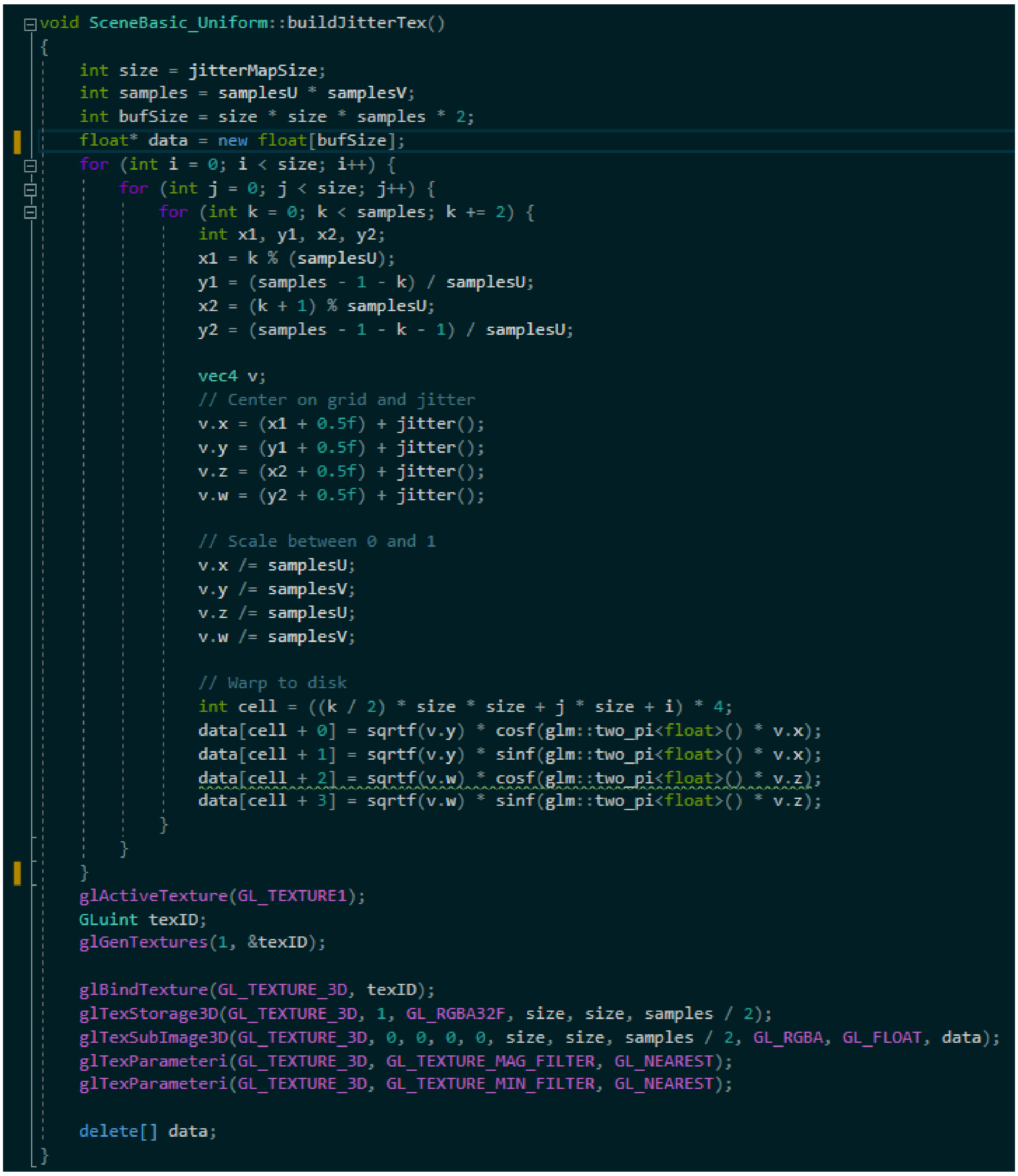
```cpp
//constructor for torus
SceneBasic_Uniform::SceneBasic_Uniform() : plane(20.0f, 20.0f, 2, 2)
{
    shadowMapWidth = 512;
    shadowMapHeight = 512;

    samplesU = 4;
    samplesV = 8;
    jitterMapSize = 8;
    radius = 7.0f;

    mesh = ObjMesh::load("../Project_Template/media/building.obj");
}
```

Scene Initialisation



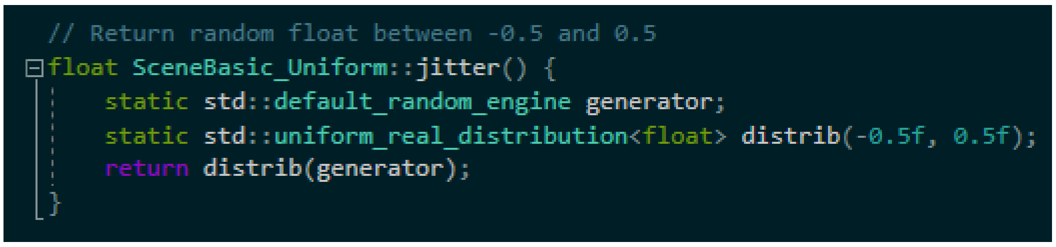
```cpp
void SceneBasic_Uniform::initScene()
{
    compile(); //compile, link and use shaders

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);

    glEnable(GL_DEPTH_TEST);

    angle = glm::two_pi<float>() * 0.85f;

    // Set up the framebuffer object
    setupFBO();
    buildJitterTex();

    GLuint programHandle = prog.getHandle();
    pass1Index = glGetSubroutineIndex(programHandle, GL_FRAGMENT_SHADER, "recordDepth");
    pass2Index = glGetSubroutineIndex(programHandle, GL_FRAGMENT_SHADER, "shadeWithShadow");

    shadowScale = mat4(vec4(0.5f, 0.0f, 0.0f, 0.0f),
        vec4(0.0f, 0.5f, 0.0f, 0.0f),
        vec4(0.0f, 0.0f, 0.5f, 0.0f),
        vec4(0.5f, 0.5f, 0.5f, 1.0f)
    );

    lightPos = vec3(-2.5f, 2.0f, -2.5f);  // World coords
    lightFrustum.orient(lightPos, vec3(0.0f), vec3(0.0f, 1.0f, 0.0f));
    lightFrustum.setPerspective(40.0f, 1.0f, 0.1f, 100.0f);

    lightPV = shadowScale * lightFrustum.getProjectionMatrix() * lightFrustum.getViewMatrix();

    prog.setUniform("Light.Intensity", vec3(0.85f));

    prog.setUniform("ShadowMap", 0);
    prog.setUniform("OffsetTex", 1);
    prog.setUniform("Radius", radius / 512.0f);
    prog.setUniform("OffsetTexSize", vec3(jitterMapSize, jitterMapSize, samplesU * samplesV / 2.0f));
}
```

Jitter Texture Building



```cpp
void SceneBasic_Uniform::buildJitterTex()
{
    int size = jitterMapSize;
    int samples = samplesU * samplesV;
    int bufSize = size * size * samples * 2;
    float* data = new float[bufSize];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < samples; k += 2) {
                int x1, y1, x2, y2;
                x1 = k % (samplesU);
                y1 = (samples - 1 - k) / samplesU;
                x2 = (k + 1) % samplesU;
                y2 = (samples - 1 - k - 1) / samplesU;

                vec4 v;
                // Center on grid and jitter
                v.x = (x1 + 0.5f) + jitter();
                v.y = (y1 + 0.5f) + jitter();
                v.z = (x2 + 0.5f) + jitter();
                v.w = (y2 + 0.5f) + jitter();

                // Scale between 0 and 1
                v.x /= samplesU;
                v.y /= samplesV;
                v.z /= samplesU;
                v.w /= samplesV;

                // Warp to disk
                int cell = ((k / 2) * size * size + j * size + i) * 4;
                data[cell + 0] = sqrtf(v.y) * cosf(glm::two_pi<float>() * v.x);
                data[cell + 1] = sqrtf(v.y) * sinf(glm::two_pi<float>() * v.x);
                data[cell + 2] = sqrtf(v.w) * cosf(glm::two_pi<float>() * v.z);
                data[cell + 3] = sqrtf(v.w) * sinf(glm::two_pi<float>() * v.z);
            }
        }
    }
    glActiveTexture(GL_TEXTURE1);
    GLuint texID;
    glGenTextures(1, &texID);

    glBindTexture(GL_TEXTURE_3D, texID);
    glTexStorage3D(GL_TEXTURE_3D, 1, GL_RGBA32F, size, size, samples / 2);
    glTexSubImage3D(GL_TEXTURE_3D, 0, 0, 0, 0, size, size, samples / 2, GL_RGBA, GL_FLOAT, data);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

    delete[] data;
}
```

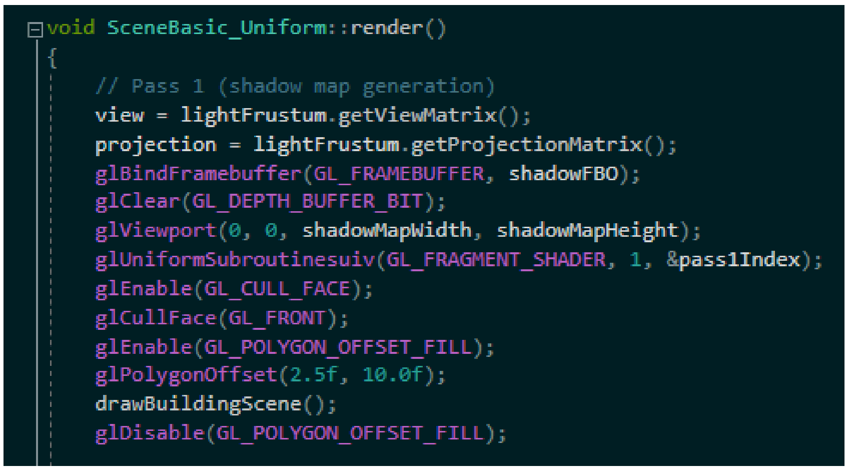
Jittering



```cpp
// Return random float between -0.5 and 0.5
float SceneBasic_Uniform::jitter() {
    static std::default_random_engine generator;
    static std::uniform_real_distribution<float> distrib(-0.5f, 0.5f);
    return distrib(generator);
}
```

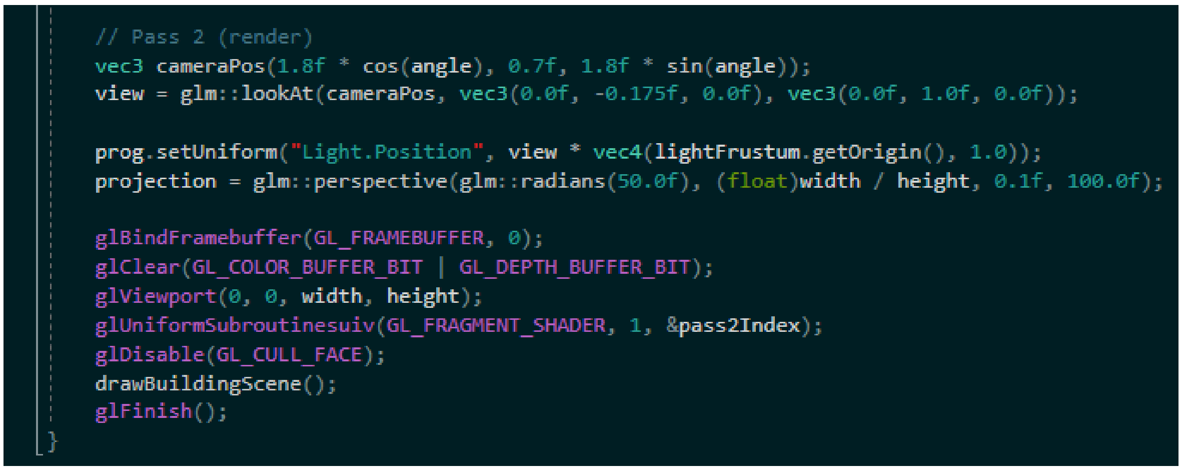
Rendering

Shadow Map Generation Pass (1)



```
void SceneBasic_Uniform::render()
{
    // Pass 1 (shadow map generation)
    view = lightFrustum.getViewMatrix();
    projection = lightFrustum.getProjectionMatrix();
    glBindFramebuffer(GL_FRAMEBUFFER, shadowFBO);
    glClear(GL_DEPTH_BUFFER_BIT);
    glViewport(0, 0, shadowMapWidth, shadowMapHeight);
    glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, &pass1Index);
    glEnable(GL_CULL_FACE);
    glCullFace(GL_FRONT);
    glEnable(GL_POLYGON_OFFSET_FILL);
    glPolygonOffset(2.5f, 10.0f);
    drawBuildingScene();
    glDisable(GL_POLYGON_OFFSET_FILL);
```

Render Pass (2)



```
    // Pass 2 (render)
    vec3 cameraPos(1.8f * cos(angle), 0.7f, 1.8f * sin(angle));
    view = glm::lookAt(cameraPos, vec3(0.0f, -0.175f, 0.0f), vec3(0.0f, 1.0f, 0.0f));

    prog.setUniform("Light.Position", view * vec4(lightFrustum.getOrigin(), 1.0));
    projection = glm::perspective(glm::radians(50.0f), (float)width / height, 0.1f, 100.0f);

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glViewport(0, 0, width, height);
    glUniformSubroutinesuiv(GL_FRAGMENT_SHADER, 1, &pass2Index);
    glDisable(GL_CULL_FACE);
    drawBuildingScene();
    glFinish();
}
```

FBO Setup

Use the setupFBO() function from the anti-aliasing shadow edges with percentage-closer filtering implementation.

Shadow Volumes

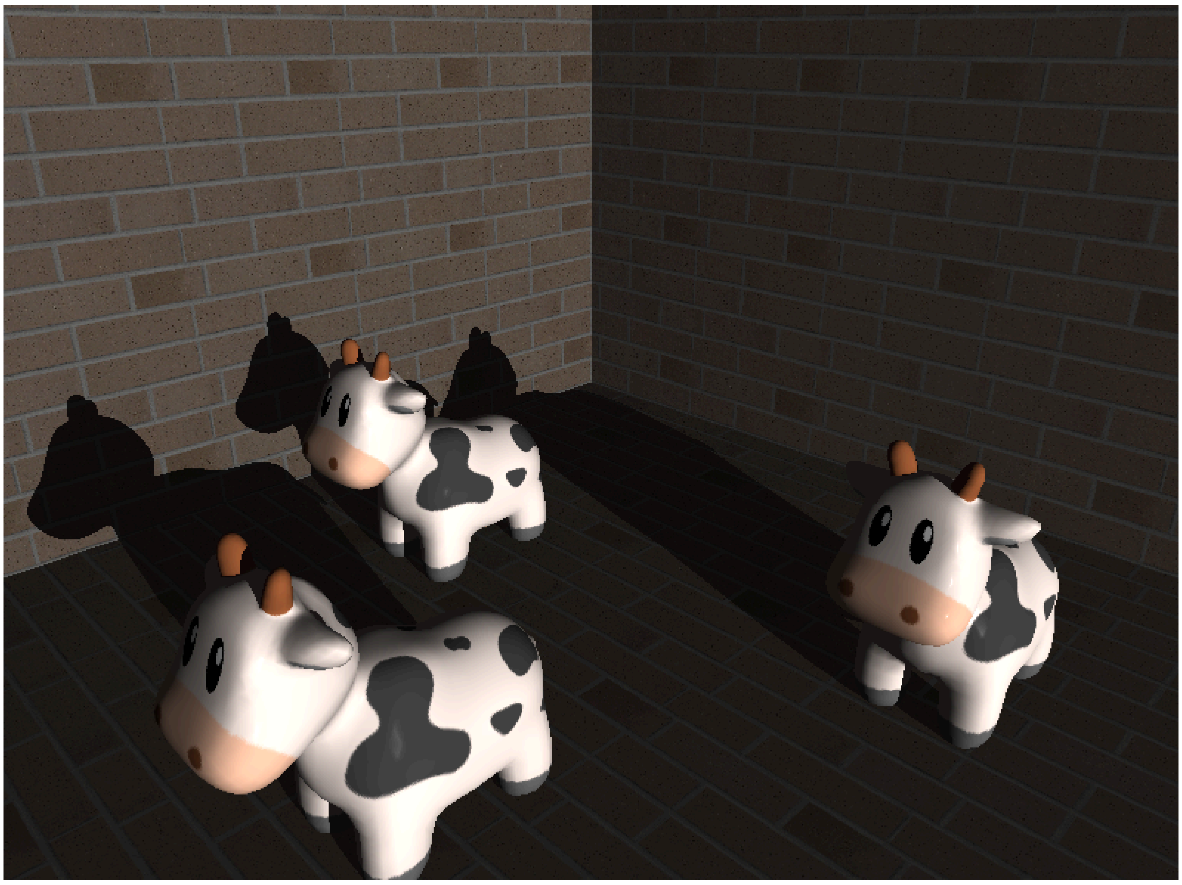
Drawing

The shadow volumes will be drawn with aid from the geometry shader. The geometry will be rendered normally & the geometry shader will produce the shadow volumes. This is as opposed to computing the shadow volumes on the CPU.

Silhouette Edges

Adjacency information can be used to determine whether a triangle has a silhouette edge. If a triangle faces towards the given light & a neighbouring triangle faces away from it, the shared edge can be considered a silhouette edge. This is used to create a polygon for the shadow volume.

Result



Vertex Shader

```
layout (location=0) in vec3 VertexPosition;
layout (location=1) in vec3 VertexNormal;
layout (location=2) in vec2 VertexTexCoord;

out vec3 VPosition;
out vec3 VNormal;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjMatrix;

void main()
{
    VNormal = NormalMatrix * VertexNormal;
    VPosition = (ModelViewMatrix * vec4(VertexPosition,1.0)).xyz;
    gl_Position = ProjMatrix * ModelViewMatrix * vec4(VertexPosition,1.0);
}
```

Fragment Shader

The fragment shader is left empty:

```
void main()

{

        // Nothing to see here, move along

}
```

Geometry Shader

Globals

```
layout( triangles_adjacency ) in;
layout( triangle_strip, max_vertices = 18 ) out;

in vec3 VPosition[];
in vec3 VNormal[];

uniform vec4 LightPosition;  // Light position (eye coords)
uniform mat4 ProjMatrix;     // Projection matrix
```

Faces Light

```
bool facesLight( vec3 a, vec3 b, vec3 c )
{
  vec3 n = cross( b - a, c - a );
  vec3 da = LightPosition.xyz - a;
  vec3 db = LightPosition.xyz - b;
  vec3 dc = LightPosition.xyz - c;

  return dot(n, da) > 0 || dot(n, db) > 0 || dot(n, dc) > 0;
}
```

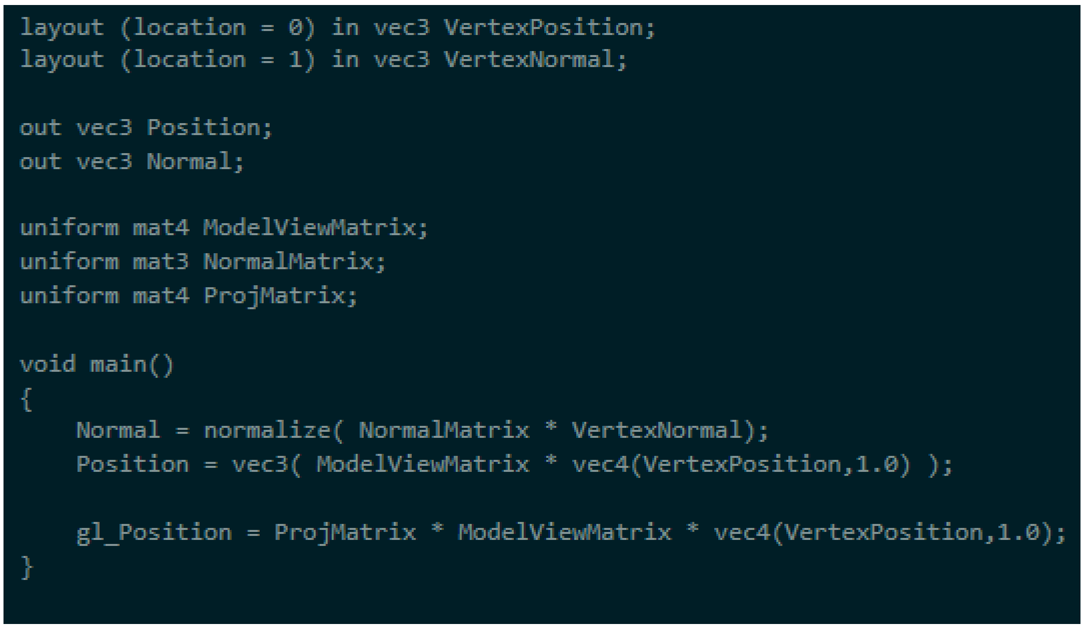
Edge Quad Emitting

```
void emitEdgeQuad( vec3 a, vec3 b ) {
  gl_Position = ProjMatrix * vec4(a, 1);
  EmitVertex();

  gl_Position = ProjMatrix * vec4(a - LightPosition.xyz, 0);
  EmitVertex();

  gl_Position = ProjMatrix * vec4(b, 1);
  EmitVertex();

  gl_Position = ProjMatrix * vec4(b - LightPosition.xyz, 0);
  EmitVertex();
  EndPrimitive();
}
```

Main Function

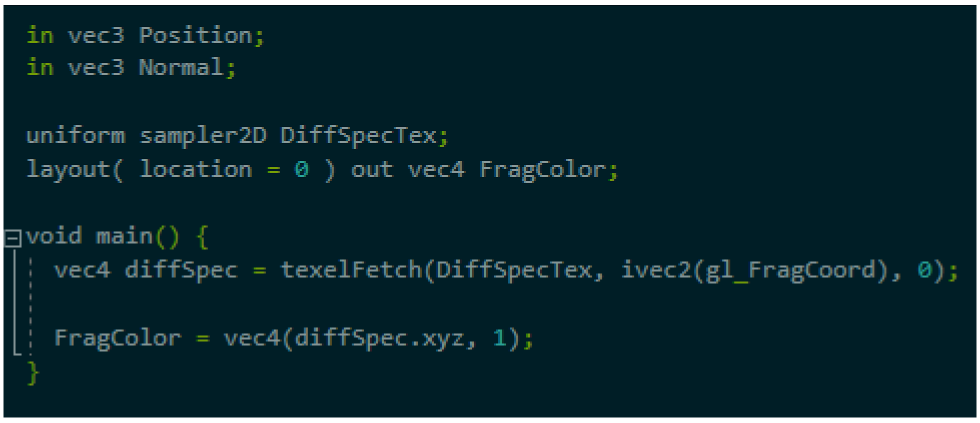
```
void main()
{
    // If the main triangle faces the light, check each adjacent
    // triangle.  If an adjacent triangle does not face the light
    // we output a sihlouette edge quad for the corresponding edge.
    if( facesLight(VPosition[0], VPosition[2], VPosition[4]) ) {
        if( ! facesLight(VPosition[0],VPosition[1],VPosition[2]) )
          emitEdgeQuad(VPosition[0],VPosition[2]);
        if( ! facesLight(VPosition[2],VPosition[3],VPosition[4]) )
          emitEdgeQuad(VPosition[2],VPosition[4]);
        if( ! facesLight(VPosition[4],VPosition[5],VPosition[0]) )
          emitEdgeQuad(VPosition[4],VPosition[0]);
    }
}
```

Shadow Volume Comp Vertex Shader



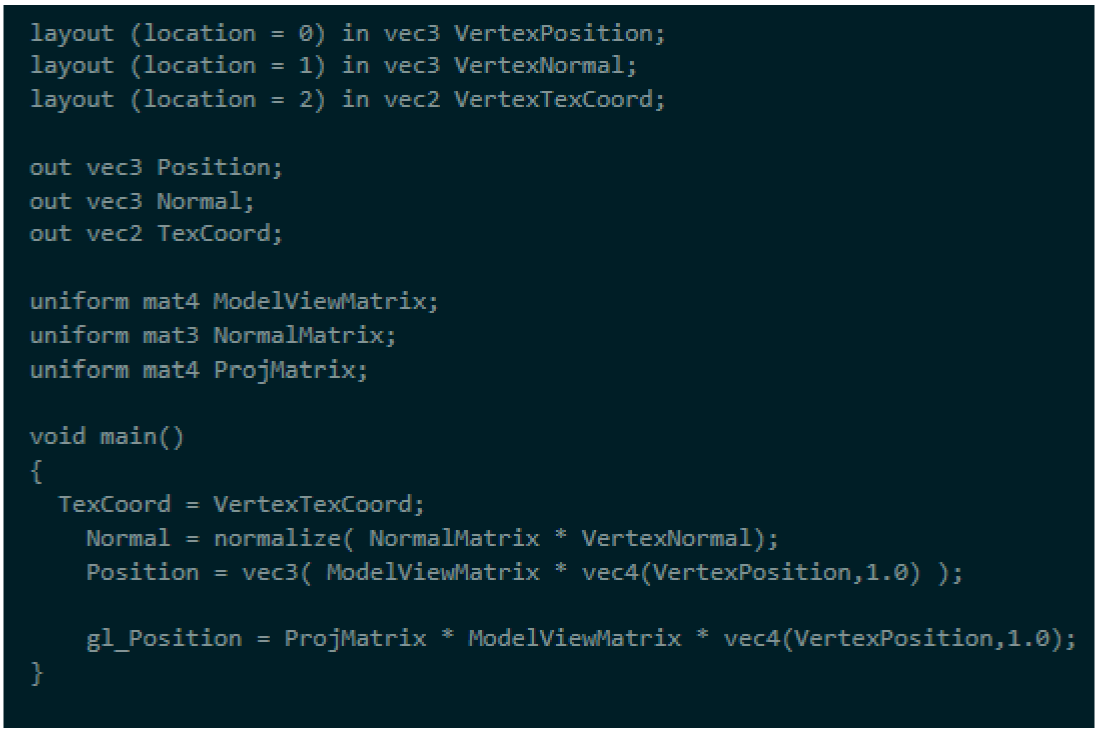
```
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;

out vec3 Position;
out vec3 Normal;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjMatrix;

void main()
{
    Normal = normalize( NormalMatrix * VertexNormal);
    Position = vec3( ModelViewMatrix * vec4(VertexPosition,1.0) );

    gl_Position = ProjMatrix * ModelViewMatrix * vec4(VertexPosition,1.0);
}
```

Shadow Volume Comp Fragment Shader



```
in vec3 Position;
in vec3 Normal;

uniform sampler2D DiffSpecTex;
layout( location = 0 ) out vec4 FragColor;

void main() {
    vec4 diffSpec = texelFetch(DiffSpecTex, ivec2(gl_FragCoord), 0);

    FragColor = vec4(diffSpec.xyz, 1);
}
```

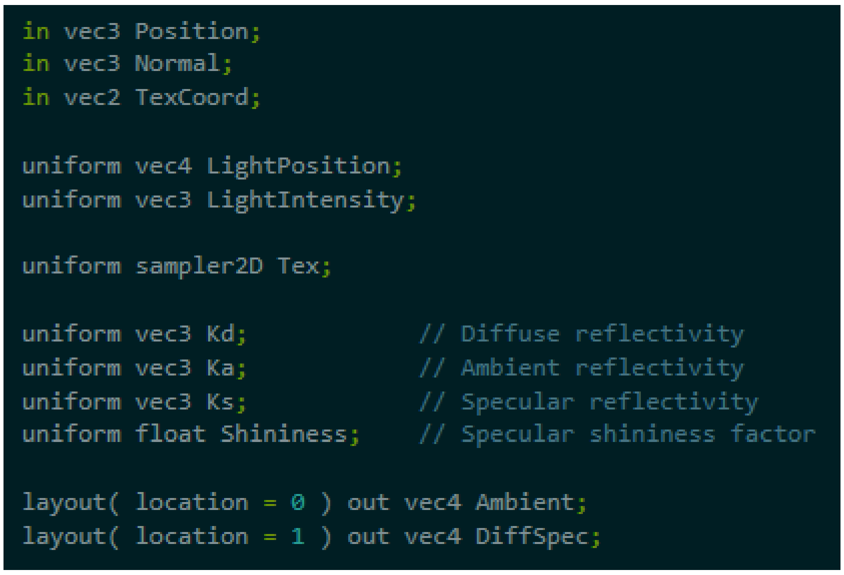
Shader Volume Render Vertex Shader



```glsl
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexNormal;
layout (location = 2) in vec2 VertexTexCoord;

out vec3 Position;
out vec3 Normal;
out vec2 TexCoord;

uniform mat4 ModelViewMatrix;
uniform mat3 NormalMatrix;
uniform mat4 ProjMatrix;

void main()
{
  TexCoord = VertexTexCoord;
    Normal = normalize( NormalMatrix * VertexNormal);
    Position = vec3( ModelViewMatrix * vec4(VertexPosition,1.0) );

    gl_Position = ProjMatrix * ModelViewMatrix * vec4(VertexPosition,1.0);
}
```

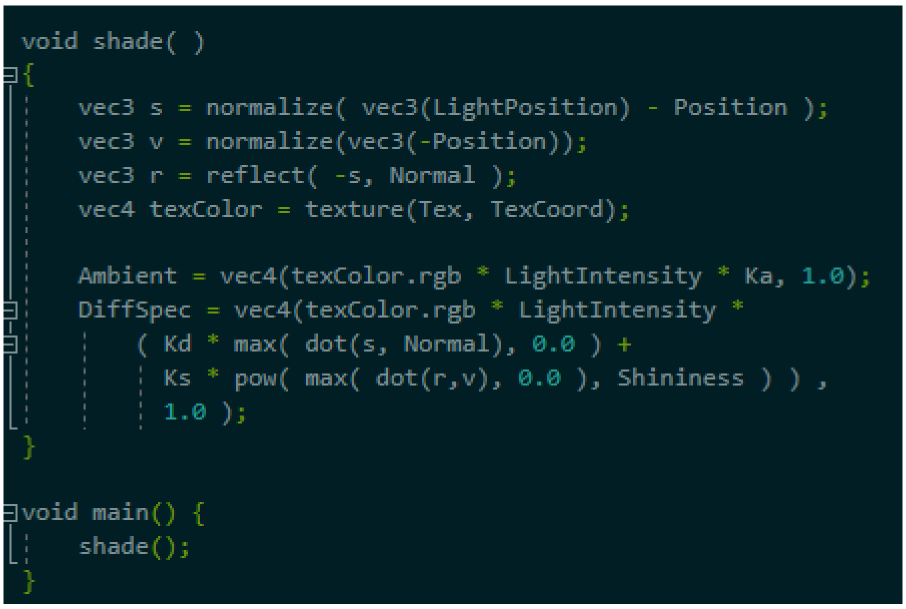
Shadow Volume Render Fragment Shader

Globals



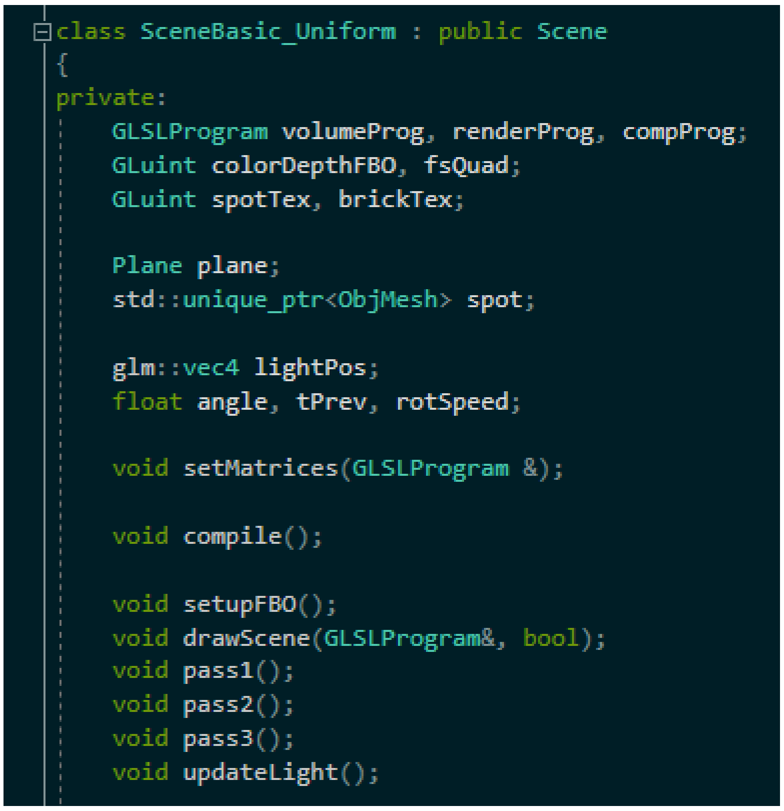
```glsl
in vec3 Position;
in vec3 Normal;
in vec2 TexCoord;

uniform vec4 LightPosition;
uniform vec3 LightIntensity;

uniform sampler2D Tex;

uniform vec3 Kd;            // Diffuse reflectivity
uniform vec3 Ka;            // Ambient reflectivity
uniform vec3 Ks;            // Specular reflectivity
uniform float Shininess;    // Specular shininess factor

layout( location = 0 ) out vec4 Ambient;
layout( location = 1 ) out vec4 DiffSpec;
```

Shade & Main Functions



```
void shade( )
{
    vec3 s = normalize( vec3(LightPosition) - Position );
    vec3 v = normalize(vec3(-Position));
    vec3 r = reflect( -s, Normal );
    vec4 texColor = texture(Tex, TexCoord);

    Ambient = vec4(texColor.rgb * LightIntensity * Ka, 1.0);
    DiffSpec = vec4(texColor.rgb * LightIntensity *
        ( Kd * max( dot(s, Normal), 0.0 ) +
          Ks * pow( max( dot(r,v), 0.0 ), Shininess ) ) ,
          1.0 );
}

void main() {
    shade();
}
```

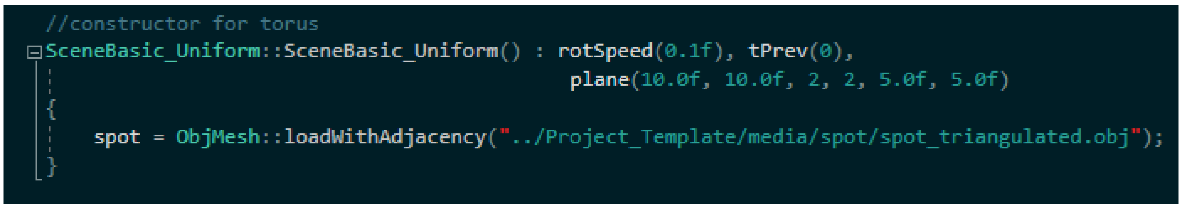
Scenebasic Uniform Header



```
class SceneBasic_Uniform : public Scene
{
private:
    GLSLProgram volumeProg, renderProg, compProg;
    GLuint colorDepthFBO, fsQuad;
    GLuint spotTex, brickTex;

    Plane plane;
    std::unique_ptr<ObjMesh> spot;

    glm::vec4 lightPos;
    float angle, tPrev, rotSpeed;

    void setMatrices(GLSLProgram &);

    void compile();

    void setupFBO();
    void drawScene(GLSLProgram&, bool);
    void pass1();
    void pass2();
    void pass3();
    void updateLight();
```

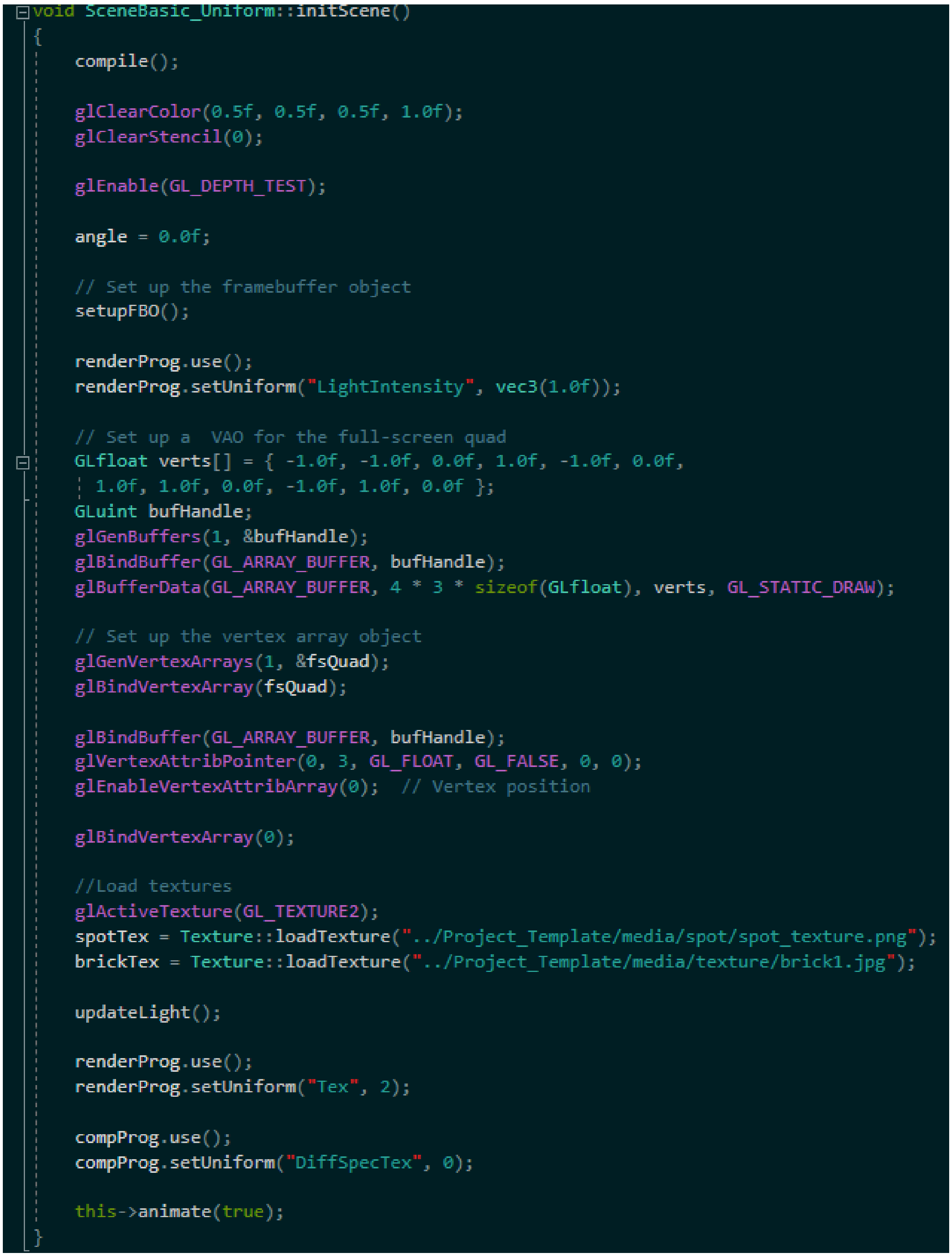
Scenebasic Uniform CPP

Constructor



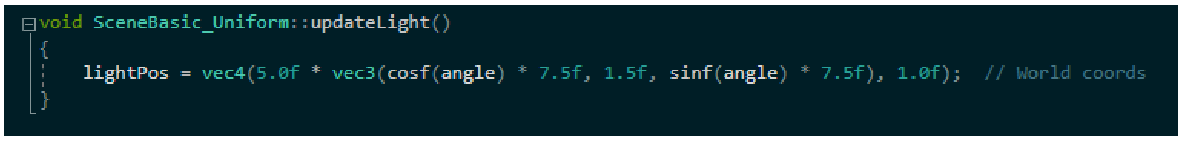
```
//constructor for torus
SceneBasic_Uniform::SceneBasic_Uniform() : rotSpeed(0.1f), tPrev(0),
                                           plane(10.0f, 10.0f, 2, 2, 5.0f, 5.0f)
{
    spot = ObjMesh::loadWithAdjacency("../Project_Template/media/spot/spot_triangulated.obj");
}
```

Scene Initialisation



```
void SceneBasic_Uniform::initScene()
{
    compile();

    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glClearStencil(0);

    glEnable(GL_DEPTH_TEST);

    angle = 0.0f;

    // Set up the framebuffer object
    setupFBO();

    renderProg.use();
    renderProg.setUniform("LightIntensity", vec3(1.0f));

    // Set up a  VAO for the full-screen quad
    GLfloat verts[] = { -1.0f, -1.0f, 0.0f, 1.0f, -1.0f, 0.0f,
        1.0f, 1.0f, 0.0f, -1.0f, 1.0f, 0.0f };
    GLuint bufHandle;
    glGenBuffers(1, &bufHandle);
    glBindBuffer(GL_ARRAY_BUFFER, bufHandle);
    glBufferData(GL_ARRAY_BUFFER, 4 * 3 * sizeof(GLfloat), verts, GL_STATIC_DRAW);

    // Set up the vertex array object
    glGenVertexArrays(1, &fsQuad);
    glBindVertexArray(fsQuad);

    glBindBuffer(GL_ARRAY_BUFFER, bufHandle);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(0);  // Vertex position

    glBindVertexArray(0);

    //Load textures
    glActiveTexture(GL_TEXTURE2);
    spotTex = Texture::loadTexture("../Project_Template/media/spot/spot_texture.png");
    brickTex = Texture::loadTexture("../Project_Template/media/texture/brick1.jpg");

    updateLight();

    renderProg.use();
    renderProg.setUniform("Tex", 2);

    compProg.use();
    compProg.setUniform("DiffSpecTex", 0);

    this->animate(true);
}
```

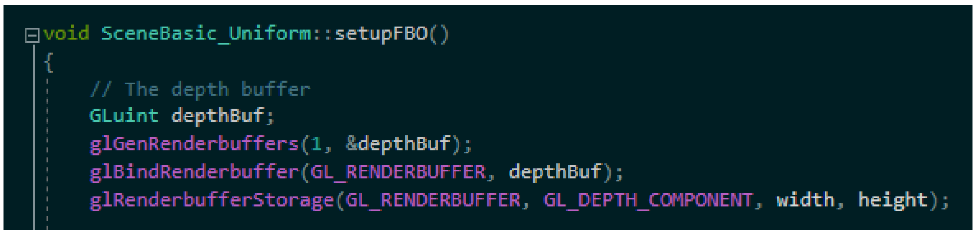
Light Updating



```
void SceneBasic_Uniform::updateLight()
{
    lightPos = vec4(5.0f * vec3(cosf(angle) * 7.5f, 1.5f, sinf(angle) * 7.5f), 1.0f);  // World coords
}
```

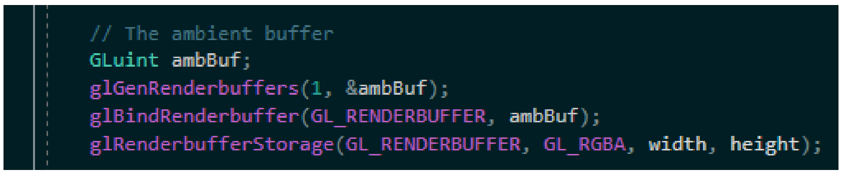
FBO Setup

Depth Buffer



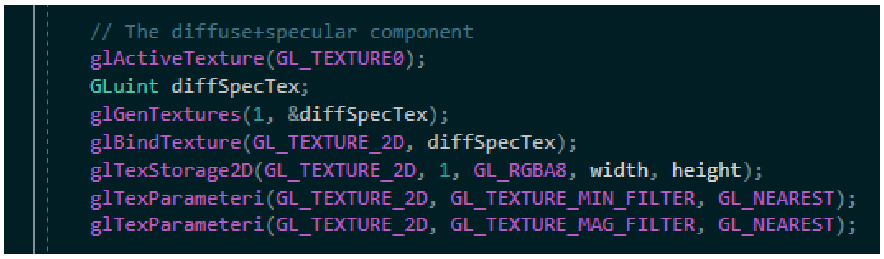
```
void SceneBasic_Uniform::setupFBO()
{
    // The depth buffer
    GLuint depthBuf;
    glGenRenderbuffers(1, &depthBuf);
    glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);
```

Ambient Buffer



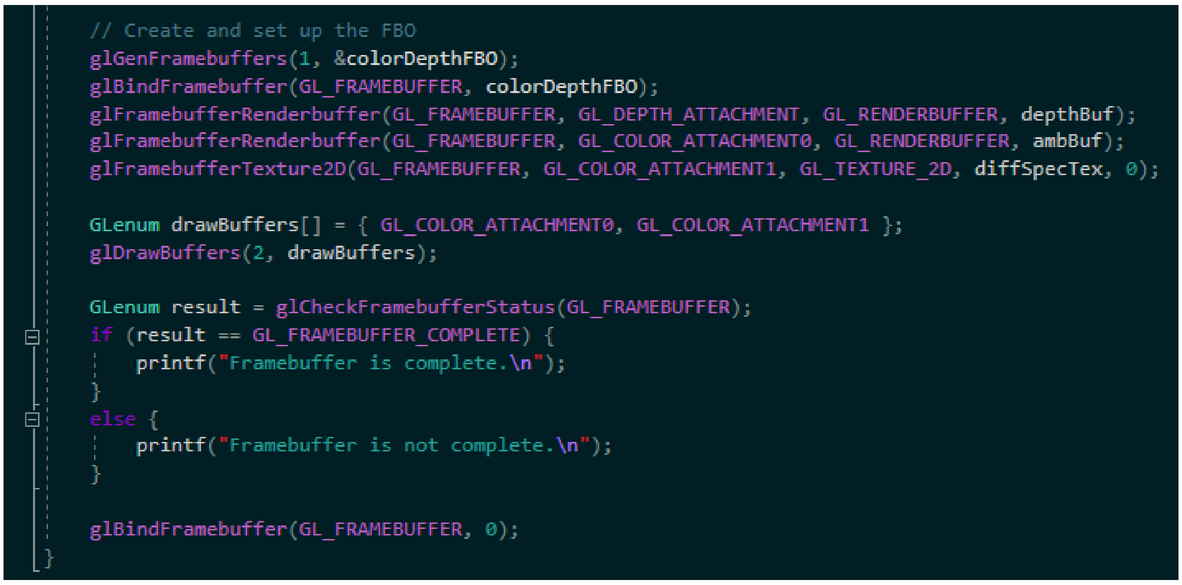
```
    // The ambient buffer
    GLuint ambBuf;
    glGenRenderbuffers(1, &ambBuf);
    glBindRenderbuffer(GL_RENDERBUFFER, ambBuf);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA, width, height);
```

Diffuse & Specular Component



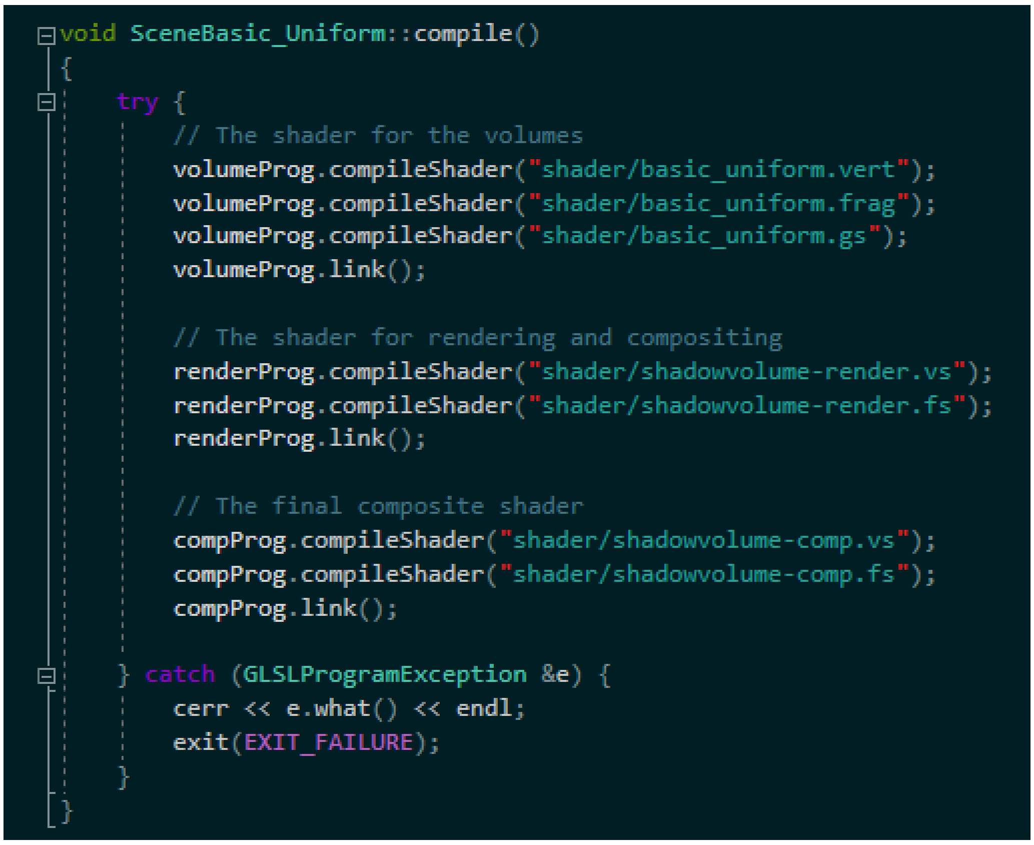
```
    // The diffuse+specular component
    glActiveTexture(GL_TEXTURE0);
    GLuint diffSpecTex;
    glGenTextures(1, &diffSpecTex);
    glBindTexture(GL_TEXTURE_2D, diffSpecTex);
    glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGBA8, width, height);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

Creation & Initialisation of FBO



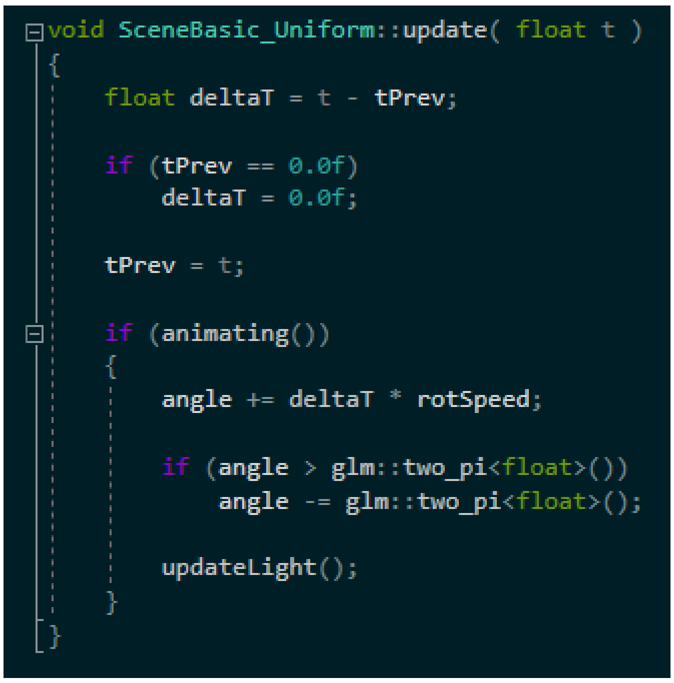
```
    // Create and set up the FBO
    glGenFramebuffers(1, &colorDepthFBO);
    glBindFramebuffer(GL_FRAMEBUFFER, colorDepthFBO);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuf);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_RENDERBUFFER, ambBuf);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, diffSpecTex, 0);

    GLenum drawBuffers[] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };
    glDrawBuffers(2, drawBuffers);

    GLenum result = glCheckFramebufferStatus(GL_FRAMEBUFFER);
    if (result == GL_FRAMEBUFFER_COMPLETE) {
        printf("Framebuffer is complete.\n");
    }
    else {
        printf("Framebuffer is not complete.\n");
    }

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Shader Compilation



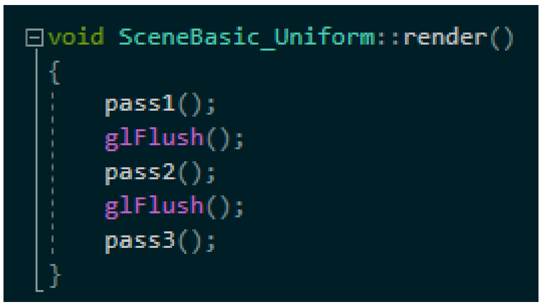
```
void SceneBasic_Uniform::compile()
{
    try {
        // The shader for the volumes
        volumeProg.compileShader("shader/basic_uniform.vert");
        volumeProg.compileShader("shader/basic_uniform.frag");
        volumeProg.compileShader("shader/basic_uniform.gs");
        volumeProg.link();

        // The shader for rendering and compositing
        renderProg.compileShader("shader/shadowvolume-render.vs");
        renderProg.compileShader("shader/shadowvolume-render.fs");
        renderProg.link();

        // The final composite shader
        compProg.compileShader("shader/shadowvolume-comp.vs");
        compProg.compileShader("shader/shadowvolume-comp.fs");
        compProg.link();

    } catch (GLSLProgramException &e) {
        cerr << e.what() << endl;
        exit(EXIT_FAILURE);
    }
}
```

Updating



```
void SceneBasic_Uniform::update( float t )
{
    float deltaT = t - tPrev;

    if (tPrev == 0.0f)
        deltaT = 0.0f;

    tPrev = t;

    if (animating())
    {
        angle += deltaT * rotSpeed;

        if (angle > glm::two_pi<float>())
            angle -= glm::two_pi<float>();

        updateLight();
    }
}
```

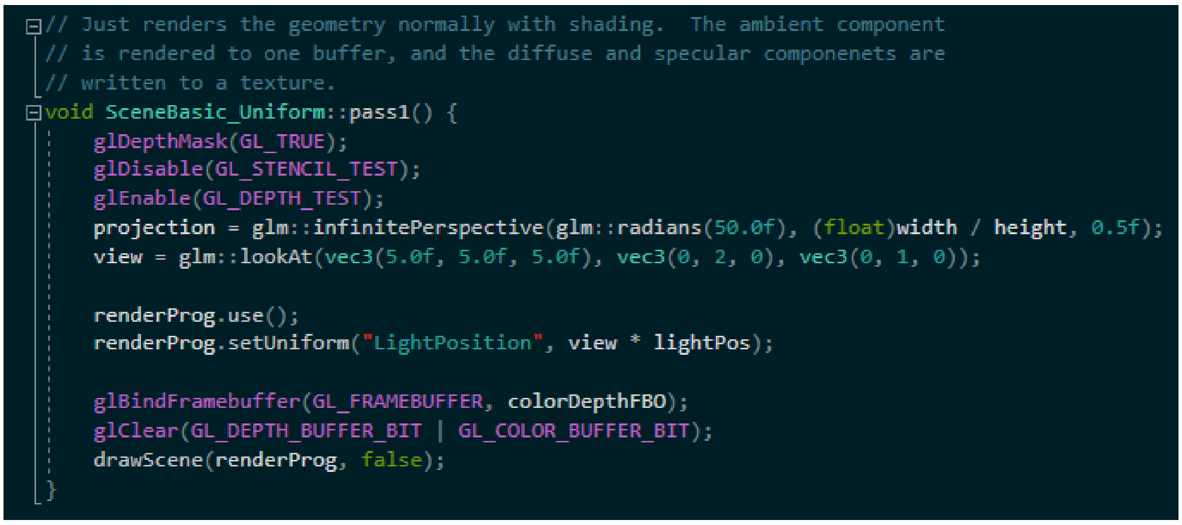
Rendering



```
void SceneBasic_Uniform::render()
{
    pass1();
    glFlush();
    pass2();
    glFlush();
    pass3();
}
```

First Pass

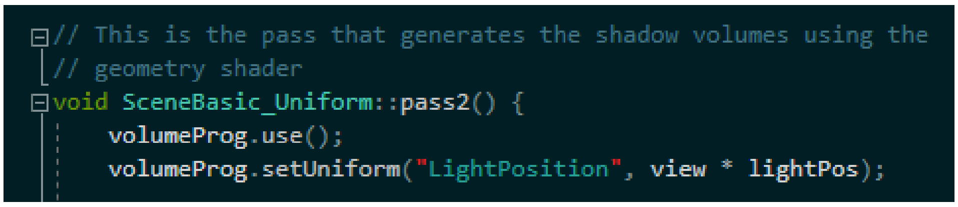
Render the scene normally, but write the shaded colour to two separate buffers. The ambient component will be stored in one buffer & the diffuse & specular components will be stored in the other.



```cpp
// Just renders the geometry normally with shading.  The ambient component
// is rendered to one buffer, and the diffuse and specular componenets are
// written to a texture.
void SceneBasic_Uniform::pass1() {
    glDepthMask(GL_TRUE);
    glDisable(GL_STENCIL_TEST);
    glEnable(GL_DEPTH_TEST);
    projection = glm::infinitePerspective(glm::radians(50.0f), (float)width / height, 0.5f);
    view = glm::lookAt(vec3(5.0f, 5.0f, 5.0f), vec3(0, 2, 0), vec3(0, 1, 0));

    renderProg.use();
    renderProg.setUniform("LightPosition", view * lightPos);

    glBindFramebuffer(GL_FRAMEBUFFER, colorDepthFBO);
    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
    drawScene(renderProg, false);
}
```

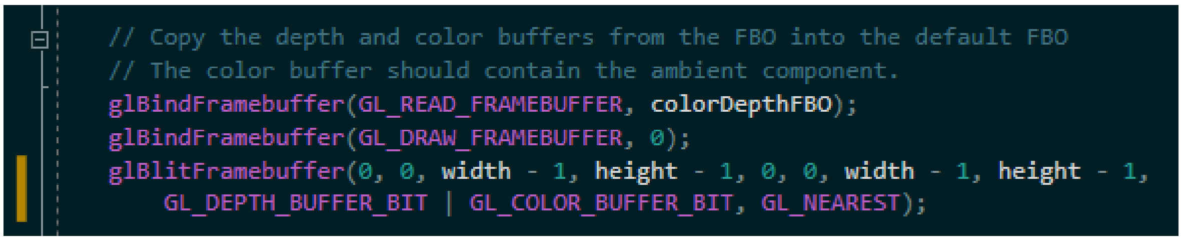
Second Pass

Volume Shader Program



```cpp
// This is the pass that generates the shadow volumes using the
// geometry shader
void SceneBasic_Uniform::pass2() {
    volumeProg.use();
    volumeProg.setUniform("LightPosition", view * lightPos);
```

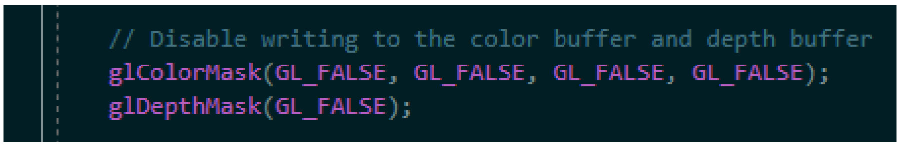
Buffer Copying



```cpp
    // Copy the depth and color buffers from the FBO into the default FBO
    // The color buffer should contain the ambient component.
    glBindFramebuffer(GL_READ_FRAMEBUFFER, colorDepthFBO);
    glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
    glBlitFramebuffer(0, 0, width - 1, height - 1, 0, 0, width - 1, height - 1,
        GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT, GL_NEAREST);
```

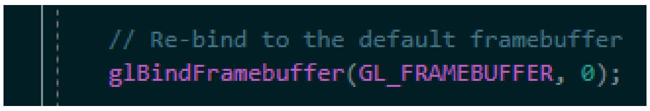
Disabling Buffer Writing

The depth-buffer should be read-only & only shadow-casting objects should be rendered.



```
    // Disable writing to the color buffer and depth buffer
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
    glDepthMask(GL_FALSE);
```

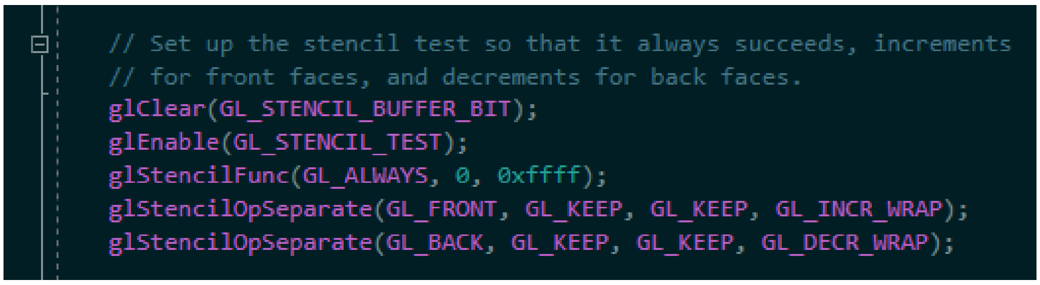
Default Framebuffer Rebinding



```
    // Re-bind to the default framebuffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Stencil Test Setup

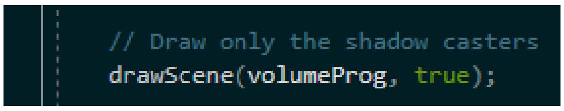
The stencil buffer should be setup to ensure that the stencil test always passes & that front faces cause an increment & back faces cause a decrement.



```
    // Set up the stencil test so that it always succeeds, increments
    // for front faces, and decrements for back faces.
    glClear(GL_STENCIL_BUFFER_BIT);
    glEnable(GL_STENCIL_TEST);
    glStencilFunc(GL_ALWAYS, 0, 0xffff);
    glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_KEEP, GL_INCR_WRAP);
    glStencilOpSeparate(GL_BACK, GL_KEEP, GL_KEEP, GL_DECR_WRAP);
```

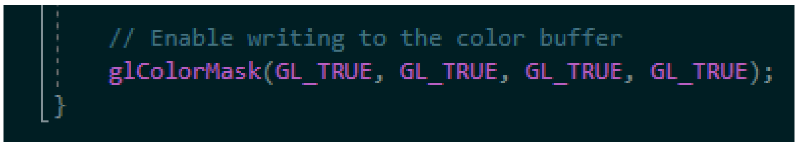
Scene Drawing

The shadow volumes will be produced & only the shadow volumes will be rendered to the fragment shader.



```
    // Draw only the shadow casters
    drawScene(volumeProg, true);
```

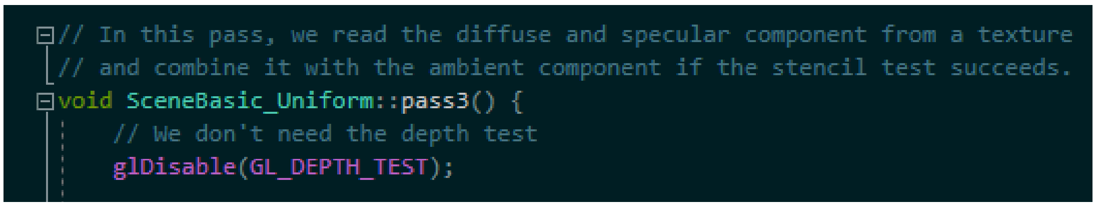
Re-Enabling Buffer Writing



```
    // Enable writing to the color buffer
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
}
```

Third Pass

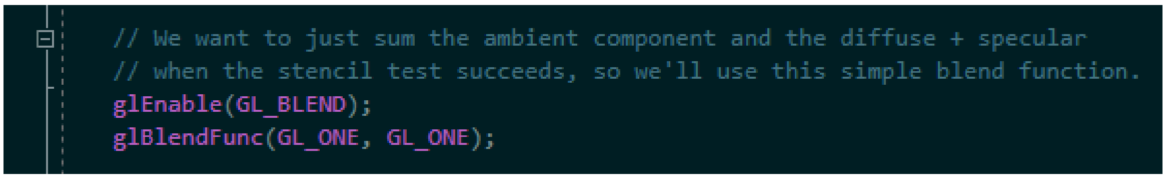
Disabling Depth Test



```
// In this pass, we read the diffuse and specular component from a texture
// and combine it with the ambient component if the stencil test succeeds.
void SceneBasic_Uniform::pass3() {
    // We don't need the depth test
    glDisable(GL_DEPTH_TEST);
```

Blending

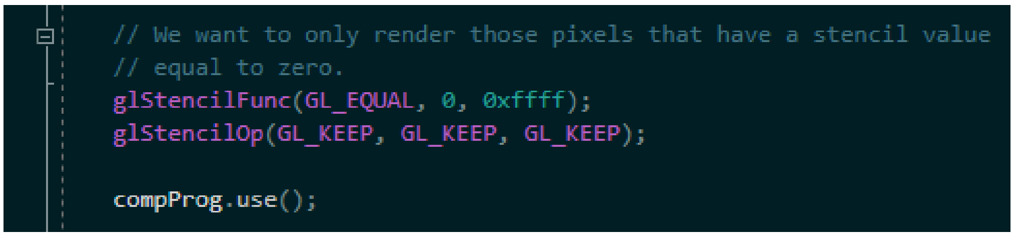
The values of both buffers used in the first pass should be combined in the case where the stencil test is successful.



```
    // We want to just sum the ambient component and the diffuse + specular
    // when the stencil test succeeds, so we'll use this simple blend function.
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);
```

Stencil Value Rendering Condition

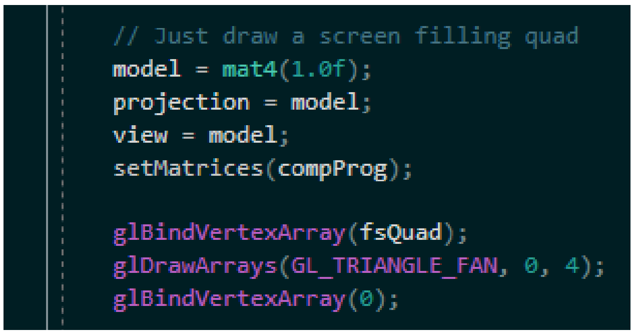
The stencil buffer should be setup so that the test succeeds only when the given value is equal to 0.



```
    // We want to only render those pixels that have a stencil value
    // equal to zero.
    glStencilFunc(GL_EQUAL, 0, 0xffff);
    glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

    compProg.use();
```

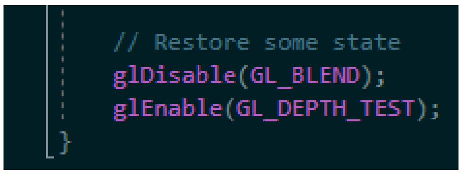
Drawing of Screen Filling Quad

A screen-filling quad needs to be drawn.



```
    // Just draw a screen filling quad
    model = mat4(1.0f);
    projection = model;
    view = model;
    setMatrices(compProg);

    glBindVertexArray(fsQuad);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    glBindVertexArray(0);
```

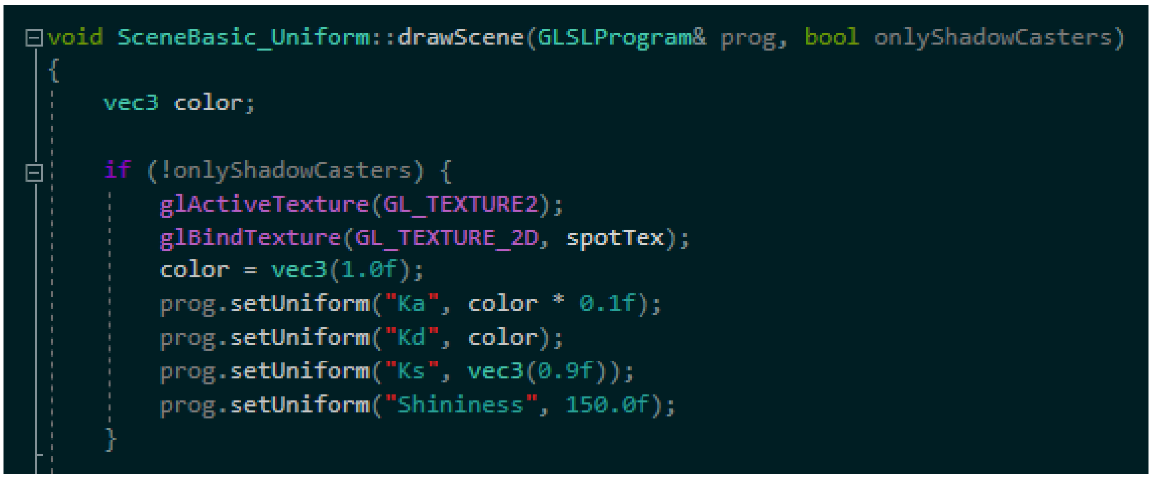
Restoring of Some State



```
    // Restore some state
    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);
}
```

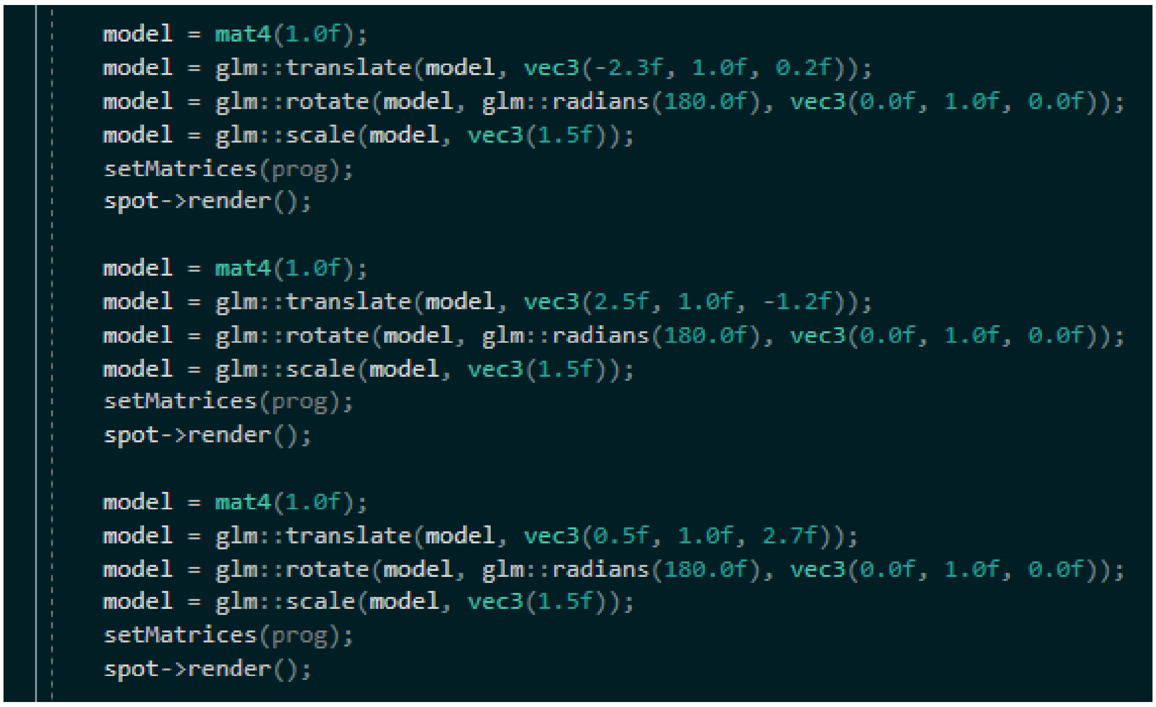
Scene Drawing

Texture & Lighting of Cows



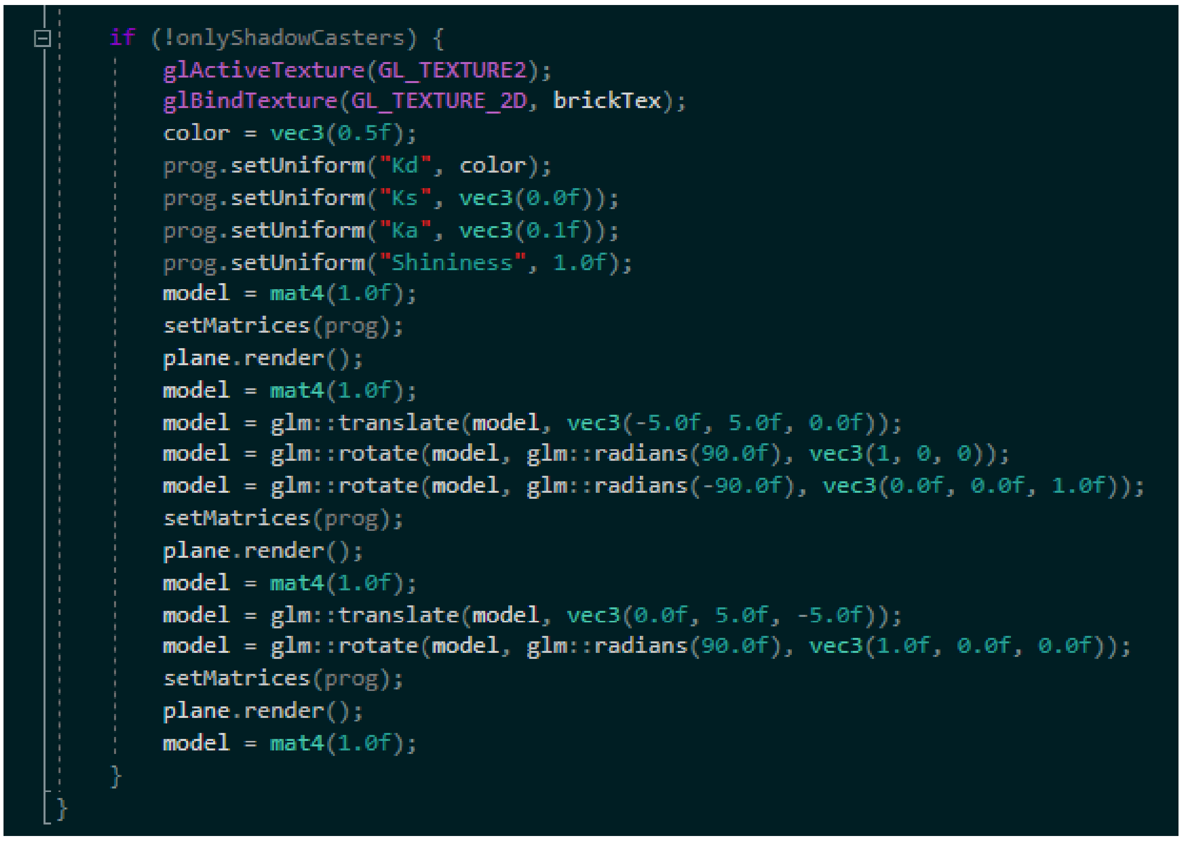
```
void SceneBasic_Uniform::drawScene(GLSLProgram& prog, bool onlyShadowCasters)
{
    vec3 color;

    if (!onlyShadowCasters) {
        glActiveTexture(GL_TEXTURE2);
        glBindTexture(GL_TEXTURE_2D, spotTex);
        color = vec3(1.0f);
        prog.setUniform("Ka", color * 0.1f);
        prog.setUniform("Kd", color);
        prog.setUniform("Ks", vec3(0.9f));
        prog.setUniform("Shininess", 150.0f);
    }
```

Rendering of Cows



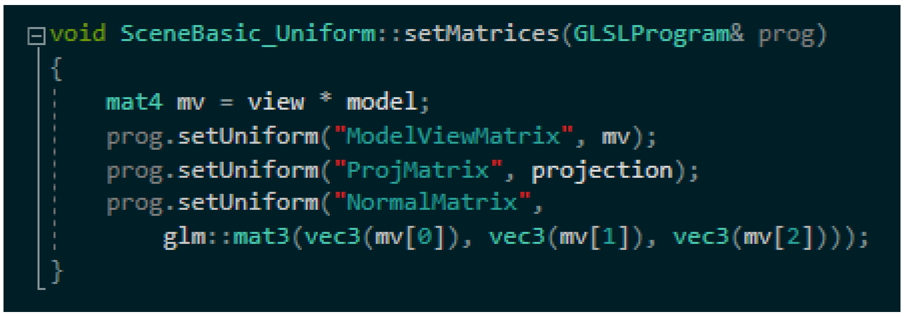
```
    model = mat4(1.0f);
    model = glm::translate(model, vec3(-2.3f, 1.0f, 0.2f));
    model = glm::rotate(model, glm::radians(180.0f), vec3(0.0f, 1.0f, 0.0f));
    model = glm::scale(model, vec3(1.5f));
    setMatrices(prog);
    spot->render();

    model = mat4(1.0f);
    model = glm::translate(model, vec3(2.5f, 1.0f, -1.2f));
    model = glm::rotate(model, glm::radians(180.0f), vec3(0.0f, 1.0f, 0.0f));
    model = glm::scale(model, vec3(1.5f));
    setMatrices(prog);
    spot->render();

    model = mat4(1.0f);
    model = glm::translate(model, vec3(0.5f, 1.0f, 2.7f));
    model = glm::rotate(model, glm::radians(180.0f), vec3(0.0f, 1.0f, 0.0f));
    model = glm::scale(model, vec3(1.5f));
    setMatrices(prog);
    spot->render();
```

Texture, Lighting & Rendering of Planes



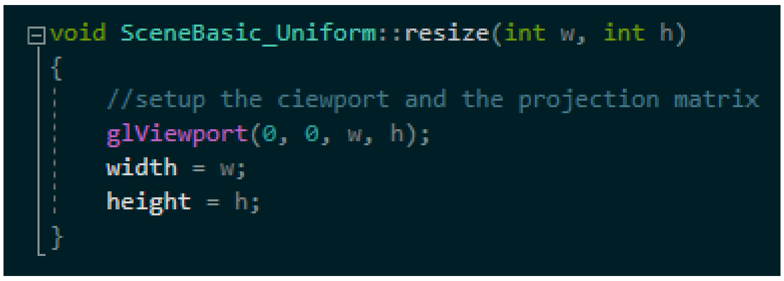
```
    if (!onlyShadowCasters) {
        glActiveTexture(GL_TEXTURE2);
        glBindTexture(GL_TEXTURE_2D, brickTex);
        color = vec3(0.5f);
        prog.setUniform("Kd", color);
        prog.setUniform("Ks", vec3(0.0f));
        prog.setUniform("Ka", vec3(0.1f));
        prog.setUniform("Shininess", 1.0f);
        model = mat4(1.0f);
        setMatrices(prog);
        plane.render();
        model = mat4(1.0f);
        model = glm::translate(model, vec3(-5.0f, 5.0f, 0.0f));
        model = glm::rotate(model, glm::radians(90.0f), vec3(1, 0, 0));
        model = glm::rotate(model, glm::radians(-90.0f), vec3(0.0f, 0.0f, 1.0f));
        setMatrices(prog);
        plane.render();
        model = mat4(1.0f);
        model = glm::translate(model, vec3(0.0f, 5.0f, -5.0f));
        model = glm::rotate(model, glm::radians(90.0f), vec3(1.0f, 0.0f, 0.0f));
        setMatrices(prog);
        plane.render();
        model = mat4(1.0f);
    }
}
```

Setting Matrices



```
void SceneBasic_Uniform::setMatrices(GLSLProgram& prog)
{
    mat4 mv = view * model;
    prog.setUniform("ModelViewMatrix", mv);
    prog.setUniform("ProjMatrix", projection);
    prog.setUniform("NormalMatrix",
        glm::mat3(vec3(mv[0]), vec3(mv[1]), vec3(mv[2])));
}
```

Resizing



```
void SceneBasic_Uniform::resize(int w, int h)
{
    //setup the ciewport and the projection matrix
    glViewport(0, 0, w, h);
    width = w;
    height = h;
}
```

Appendix

Shadow Maps with Soft Edges

Shading with Shadow

```
subroutine (RenderPassType)
void shadeWithShadow()
{
  vec3 ambient = Light.Intensity * Material.Ka;
  vec3 diffAndSpec = phongModelDiffAndSpec();

  ivec3 offsetCoord;
  offsetCoord.xy = ivec2( mod( gl_FragCoord.xy, OffsetTexSize.xy ) );

  float sum = 0.0, shadow = 1.0;
  int samplesDiv2 = int(OffsetTexSize.z);
  vec4 sc = ShadowCoord;

  // Don't test points behind the light source.
  if( sc.z >= 0 ) {
    for( int i = 0 ; i < 4; i++ ) {
      offsetCoord.z = i;
      vec4 offsets = texelFetch(OffsetTex,offsetCoord,0) * Radius * ShadowCoord.w;

      sc.xy = ShadowCoord.xy + offsets.xy;
      sum += textureProj(ShadowMap, sc);
      sc.xy = ShadowCoord.xy + offsets.zw;
      sum += textureProj(ShadowMap, sc);
    }
    shadow = sum / 8.0;

    if( shadow != 1.0 && shadow != 0.0 ) {
      for( int i = 4; i < samplesDiv2; i++ ) {
        offsetCoord.z = i;
        vec4 offsets = texelFetch(OffsetTex, offsetCoord,0) * Radius * ShadowCoord.w;

        sc.xy = ShadowCoord.xy + offsets.xy;
        sum += textureProj(ShadowMap, sc);
        sc.xy = ShadowCoord.xy + offsets.zw;
        sum += textureProj(ShadowMap, sc);
      }
      shadow = sum / float(samplesDiv2 * 2.0);
    }
  }

  FragColor = vec4(diffAndSpec * shadow + ambient, 1.0);

  // Gamma correct
  FragColor = pow( FragColor, vec4(1.0 / 2.2) );
}
```

Jitter Texture Building

```
void SceneBasic_Uniform::buildJitterTex()
{
    int size = jitterMapSize;
    int samples = samplesU * samplesV;
    int bufSize = size * size * samples * 2;
    float* data = new float[bufSize];

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            for (int k = 0; k < samples; k += 2) {
                int x1, y1, x2, y2;
                x1 = k % (samplesU);
                y1 = (samples - 1 - k) / samplesU;
                x2 = (k + 1) % samplesU;
                y2 = (samples - 1 - k - 1) / samplesU;

                vec4 v;
                // Center on grid and jitter
                v.x = (x1 + 0.5f) + jitter();
                v.y = (y1 + 0.5f) + jitter();
                v.z = (x2 + 0.5f) + jitter();
                v.w = (y2 + 0.5f) + jitter();

                // Scale between 0 and 1
                v.x /= samplesU;
                v.y /= samplesV;
                v.z /= samplesU;
                v.w /= samplesV;

                // Warp to disk
                int cell = ((k / 2) * size * size + j * size + i) * 4;
                data[cell + 0] = sqrtf(v.y) * cosf(glm::two_pi<float>() * v.x);
                data[cell + 1] = sqrtf(v.y) * sinf(glm::two_pi<float>() * v.x);
                data[cell + 2] = sqrtf(v.w) * cosf(glm::two_pi<float>() * v.z);
                data[cell + 3] = sqrtf(v.w) * sinf(glm::two_pi<float>() * v.z);
            }
        }
    }

    glActiveTexture(GL_TEXTURE1);
    GLuint texID;
    glGenTextures(1, &texID);

    glBindTexture(GL_TEXTURE_3D, texID);
    glTexStorage3D(GL_TEXTURE_3D, 1, GL_RGBA32F, size, size, samples / 2);
    glTexSubImage3D(GL_TEXTURE_3D, 0, 0, 0, 0, size, size, samples / 2, GL_RGBA, GL_FLOAT, data);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

    delete[] data;
}
```