Image Processing Techniques

COMP3015 Lab 5

Edge Detection

Setup

Download the ‘Additional files’ folder & unzip it. Next, add its contained ‘bs_ears’ file to your project’s media folder. In addition, add the ‘random.h’ & sphere cpp & header files to your helper folder & add them to your project solution.

Vertex Shader

Prerequisites

The vertex shader requires the same standard code that has been used prior, with exception to code for edge detection & texture coordinates. Ensure two input vec3 variables named ‘VertexPosition’ & ‘VertexNormal’ are present, as well as two vec3 output variables named ‘Position’ & ‘Normal’ that will be passed to the fragment shader.

Main Function

Transform the normal & position to eye coordinates, & setup ‘gl_Position’ for clipping stage:

```
gl_Position = MVP * vec4(VertexPosition,1.0);
```

Fragment Shader

Global Variables

Two input vec3 variables named ‘Position’ & ‘Normal’ are required. Additionally, a uniform named ‘RenderTex’ of type sampler2D with a layout index of 0 is necessary. We also need two other uniforms, one of type float named ‘EdgeThreshold’ & the other of type int named ‘Pass.’

```
layout( binding=0 ) uniform sampler2D RenderTex;

uniform float EdgeThreshold;

uniform int Pass;
```

Setup the Light & Material uniform structs. The ‘Light’ struct needs 3 variables, one of type vec4 named ‘Position’ & two of type vec3 named ‘L’ (diffuse & specular lighting) & ‘La’ (ambient lighting). The ‘Material’ struct needs all the elements used within the Blinn-Phong lighting model. Lastly, we need an output variable of type vec4 named ‘FragColor’ with a layout index of location 0.

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

Add a constant named ‘lum’ for calculating the luminance:

```
const vec3 lum = vec3(0.2126, 0.7152, 0.0722);
```

Main Function

Two types of passes need to be executed.

```
void main()

{

        if( Pass == 1 ) FragColor = pass1();

        if( Pass == 2 ) FragColor = pass2();

}
```

Pass 1 Function

The first pass calculates the Blinn-Phong lighting & the result is assigned to a RenderTexture.

```
vec4 pass1()

{

        return vec4(blinnPhong( Position, normalize(Normal) ),1.0);

}
```

Pass 2 Function

The second pass calculates the luminance of each pixel & determines if any given pixel is on an edge or not. Pixels on the edge are rendered white, otherwise they are rendered black.

```
vec4 pass2()

{

        ivec2 pix = ivec2(gl_FragCoord.xy); //we grab a pixel to check if edge

        //pick neighbouring pixels for convolution filter

        //check lecture slides

        float s00 = luminance(texelFetchOffset(RenderTex, pix, 0, ivec2(-1,1)).rgb);

        float s10 = luminance(texelFetchOffset(RenderTex, pix, 0, ivec2(-1,0)).rgb);

        float s20 = luminance(texelFetchOffset(RenderTex, pix, 0, ivec2(-1,-1)).rgb);

        float s01 = luminance(texelFetchOffset(RenderTex, pix, 0, ivec2(0,1)).rgb);

        float s21 = luminance(texelFetchOffset(RenderTex, pix, 0, ivec2(0,-1)).rgb);

        float s02 = luminance(texelFetchOffset(RenderTex, pix, 0, ivec2(1,1)).rgb);

        float s12 = luminance(texelFetchOffset(RenderTex, pix, 0, ivec2(1,0)).rgb);

        float s22 = luminance(texelFetchOffset(RenderTex, pix, 0, ivec2(1,-1)).rgb);

        float sx = s00 + 2 * s10 + s20 - (s02 + 2 * s12 + s22);

        float sy = s00 + 2 * s01 + s02 - (s20 + 2 * s21 + s22);

        float g = sx * sx + sy * sy;

        if( g > EdgeThreshold )

        return vec4(1.0); //edge

        else

        return vec4(0.0,0.0,0.0,1.0); //no edge

}
```

Luminance Function

```
float luminance( vec3 color )
{
        return dot(lum,color);
}
```

Scenebasic Uniform Header

3 private ‘Gluint’ variables named ‘fsQuad,’ ‘fboHandle’ & ‘renderTex’ are required for setting up elements: In addition, declare plane, torus & teapot objects privately & 3 private functions named ‘setupFBO(),’ ‘pass1()’ & ‘pass2():’

```
GLuint fsQuad, fboHandle, renderTex;
Plane plane;
Torus torus;
Teapot teapot;
void setupFBO();
void pass1();
void pass2();
```

Your private section should look something like this:

```
GLSLProgram prog;
GLuint fsQuad, fboHandle, renderTex;
Plane plane;
Torus torus;
Teapot teapot;
float angle;
float tPrev, rotSpeed;
void setMatrices();
void compile();
void setupFBO();
void pass1();
void pass2();
```

Scenebasic Uniform CPP

Constructor

Initialise your plane, teapot and torus & setup a rotation speed if you want the camera to rotate:

```
angle(0.0f), tPrev(0.0f), rotSpeed(glm::pi<float>() / 8.0f), plane(50.0f, 50.0f, 1, 1), teapot(14, mat4(1.0f)), torus(0.7f * 1.5f, 0.3f * 1.5f, 50, 50)
```

Init Scene Function

Initialise all the elements needed for setting up the texture, including a quad for rendering. The code should look something like this:

Compile, FBO Setup, ETC

```
compile();

glClearColor(1.0f, 0.0f, 0.0f, 1.0f);

glEnable(GL_DEPTH_TEST);

projection = mat4(1.0f);

angle = glm::pi<float>() / 4.0f;

setupFBO();
```

Full Screen Quad

```
// Array for 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, 1.0f, 1.0f, 0.0f, -1.0f, 1.0f, 0.0f

};

GLfloat tc[] = {

        0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f,

        0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f

};
```

Buffer Setup

```
// Set up the buffers

unsigned int handle[2];

glGenBuffers(2, handle);

glBindBuffer(GL_ARRAY_BUFFER, handle[0]);

glBufferData(GL_ARRAY_BUFFER, 6 * 3 * sizeof(float), verts, GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, handle[1]);

glBufferData(GL_ARRAY_BUFFER, 6 * 2 * sizeof(float), tc, GL_STATIC_DRAW);
```

Vertex Array Object Setup

```
// Set up the vertex array object

glGenVertexArrays(1, &fsQuad);

glBindVertexArray(fsQuad);

glBindBuffer(GL_ARRAY_BUFFER, handle[0]);

glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(0); // Vertex position

glBindBuffer(GL_ARRAY_BUFFER, handle[1]);

glVertexAttribPointer((GLuint)2, 2, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(2); // Texture coordinates

glBindVertexArray(0);

prog.setUniform("EdgeThreshold", 0.05f);

prog.setUniform("Light.L", vec3(1.0f));

prog.setUniform("Light.La", vec3(0.2f));
```

FBO Setup Function

```
void SceneBasic_Uniform::setupFBO()

{

        // Generate and bind the framebuffer

        glGenFramebuffers(1, &fboHandle);

        glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);

        // Create the texture object

        glGenTextures(1, &renderTex);

        glBindTexture(GL_TEXTURE_2D, renderTex);

        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);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);

        // Bind the texture to the FBO

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, renderTex, 0);

        // Create the depth buffer

        GLuint depthBuf;

        glGenRenderbuffers(1, &depthBuf);

        glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);

        glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);

        // Bind the depth buffer to the FBO

        glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,

        GL_RENDERBUFFER, depthBuf);

        // Set the targets for the fragment output variables

        GLenum drawBuffers[] = { GL_COLOR_ATTACHMENT0 };

        glDrawBuffers(1, drawBuffers);

        // Unbind the framebuffer, and revert to default framebuffer

        glBindFramebuffer(GL_FRAMEBUFFER, 0);

}
```

Update Function

Use this if you want to animate the camera (use the angle update):

```
void SceneBasic_Uniform::update( float t )
{
        float deltaT = t - tPrev;
        if (tPrev == 0.0f)
        deltaT = 0.0f;
        tPrev = t;
        angle += rotSpeed * deltaT;
        if (angle > glm::two_pi<float>())
        angle -= glm::two_pi<float>();
}
```

Render Function

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

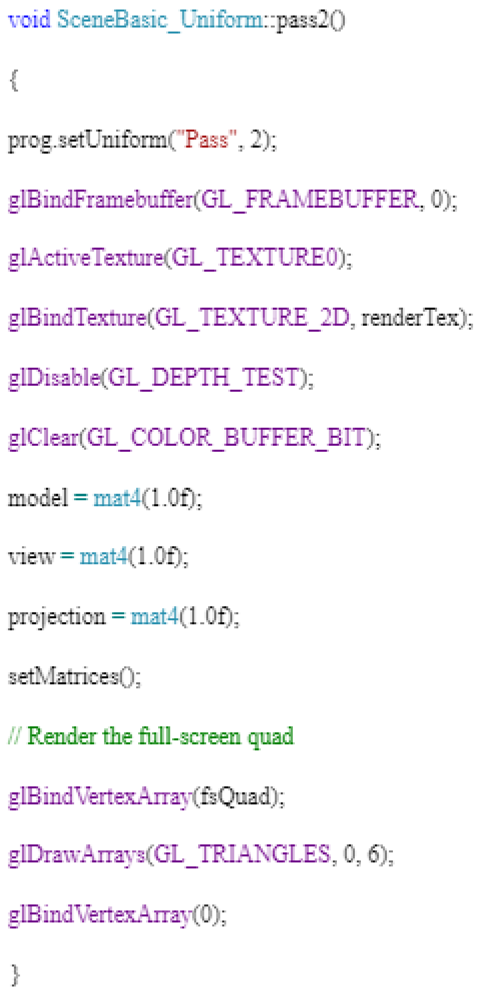
Pass 1 Function

Set the uniforms for the Blinn-Phong lighting for the teapot, torus & plane models & render it to the texture:

```
prog.setUniform("Pass", 1);
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);
glEnable(GL_DEPTH_TEST);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
view = glm::lookAt(vec3(7.0f * cos(angle), 4.0f, 7.0f * sin(angle)), vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 1.0f, 0.0f));
projection = glm::perspective(glm::radians(60.0f), (float)width / height, 0.3f, 100.0f);
prog.setUniform("Light.Position", vec4(0.0f, 0.0f, 0.0f, 1.0f));
prog.setUniform("Material.Kd", 0.9f, 0.9f, 0.9f);
prog.setUniform("Material.Ks", 0.95f, 0.95f, 0.95f);
prog.setUniform("Material.Ka", 0.1f, 0.1f, 0.1f);
prog.setUniform("Material.Shininess", 100.0f);
model = mat4(1.0f);
model = glm::rotate(model, glm::radians(-90.0f), vec3(1.0f, 0.0f, 0.0f));
setMatrices();
teapot.render();
prog.setUniform("Material.Kd", 0.4f, 0.4f, 0.4f);
prog.setUniform("Material.Ks", 0.0f, 0.0f, 0.0f);
prog.setUniform("Material.Ka", 0.1f, 0.1f, 0.1f);
prog.setUniform("Material.Shininess", 1.0f);
model = mat4(1.0f);
model = glm::translate(model, vec3(0.0f, -0.75f, 0.0f));
setMatrices();
plane.render();
prog.setUniform("Light.Position", vec4(0.0f, 0.0f, 0.0f, 1.0f));
prog.setUniform("Material.Kd", 0.9f, 0.5f, 0.2f);
prog.setUniform("Material.Ks", 0.95f, 0.95f, 0.95f);
prog.setUniform("Material.Ka", 0.1f, 0.1f, 0.1f);
prog.setUniform("Material.Shininess", 100.0f);
model = mat4(1.0f);
model = glm::translate(model, vec3(1.0f, 1.0f, 3.0f));
model = glm::rotate(model, glm::radians(90.0f), vec3(1.0f, 0.0f, 0.0f));
setMatrices();
torus.render();
```

Pass 2 Function

Render a full screen quad with our texture in place. Notably, this occurs after the pixels have been rendered in either black or white:



```
void SceneBasic_Uniform::pass2()

{

prog.setUniform("Pass", 2);

glBindFramebuffer(GL_FRAMEBUFFER, 0);

glActiveTexture(GL_TEXTURE0);

glBindTexture(GL_TEXTURE_2D, renderTex);

glDisable(GL_DEPTH_TEST);

glClear(GL_COLOR_BUFFER_BIT);

model = mat4(1.0f);

view = mat4(1.0f);

projection = mat4(1.0f);

setMatrices();

// Render the full-screen quad

glBindVertexArray(fsQuad);

glDrawArrays(GL_TRIANGLES, 0, 6);

glBindVertexArray(0);

}
```

Gaussian Blur

Vertex Shader

The vertex shader implementation for Gaussian blurring is identical to that of the Edge Detection vertex shader implementation.

Fragment Shader

Global Variables

The fragment shader implementation is similar to the Edge Detection shader. It makes use of 3 passes, as opposed to 2. The following code uses the name ‘Texture0’ for the texture variable, instead of the name ‘RenderTexture’ as in the case of the edge detection code. The texture variable may be renamed for consistency with the following code if desired. Regardless, we need a new uniform array of type float:

```
uniform float Weight[5];
```

Main Function

```
void main()

{

        if( Pass == 1 )

                FragColor = pass1();

        else if( Pass == 2 )

                FragColor = pass2();

        else if( Pass == 3 )

                FragColor = pass3();

}
```

Pass 1 Function

The first pass function is identical to the previous implementation; just calculate the Blinn-Phong model.

Pass 2 Function

```
vec4 pass2()

{

        ivec2 pix = ivec2( gl_FragCoord.xy );

        vec4 sum = texelFetch(Texture0, pix, 0) * Weight[0];

        sum += texelFetchOffset( Texture0, pix, 0, ivec2(0,1) ) * Weight[1];

        sum += texelFetchOffset( Texture0, pix, 0, ivec2(0,-1) ) * Weight[1];

        sum += texelFetchOffset( Texture0, pix, 0, ivec2(0,2) ) * Weight[2];

        sum += texelFetchOffset( Texture0, pix, 0, ivec2(0,-2) ) * Weight[2];

        sum += texelFetchOffset( Texture0, pix, 0, ivec2(0,3) ) * Weight[3];

        sum += texelFetchOffset( Texture0, pix, 0, ivec2(0,-3) ) * Weight[3];

        sum += texelFetchOffset( Texture0, pix, 0, ivec2(0,4) ) * Weight[4];

        sum += texelFetchOffset( Texture0, pix, 0, ivec2(0,-4) ) * Weight[4];

        return sum;

}
```

Pass 3 Function

```
vec4 pass3()
{
        ivec2 pix = ivec2( gl_FragCoord.xy );
        vec4 sum = texelFetch(Texture0, pix, 0) * Weight[0];
        sum += texelFetchOffset( Texture0, pix, 0, ivec2(1,0) ) * Weight[1];
        sum += texelFetchOffset( Texture0, pix, 0, ivec2(-1,0) ) * Weight[1];
        sum += texelFetchOffset( Texture0, pix, 0, ivec2(2,0) ) * Weight[2];
        sum += texelFetchOffset( Texture0, pix, 0, ivec2(-2,0) ) * Weight[2];
        sum += texelFetchOffset( Texture0, pix, 0, ivec2(3,0) ) * Weight[3];
        sum += texelFetchOffset( Texture0, pix, 0, ivec2(-3,0) ) * Weight[3];
        sum += texelFetchOffset( Texture0, pix, 0, ivec2(4,0) ) * Weight[4];
        sum += texelFetchOffset( Texture0, pix, 0, ivec2(-4,0) ) * Weight[4];
        return sum;
}
```

Scenebasic Uniform Header

Your private section should look like the following:

```
private:

GLSLProgram prog;

GLuint fsQuad;

GLuint renderFBO, intermediateFBO;

GLuint renderTex, intermediateTex;

Plane plane;

Torus torus;

Teapot teapot;

float angle;

float tPrev, rotSpeed;

void setMatrices();

void compile();

void setupFBO();

void pass1();

void pass2();

void pass3();

float gauss(float, float);
```

Scenebasic Uniform CPP

Init Scene Function

Replace the code for initialising the ‘EdgeThreshold,’ ‘Light.L’ & ‘Light.La’ uniform variables with the following:

```
prog.setUniform("Light.L", vec3(1.0f));

prog.setUniform("Light.La", vec3(0.2f));

float weights[5], sum, sigma2 = 8.0f;

// Compute and sum the weights

weights[0] = gauss(0, sigma2);

sum = weights[0];

for (int i = 1; i < 5; i++) {

        weights[i] = gauss(float(i), sigma2);

        sum += 2 * weights[i];

}

// Normalize the weights and set the uniform

for (int i = 0; i < 5; i++) {

        std::stringstream uniName;

        uniName << "Weight[" << i << "]";

        float val = weights[i] / sum;

        prog.setUniform(uniName.str().c_str(), val);
```

FBO Setup Function

The principle of the FBO setup function is still applied, however it is applied to making use of 2 textures & FBOs:

```
// Generate and bind the framebuffer
glGenFramebuffers(1, &renderFBO);
glBindFramebuffer(GL_FRAMEBUFFER, renderFBO);
// Create the texture object
glGenTextures(1, &renderTex);
glBindTexture(GL_TEXTURE_2D, renderTex);
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);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);
// Bind the texture to the FBO
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, renderTex, 0);
// Create the depth buffer
GLuint depthBuf;
glGenRenderbuffers(1, &depthBuf);
glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);
// Bind the depth buffer to the FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, depthBuf);
// Set the targets for the fragment output variables
GLenum drawBuffers[] = { GL_COLOR_ATTACHMENT0 };
glDrawBuffers(1, drawBuffers);
// Unbind the framebuffer, and revert to default framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
// Generate and bind the framebuffer
glGenFramebuffers(1, &intermediateFBO);
glBindFramebuffer(GL_FRAMEBUFFER, intermediateFBO);
// Create the texture object
glGenTextures(1, &intermediateTex);
glActiveTexture(GL_TEXTURE0); // Use texture unit 0
glBindTexture(GL_TEXTURE_2D, intermediateTex);
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);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);
// Bind the texture to the FBO
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, intermediateTex, 0);
// No depth buffer needed for this FBO
// Set the targets for the fragment output variables
glDrawBuffers(1, drawBuffers);
// Unbind the framebuffer, and revert to default framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Render Function

Call all 3 passes in order:

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

Pass 1 Function

The pass 1 function is identical to the one used in the edge detection implementation, with the exception that the ‘renderFBO’ is used to write to the texture.

Pass 2 Function

Use the ‘renderTex’ & ‘intermediateFBO:’

```
void SceneBasic_Uniform::pass2()
{
    prog.setUniform("Pass", 2);
    glBindFramebuffer(GL_FRAMEBUFFER, intermediateFBO);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, renderTex);
    glDisable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT);
    model = mat4(1.0f);
    view = mat4(1.0f);
    projection = mat4(1.0f);
    setMatrices();
    // Render the full-screen quad
    glBindVertexArray(fsQuad);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

Pass 3 Function

Use ‘intermediateTex’ & the default buffer:

```
void SceneBasic_Uniform::pass3()
{
    prog.setUniform("Pass", 3);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, intermediateTex);
    glClear(GL_COLOR_BUFFER_BIT);
    model = mat4(1.0f);
    view = mat4(1.0f);
    projection = mat4(1.0f);
    setMatrices();
    // Render the full-screen quad
    glBindVertexArray(fsQuad);
    glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

Gaussian Blur Function

```
float SceneBasic_Uniform::gauss(float x, float sigma2)
{
    double coeff = 1.0 / (glm::two_pi<double>() * sigma2);
    double expon = -(x * x) / (2.0 * sigma2);
    return (float)(coeff * exp(expon));
}
```

HDR with Tone Mapping

Vertex Shader

The vertex shader is identical to the edge detection vertex shader implementation, with exception to that it needs to receive the vertex texture coordinates, pass them & output them to the fragment shader:

```
layout( location = 2 ) in vec2 VertexTexCoord;

out vec2 TexCoord;

TexCoord = VertexTexCoord;
```

Fragment Shader

Global Variables

3 input variables are required, a vec3 named ‘Position’ & 2 vec2 variables named ‘Normal’ & ‘TexCoord.’ Two output variables are also necessary, one vec4 at layout location 0 named ‘FragColor’ & one vec3 at layout location 1 named ‘HdrColor:’

```
in vec3 Position;

in vec3 Normal;

in vec2 TexCoord;

layout (location = 0) out vec4 FragColor;

layout (location = 1) out vec3 HdrColor;
```

Various additional uniforms are required:

```
uniform int Pass; // Pass number

uniform float AveLum;

layout(binding=0) uniform sampler2D HdrTex;

// XYZ/RGB conversion matrices from:

// http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.html

uniform mat3 rgb2xyz = mat3(

0.4124564, 0.2126729, 0.0193339,

0.3575761, 0.7151522, 0.1191920,

0.1804375, 0.0721750, 0.9503041 );

uniform mat3 xyz2rgb = mat3(

3.2404542, -0.9692660, 0.0556434,

-1.5371385, 1.8760108, -0.2040259,

-0.4985314, 0.0415560, 1.0572252 );

uniform float Exposure = 0.35;

uniform float White = 0.928;

uniform bool DoToneMap = true;
```

Main Function

```
void main()

{

        if( Pass == 1 )

        pass1();

        else if( Pass == 2)

        pass2();

}
```

Pass 1 Function

```
void pass1()
{
        vec3 n = normalize(Normal);
        // Compute shading and store result in high-res framebuffer
        HdrColor = vec3(0.0);
        for( int i = 0; i < 3; i++)
        HdrColor += blinnPhong(Position, n, i);
}
```

Pass 2 Function

```
// This pass computes the sum of the luminance of all pixels

void pass2()

{

        // Retrieve high-res color from texture

        vec4 color = texture( HdrTex, TexCoord );

        // Convert to XYZ

        vec3 xyzCol = rgb2xyz * vec3(color);

        // Convert to xyY

        float xyzSum = xyzCol.x + xyzCol.y + xyzCol.z;

        vec3 xyYCol = vec3( xyzCol.x / xyzSum, xyzCol.y / xyzSum, xyzCol.y);

        // Apply the tone mapping operation to the luminance (xyYCol.z or xyzCol.y)

        float L = (Exposure * xyYCol.z) / AveLum;

        L = (L * ( 1 + L / (White * White) )) / ( 1 + L );

        // Using the new luminance, convert back to XYZ

        xyzCol.x = (L * xyYCol.x) / (xyYCol.y);

        xyzCol.y = L;

        xyzCol.z = (L * (1 - xyYCol.x - xyYCol.y))/xyYCol.y;

        // Convert back to RGB and send to output buffer

        if( DoToneMap )

                FragColor = vec4( xyz2rgb * xyzCol, 1.0);

        else

                FragColor = color;

}
```

Scenebasic Uniform Header

The models that are being rendered are the plane, sphere & teapot. Therefore, your private section should appear like the following:

```
private:

GLSLProgram prog;

GLuint hdrFBO;

GLuint quad;

GLuint hdrTex, avgTex;

Plane plane;

Sphere sphere;

Teapot teapot;

void setMatrices();

void compile();

void setupFBO();

void pass1();

void pass2();

void computeLogAveLuminance();

void drawScene();
```

Scenebasic Uniform CPP

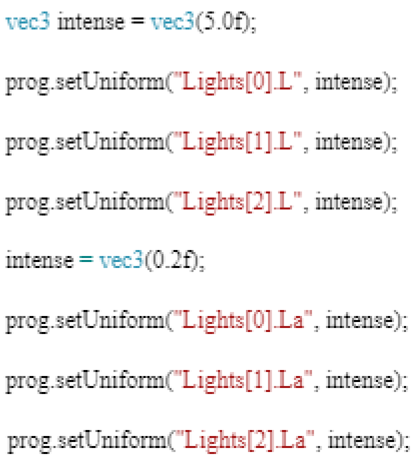
Constructor

Initialise your constructor with these values:

```
plane(20.0f, 50.0f, 1, 1), teapot(14, mat4(1.0f)), sphere(2.0f, 50, 50)
```

Init Scene Function

Setup the quad as in the previous implementations & add the settings of the uniforms after the call to the FBO setup function:



```
vec3 intense = vec3(5.0f);

prog.setUniform("Lights[0].L", intense);

prog.setUniform("Lights[1].L", intense);

prog.setUniform("Lights[2].L", intense);

intense = vec3(0.2f);

prog.setUniform("Lights[0].La", intense);

prog.setUniform("Lights[1].La", intense);

prog.setUniform("Lights[2].La", intense);
```

FBO Setup Function

```
void SceneBasic_Uniform::setupFBO()

{

GLuint depthBuf;

// Create and bind the FBO

glGenFramebuffers(1, &hdrFBO);

glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);

// The depth buffer

glGenRenderbuffers(1, &depthBuf);

glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);

glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);

// The HDR color buffer

glActiveTexture(GL_TEXTURE0);

glGenTextures(1, &hdrTex);

glBindTexture(GL_TEXTURE_2D, hdrTex);

glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGB32F, width, height);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

// Attach the images to the framebuffer

glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuf);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, hdrTex, 0);

GLenum drawBuffers[] = { GL_NONE, GL_COLOR_ATTACHMENT0 };

glDrawBuffers(2, drawBuffers);

glBindFramebuffer(GL_FRAMEBUFFER, 0);

}
```

Render Function

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

Pass 1 Function

Use the ‘hdrFBO’ & draw the scene with all the models:

```
void SceneBasic_Uniform::pass1()
{
    prog.setUniform("Pass", 1);
    glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
    glViewport(0, 0, width, height);
    glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);
    view = glm::lookAt(vec3(2.0f, 0.0f, 14.0f), vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 1.0f, 0.0f));
    projection = glm::perspective(glm::radians(60.0f), (float)width / height, 0.3f, 100.0f);
    drawScene();
}
```

Pass 2 Function

Move to the default buffer & render the quad:

```
void SceneBasic_Uniform::pass2()

{

        prog.setUniform("Pass", 2);

        // Revert to default framebuffer

        glBindFramebuffer(GL_FRAMEBUFFER, 0);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glDisable(GL_DEPTH_TEST);

        view = mat4(1.0);

        model = mat4(1.0);

        projection = mat4(1.0);

        setMatrices();

        // Render the quad

        glBindVertexArray(quad);

        glDrawArrays(GL_TRIANGLES, 0, 6);

}
```

Average Luminance Function

For computing the average luminance, a new function named ‘computeLogAveLuminance’ needs to be implemented as shown:

```
void SceneBasic_Uniform::computeLogAveLuminance()

{

    int size = width * height;

    std::vector<GLfloat> texData(size * 3);

    glActiveTexture(GL_TEXTURE0);

    glBindTexture(GL_TEXTURE_2D, hdrTex);

    glGetTexImage(GL_TEXTURE_2D, 0, GL_RGB, GL_FLOAT, texData.data());

    float sum = 0.0f;

    for (int i = 0; i < size; i++) {

        float lum = glm::dot(vec3(texData[i * 3 + 0], texData[i * 3 + 1], texData[i * 3 + 2]),

        vec3(0.2126f, 0.7152f, 0.0722f));

        sum += logf(lum + 0.00001f);

    }

    prog.setUniform("AveLum", expf(sum / size));

}
```

Draw Scene Function

Draw the models into the scene:

```
void SceneBasic_Uniform::drawScene()
{
vec3 intense = vec3(1.0f);
prog.setUniform("Lights[0].L", intense);
prog.setUniform("Lights[1].L", intense);
prog.setUniform("Lights[2].L", intense);
vec4 lightPos = vec4(0.0f, 4.0f, 2.5f, 1.0f);
lightPos.x = -7.0f;
prog.setUniform("Lights[0].Position", view * lightPos);
lightPos.x = 0.0f;
prog.setUniform("Lights[1].Position", view * lightPos);
lightPos.x = 7.0f;
prog.setUniform("Lights[2].Position", view * lightPos);
prog.setUniform("Material.Kd", 0.9f, 0.3f, 0.2f);
prog.setUniform("Material.Ks", 1.0f, 1.0f, 1.0f);
prog.setUniform("Material.Ka", 0.2f, 0.2f, 0.2f);
prog.setUniform("Material.Shininess", 100.0f);
// The backdrop plane
model = glm::rotate(mat4(1.0f), glm::radians(90.0f), vec3(1.0f, 0.0f, 0.0f));
setMatrices();
plane.render();
// The bottom plane
model = glm::translate(mat4(1.0f), vec3(0.0f, -5.0f, 0.0f));
setMatrices();
plane.render();
// Top plane
model = glm::translate(mat4(1.0f), vec3(0.0f, 5.0f, 0.0f));
model = glm::rotate(model, glm::radians(180.0f), vec3(1.0f, 0.0f, 0.0f));
setMatrices();
plane.render();
prog.setUniform("Material.Kd", vec3(0.4f, 0.9f, 0.4f));
model = glm::translate(mat4(1.0f), vec3(-3.0f, -3.0f, 2.0f));
setMatrices();
sphere.render();
prog.setUniform("Material.Kd", vec3(0.4f, 0.4f, 0.9f));
model = glm::translate(mat4(1.0f), vec3(3.0f, -5.0f, 1.5f));
model = glm::rotate(model, glm::radians(-90.0f), vec3(1.0f, 0.0f, 0.0f));
setMatrices();
teapot.render();
}
```

Bloom Effect

Overview

The bloom effect is achieved with 5 passes:

- Pass 1: Scene rendered to a HDR texture
- Pass 2: The bright-pass filter. Parts of the image that are brighter than a certain threshold value are extracted. Notably, this filter will make use of a downsampled resolution as this generates additional blurring when the buffer is read back using a linear sampler
- Pass 3 & 4: Application of the Gaussian blur to bright parts (refer to the Gaussian blur covered earlier)
- Pass 5: Application of tone mapping & add the result to the blurred bright-pass filter result

Vertex Shader

The vertex shader implementation is identical to that of the HDR tone mapping vertex shader implementation.

Fragment Shader

Global Variables

The input & output variables within the fragment shader are typical of those in previous implementations. Notably, it needs to include the Light & Material uniform structs. In addition to this, the following uniforms must be added as shown:

```
uniform int Pass; // Pass number

layout (binding=0) uniform sampler2D HdrTex;

layout (binding=1) uniform sampler2D BlurTex1;

layout (binding=2) uniform sampler2D BlurTex2;

uniform float LumThresh; // Luminance threshold

uniform float PixOffset[10] = float[](0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0);

uniform float Weight[10];

// XYZ/RGB conversion matrices from:

// http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.html

uniform mat3 rgb2xyz = mat3(

0.4124564, 0.2126729, 0.0193339,

0.3575761, 0.7151522, 0.1191920,

0.1804375, 0.0721750, 0.9503041 );

uniform mat3 xyz2rgb = mat3(

3.2404542, -0.9692660, 0.0556434,

-1.5371385, 1.8760108, -0.2040259,

-0.4985314, 0.0415560, 1.0572252 );

uniform float Exposure = 0.35;

uniform float White = 0.928;

uniform float AveLum;
```

Luminance Function

```
float luminance( vec3 color )

{

return 0.2126 * color.r + 0.7152 * color.g + 0.0722 * color.b;

}
```

Main Function

```
void main()
{
        if(Pass == 1)
                FragColor = pass1();
        else if(Pass == 2)
                FragColor = pass2();
        else if(Pass == 3)
                FragColor = pass3();
        else if(Pass == 4)
                FragColor = pass4();
        else if(Pass == 5)
                FragColor = pass5();
}
```

Render Pass Function (1)

```
// The render pass
vec4 pass1()
{
        vec3 n = normalize(Normal);
        vec3 color = vec3(0.0);
        for( int i = 0; i < 3; i++)
                color += blinnPhong(Position, n, i);
        return vec4(color,1);
}
```

Bright Pass Function (2)

```
// Bright-pass filter (write to BlurTex1)

vec4 pass2() {

        vec4 val = texture(HdrTex, TexCoord);

        if( luminance(val.rgb) > LumThresh )

                return val;

        else

                return vec4(0.0);

}
```

First Blur Pass Function (3)

```
// First blur pass (read from BlurTex1, write to BlurTex2)

vec4 pass3() {

        float dy = 1.0 / (textureSize(BlurTex1,0)).y;

        vec4 sum = texture(BlurTex1, TexCoord) * Weight[0];

        for( int i = 1; i < 10; i++ )

        {

                sum += texture( BlurTex1, TexCoord + vec2(0.0,PixOffset[i]) * dy ) * Weight[i];

                sum += texture( BlurTex1, TexCoord - vec2(0.0,PixOffset[i]) * dy ) * Weight[i];

        }

        return sum;

}
```

Second Blur Pass Function (4)

```
// Second blur (read from BlurTex2, write to BlurTex1)

vec4 pass4() {

        float dx = 1.0 / (textureSize(BlurTex2,0)).x;

        vec4 sum = texture(BlurTex2, TexCoord) * Weight[0];

        for( int i = 1; i < 10; i++ )

        {

                sum += texture( BlurTex2, TexCoord + vec2(PixOffset[i],0.0) * dx ) * Weight[i];

                sum += texture( BlurTex2, TexCoord - vec2(PixOffset[i],0.0) * dx ) * Weight[i];

        }

        return sum;

}
```

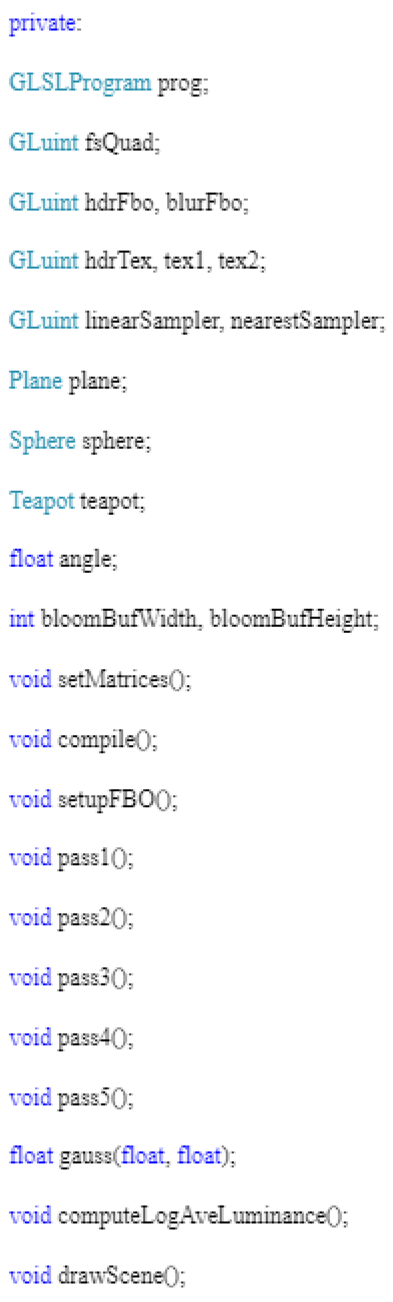
Composite Pass Function (5)

Apply the tone map to the HDR image & combine it with the blurred bright pass filter:

```
// (Read from BlurTex1 and HdrTex, write to default buffer).

vec4 pass5() {

        /////////////// Tone mapping ///////////////

        // Retrieve high-res color from texture

        vec4 color = texture( HdrTex, TexCoord );

        // Convert to XYZ

        vec3 xyzCol = rgb2xyz * vec3(color);

        // Convert to xyY

        float xyzSum = xyzCol.x + xyzCol.y + xyzCol.z;

        vec3 xyYCol = vec3( xyzCol.x / xyzSum, xyzCol.y / xyzSum, xyzCol.y);

        // Apply the tone mapping operation to the luminance (xyYCol.z or xyzCol.y)

        float L = (Exposure * xyYCol.z) / AveLum;

        L = (L * ( 1 + L / (White * White) )) / ( 1 + L );

        // Using the new luminance, convert back to XYZ

        xyzCol.x = (L * xyYCol.x) / (xyYCol.y);

        xyzCol.y = L;

        xyzCol.z = (L * (1 - xyYCol.x - xyYCol.y))/xyYCol.y;

        // Convert back to RGB

        vec4 toneMapColor = vec4( xyz2rgb * xyzCol, 1.0);

        ///////////// Combine with blurred texture /////////////

        // We want linear filtering on this texture access so that

        // we get additional blurring.

        vec4 blurTex = texture(BlurTex1, TexCoord);

        return toneMapColor + blurTex;

}
```

Scenebasic Uniform Header

As in previous implementations, the plane, sphere & teapot models are used. Additionally, the private section should look like the following:



```
private:

GLSLProgram prog;

GLuint fsQuad;

GLuint hdrFbo, blurFbo;

GLuint hdrTex, tex1, tex2;

GLuint linearSampler, nearestSampler;

Plane plane;

Sphere sphere;

Teapot teapot;

float angle;

int bloomBufWidth, bloomBufHeight;

void setMatrices();

void compile();

void setupFBO();

void pass1();

void pass2();

void pass3();

void pass4();

void pass5();

float gauss(float, float);

void computeLogAveLuminance();

void drawScene();
```

Scenebasic Uniform CPP

Init Scene Function

The initScene() function is a bit more complex than in previous implementations, however it remains similar.

Uniforms, Quad & Buffers

```
void SceneBasic_Uniform::initScene()
{
        compile();
        glClearColor(0.5f, 0.5f, 0.5f, 1.0f);
        glEnable(GL_DEPTH_TEST);
        vec3 intense = vec3(0.6f);
        prog.setUniform("Lights[0].L", intense);
        prog.setUniform("Lights[1].L", intense);
        prog.setUniform("Lights[2].L", intense);
        intense = vec3(0.2f);
        prog.setUniform("Lights[0].La", intense);
        prog.setUniform("Lights[1].La", intense);
        prog.setUniform("Lights[2].La", intense);
        projection = mat4(1.0f);
        angle = glm::pi<float>() / 2.0f;
        setupFBO();
        // Array for 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, 1.0f, 1.0f, 0.0f, -1.0f, 1.0f, 0.0f
        };
        GLfloat tc[] = {
        0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f,
        0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f
        };
        // Set up the buffers
        unsigned int handle[2];
        glGenBuffers(2, handle);
        glBindBuffer(GL_ARRAY_BUFFER, handle[0]);
        glBufferData(GL_ARRAY_BUFFER, 6 * 3 * sizeof(float), verts, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, handle[1]);
        glBufferData(GL_ARRAY_BUFFER, 6 * 2 * sizeof(float), tc, GL_STATIC_DRAW);
        // Set up the vertex array object
        glGenVertexArrays(1, &fsQuad);
        glBindVertexArray(fsQuad);
        glBindBuffer(GL_ARRAY_BUFFER, handle[0]);
        glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0);
        glEnableVertexAttribArray(0); // Vertex position
        glBindBuffer(GL_ARRAY_BUFFER, handle[1]);
        glVertexAttribPointer((GLuint)2, 2, GL_FLOAT, GL_FALSE, 0, 0);
        glEnableVertexAttribArray(2); // Texture coordinates
        glBindVertexArray(0);
        prog.setUniform("LumThresh", 1.7f);
        float weights[10], sum, sigma2 = 25.0f;
```

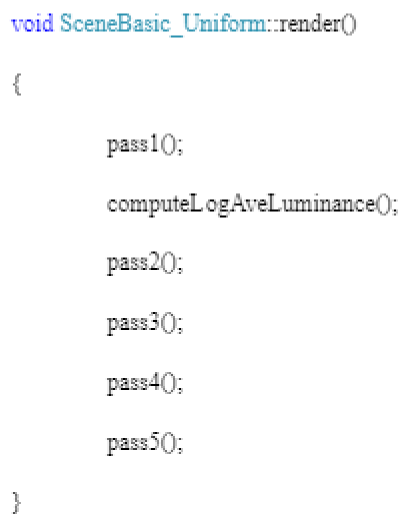
Weights, Normalisation & Sampling

```
        // Compute and sum the weights
        weights[0] = gauss(0, sigma2);
        sum = weights[0];
        for (int i = 1; i < 10; i++) {
        weights[i] = gauss(float(i), sigma2);
        sum += 2 * weights[i];
        }
        // Normalize the weights and set the uniform
        for (int i = 0; i < 10; i++) {
        std::stringstream uniName;
        uniName << "Weight[" << i << "]";
        float val = weights[i] / sum;
        prog.setUniform(uniName.str().c_str(), val);
        }
        // Set up two sampler objects for linear and nearest filtering
        GLuint samplers[2];
        glGenSamplers(2, samplers);
        linearSampler = samplers[0];
        nearestSampler = samplers[1];
        GLfloat border[] = { 0.0f,0.0f,0.0f,0.0f };
        // Set up the nearest sampler
        glSamplerParameteri(nearestSampler, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glSamplerParameteri(nearestSampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glSamplerParameteri(nearestSampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
        glSamplerParameteri(nearestSampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
        glSamplerParameterfv(nearestSampler, GL_TEXTURE_BORDER_COLOR, border);
        // Set up the linear sampler
        glSamplerParameteri(linearSampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glSamplerParameteri(linearSampler, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glSamplerParameteri(linearSampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
        glSamplerParameteri(linearSampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
        glSamplerParameterfv(linearSampler, GL_TEXTURE_BORDER_COLOR, border);
        // We want nearest sampling except for the last pass.
        glBindSampler(0, nearestSampler);
        glBindSampler(1, nearestSampler);
        glBindSampler(2, nearestSampler);
}
```

FBO Setup Function

```
//sets up the fbo for rendering to a texture
void SceneBasic_Uniform::setupFBO() {
        // Generate and bind the framebuffer
        glGenFramebuffers(1, &hdrFbo);
        glBindFramebuffer(GL_FRAMEBUFFER, hdrFbo);
        // Create the texture object
        glGenTextures(1, &hdrTex);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, hdrTex);
        glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGB32F, width, height);
        // Bind the texture to the FBO
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, hdrTex, 0);
        // Create the depth buffer
        GLuint depthBuf;
        glGenRenderbuffers(1, &depthBuf);
        glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
        glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);
        // Bind the depth buffer to the FBO
        glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
        GL_RENDERBUFFER, depthBuf);
        // Set the targets for the fragment output variables
        GLenum drawBuffers[] = { GL_COLOR_ATTACHMENT0 };
        glDrawBuffers(1, drawBuffers);
        // Create an FBO for the bright-pass filter and blur
        glGenFramebuffers(1, &blurFbo);
        glBindFramebuffer(GL_FRAMEBUFFER, blurFbo);
        // Create two texture objects to ping-pong for the bright-pass filter
        // and the two-pass blur
        bloomBufWidth = width / 8;
        bloomBufHeight = height / 8;
        glGenTextures(1, &tex1);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, tex1);
        glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGB32F, bloomBufWidth, bloomBufHeight);
        glActiveTexture(GL_TEXTURE2);
        glGenTextures(1, &tex2);
        glBindTexture(GL_TEXTURE_2D, tex2);
        glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGB32F, bloomBufWidth, bloomBufHeight);
        // Bind tex1 to the FBO
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tex1, 0);
        glDrawBuffers(1, drawBuffers);
        // Unbind the framebuffer, and revert to default framebuffer
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Render Function



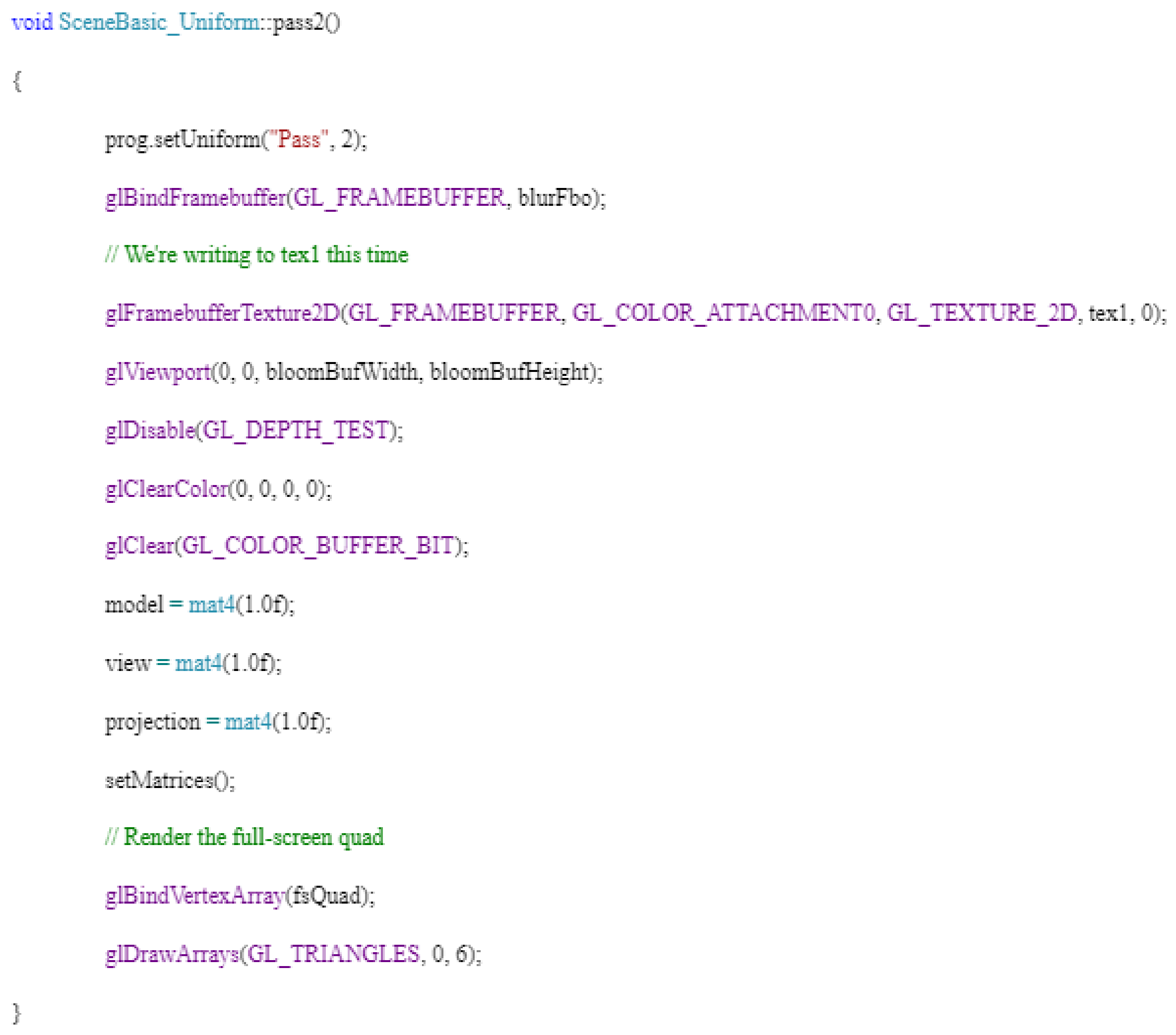
```
void SceneBasic_Uniform::render()

{

        pass1();

        computeLogAveLuminance();

        pass2();

        pass3();

        pass4();

        pass5();

}
```

Render Pass Function (1)

Use the same code from the HDR tone mapping pass 1 function implementation.

Bright Pass Function (2)

Use the same code from the Gaussian blur pass 2 function implementation, but make use of the blurFBO & write to tex1 instead:



```
void SceneBasic_Uniform::pass2()
{
        prog.setUniform("Pass", 2);
        glBindFramebuffer(GL_FRAMEBUFFER, blurFbo);
        // We're writing to tex1 this time
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tex1, 0);
        glViewport(0, 0, bloomBufWidth, bloomBufHeight);
        glDisable(GL_DEPTH_TEST);
        glClearColor(0, 0, 0, 0);
        glClear(GL_COLOR_BUFFER_BIT);
        model = mat4(1.0f);
        view = mat4(1.0f);
        projection = mat4(1.0f);
        setMatrices();
        // Render the full-screen quad
        glBindVertexArray(fsQuad);
        glDrawArrays(GL_TRIANGLES, 0, 6);
}
```

First Blur Pass Function (3)

Write to tex2 instead:

```
void SceneBasic_Uniform::pass3()

{

        prog.setUniform("Pass", 3);

        // We're writing to tex2 this time

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tex2, 0);

        // Render the full-screen quad

        glBindVertexArray(fsQuad);

        glDrawArrays(GL_TRIANGLES, 0, 6);

}
```

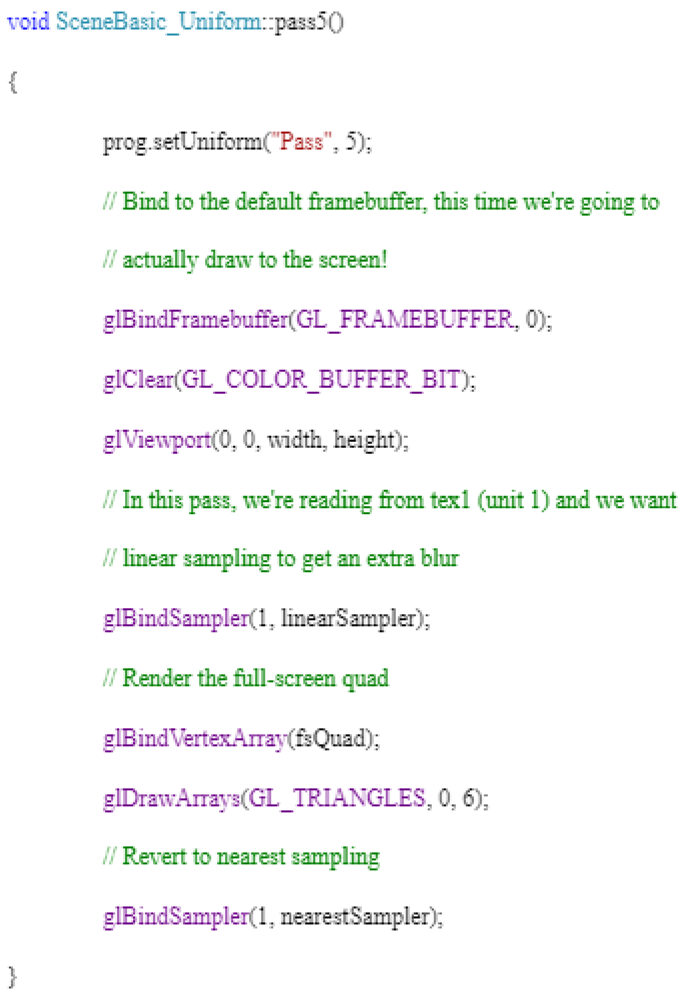
Second Blur Pass Function (4)

Do a second pass & write to tex2:

```
void SceneBasic_Uniform::pass4()

{

        prog.setUniform("Pass", 4);

        // We're writing to tex1 this time

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tex1, 0);

        // Render the full-screen quad

        glBindVertexArray(fsQuad);

        glDrawArrays(GL_TRIANGLES, 0, 6);

}
```

Composite Pass Function (5)

Use the default buffer, write to the screen & bring everything together:



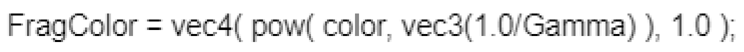
```
void SceneBasic_Uniform::pass5()
{
        prog.setUniform("Pass", 5);
        // Bind to the default framebuffer, this time we're going to
        // actually draw to the screen!
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
        glClear(GL_COLOR_BUFFER_BIT);
        glViewport(0, 0, width, height);
        // In this pass, we're reading from tex1 (unit 1) and we want
        // linear sampling to get an extra blur
        glBindSampler(1, linearSampler);
        // Render the full-screen quad
        glBindVertexArray(fsQuad);
        glDrawArrays(GL_TRIANGLES, 0, 6);
        // Revert to nearest sampling
        glBindSampler(1, nearestSampler);
}
```

Previous Functions

Use the same average luminance & scene drawing function implementations used for HDR tone mapping. Additionally, use the same gaussian blur function used for the gaussian blur implementation.

Gamma Correction

Implementing gamma correction is very simple. Create a variable named ‘Gamma’ & set it to an appropriate value. Experiment with different values to determine a good one. In the fragment shader, add it to the final colour before passing it to ‘FragColour:’



```
FragColor = vec4( pow( color, vec3(1.0/Gamma) ), 1.0 );
```

Deferred Shading

Overview

Pass 1

Render the scene, but as opposed to evaluating the reflection model to determine the fragment colour, store the geometry information in an intermediate set of buffers, collectively referred to as the geometry buffer (g-buffer). Geometric information required includes but is not limited to the position, normal, texture coordinate & reflectivity.

Pass 2

Read from the g-buffer, evaluate the reflection model & produce a final colour for each pixel.

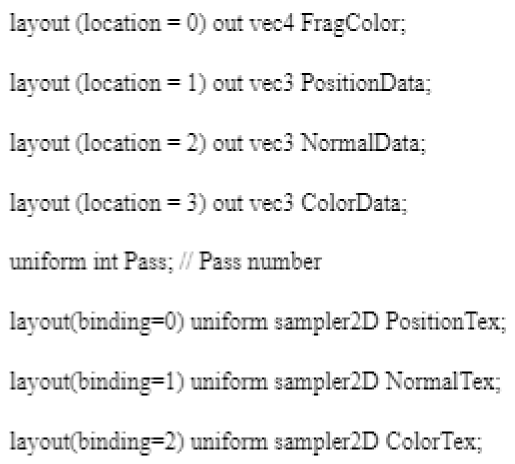
Vertex Shader

The vertex shader implementation should be similar to the implementation used for the Bloom effect.

Fragment Shader

Global Variables

The fragment shader includes the typical input & output variables as used in previous implementations. Notably, it needs to include the Light & Material uniform structs. Additionally, it requires the following uniforms:



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

layout (location = 1) out vec3 PositionData;

layout (location = 2) out vec3 NormalData;

layout (location = 3) out vec3 ColorData;

uniform int Pass; // Pass number

layout(binding=0) uniform sampler2D PositionTex;

layout(binding=1) uniform sampler2D NormalTex;

layout(binding=2) uniform sampler2D ColorTex;
```

Lighting Model

You can opt between the Blinn-Phong or diffuse models for light calculations. This lab will make use of the diffuse model.

Main Function

```
void main()

{

        if( Pass == 1)

                pass1();

        else if(Pass==2)

                pass2();

}
```

Pass 1 Function

The first pass stores the position, normal and diffuse in textures:

```
void pass1()

{

        // Store position, normal, and diffuse color in textures

        PositionData = Position;

        NormalData = normalize(Normal);

        ColorData = Material.Kd;

}
```

Pass 2 Function

The second pass retrieves the position, normal & colour:

```
void pass2()

{

        // Retrieve position and normal information from textures

        vec3 pos = vec3( texture( PositionTex, TexCoord ) );

        vec3 norm = vec3( texture( NormalTex, TexCoord ) );

        vec3 diffColor = vec3( texture(ColorTex, TexCoord) );

        FragColor = vec4( diffuseModel(pos,norm,diffColor), 1.0 );

}
```

Scenebasic Uniform Header

The following private object variables are required:

```
private:

GLSLProgram prog;

GLuint deferredFBO;

GLuint quad;

Plane plane;

Torus torus;

Teapot teapot;

float angle, tPrev, rotSpeed;

void setMatrices();

void compile();

void setupFBO();

void createGBufTex(GLenum, GLenum, GLuint &);

void pass1();

void pass2();
```

Scenebasic Uniform CPP

Constructor

The same constructor used in the edge detection implementation is used:

Init Scene Function

```
void SceneBasic_Uniform::initScene()

{

compile();

glEnable(GL_DEPTH_TEST);

float c = 1.5f;

angle = glm::pi<float>() / 2.0f;

// Array for 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, 1.0f, 1.0f, 0.0f, -1.0f, 1.0f, 0.0f

};

GLfloat tc[] = {

0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f,

0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f

};

// Set up the buffers

unsigned int handle[2];

glGenBuffers(2, handle);

glBindBuffer(GL_ARRAY_BUFFER, handle[0]);

glBufferData(GL_ARRAY_BUFFER, 6 * 3 * sizeof(float), verts, GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, handle[1]);

glBufferData(GL_ARRAY_BUFFER, 6 * 2 * sizeof(float), tc, GL_STATIC_DRAW);

// Set up the vertex array object

glGenVertexArrays(1, &quad);

glBindVertexArray(quad);

glBindBuffer(GL_ARRAY_BUFFER, handle[0]);

glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(0); // Vertex position

glBindBuffer(GL_ARRAY_BUFFER, handle[1]);

glVertexAttribPointer((GLuint)2, 2, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(2); // Texture coordinates

glBindVertexArray(0);

setupFBO();

prog.setUniform("Light.L", vec3(1.0f));

}
```

FBO Setup Function

```
void SceneBasic_Uniform::setupFBO()

{

        GLuint depthBuf, posTex, normTex, colorTex;

        // Create and bind the FBO

        glGenFramebuffers(1, &deferredFBO);

        glBindFramebuffer(GL_FRAMEBUFFER, deferredFBO);

        // The depth buffer

        glGenRenderbuffers(1, &depthBuf);

        glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);

        glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);

        // Create the textures for position, normal and color

        createGBufTex(GL_TEXTURE0, GL_RGB32F, posTex); // Position

        createGBufTex(GL_TEXTURE1, GL_RGB32F, normTex); // Normal

        createGBufTex(GL_TEXTURE2, GL_RGB8, colorTex); // Color

        // Attach the textures to the framebuffer

        glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER,
depthBuf);

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, posTex,
0);

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, normTex,
0);

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, colorTex,
0);

        GLenum drawBuffers[] = { GL_NONE, GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,

        GL_COLOR_ATTACHMENT2 };

        glDrawBuffers(4, drawBuffers);

        glBindFramebuffer(GL_FRAMEBUFFER, 0);

}
```

Geometry Buffer Texture Creation Function

```
void SceneBasic_Uniform::createGBufTex(GLenum texUnit, GLenum format, GLuint &texid)

{

        glActiveTexture(texUnit);

        glGenTextures(1, &texid);

        glBindTexture(GL_TEXTURE_2D, texid);

        glTexStorage2D(GL_TEXTURE_2D, 1, format, width, height);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);

}
```

Update Function

Use the update function from the edge detection implementation

Render Function

Call the first & second pass functions in order.

Pass 1 Function

Render the scene & send it to the ‘deferredFBO:’

```
void SceneBasic_Uniform::pass1()

{

            prog.setUniform("Pass", 1);

            glBindFramebuffer(GL_FRAMEBUFFER, deferredFBO);

            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

            glEnable(GL_DEPTH_TEST);

            view = glm::lookAt(vec3(7.0f * cos(angle), 4.0f, 7.0f * sin(angle)), vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 1.0f, 0.0f));

            projection = glm::perspective(glm::radians(60.0f), (float)width / height, 0.3f, 100.0f);

            prog.setUniform("Light.Position", glm::vec4(0.0f, 0.0f, 0.0f, 1.0f));

            prog.setUniform("Material.Kd", 0.9f, 0.9f, 0.9f);

            model = mat4(1.0f);

            model = glm::translate(model, vec3(0.0f, 0.0f, 0.0f));

            model = glm::rotate(model, glm::radians(-90.0f), vec3(1.0f, 0.0f, 0.0f));

            setMatrices();

            teapot.render();

            prog.setUniform("Material.Kd", 0.4f, 0.4f, 0.4f);

            model = mat4(1.0f);

            model = glm::translate(model, vec3(0.0f, -0.75f, 0.0f));

            setMatrices();

            plane.render();

            prog.setUniform("Light.Position", glm::vec4(0.0f, 0.0f, 0.0f, 1.0f));

            prog.setUniform("Material.Kd", 0.2f, 0.5f, 0.9f);

            model = mat4(1.0f);

            model = glm::translate(model, vec3(1.0f, 1.0f, 3.0f));

            model = glm::rotate(model, glm::radians(90.0f), vec3(1.0f, 0.0f, 0.0f));

            setMatrices();

            torus.render();

            glFinish();

}
```

Pass 2 Function

Use the default buffer & render the quad:

```
void SceneBasic_Uniform::pass2()

{

        prog.setUniform("Pass", 2);

        // Revert to default framebuffer

        glBindFramebuffer(GL_FRAMEBUFFER, 0);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glDisable(GL_DEPTH_TEST);

        view = mat4(1.0);

        model = mat4(1.0);

        projection = mat4(1.0);

        setMatrices();

        // Render the quad

        glBindVertexArray(quad);

        glDrawArrays(GL_TRIANGLES, 0, 6);

}
```

Appendix

Bloom Effect

Composite Pass Function (5)

```
// (Read from BlurTex1 and HdrTex, write to default buffer).

vec4 pass5() {

        /////////////// Tone mapping ///////////////

        // Retrieve high-res color from texture

        vec4 color = texture( HdrTex, TexCoord );

        // Convert to XYZ

        vec3 xyzCol = rgb2xyz * vec3(color);

        // Convert to xyY

        float xyzSum = xyzCol.x + xyzCol.y + xyzCol.z;

        vec3 xyYCol = vec3( xyzCol.x / xyzSum, xyzCol.y / xyzSum, xyzCol.y);

        // Apply the tone mapping operation to the luminance (xyYCol.z or xyzCol.y)

        float L = (Exposure * xyYCol.z) / AveLum;

        L = (L * ( 1 + L / (White * White) )) / ( 1 + L );

        // Using the new luminance, convert back to XYZ

        xyzCol.x = (L * xyYCol.x) / (xyYCol.y);

        xyzCol.y = L;

        xyzCol.z = (L * (1 - xyYCol.x - xyYCol.y))/xyYCol.y;

        // Convert back to RGB

        vec4 toneMapColor = vec4( xyz2rgb * xyzCol, 1.0);

        ///////////// Combine with blurred texture /////////////

        // We want linear filtering on this texture access so that

        // we get additional blurring.

        vec4 blurTex = texture(BlurTex1, TexCoord);

        return toneMapColor + blurTex;

}
```

Uniforms, Quad & Buffers

```
void SceneBasic_Uniform::initScene()

{

compile();

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

glEnable(GL_DEPTH_TEST);

vec3 intense = vec3(0.6f);

prog.setUniform("Lights[0].L", intense);

prog.setUniform("Lights[1].L", intense);

prog.setUniform("Lights[2].L", intense);

intense = vec3(0.2f);

prog.setUniform("Lights[0].La", intense);

prog.setUniform("Lights[1].La", intense);

prog.setUniform("Lights[2].La", intense);

projection = mat4(1.0f);

angle = glm::pi<float>() / 2.0f;

setupFBO();

// Array for 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, 1.0f, 1.0f, 0.0f, -1.0f, 1.0f, 0.0f

};

GLfloat tc[] = {

0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f,

0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f

};

// Set up the buffers

unsigned int handle[2];

glGenBuffers(2, handle);

glBindBuffer(GL_ARRAY_BUFFER, handle[0]);

glBufferData(GL_ARRAY_BUFFER, 6 * 3 * sizeof(float), verts, GL_STATIC_DRAW);
```

```
glBindBuffer(GL_ARRAY_BUFFER, handle[1]);

glBufferData(GL_ARRAY_BUFFER, 6 * 2 * sizeof(float), tc, GL_STATIC_DRAW);

// Set up the vertex array object

glGenVertexArrays(1, &fsQuad);

glBindVertexArray(fsQuad);

glBindBuffer(GL_ARRAY_BUFFER, handle[0]);

glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(0); // Vertex position

glBindBuffer(GL_ARRAY_BUFFER, handle[1]);

glVertexAttribPointer((GLuint)2, 2, GL_FLOAT, GL_FALSE, 0, 0);

glEnableVertexAttribArray(2); // Texture coordinates

glBindVertexArray(0);

prog.setUniform("LumThresh", 1.7f);

float weights[10], sum, sigma2 = 25.0f;

// Compute and sum the weights

weights[0] = gauss(0, sigma2);

sum = weights[0];

for (int i = 1; i < 10; i++) {

weights[i] = gauss(float(i), sigma2);

sum += 2 * weights[i];

}

// Normalize the weights and set the uniform

for (int i = 0; i < 10; i++) {

std::stringstream uniName;

uniName << "Weight[" << i << "]";

float val = weights[i] / sum;

prog.setUniform(uniName.str().c_str(), val);

}

// Set up two sampler objects for linear and nearest filtering

GLuint samplers[2];

glGenSamplers(2, samplers);
```

```
linearSampler = samplers[0];

nearestSampler = samplers[1];

GLfloat border[] = { 0.0f,0.0f,0.0f,0.0f };

// Set up the nearest sampler

glSamplerParameteri(nearestSampler, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glSamplerParameteri(nearestSampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glSamplerParameteri(nearestSampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);

glSamplerParameteri(nearestSampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);

glSamplerParameterfv(nearestSampler, GL_TEXTURE_BORDER_COLOR, border);

// Set up the linear sampler

glSamplerParameteri(linearSampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glSamplerParameteri(linearSampler, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

glSamplerParameteri(linearSampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);

glSamplerParameteri(linearSampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);

glSamplerParameterfv(linearSampler, GL_TEXTURE_BORDER_COLOR, border);

// We want nearest sampling except for the last pass.

glBindSampler(0, nearestSampler);

glBindSampler(1, nearestSampler);

glBindSampler(2, nearestSampler);

}
```

FBO Setup Function

```
//sets up the fbo for rendering to a texture

void SceneBasic_Uniform::setupFBO() {

        // Generate and bind the framebuffer

        glGenFramebuffers(1, &hdrFbo);

        glBindFramebuffer(GL_FRAMEBUFFER, hdrFbo);

        // Create the texture object

        glGenTextures(1, &hdrTex);

        glActiveTexture(GL_TEXTURE0);

        glBindTexture(GL_TEXTURE_2D, hdrTex);

        glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGB32F, width, height);
```

```
    // Bind the texture to the FBO

    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, hdrTex,
0);

    // Create the depth buffer

    GLuint depthBuf;

    glGenRenderbuffers(1, &depthBuf);

    glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);

    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);

    // Bind the depth buffer to the FBO

    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,

    GL_RENDERBUFFER, depthBuf);

    // Set the targets for the fragment output variables

    GLenum drawBuffers[] = { GL_COLOR_ATTACHMENT0 };

    glDrawBuffers(1, drawBuffers);

    // Create an FBO for the bright-pass filter and blur

    glGenFramebuffers(1, &blurFbo);

    glBindFramebuffer(GL_FRAMEBUFFER, blurFbo);

    // Create two texture objects to ping-pong for the bright-pass filter

    // and the two-pass blur

    bloomBufWidth = width / 8;

    bloomBufHeight = height / 8;

    glGenTextures(1, &tex1);

    glActiveTexture(GL_TEXTURE1);

    glBindTexture(GL_TEXTURE_2D, tex1);

    glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGB32F, bloomBufWidth, bloomBufHeight);

    glActiveTexture(GL_TEXTURE2);

    glGenTextures(1, &tex2);

    glBindTexture(GL_TEXTURE_2D, tex2);

    glTexStorage2D(GL_TEXTURE_2D, 1, GL_RGB32F, bloomBufWidth, bloomBufHeight);

    // Bind tex1 to the FBO

    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tex1, 0);

    glDrawBuffers(1, drawBuffers);
```

```
        // Unbind the framebuffer, and revert to default framebuffer
        glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Deferred Shading

Init Scene Function

```
void SceneBasic_Uniform::initScene()
{
        compile();
        glEnable(GL_DEPTH_TEST);
        float c = 1.5f;
        angle = glm::pi<float>() / 2.0f;
        // Array for 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, 1.0f, 1.0f, 0.0f, -1.0f, 1.0f, 0.0f
        };
        GLfloat tc[] = {
                0.0f, 0.0f, 1.0f, 0.0f, 1.0f, 1.0f,
                0.0f, 0.0f, 1.0f, 1.0f, 0.0f, 1.0f
        };
        // Set up the buffers
        unsigned int handle[2];
        glGenBuffers(2, handle);
        glBindBuffer(GL_ARRAY_BUFFER, handle[0]);
        glBufferData(GL_ARRAY_BUFFER, 6 * 3 * sizeof(float), verts, GL_STATIC_DRAW);
        glBindBuffer(GL_ARRAY_BUFFER, handle[1]);
        glBufferData(GL_ARRAY_BUFFER, 6 * 2 * sizeof(float), tc, GL_STATIC_DRAW);
        // Set up the vertex array object
        glGenVertexArrays(1, &quad);
        glBindVertexArray(quad);
        glBindBuffer(GL_ARRAY_BUFFER, handle[0]);

        glVertexAttribPointer((GLuint)0, 3, GL_FLOAT, GL_FALSE, 0, 0);

        glEnableVertexAttribArray(0); // Vertex position

        glBindBuffer(GL_ARRAY_BUFFER, handle[1]);

        glVertexAttribPointer((GLuint)2, 2, GL_FLOAT, GL_FALSE, 0, 0);

        glEnableVertexAttribArray(2); // Texture coordinates

        glBindVertexArray(0);

        setupFBO();

        prog.setUniform("Light.L", vec3(1.0f));

}
```

FBO Setup Function

```
void SceneBasic_Uniform::setupFBO()

{

        GLuint depthBuf, posTex, normTex, colorTex;

        // Create and bind the FBO

        glGenFramebuffers(1, &deferredFBO);

        glBindFramebuffer(GL_FRAMEBUFFER, deferredFBO);

        // The depth buffer

        glGenRenderbuffers(1, &depthBuf);

        glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);

        glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);

        // Create the textures for position, normal and color

        createGBufTex(GL_TEXTURE0, GL_RGB32F, posTex); // Position

        createGBufTex(GL_TEXTURE1, GL_RGB32F, normTex); // Normal

        createGBufTex(GL_TEXTURE2, GL_RGB8, colorTex); // Color

        // Attach the textures to the framebuffer

        glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER,
depthBuf);

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, posTex,
0);

        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE_2D, normTex,
0);
```

```
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT2, GL_TEXTURE_2D, colorTex,
0);

        GLenum drawBuffers[] = { GL_NONE, GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,

        GL_COLOR_ATTACHMENT2 };

        glDrawBuffers(4, drawBuffers);

        glBindFramebuffer(GL_FRAMEBUFFER, 0);

}
```

Pass 1 Function

```
void SceneBasic_Uniform::pass1()

{

        prog.setUniform("Pass", 1);

        glBindFramebuffer(GL_FRAMEBUFFER, deferredFBO);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glEnable(GL_DEPTH_TEST);

        view = glm::lookAt(vec3(7.0f * cos(angle), 4.0f, 7.0f * sin(angle)), vec3(0.0f, 0.0f, 0.0f), vec3(0.0f, 1.0f, 0.0f));

        projection = glm::perspective(glm::radians(60.0f), (float)width / height, 0.3f, 100.0f);

        prog.setUniform("Light.Position", glm::vec4(0.0f, 0.0f, 0.0f, 1.0f));

        prog.setUniform("Material.Kd", 0.9f, 0.9f, 0.9f);

        model = mat4(1.0f);

        model = glm::translate(model, vec3(0.0f, 0.0f, 0.0f));

        model = glm::rotate(model, glm::radians(-90.0f), vec3(1.0f, 0.0f, 0.0f));

        setMatrices();

        teapot.render();

        prog.setUniform("Material.Kd", 0.4f, 0.4f, 0.4f);

        model = mat4(1.0f);

        model = glm::translate(model, vec3(0.0f, -0.75f, 0.0f));

        setMatrices();

        plane.render();

        prog.setUniform("Light.Position", glm::vec4(0.0f, 0.0f, 0.0f, 1.0f));

        prog.setUniform("Material.Kd", 0.2f, 0.5f, 0.9f);

        model = mat4(1.0f);
        model = glm::translate(model, vec3(1.0f, 1.0f, 3.0f));

        model = glm::rotate(model, glm::radians(90.0f), vec3(1.0f, 0.0f, 0.0f));

        setMatrices();

        torus.render();

        glFinish();

}
```