# CMP301 Graphics Programming with Shaders

By Matthew Wallace

1502616@abertay.ac.uk

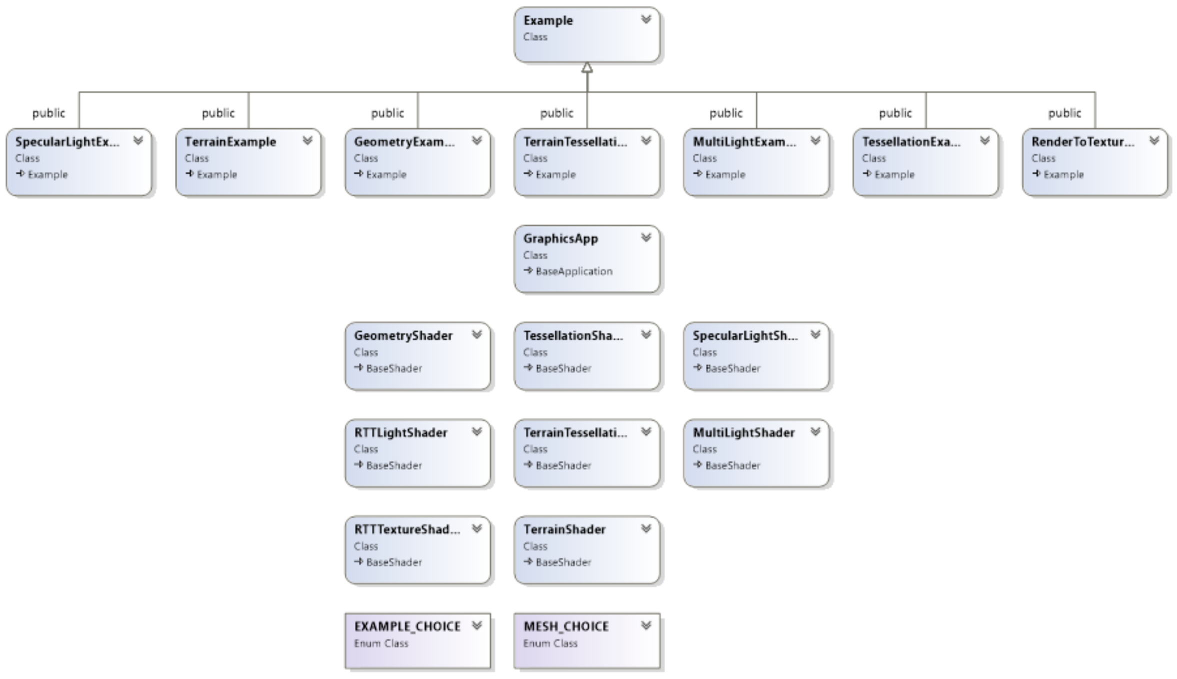
## Contents

## 1. Information on user controls

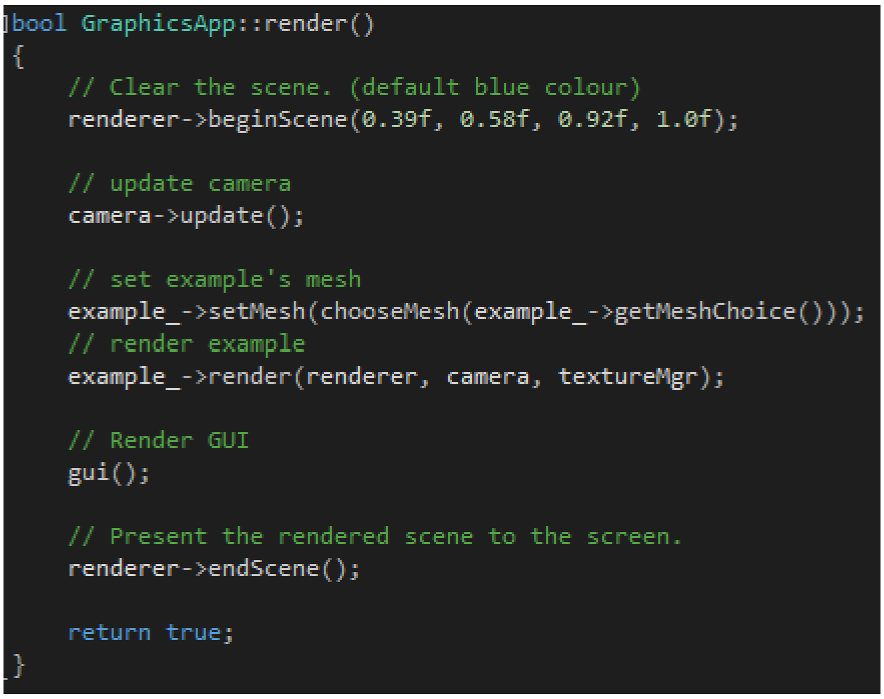
User controls don't differ from normal functionality of the framework provided (meaning camera controls ect.). The application makes full use of ImGUI and it's quite intuitive to use. But I'll explain how to use while also talking about its structure.

### Application structure

The application differes from the normla 'App1' structure given to us during the labs. It's divided into 'Examples' which showcase different shader techniques. 'GraphicsApp' has all example objects ('SpecularLightExample', 'Terrain Example', 'GeometryExample', 'TerrainTessellationExample', 'RenderToTextureExample') which inherit from a virtual class called 'Example' to enable polimorphic game state machine. 'GraphicsApp' also has all mesh objects that are later being passed to different exmaples, which limits the times that those have to created.



### Render function

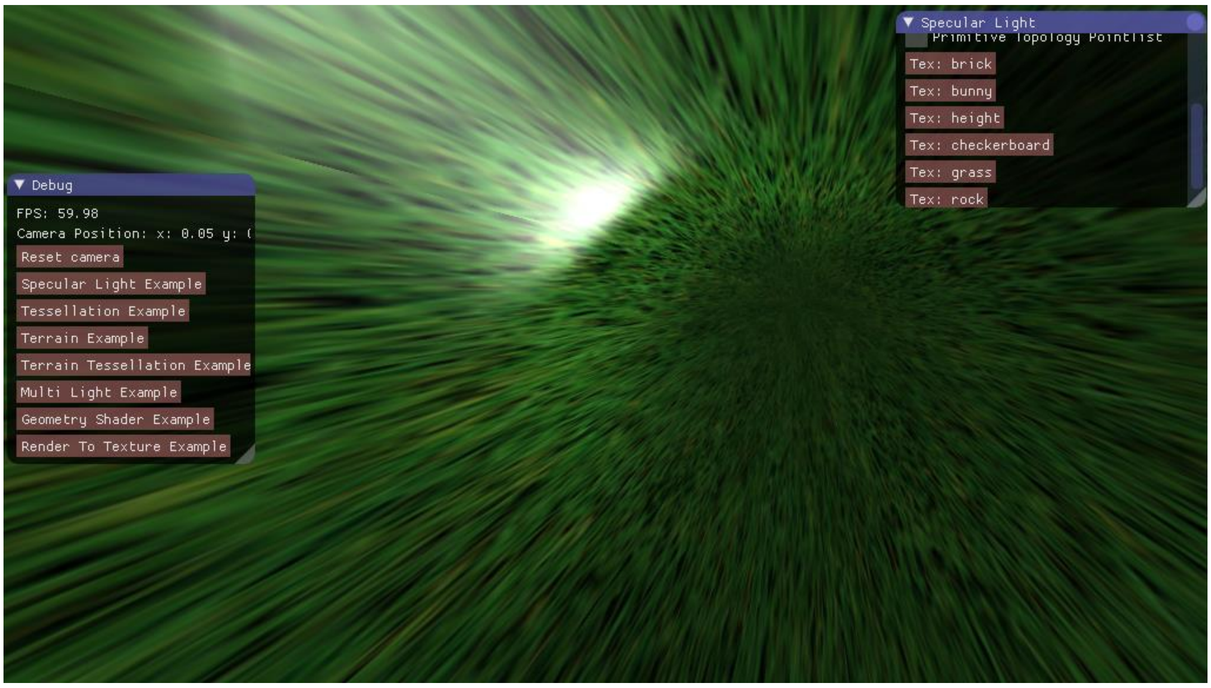


```
bool GraphicsApp::render()
{
    // Clear the scene. (default blue colour)
    renderer->beginScene(0.39f, 0.58f, 0.92f, 1.0f);

    // update camera
    camera->update();

    // set example's mesh
    example_->setMesh(chooseMesh(example_->getMeshChoice()));
    // render example
    example_->render(renderer, camera, textureMgr);

    // Render GUI
    gui();

    // Present the rendered scene to the screen.
    renderer->endScene();

    return true;
}
```

Previously presented code structure enables for a single call to example's render function like also for a single scene for all the examples.

## 2. Shaders

### Specular Light Example



```
FPS: 59.98
Camera Position: x: 0.05 y: (
Primitive Topology PointList
```

This was a test example to develop the application's structure hence there's no major changes to it from the lab exercise apart from applying different textures, scaling and using different primitives. More elaborate light explanation can be found in the Multi Light Example section.

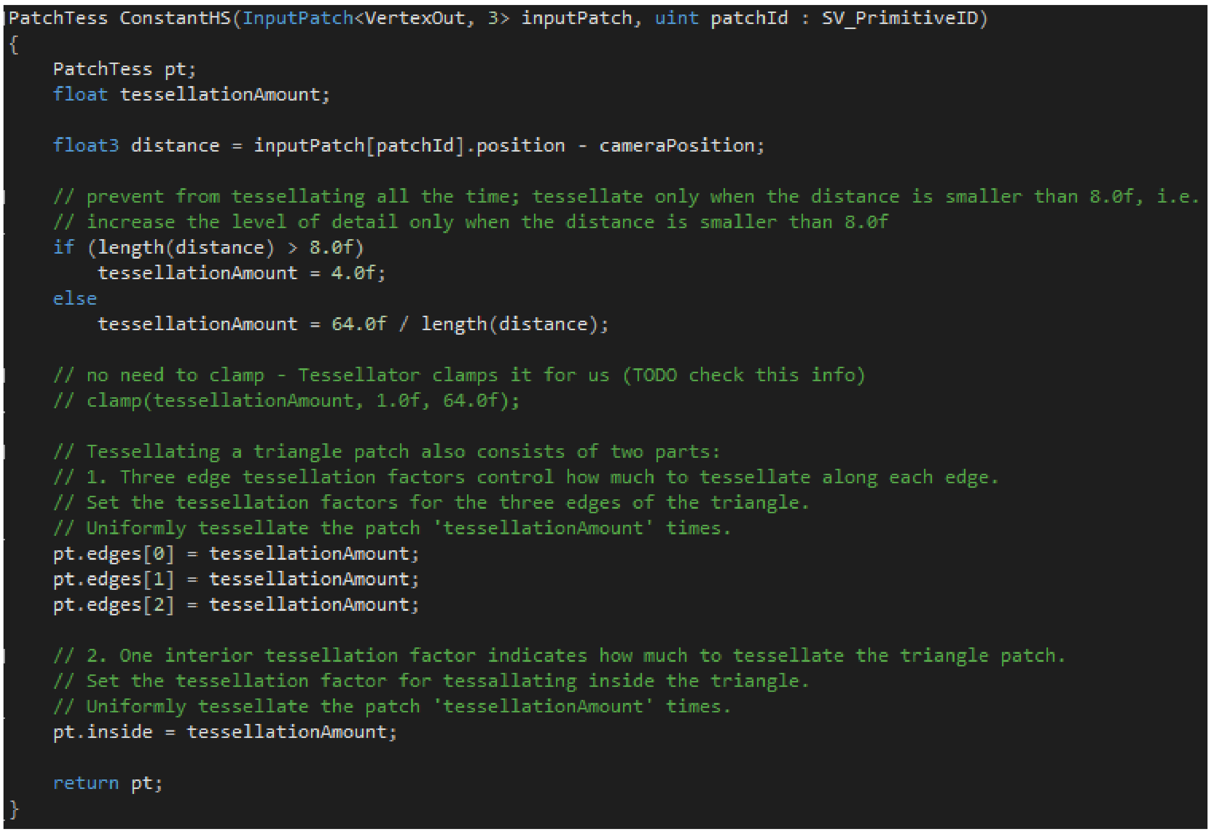
## Tessellation Example

### Shader stages

- Vertex shader
- Hull shader
- Domain shader
- Pixel shader

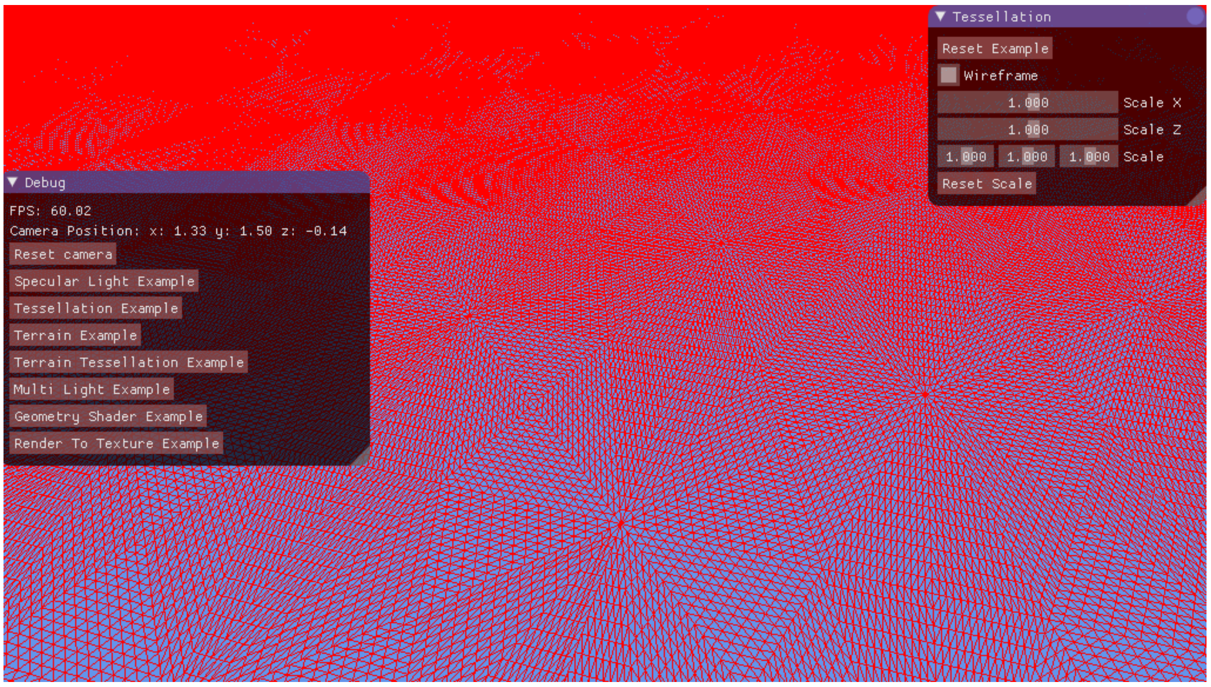
### Shader algorithms and calculations

#### *Hull shader*



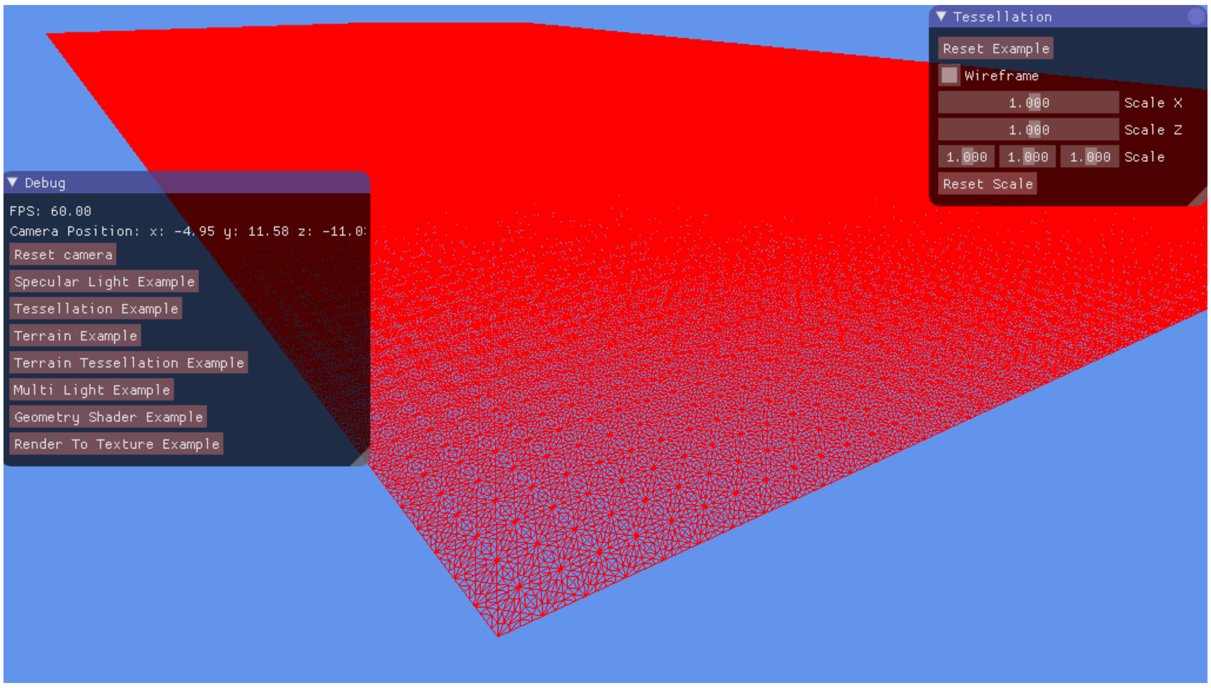
```
PatchTess ConstantHS(InputPatch<VertexOut, 3> inputPatch, uint patchId : SV_PrimitiveID)
{
    PatchTess pt;
    float tessellationAmount;

    float3 distance = inputPatch[patchId].position - cameraPosition;

    // prevent from tessellating all the time; tessellate only when the distance is smaller than 8.0f, i.e.
    // increase the level of detail only when the distance is smaller than 8.0f
    if (length(distance) > 8.0f)
        tessellationAmount = 4.0f;
    else
        tessellationAmount = 64.0f / length(distance);

    // no need to clamp - Tessellator clamps it for us (TODO check this info)
    // clamp(tessellationAmount, 1.0f, 64.0f);

    // Tessellating a triangle patch also consists of two parts:
    // 1. Three edge tessellation factors control how much to tessellate along each edge.
    // Set the tessellation factors for the three edges of the triangle.
    // Uniformly tessellate the patch 'tessellationAmount' times.
    pt.edges[0] = tessellationAmount;
    pt.edges[1] = tessellationAmount;
    pt.edges[2] = tessellationAmount;

    // 2. One interior tessellation factor indicates how much to tessellate the triangle patch.
    // Set the tessellation factor for tessalating inside the triangle.
    // Uniformly tessellate the patch 'tessellationAmount' times.
    pt.inside = tessellationAmount;

    return pt;
}
```

The mesh gets tessellated based on the camera's position and the distance from the patch. The value is then used to tessellate the patch on the edges and on the inside. The draw back here is that the application does not take into account centre of the patch but centre of the mesh which would cause a lot of problems if the vertices were being manipulated (not accurate distance).



```
1.000 Scale X
1.000 Scale Z
1.000 1.000 1.000 Scale
FPS: 60.02
Camera Position: x: 1.33 y: 1.50 z: -0.14
```

*High tessellation amount near to the mesh*

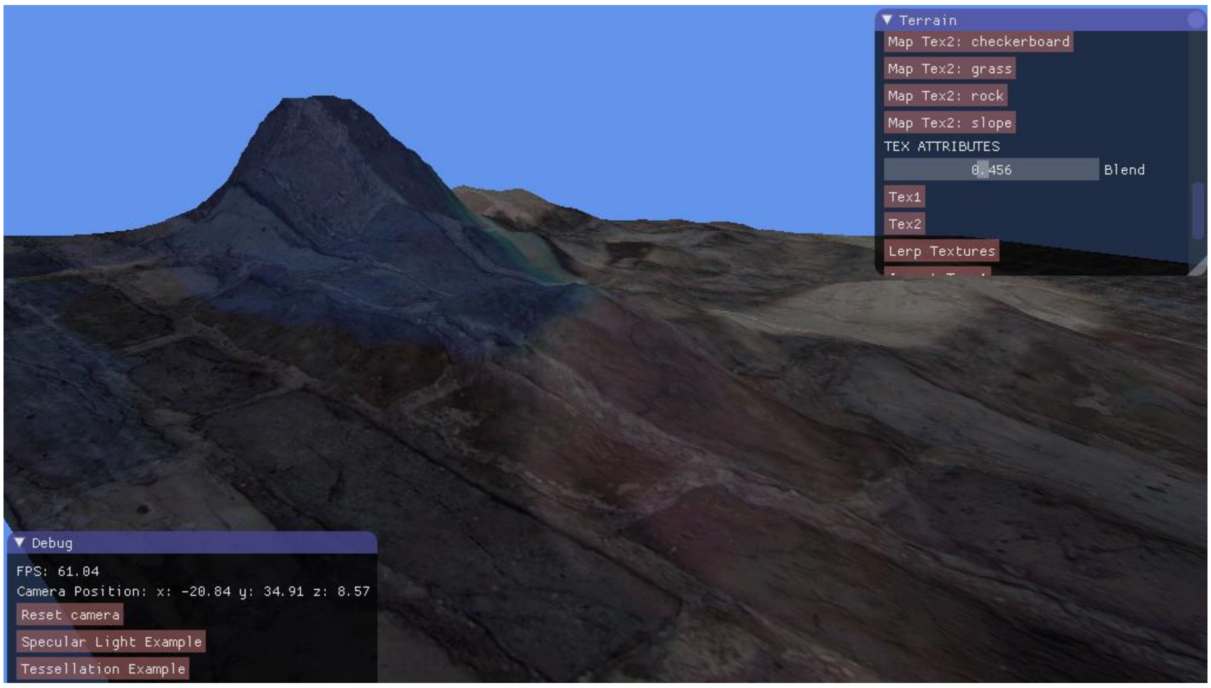


```
1.000 Scale X
1.000 Scale Z
1.000 1.000 1.000 Scale
FPS: 60.00
Camera Position: x: -4.95 y: 11.58 z: -11.0
```

*Low tessellation amount father from the mesh*

## Shader algorithms and calculations

## Terrain Example



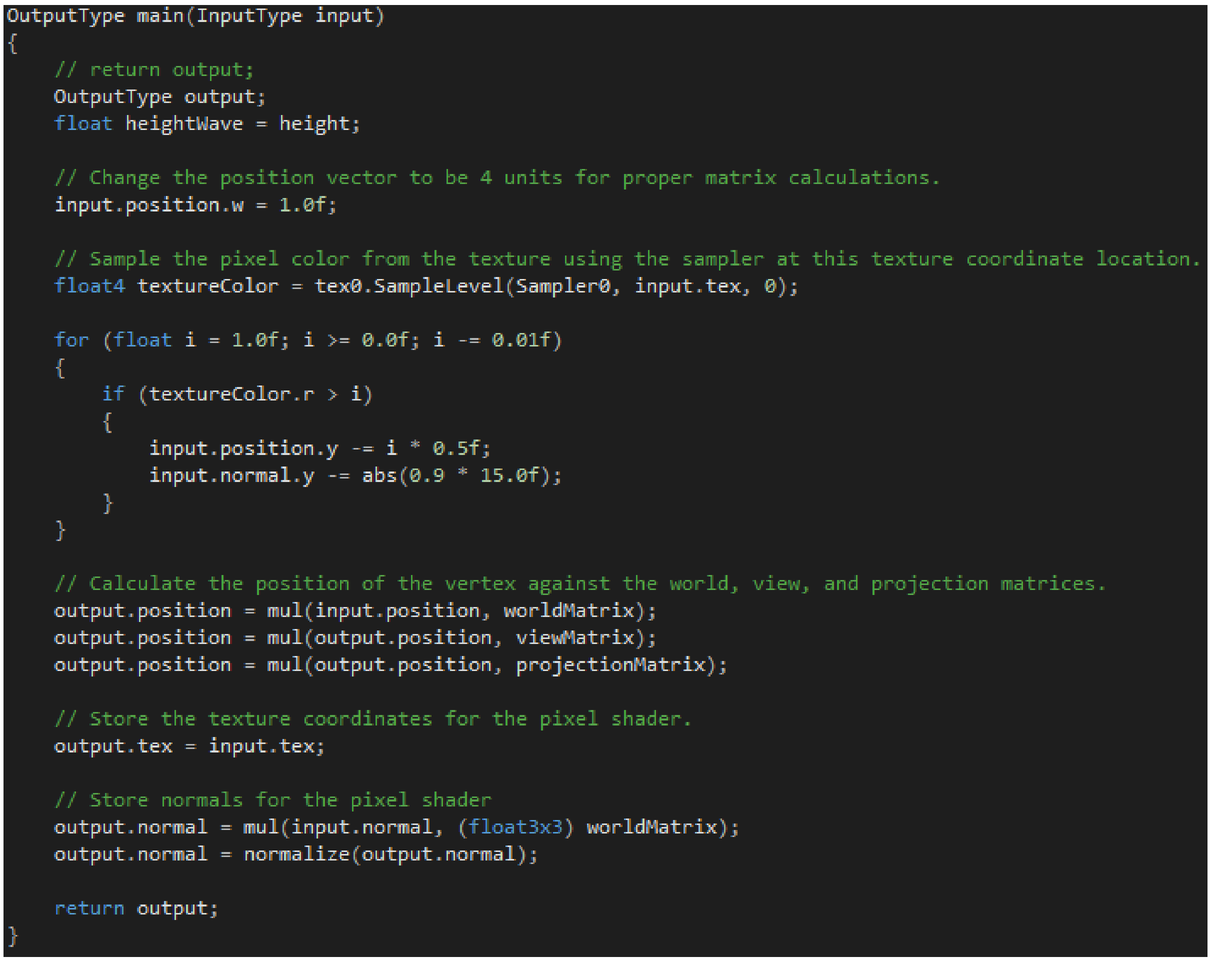
```
0.456 Blend
FPS: 61.04
Camera Position: x: -20.84 y: 34.91 z: 8.57
```

## Shader stages

- Vertex shader
- Pixel shader

## Shader algorithms and calculations

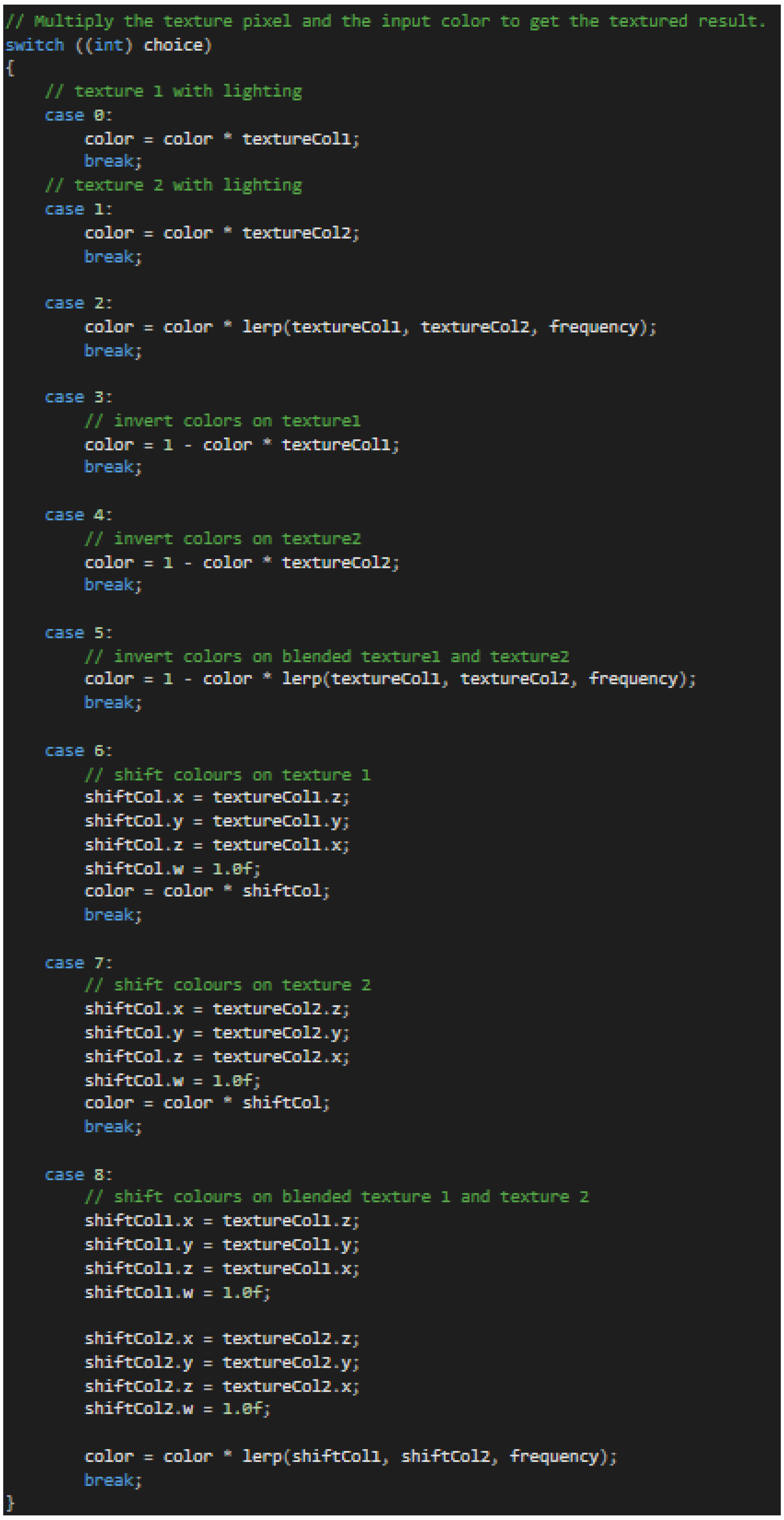
### *Vertex Shader*



```
OutputType main(InputType input)
{
    // return output;
    OutputType output;
    float heightWave = height;

    // Change the position vector to be 4 units for proper matrix calculations.
    input.position.w = 1.0f;

    // Sample the pixel color from the texture using the sampler at this texture coordinate location.
    float4 textureColor = tex0.SampleLevel(Sampler0, input.tex, 0);

    for (float i = 1.0f; i >= 0.0f; i -= 0.01f)
    {
        if (textureColor.r > i)
        {
            input.position.y -= i * 0.5f;
            input.normal.y -= abs(0.9 * 15.0f);
        }
    }

    // Calculate the position of the vertex against the world, view, and projection matrices.
    output.position = mul(input.position, worldMatrix);
    output.position = mul(output.position, viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    // Store the texture coordinates for the pixel shader.
    output.tex = input.tex;

    // Store normals for the pixel shader
    output.normal = mul(input.normal, (float3x3) worldMatrix);
    output.normal = normalize(output.normal);

    return output;
}
```

Vertex shader uses SampleLevel function (on contrary to Sample which is Pixel Shader function only) to sample the texture. Since the height map is in the shades of grey we don't need to worry about RGB colours so taking only one of the factors is sufficient. In this case it is 'r' component of the texture's sampled colour. Then a for loop runs from 1.0 to 0.0 assigning a new 'y' position to each vertex. The brighter the pixel is the higher position it gets. The normals are also being offset.

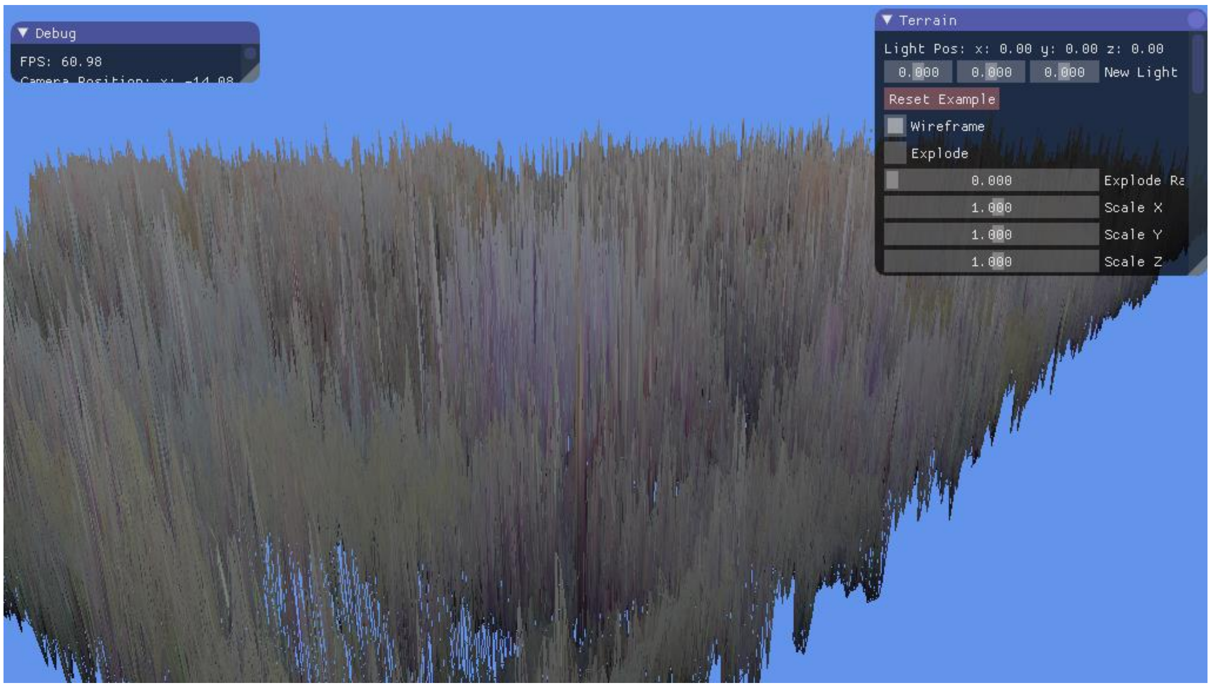
## *Pixel shader*



```
// Multiply the texture pixel and the input color to get the textured result.
switch ((int) choice)
{
    // texture 1 with lighting
    case 0:
        color = color * textureCol1;
        break;
    // texture 2 with lighting
    case 1:
        color = color * textureCol2;
        break;

    case 2:
        color = color * lerp(textureCol1, textureCol2, frequency);
        break;

    case 3:
        // invert colors on texture1
        color = 1 - color * textureCol1;
        break;

    case 4:
        // invert colors on texture2
        color = 1 - color * textureCol2;
        break;

    case 5:
        // invert colors on blended texture1 and texture2
        color = 1 - color * lerp(textureCol1, textureCol2, frequency);
        break;

    case 6:
        // shift colours on texture 1
        shiftCol.x = textureCol1.z;
        shiftCol.y = textureCol1.y;
        shiftCol.z = textureCol1.x;
        shiftCol.w = 1.0f;
        color = color * shiftCol;
        break;

    case 7:
        // shift colours on texture 2
        shiftCol.x = textureCol2.z;
        shiftCol.y = textureCol2.y;
        shiftCol.z = textureCol2.x;
        shiftCol.w = 1.0f;
        color = color * shiftCol;
        break;

    case 8:
        // shift colours on blended texture 1 and texture 2
        shiftCol1.x = textureCol1.z;
        shiftCol1.y = textureCol1.y;
        shiftCol1.z = textureCol1.x;
        shiftCol1.w = 1.0f;

        shiftCol2.x = textureCol2.z;
        shiftCol2.y = textureCol2.y;
        shiftCol2.z = textureCol2.x;
        shiftCol2.w = 1.0f;

        color = color * lerp(shiftCol1, shiftCol2, frequency);
        break;
}
```

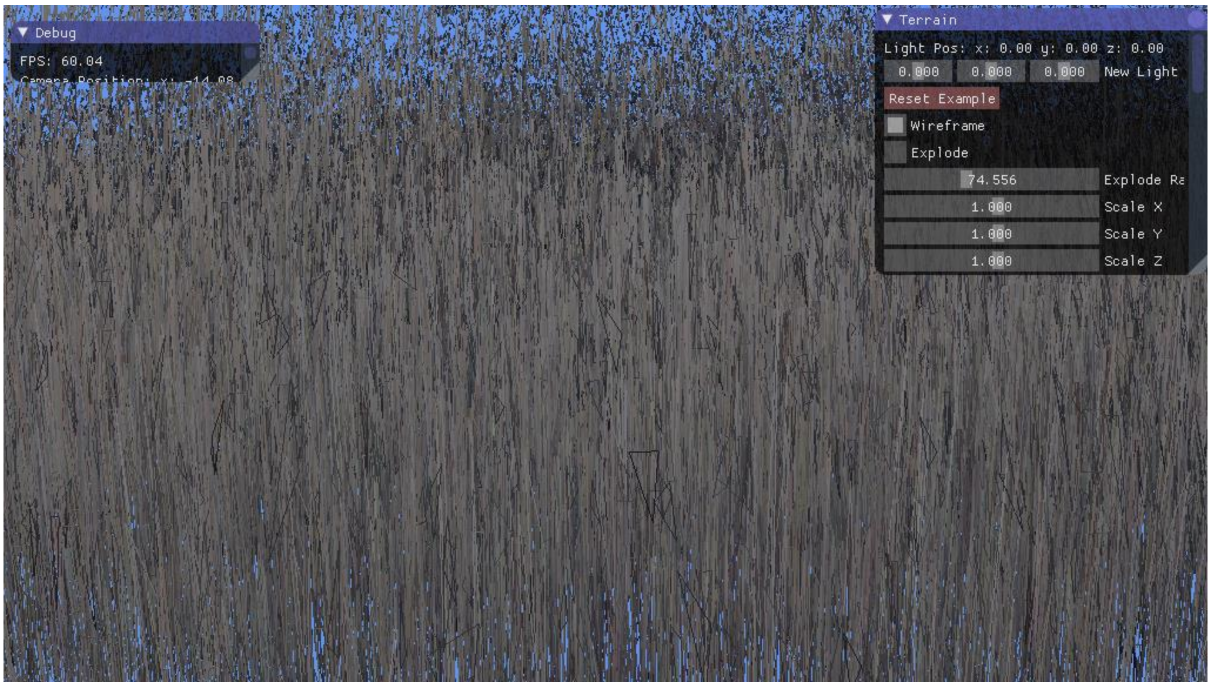
The pixel shader receives two textures which ten are being sampled and applied to the mesh. The textures get blended, inverted, shifted and then mixed with the lighting.

## Terrain Tessellation Example



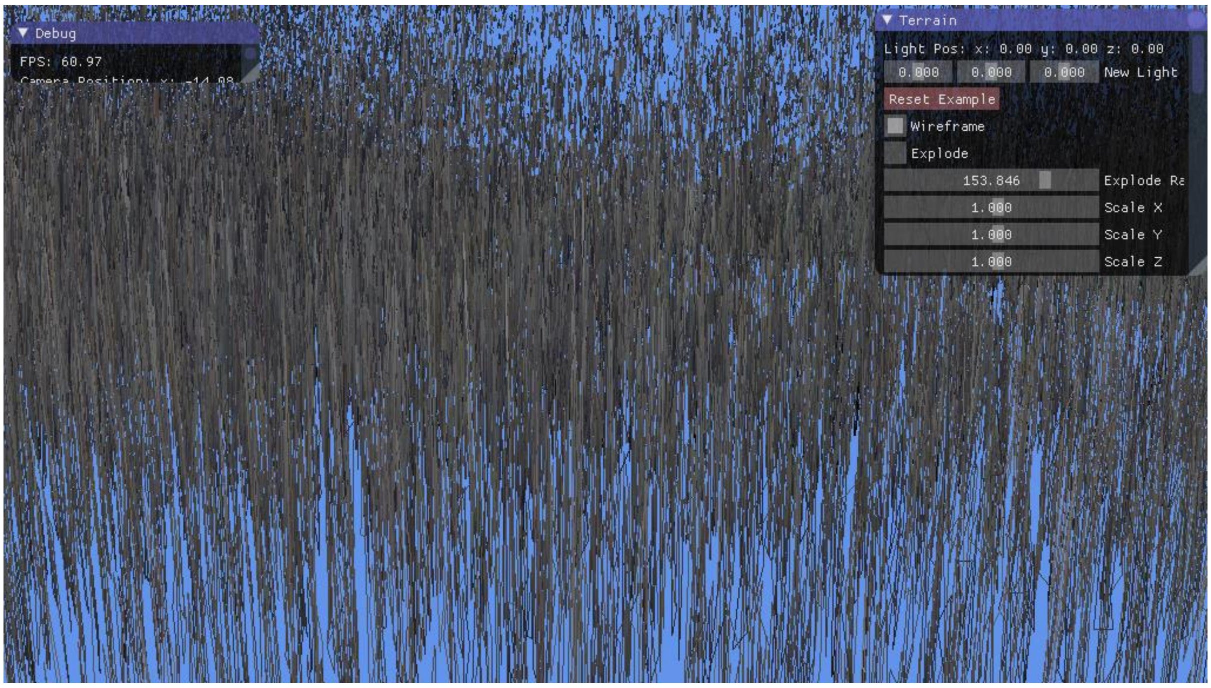
```
▼ Debug
FPS: 60.98
Camera Position: x: -14.08
```

```
▼ Terrain
Light Pos: x: 0.00 y: 0.00 z: 0.00
  0.000     0.000     0.000   New Light
Reset Example
Wireframe
Explode
          0.000             Explode Ra
          1.000             Scale X
          1.000             Scale Y
          1.000             Scale Z
```

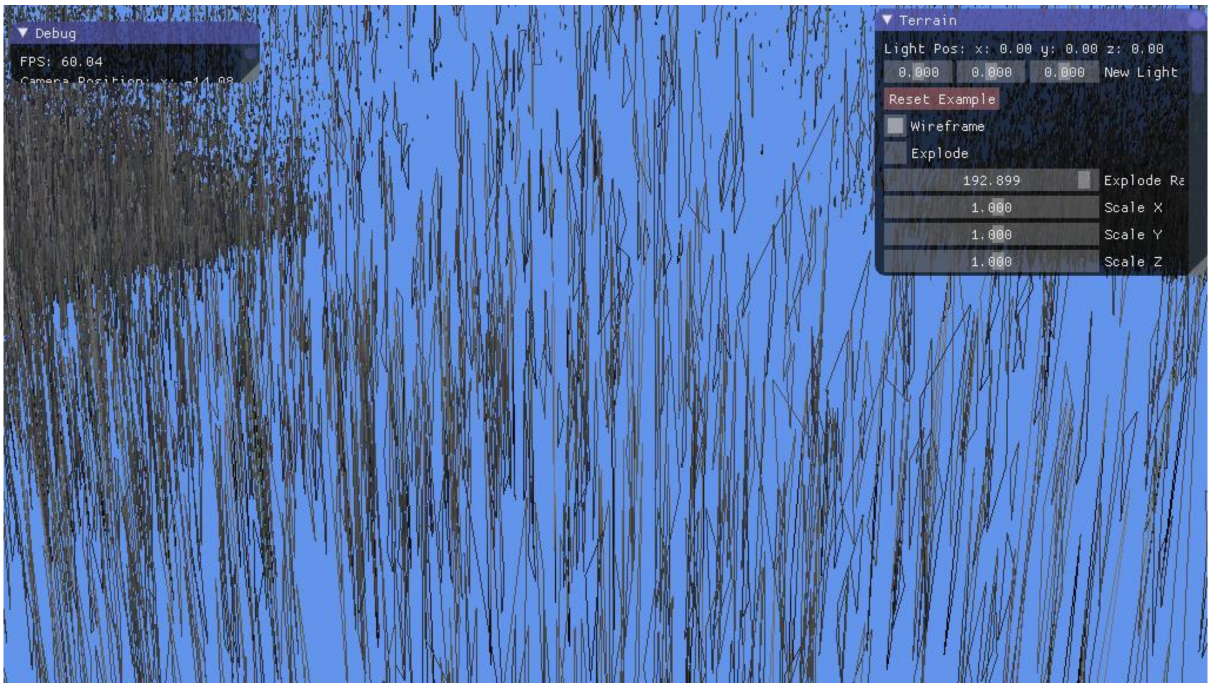


```
▼ Debug
FPS: 60.04
Camera Position: x: -14.08
```

```
▼ Terrain
Light Pos: x: 0.00 y: 0.00 z: 0.00
  0.000     0.000     0.000   New Light
Reset Example
Wireframe
Explode
          74.556            Explode Ra
          1.000             Scale X
          1.000             Scale Y
          1.000             Scale Z
```



```
FPS: 60.97
Camera Position: x: -14.08
Light Pos: x: 0.00 y: 0.00 z: 0.00
0.000   0.000   0.000   New Light
153.846   Explode Ra
1.000   Scale X
1.000   Scale Y
1.000   Scale Z
```



```
FPS: 60.04
Camera Position: x: -14.08
Light Pos: x: 0.00 y: 0.00 z: 0.00
0.000   0.000   0.000   New Light
192.899   Explode Ra
1.000   Scale X
1.000   Scale Y
1.000   Scale Z
```

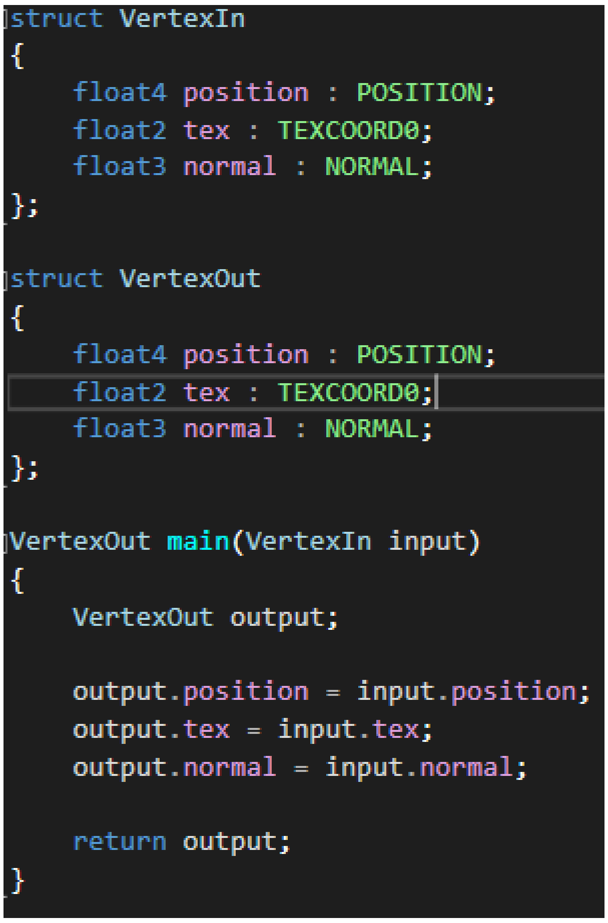
With different height maps the explosion looks different. This example demonstrates

## Shader stages

- Vertex shader
- Hull shader
- Domain shader
- Geometry shader
- Pixel shader

## Shader algorithms and calculations

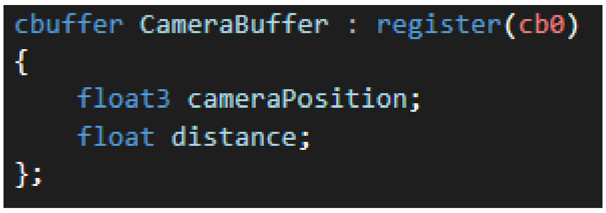
### *Vertex shader*



```
struct VertexIn
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};

struct VertexOut
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};

VertexOut main(VertexIn input)
{
    VertexOut output;

    output.position = input.position;
    output.tex = input.tex;
    output.normal = input.normal;

    return output;
}
```

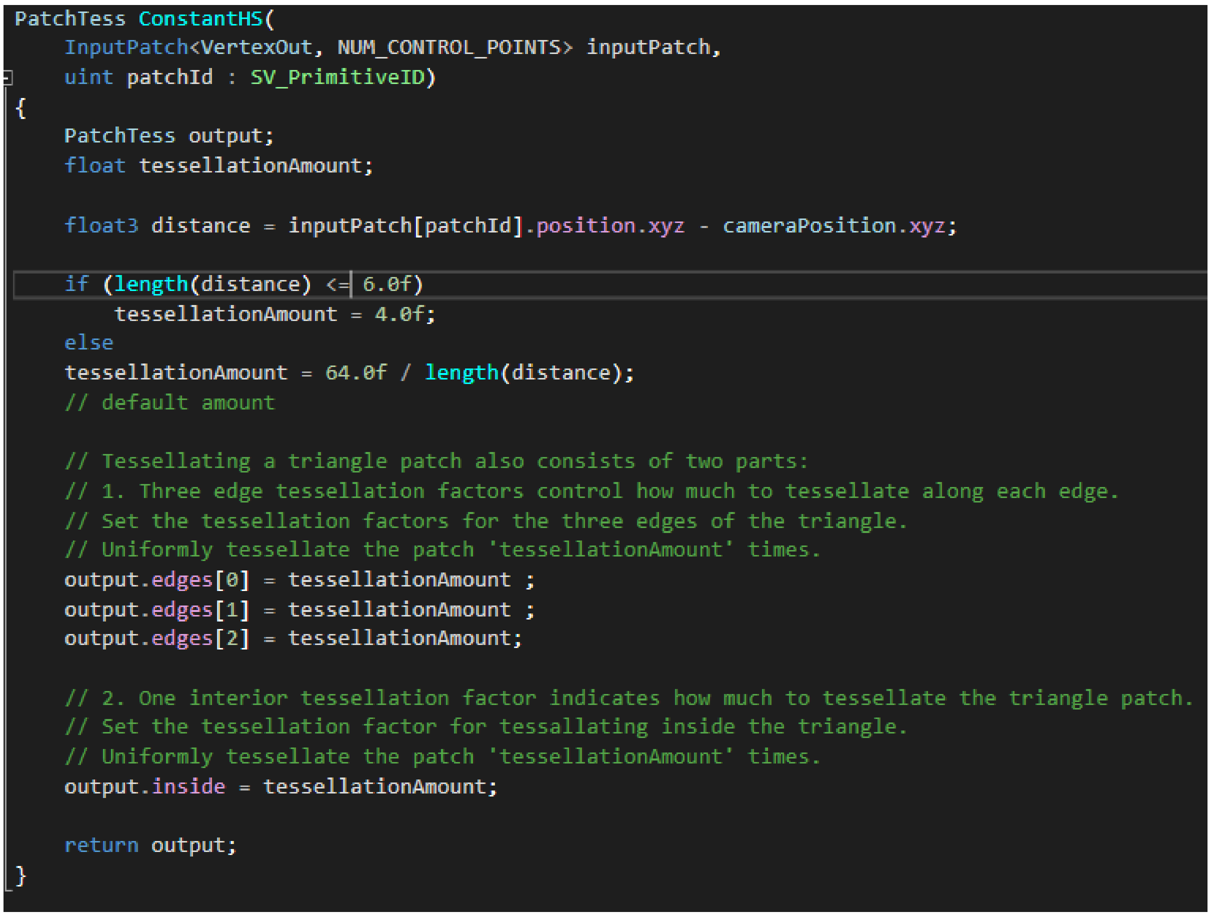
The vertex shader simply passes a structure holding positions, texture coordinate and normal to the next stage, which is the hull shader.

### *Hull shader*



```
cbuffer CameraBuffer : register(cb0)
{
    float3 cameraPosition;
    float distance;
};
```

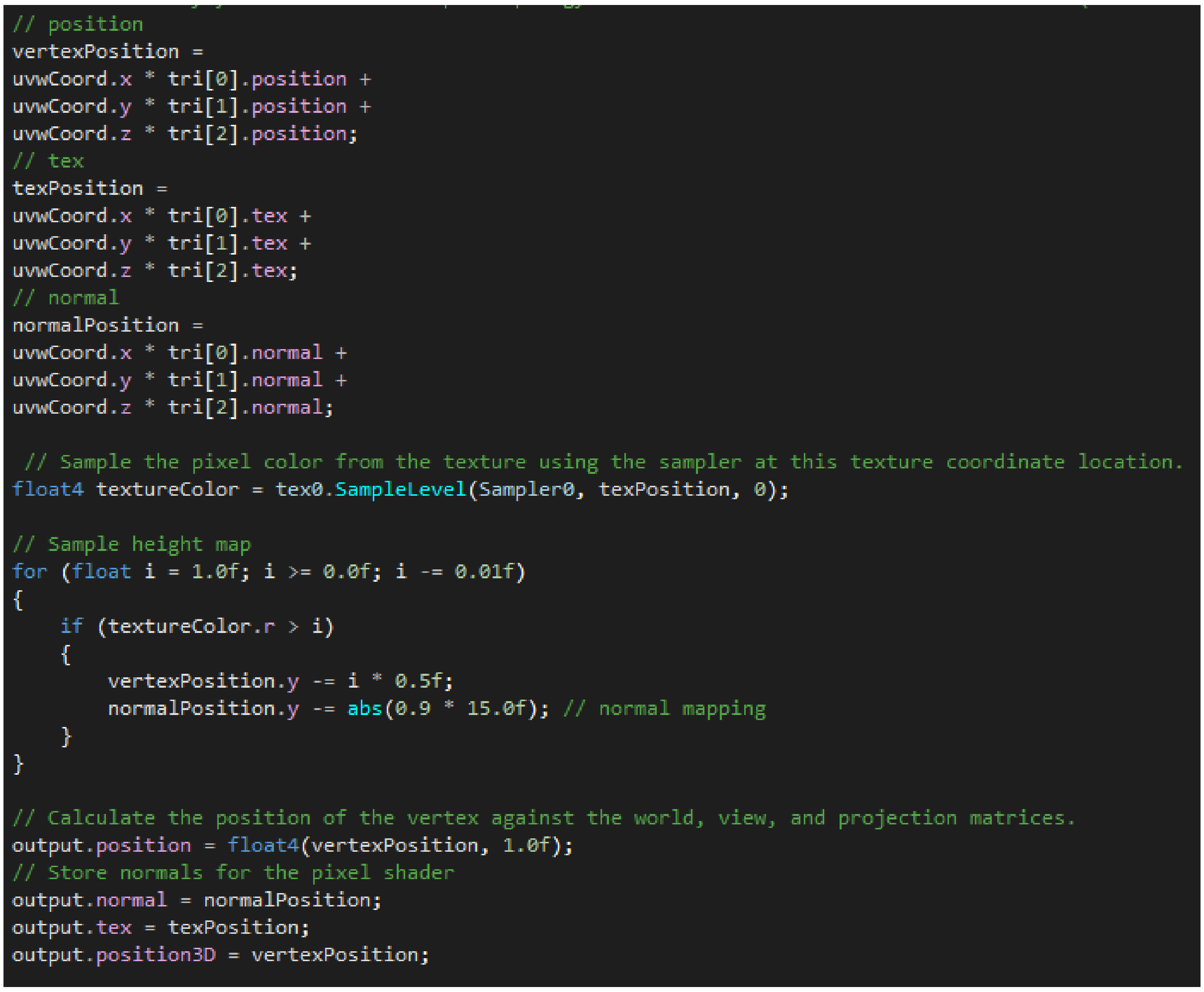
The hull shader receives the camera's position through the constant buffer.



```
PatchTess ConstantHS(
    InputPatch<VertexOut, NUM_CONTROL_POINTS> inputPatch,
    uint patchId : SV_PrimitiveID)
{
    PatchTess output;
    float tessellationAmount;

    float3 distance = inputPatch[patchId].position.xyz - cameraPosition.xyz;

    if (length(distance) <= 6.0f)
        tessellationAmount = 4.0f;
    else
    tessellationAmount = 64.0f / length(distance);
    // default amount

    // Tessellating a triangle patch also consists of two parts:
    // 1. Three edge tessellation factors control how much to tessellate along each edge.
    // Set the tessellation factors for the three edges of the triangle.
    // Uniformly tessellate the patch 'tessellationAmount' times.
    output.edges[0] = tessellationAmount ;
    output.edges[1] = tessellationAmount ;
    output.edges[2] = tessellationAmount;

    // 2. One interior tessellation factor indicates how much to tessellate the triangle patch.
    // Set the tessellation factor for tessallating inside the triangle.
    // Uniformly tessellate the patch 'tessellationAmount' times.
    output.inside = tessellationAmount;

    return output;
}
```

And then uses it to tessellate the terrain based on the distance from the camera. Unfortunately the way it is set up now, the tessellation will not be accurate in the camera is moved to the centre of the terrain.

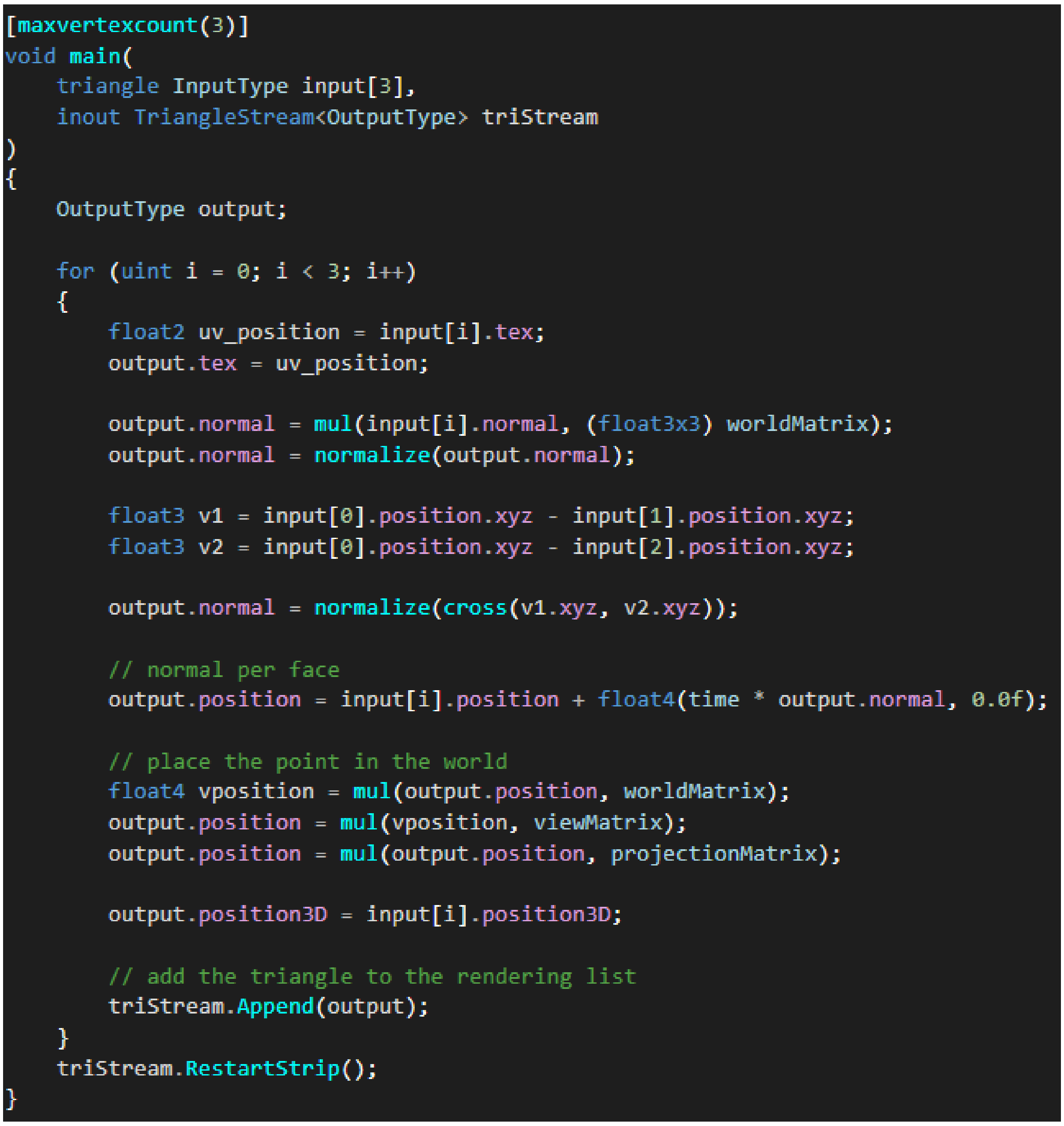
*Domain shader*



```
// position
vertexPosition =
uvwCoord.x * tri[0].position +
uvwCoord.y * tri[1].position +
uvwCoord.z * tri[2].position;
// tex
texPosition =
uvwCoord.x * tri[0].tex +
uvwCoord.y * tri[1].tex +
uvwCoord.z * tri[2].tex;
// normal
normalPosition =
uvwCoord.x * tri[0].normal +
uvwCoord.y * tri[1].normal +
uvwCoord.z * tri[2].normal;

 // Sample the pixel color from the texture using the sampler at this texture coordinate location.
float4 textureColor = tex0.SampleLevel(Sampler0, texPosition, 0);

// Sample height map
for (float i = 1.0f; i >= 0.0f; i -= 0.01f)
{
    if (textureColor.r > i)
    {
        vertexPosition.y -= i * 0.5f;
        normalPosition.y -= abs(0.9 * 15.0f); // normal mapping
    }
}

// Calculate the position of the vertex against the world, view, and projection matrices.
output.position = float4(vertexPosition, 1.0f);
// Store normals for the pixel shader
output.normal = normalPosition;
output.tex = texPosition;
output.position3D = vertexPosition;
```

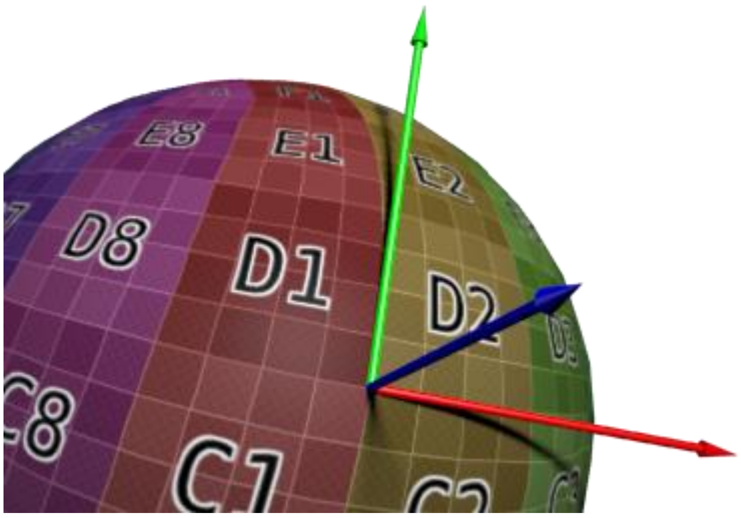
The domain shader offsets the vertex positions and the normal positions based on the height map and passes those newly calculated values to the geometry shader.

### *Geometry shader*



```
[maxvertexcount(3)]
void main(
    triangle InputType input[3],
    inout TriangleStream<OutputType> triStream
)
{
    OutputType output;

    for (uint i = 0; i < 3; i++)
    {
        float2 uv_position = input[i].tex;
        output.tex = uv_position;

        output.normal = mul(input[i].normal, (float3x3) worldMatrix);
        output.normal = normalize(output.normal);

        float3 v1 = input[0].position.xyz - input[1].position.xyz;
        float3 v2 = input[0].position.xyz - input[2].position.xyz;

        output.normal = normalize(cross(v1.xyz, v2.xyz));

        // normal per face
        output.position = input[i].position + float4(time * output.normal, 0.0f);

        // place the point in the world
        float4 vposition = mul(output.position, worldMatrix);
        output.position = mul(vposition, viewMatrix);
        output.position = mul(output.position, projectionMatrix);

        output.position3D = input[i].position3D;

        // add the triangle to the rendering list
        triStream.Append(output);
    }
    triStream.RestartStrip();
}
```

The geometry shader calculates positions of the vertices and texture coordinates. Because it is a triangle stream, the normals are calculated by creating two vectors (out of the triangle's vertices) with the same origin and taking the cross product of those two. This is not the most accurate way to calculate normals (e.g., normal mapping would be more accurate) but it is good enough for the sake of this example.

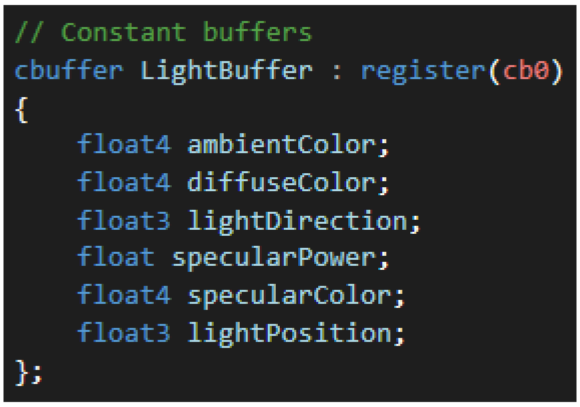


*How normals are calculated (Picture source)*

### *Pixel shader*

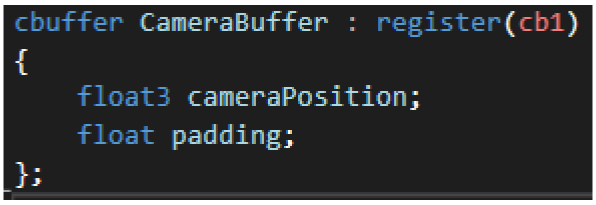
The pixel shader receives 3 constant buffers.

The Light Buffer, which is used for light calculations.



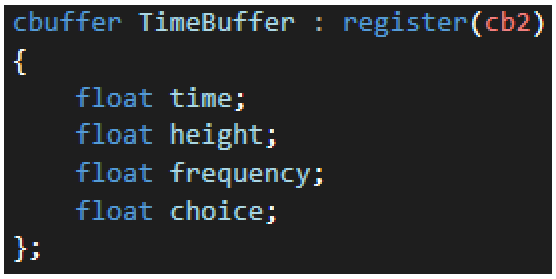
```
// Constant buffers
cbuffer LightBuffer : register(cb0)
{
    float4 ambientColor;
    float4 diffuseColor;
    float3 lightDirection;
    float specularPower;
    float4 specularColor;
    float3 lightPosition;
};
```

The camera buffer, which is used to calculate the view direction.



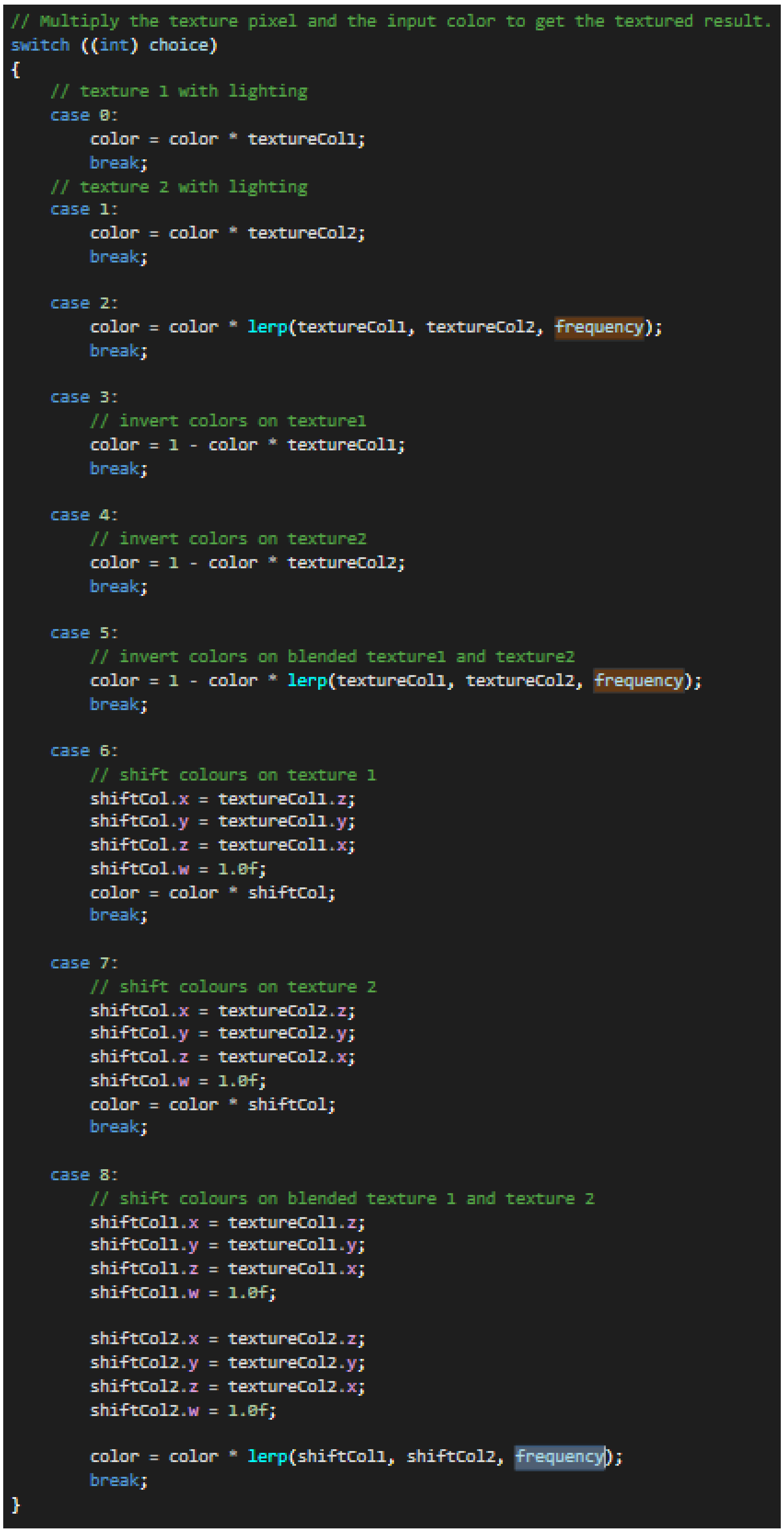
```
cbuffer CameraBuffer : register(cb1)
{
    float3 cameraPosition;
    float padding;
};
```

And the Time Buffer from which only two variables are used – choice and frequency.



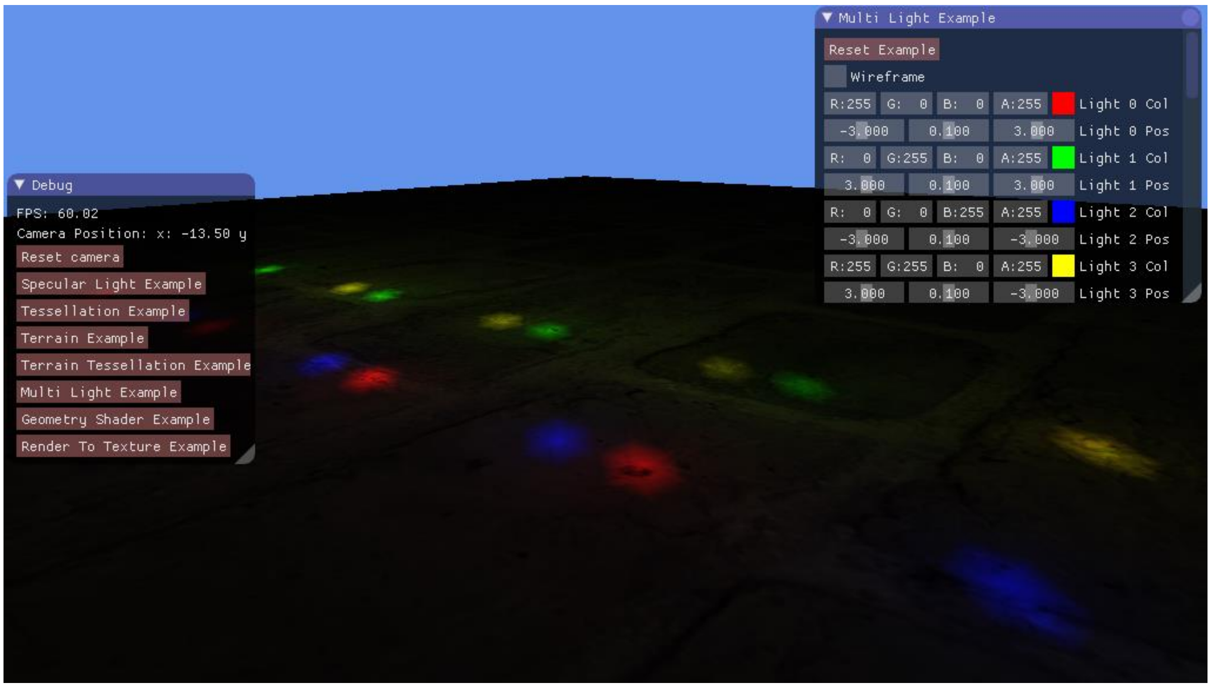
```
cbuffer TimeBuffer : register(cb2)
{
    float time;
    float height;
    float frequency;
    float choice;
};
```

Apart from standard light calculations there's quite an extensive state machine for blending (the blending intensity can be adjusted), colour shifting and inverting, of one or two textures.



```
// Multiply the texture pixel and the input color to get the textured result.
switch ((int) choice)
{
    // texture 1 with lighting
    case 0:
        color = color * textureCol1;
        break;
    // texture 2 with lighting
    case 1:
        color = color * textureCol2;
        break;

    case 2:
        color = color * lerp(textureCol1, textureCol2, frequency);
        break;

    case 3:
        // invert colors on texture1
        color = 1 - color * textureCol1;
        break;

    case 4:
        // invert colors on texture2
        color = 1 - color * textureCol2;
        break;

    case 5:
        // invert colors on blended texture1 and texture2
        color = 1 - color * lerp(textureCol1, textureCol2, frequency);
        break;

    case 6:
        // shift colours on texture 1
        shiftCol.x = textureCol1.z;
        shiftCol.y = textureCol1.y;
        shiftCol.z = textureCol1.x;
        shiftCol.w = 1.0f;
        color = color * shiftCol;
        break;

    case 7:
        // shift colours on texture 2
        shiftCol.x = textureCol2.z;
        shiftCol.y = textureCol2.y;
        shiftCol.z = textureCol2.x;
        shiftCol.w = 1.0f;
        color = color * shiftCol;
        break;

    case 8:
        // shift colours on blended texture 1 and texture 2
        shiftCol1.x = textureCol1.z;
        shiftCol1.y = textureCol1.y;
        shiftCol1.z = textureCol1.x;
        shiftCol1.w = 1.0f;

        shiftCol2.x = textureCol2.z;
        shiftCol2.y = textureCol2.y;
        shiftCol2.z = textureCol2.x;
        shiftCol2.w = 1.0f;

        color = color * lerp(shiftCol1, shiftCol2, frequency);
        break;
}
```

## Multi Light Example



```
FPS: 60.02
Camera Position: x: -13.50 y
```

```
Wireframe
R:255  G:  0  B:  0  A:255  🟥 Light 0 Col
  -3.000    0.100     3.000    Light 0 Pos
R:  0  G:255  B:  0  A:255  🟩 Light 1 Col
   3.000    0.100     3.000    Light 1 Pos
R:  0  G:  0  B:255  A:255  🟦 Light 2 Col
  -3.000    0.100    -3.000    Light 2 Pos
R:255  G:255  B:  0  A:255  🟨 Light 3 Col
   3.000    0.100    -3.000    Light 3 Pos
```

This example contains of 16 lights and different meshes with a brick texture. Each light's colours and position can be changed.

### Shader stages

- Vertex shader
- Pixel shader

### Shader algorithms and calculations

#### *Shader handler*

```
// When the pipeline or a resource is being manipulated,
// the 'device context' is used.
void setShaderParameters(ID3D11DeviceContext* deviceContext, const XMMATRIX &world, const XMMATRIX &view, const XMMATRIX &projection,
    ID3D11ShaderResourceView* texture,
    const std::vector<XMFLOAT4*>& lightColour,
    const std::vector<XMFLOAT4*>& lightPosition);
void render(ID3D11DeviceContext* deviceContext, int vertexCount);
```

There is a XMFLOAT4 vector of light colours and light positions of size 16 which gets populated when setShaderParameters functions is being called. This way the number of lights can be relatively easily changed to more or less than 16 (although there were some memory issues when I tried to have 32 lights).

#### *Vertex Shader and Pixel Shader*

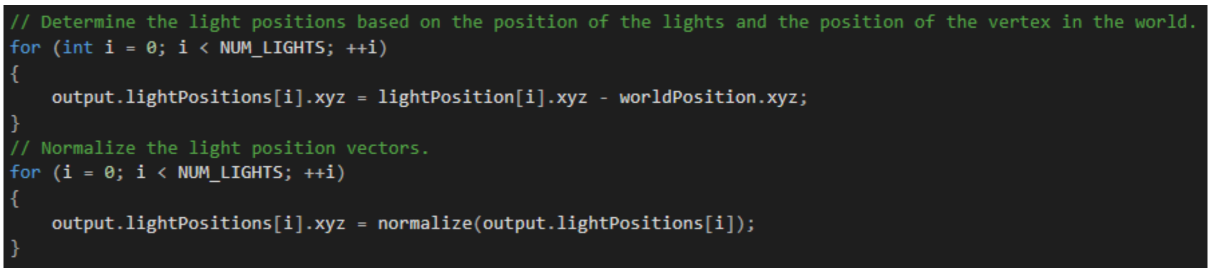
```
// Defines
#define NUM_LIGHTS 16
```

Both vertex and pixel shaders have a define for number of lights called 'NUM_LIGHTS'.

#### *Vertex Shader*

```
cbuffer LightPositionBuffer
{
    float3 lightPosition[NUM_LIGHTS];
};
```

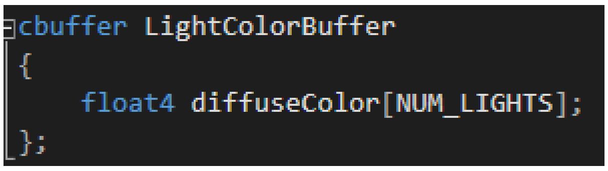
Then in the vertex shader this define is used to create (in the constant buffer) an array of light positions of size 16.



```
// Determine the light positions based on the position of the lights and the position of the vertex in the world.
for (int i = 0; i < NUM_LIGHTS; ++i)
{
    output.lightPositions[i].xyz = lightPosition[i].xyz - worldPosition.xyz;
}
// Normalize the light position vectors.
for (i = 0; i < NUM_LIGHTS; ++i)
{
    output.lightPositions[i].xyz = normalize(output.lightPositions[i]);
}
```

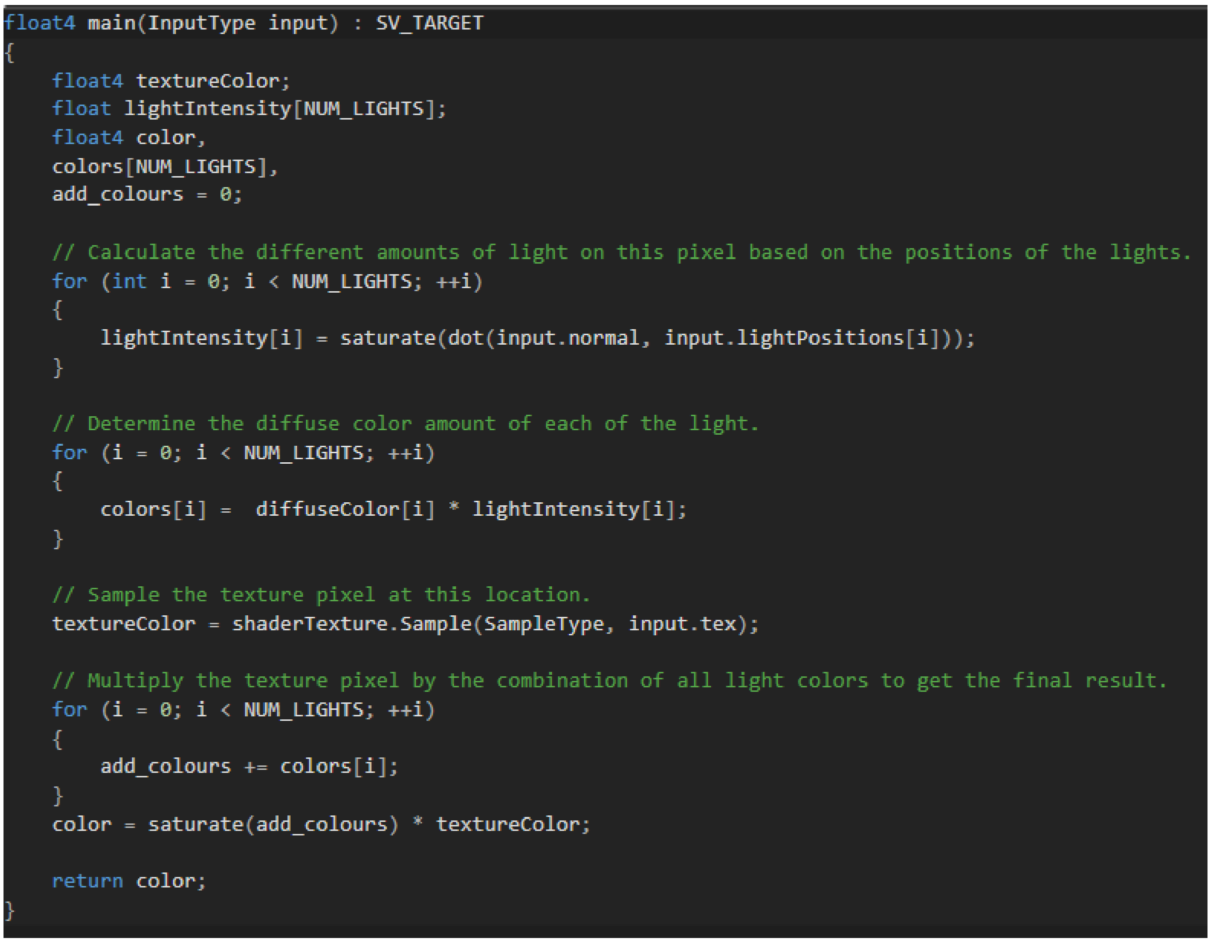
After calculating the vertex's position in the world, the light's position vector is calculated and then normalized.

### *Pixel Shader*



```
cbuffer LightColorBuffer
{
    float4 diffuseColor[NUM_LIGHTS];
};
```

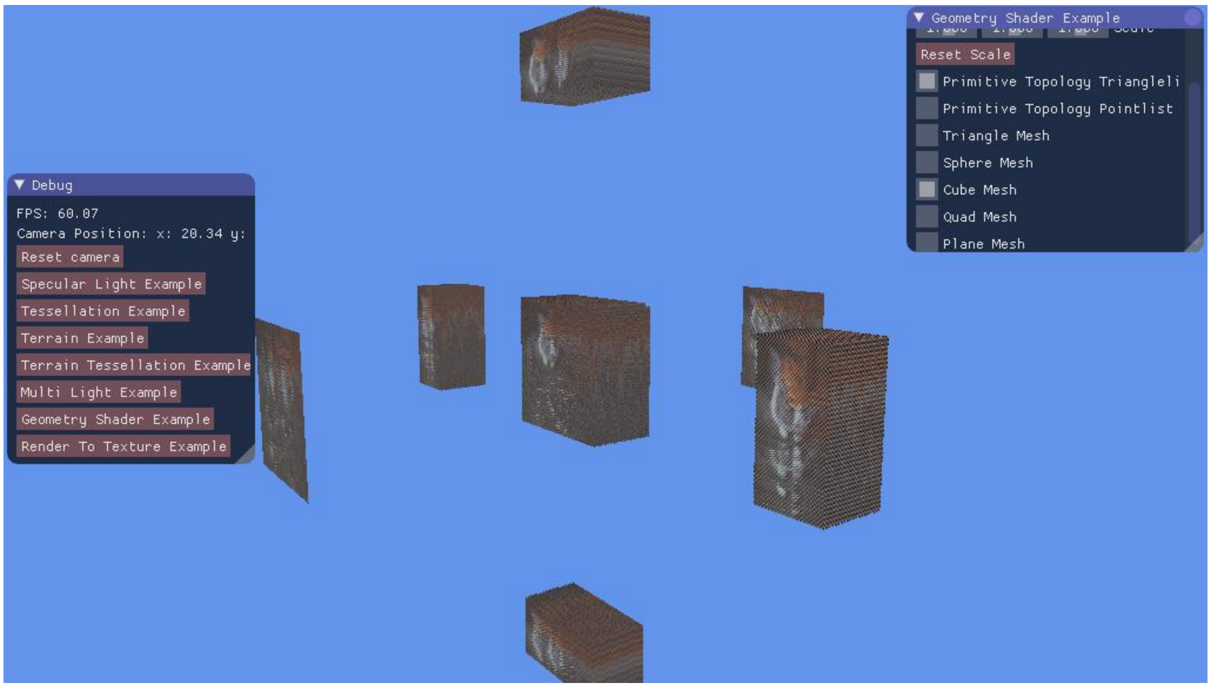
In the pixel shader this define is used to create (in the constant buffer) an array of light colours of size 16.



```
float4 main(InputType input) : SV_TARGET
{
    float4 textureColor;
    float lightIntensity[NUM_LIGHTS];
    float4 color,
    colors[NUM_LIGHTS],
    add_colours = 0;

    // Calculate the different amounts of light on this pixel based on the positions of the lights.
    for (int i = 0; i < NUM_LIGHTS; ++i)
    {
        lightIntensity[i] = saturate(dot(input.normal, input.lightPositions[i]));
    }

    // Determine the diffuse color amount of each of the light.
    for (i = 0; i < NUM_LIGHTS; ++i)
    {
        colors[i] =  diffuseColor[i] * lightIntensity[i];
    }

    // Sample the texture pixel at this location.
    textureColor = shaderTexture.Sample(SampleType, input.tex);

    // Multiply the texture pixel by the combination of all light colors to get the final result.
    for (i = 0; i < NUM_LIGHTS; ++i)
    {
        add_colours += colors[i];
    }
    color = saturate(add_colours) * textureColor;

    return color;
}
```

Then each light's light intensity and diffuse colour is calculated. At the end the texture's colour is sampled and multiplied with light's colour to get the final pixel colour.

## Geometry Shader Example



```
FPS: 60.07
Camera Position: x: 20.34 y:
```

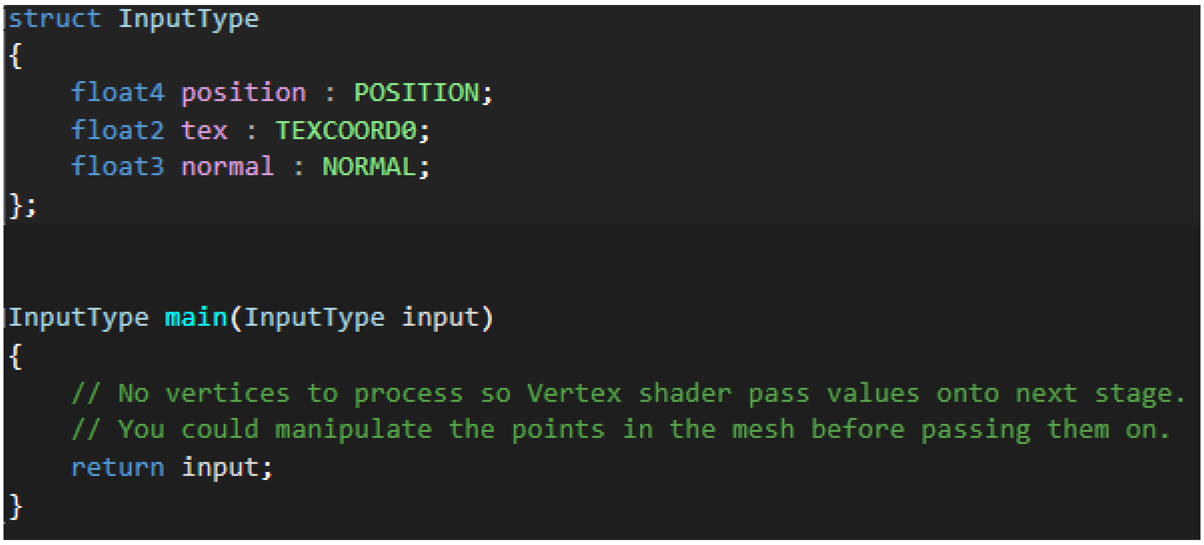
This example is more of an exercise to learn how to use geometry shader and how to properly texture in it by making a simple example of instantiating meshes in the place of a vertex and then instantiating the same mesh again but offset in the normal's direction.

### Shader stages

- Vertex shader
- Geometry shader
- Pixel shader

### Shader algorithms and calculations

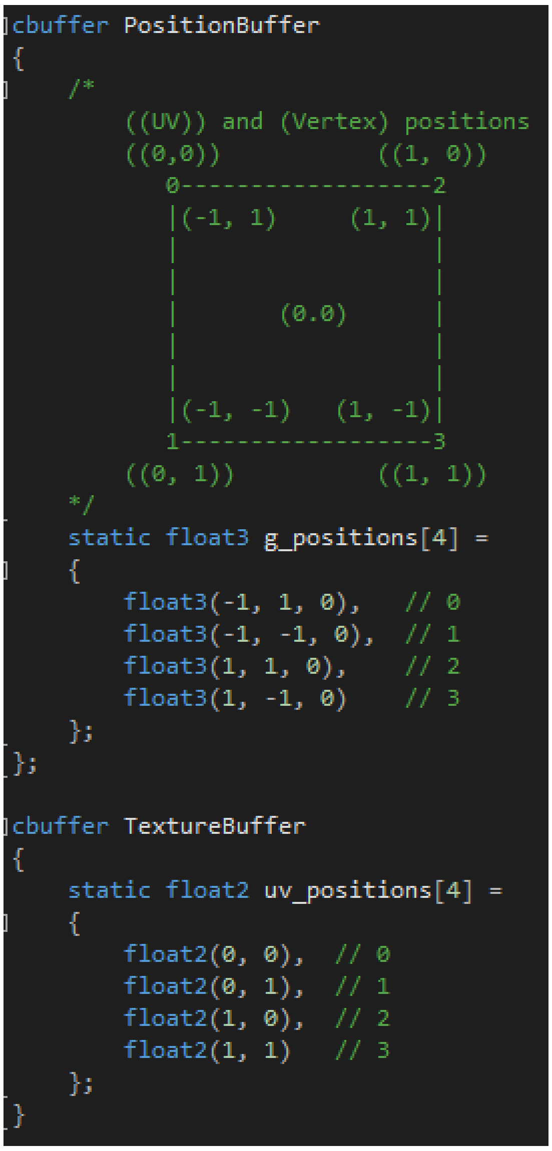
#### *Vertex shader*



```
struct InputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};


InputType main(InputType input)
{
    // No vertices to process so Vertex shader pass values onto next stage.
    // You could manipulate the points in the mesh before passing them on.
    return input;
}
```

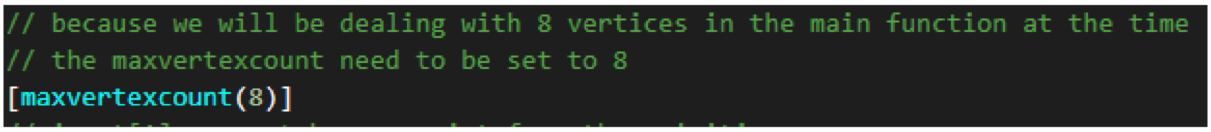
Vertex shader just passes a simple struct of position, texture coordinates and normal to the next stage, which is the geometry shader.

## *Geometry shader*



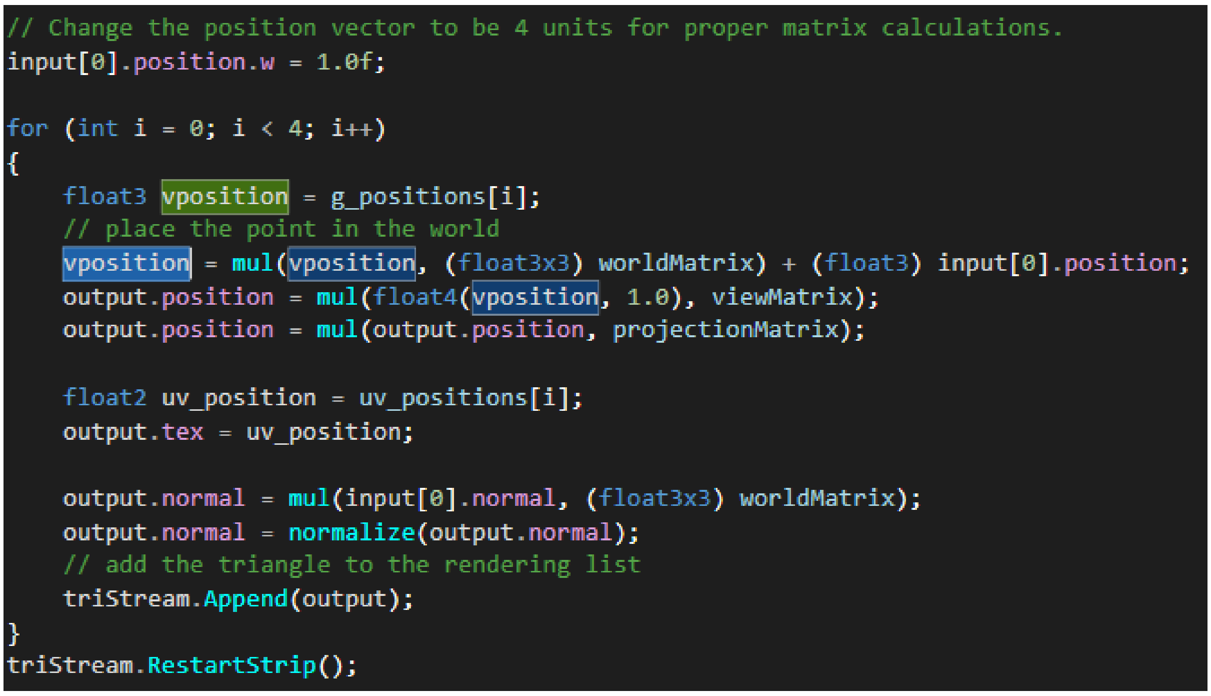
```
cbuffer PositionBuffer
{
    /*
        ((UV)) and (Vertex) positions
        ((0,0))           ((1, 0))
           0------------------2
           |(-1, 1)     (1, 1)|
           |                  |
           |                  |
           |       (0.0)      |
           |                  |
           |                  |
           |(-1, -1)   (1, -1)|
           1------------------3
        ((0, 1))          ((1, 1))
    */
    static float3 g_positions[4] =
    {
        float3(-1, 1, 0),   // 0
        float3(-1, -1, 0),  // 1
        float3(1, 1, 0),    // 2
        float3(1, -1, 0)    // 3
    };
};

cbuffer TextureBuffer
{
    static float2 uv_positions[4] =
    {
        float2(0, 0),  // 0
        float2(0, 1),  // 1
        float2(1, 0),  // 2
        float2(1, 1)   // 3
    };
}
```

Two buffers: Position Buffer and Texture Buffer. The first one is used to instantiate a quad in a place of 4 vertices and the second one is used to properly map this quad.



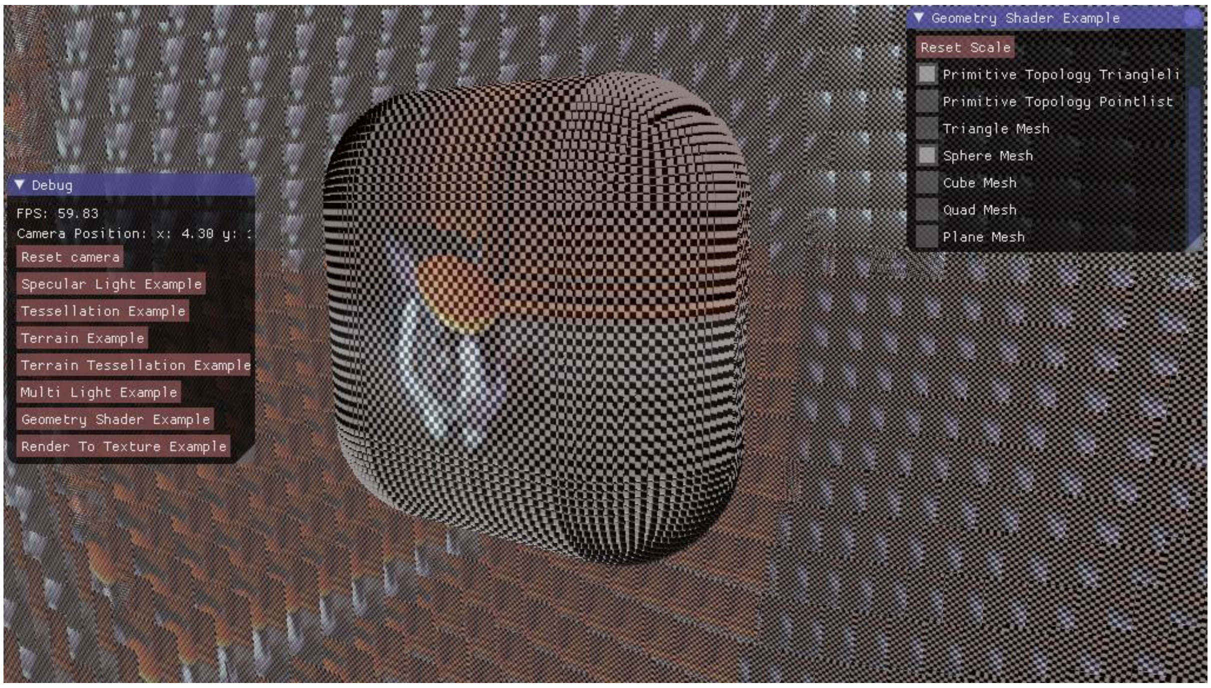
```
// because we will be dealing with 8 vertices in the main function at the time
// the maxvertexcount need to be set to 8
[maxvertexcount(8)]
```

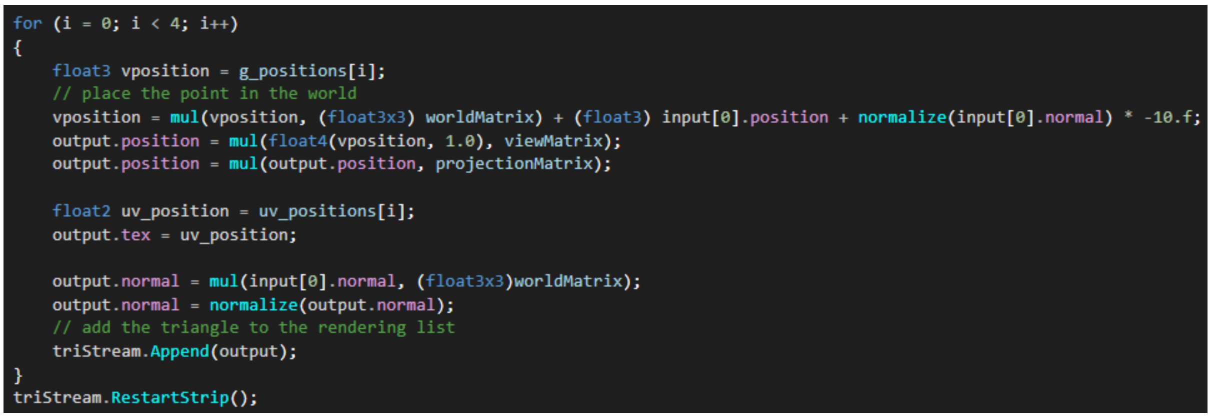
Because the main function deals with 8 vertices at the time, the max vertex count needs to be set to 8, otherwise we will not see all the geometry generated.



```
// Change the position vector to be 4 units for proper matrix calculations.
input[0].position.w = 1.0f;

for (int i = 0; i < 4; i++)
{
    float3 vposition = g_positions[i];
    // place the point in the world
    vposition = mul(vposition, (float3x3) worldMatrix) + (float3) input[0].position;
    output.position = mul(float4(vposition, 1.0), viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    float2 uv_position = uv_positions[i];
    output.tex = uv_position;

    output.normal = mul(input[0].normal, (float3x3) worldMatrix);
    output.normal = normalize(output.normal);
    // add the triangle to the rendering list
    triStream.Append(output);
}
triStream.RestartStrip();
```

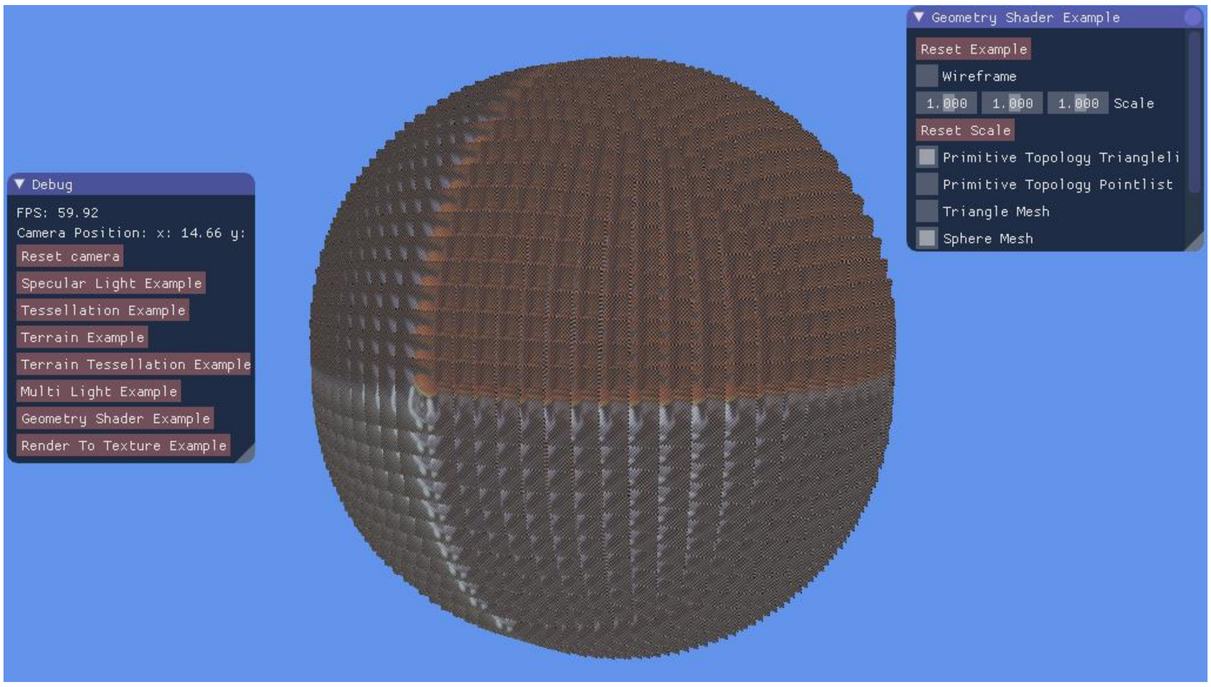
The first for loop generates a quad in a place of four vertices.



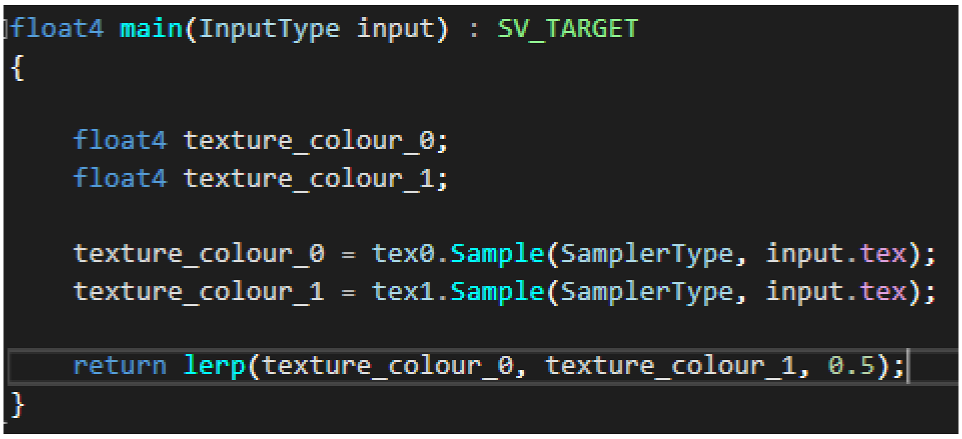


```
for (i = 0; i < 4; i++)
{
    float3 vposition = g_positions[i];
    // place the point in the world
    vposition = mul(vposition, (float3x3) worldMatrix) + (float3) input[0].position + normalize(input[0].normal) * -10.f;
    output.position = mul(float4(vposition, 1.0), viewMatrix);
    output.position = mul(output.position, projectionMatrix);

    float2 uv_position = uv_positions[i];
    output.tex = uv_position;

    output.normal = mul(input[0].normal, (float3x3)worldMatrix);
    output.normal = normalize(output.normal);
    // add the triangle to the rendering list
    triStream.Append(output);
}
triStream.RestartStrip();
```

The second loop generates a quad but offset in the direction of the normal which is also multiplied by -10.



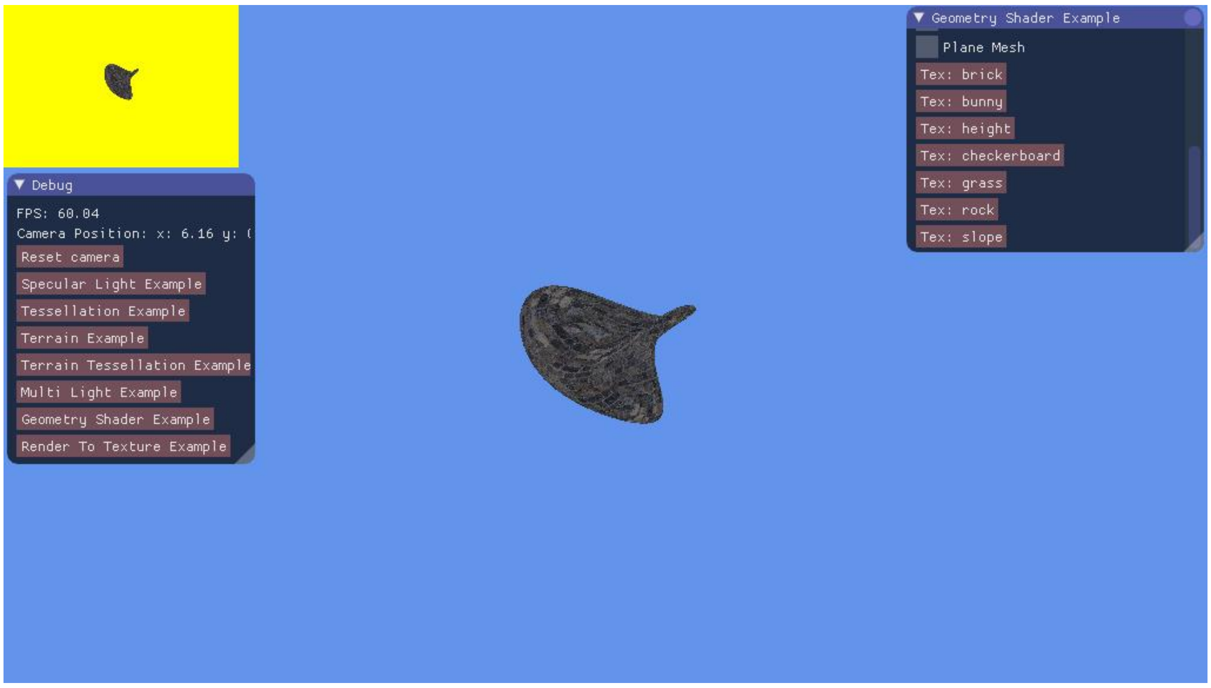
*Pixel Shader*



```
float4 main(InputType input) : SV_TARGET
{

    float4 texture_colour_0;
    float4 texture_colour_1;

    texture_colour_0 = tex0.Sample(SamplerType, input.tex);
    texture_colour_1 = tex1.Sample(SamplerType, input.tex);

    return lerp(texture_colour_0, texture_colour_1, 0.5);
}
```

Pixel shader samples two textures, blends them and applies the blended colour to the pixel.

## Render To Texture Example



```
FPS: 60.04
Camera Position: x: 6.16 y: (
```

This example is a small elaboration from the lab example. The difference is that it is possible to scale the mesh, apply different textures and use different primitive topology for the meshes with various improvement in the shaders.

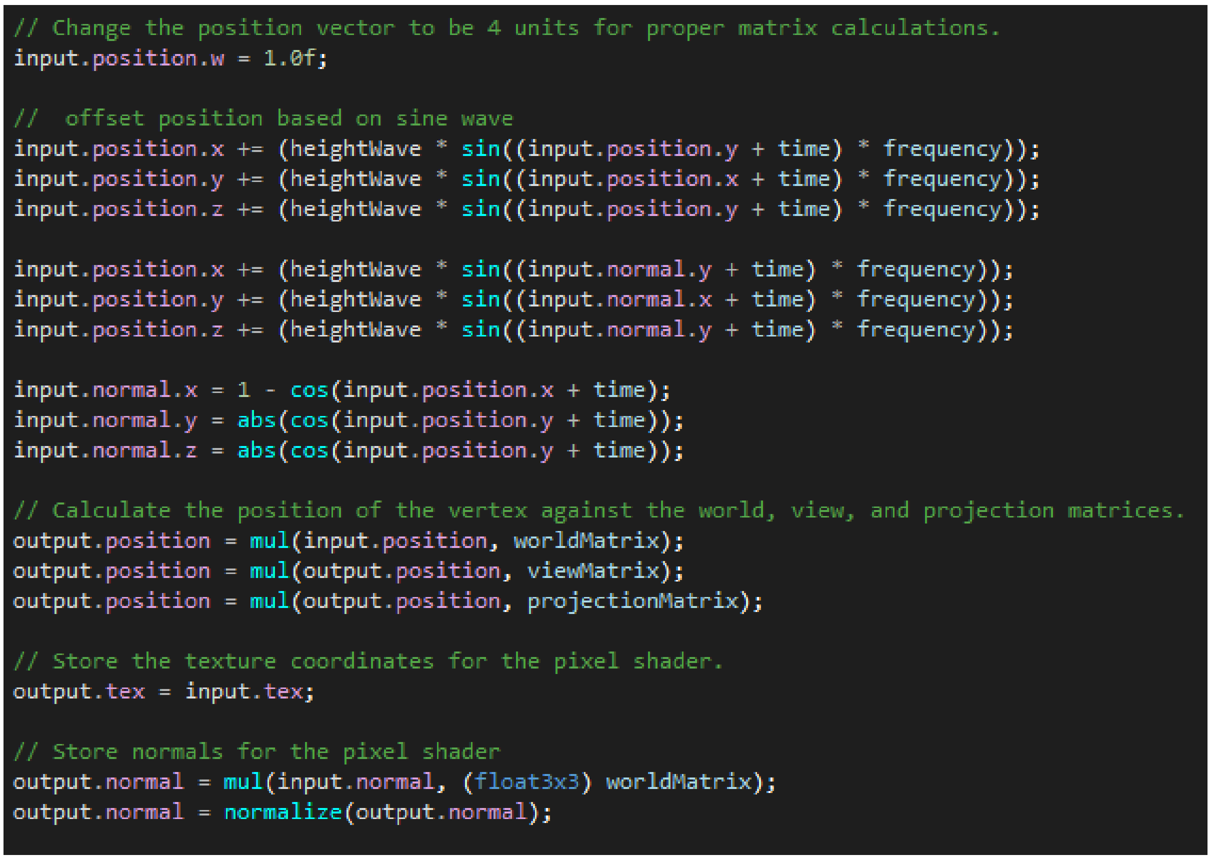
There is still two shader handlers: one for rendering the mesh called Lightshader and the second one for rendering the texture called TextureShader.

### Shader stages

- Vertex shader
- Pixel shader

## Shader algorithms and calculations

### *Light Vertex shader*



```
// Change the position vector to be 4 units for proper matrix calculations.
input.position.w = 1.0f;

//  offset position based on sine wave
input.position.x += (heightWave * sin((input.position.y + time) * frequency));
input.position.y += (heightWave * sin((input.position.x + time) * frequency));
input.position.z += (heightWave * sin((input.position.y + time) * frequency));

input.position.x += (heightWave * sin((input.normal.y + time) * frequency));
input.position.y += (heightWave * sin((input.normal.x + time) * frequency));
input.position.z += (heightWave * sin((input.normal.y + time) * frequency));

input.normal.x = 1 - cos(input.position.x + time);
input.normal.y = abs(cos(input.position.y + time));
input.normal.z = abs(cos(input.position.y + time));

// Calculate the position of the vertex against the world, view, and projection matrices.
output.position = mul(input.position, worldMatrix);
output.position = mul(output.position, viewMatrix);
output.position = mul(output.position, projectionMatrix);

// Store the texture coordinates for the pixel shader.
output.tex = input.tex;

// Store normals for the pixel shader
output.normal = mul(input.normal, (float3x3) worldMatrix);
output.normal = normalize(output.normal);
```

Shader itself has been expanded to change positions on all the axis over time.

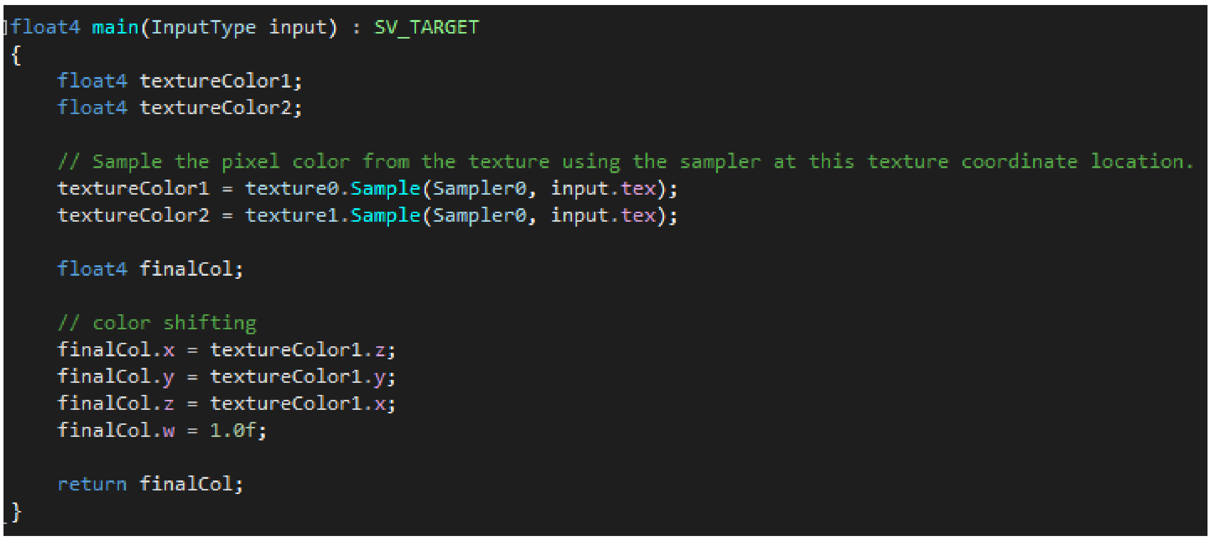
### *Light Pixel Shader*

Standard light calculations.

### *Texture Vertex Shader*

Standard vertex calculations for position, texture coordinate and normals to place them in the world and pass it to the next stage.

### *Texture Pixel Shader*



```
float4 main(InputType input) : SV_TARGET
{
    float4 textureColor1;
    float4 textureColor2;

    // Sample the pixel color from the texture using the sampler at this texture coordinate location.
    textureColor1 = texture0.Sample(Sampler0, input.tex);
    textureColor2 = texture1.Sample(Sampler0, input.tex);

    float4 finalCol;

    // color shifting
    finalCol.x = textureColor1.z;
    finalCol.y = textureColor1.y;
    finalCol.z = textureColor1.x;
    finalCol.w = 1.0f;

    return finalCol;
}
```

A simple colour shifting has been applied to the rendered texture.

## 3. Critical reflection

Overall I am quite happy with the way my application turned out although I recognize that there is still plenty of place for improvement. Here are a few things I could improve:

- Tessellated terrain based on the patch's centre and distance from the camera so it is always tessellating correctly. This might involve calculating the offset vertices in the vertex shader as well as in the domain shader to pass vertex position to the hull shader where the patch could be then correctly placed in the world.
- Lighting on the terrain is not being calculated correctly but I am calculating the normal to my best knowledge.
- I could have come up with more sophisticated example of rendering to the texture but I feel short with time and the fatigue started to get to me at this point.
- There is no example of shadowing which would definitely be a good exercise to do. I have done the lab exercise on shadow mapping but I did not want to just copy and paste it here.

## 4. References

Geeks3d.com. (2018). *Mesh Exploder with Geometry Shaders – Geeks3D*. [online] Available at: http://www.geeks3d.com/20140625/mesh-exploder-with-geometry-shaders/ [Accessed 19 Jan. 2018].

Luna, F. (n.d.). *Introduction to 3D game programming with DirectX® 11*.

Rastertek.com. (2018). *Tutorial Index*. [online] Available at: http://rastertek.com/tutindex.html [Accessed 19 Jan. 2018].

Sherrod, A. and Jones, W. (2005). *Beginning DirectX 11 game programming*. Norwood Mass.: Books24x7.com.

Taking Initiative. (2018). *DirectX10 Tutorial 9: The Geometry Shader*. [online] Available at: https://takinginitiative.wordpress.com/2011/01/12/directx10-tutorial-9-the-geometry-shader/ [Accessed 19 Jan. 2018].

Tutorial 13 : Normal Mapping, *Opengl-tutorial.org*. [online] Available at: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/ [Accessed 20 Jan. 2018].