Advanced Graphics Programming

Name: Athos van Kralingen
Studentnr: 5000685709
Date: 03-04-2016

Table of contents

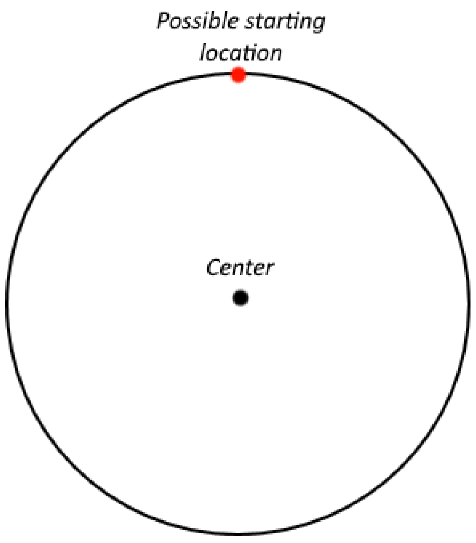
Primitives

For primitives, I would score a 10 given the rubric. The final result is a combination of a prism and the tessellation of it. This does not match the last column of the rubric to the full extent, which indicates that you should *create* geometry on the GPU, but since Tesselation shaders do not allow for introducing new vertices (you can only tell up to divide triangles), this would be near-impossible given the context. Instead, it rotates the prism at different heights, causing it to twist and become deformed.

Rendering the prism

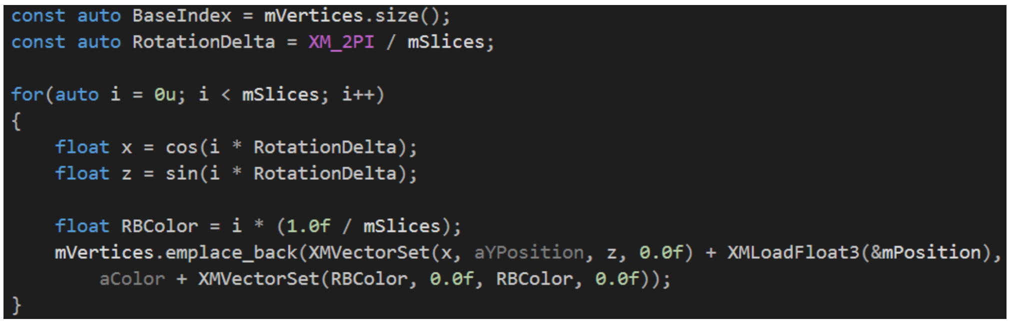
The prism, which itself is calculated on the CPU side, is dynamically calculated based on the amount of vertices for the ground plane. You could also consider this to be the 'tessellating/slices factor' given a cylinder, as the prism approaches a cylinder form when you increase the amount of vertices for the ground plane.

The base of the prism is calculated by taking a starting point for the first vertex. The starting point does not really matter, but it needs to be a point that is within the given radius of the center. In other words, we can take any point on the circle that is formed by the center and the radius of the base, as shown below.



After that, we calculate each subsequent vertex by applying a rotation to the starting point we had. The rotation needed for each vertex is determined by the total amount of vertices of base of the prism. As an example, if we have a hexagonal base (i.e. a base with 6 vertices), we would have to rotate by 360/6° (a full rotation is 360°), which is 60 degrees.

In code, I use the sin and cos functions to perform this operation. That does require us to switch to radians, rather than having to use degrees for the operation.



```
const auto BaseIndex = mVertices.size();
const auto RotationDelta = XM_2PI / mSlices;

for(auto i = 0u; i < mSlices; i++)
{
    float x = cos(i * RotationDelta);
    float z = sin(i * RotationDelta);

    float RBColor = i * (1.0f / mSlices);
    mVertices.emplace_back(XMVectorSet(x, aYPosition, z, 0.0f) + XMLoadFloat3(&mPosition),
        aColor + XMVectorSet(RBColor, 0.0f, RBColor, 0.0f));
}
```

Note that there is no multiplication by some radius, as the radius is constant for this example (1).

The color (RBColor) can be ignored, as it is a random function and only to better show the amount of vertices integrated.

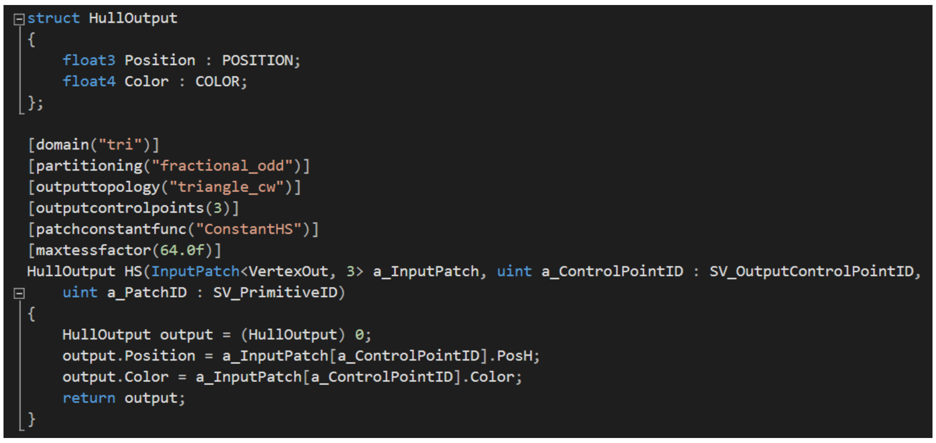
When we have our ‘rotated vertex’, we place it into the list of vertices for the prism. One vertex here consists of a position and a color, in that order.

Lastly, the indices are placed into an index buffer based on the plane normal (due to back/front facing of the triangles). After that, the sides are constructed using the existing vertices, which essentially consists of finding the right vertices to retrieve the indices from.

Tessellating the Prism

The prism is then tessellated on the GPU using the three necessary shaders: the hull shader, the constant hull shader and the domain shader.

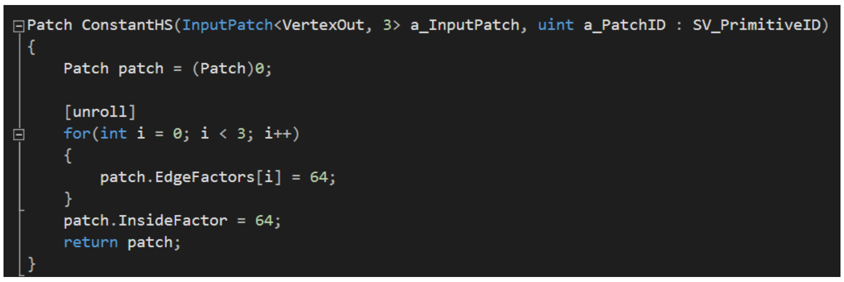
The hull shader



```
struct HullOutput
{
    float3 Position : POSITION;
    float4 Color : COLOR;
};

[domain("tri")]
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("ConstantHS")]
[maxtessfactor(64.0f)]
HullOutput HS(InputPatch<VertexOut, 3> a_InputPatch, uint a_ControlPointID : SV_OutputControlPointID,
    uint a_PatchID : SV_PrimitiveID)
{
    HullOutput output = (HullOutput) 0;
    output.Position = a_InputPatch[a_ControlPointID].PosH;
    output.Color = a_InputPatch[a_ControlPointID].Color;
    return output;
}
```

The hull shader is just a pass through, which is fairly logical since the input topology is the same as the output. This can be seen by looking at the InputPatch which consists of 3 VertexOut structs and by looking at the ‘outputcontrolpoints’ attribute, which is also 3.

The constant hull shader



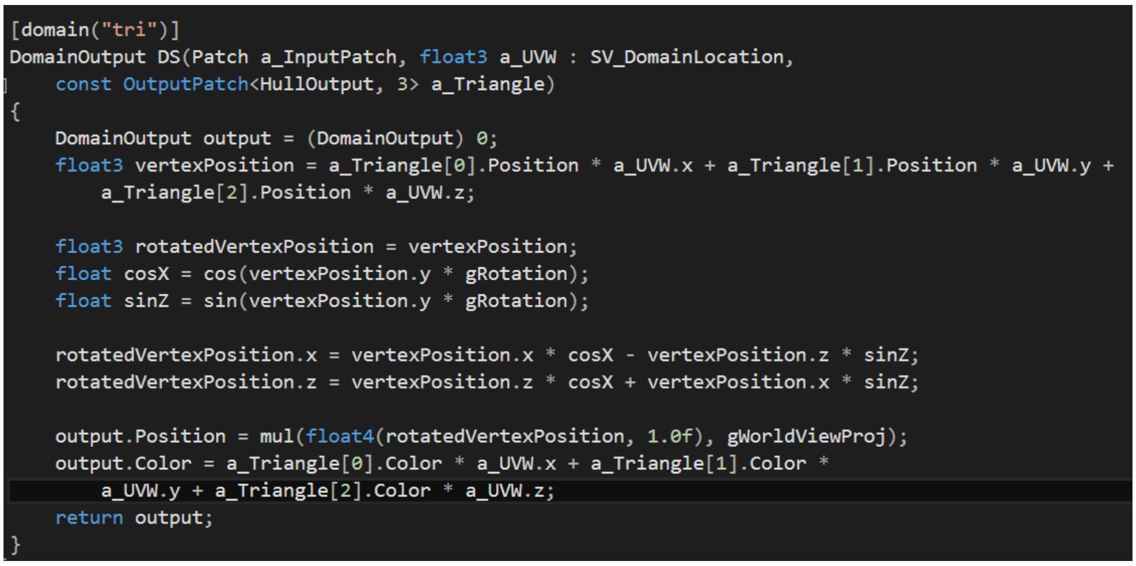
```
Patch ConstantHS(InputPatch<VertexOut, 3> a_InputPatch, uint a_PatchID : SV_PrimitiveID)
{
    Patch patch = (Patch)0;

    [unroll]
    for(int i = 0; i < 3; i++)
    {
        patch.EdgeFactors[i] = 64;
    }
    patch.InsideFactor = 64;
    return patch;
}
```

The constant hull shader is not very special either. All it does is set the tessellation factors for the center (for tessellation with respect to the triangle's center) and that of the subdivision of the edges themselves. 64 is used as this is the usual restraint of GPU's.

In case the demo crashes, though unlikely for modern GPU's, you should try reducing these factors to 32 or lower powers of two.

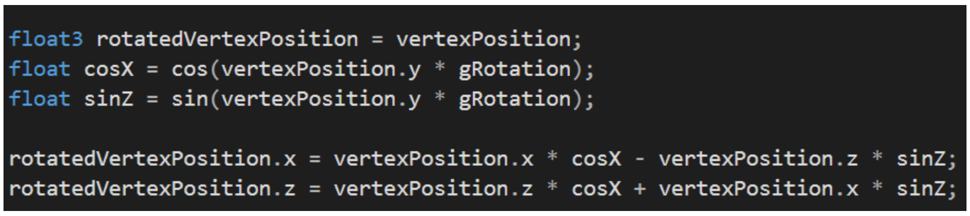
The domain shader

The domain shader is where the twisting of the prism takes place. It displaces each vertex based on its with respect to the base of the prism. The reason this works is because the vertex position hasn't been transformed to NDC yet, which will happen after the deformation.



```
[domain("tri")]
DomainOutput DS(Patch a_InputPatch, float3 a_UVW : SV_DomainLocation,
    const OutputPatch<HullOutput, 3> a_Triangle)
{
    DomainOutput output = (DomainOutput) 0;
    float3 vertexPosition = a_Triangle[0].Position * a_UVW.x + a_Triangle[1].Position * a_UVW.y +
        a_Triangle[2].Position * a_UVW.z;

    float3 rotatedVertexPosition = vertexPosition;
    float cosX = cos(vertexPosition.y * gRotation);
    float sinZ = sin(vertexPosition.y * gRotation);

    rotatedVertexPosition.x = vertexPosition.x * cosX - vertexPosition.z * sinZ;
    rotatedVertexPosition.z = vertexPosition.z * cosX + vertexPosition.x * sinZ;

    output.Position = mul(float4(rotatedVertexPosition, 1.0f), gWorldViewProj);
    output.Color = a_Triangle[0].Color * a_UVW.x + a_Triangle[1].Color *
        a_UVW.y + a_Triangle[2].Color * a_UVW.z;
    return output;
}
```

The rotation



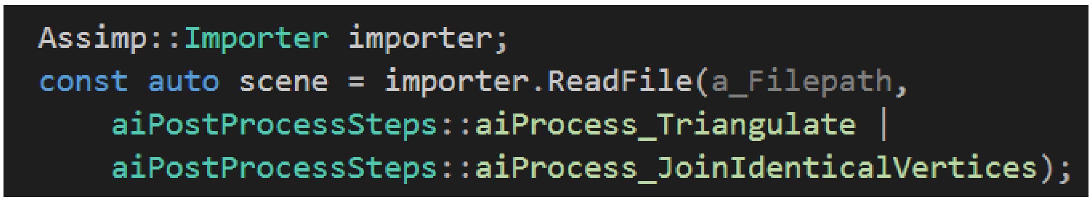
```
float3 rotatedVertexPosition = vertexPosition;
float cosX = cos(vertexPosition.y * gRotation);
float sinZ = sin(vertexPosition.y * gRotation);

rotatedVertexPosition.x = vertexPosition.x * cosX - vertexPosition.z * sinZ;
rotatedVertexPosition.z = vertexPosition.z * cosX + vertexPosition.x * sinZ;
```

This piece of code is responsible for the entire rotation of each vertex. I could elaborate on the details, but it should suffice to say that this is essentially a multiplication of a 2d vector by a 2d rotation matrix (about the y-axis). The gRotation vector is fed through a constant buffer from the CPU side, used to control the strength of the rotation.

Models

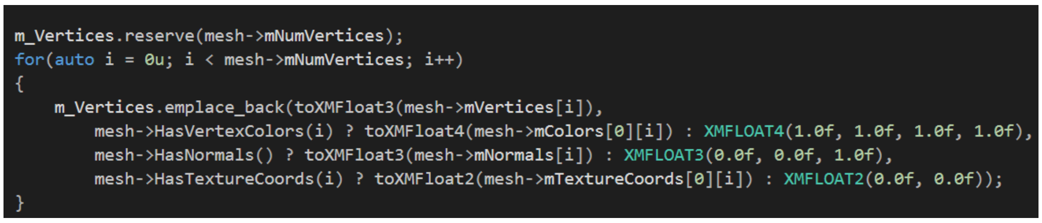
Following the rubric, I would score a 10 for the models assignment. Models are loaded by the use of the Open Asset Import Library (ASSIMP). There is not a whole lot of code to show, since the majority of getting this working is setting up the static and dynamically linked libraries to work with the program (which is not related to rendering in any way).

Importing



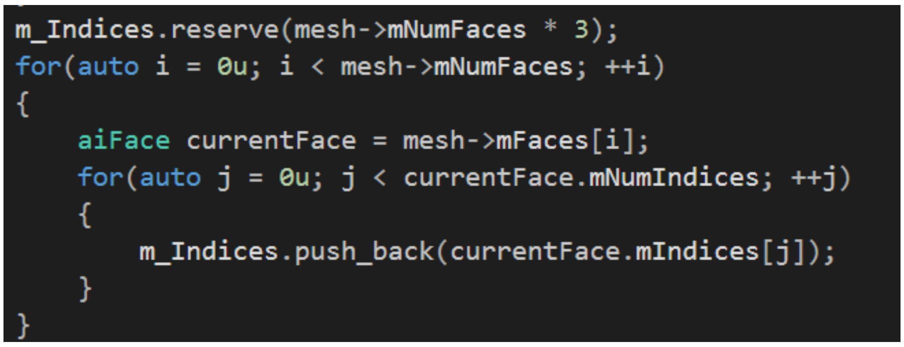
```
Assimp::Importer importer;
const auto scene = importer.ReadFile(a_Filepath,
    aiPostProcessSteps::aiProcess_Triangulate |
    aiPostProcessSteps::aiProcess_JoinIdenticalVertices);
```

First, we create an Assimp importer to load the file itself. We then instruct it to load the file itself, using several import flags. The first is used to generated triangles out of non-triangular faces. The second flag is to make sure to generate indices for the vertices, which removes the need for identical vertices. The latter can give some major performance boosts, but may have a longer loading time.



```
m_Vertices.reserve(mesh->mNumVertices);
for(auto i = 0u; i < mesh->mNumVertices; i++)
{
    m_Vertices.emplace_back(toXMFloat3(mesh->mVertices[i]),
        mesh->HasVertexColors(i) ? toXMFloat4(mesh->mColors[0][i]) : XMFLOAT4(1.0f, 1.0f, 1.0f, 1.0f),
        mesh->HasNormals() ? toXMFloat3(mesh->mNormals[i]) : XMFLOAT3(0.0f, 0.0f, 1.0f),
        mesh->HasTextureCoords(i) ? toXMFloat2(mesh->mTextureCoords[0][i]) : XMFLOAT2(0.0f, 0.0f));
}
```

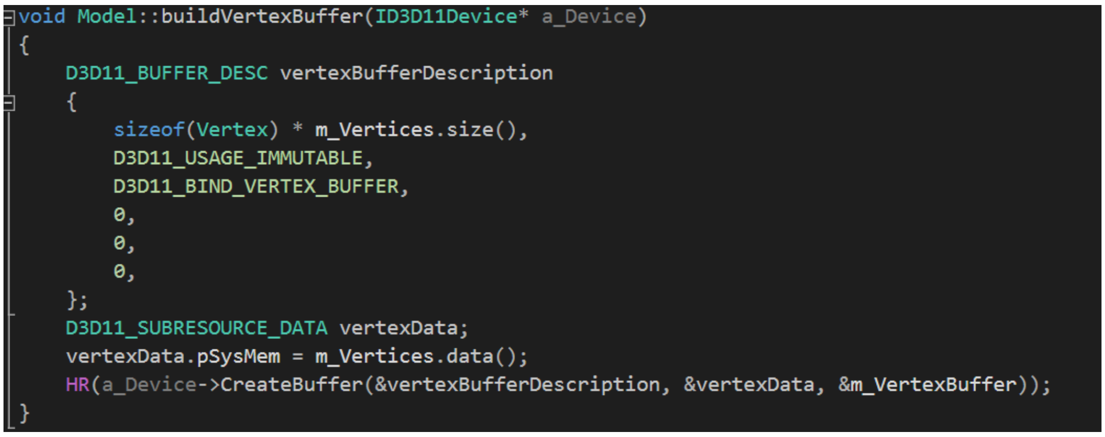
The above code then iterates over all the loaded vertices and based on whether the given data exists, adds them to the list of vertices. The reserve call before the loop is just as an optimization, so that we aren't continuously allocating and deallocating memory (which is incredibly slow).



```
m_Indices.reserve(mesh->mNumFaces * 3);
for(auto i = 0u; i < mesh->mNumFaces; ++i)
{
    aiFace currentFace = mesh->mFaces[i];
    for(auto j = 0u; j < currentFace.mNumIndices; ++j)
    {
        m_Indices.push_back(currentFace.mIndices[j]);
    }
}
```

The above code does the exact same thing, but instead iterates over the faces and its indices instead, to continue placing them in the list/vector of indices.

Finally, the index and vertex buffers are built using the loaded data.

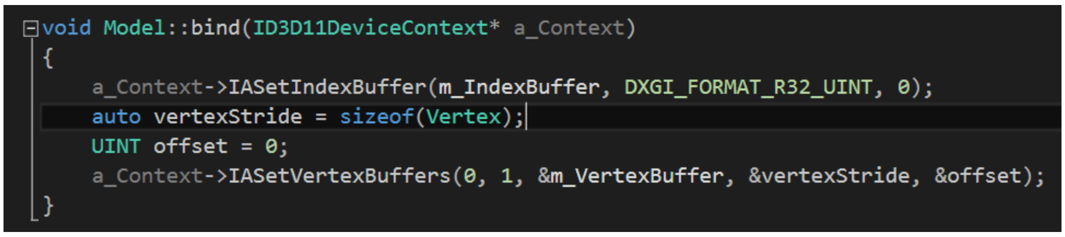


```
void Model::buildVertexBuffer(ID3D11Device* a_Device)
{
    D3D11_BUFFER_DESC vertexBufferDescription
    {
        sizeof(Vertex) * m_Vertices.size(),
        D3D11_USAGE_IMMUTABLE,
        D3D11_BIND_VERTEX_BUFFER,
        0,
        0,
        0,
    };
    D3D11_SUBRESOURCE_DATA vertexData;
    vertexData.pSysMem = m_Vertices.data();
    HR(a_Device->CreateBuffer(&vertexBufferDescription, &vertexData, &m_VertexBuffer));
}
```

The code for the index buffer is near identical, so is not shown.

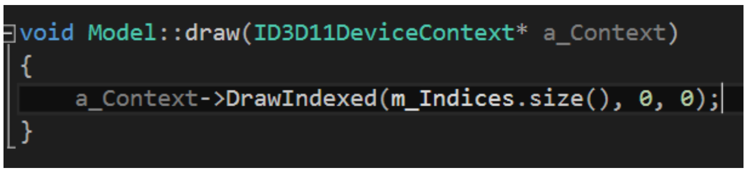
Drawing

The drawing from hereon is pretty straightforward. We first allow external code to bind the vertex and index buffers of this model.



```
void Model::bind(ID3D11DeviceContext* a_Context)
{
    a_Context->IASetIndexBuffer(m_IndexBuffer, DXGI_FORMAT_R32_UINT, 0);
    auto vertexStride = sizeof(Vertex);
    UINT offset = 0;
    a_Context->IASetVertexBuffers(0, 1, &m_VertexBuffer, &vertexStride, &offset);
}
```

And draw afterwards.



```
void Model::draw(ID3D11DeviceContext* a_Context)
{
    a_Context->DrawIndexed(m_Indices.size(), 0, 0);
}
```

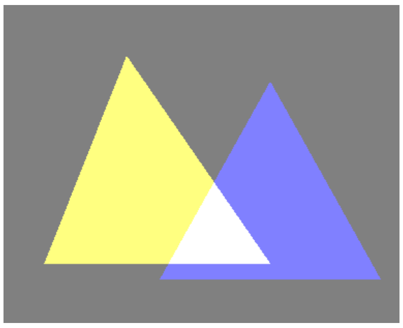
Blending & Stenciling

For this assignment, I would also score a 10 according to the rubric. Most of the relevant blending techniques have been tested (additive, multiplicative, double multiplicative and transparent) by drawing two triangles on the screen. A stencil buffer is used to block drawing to a triangular form on the screen. The latter can also cause the space inside the triangle to be blended and outside of the triangle to be solid, using multiple render passes.

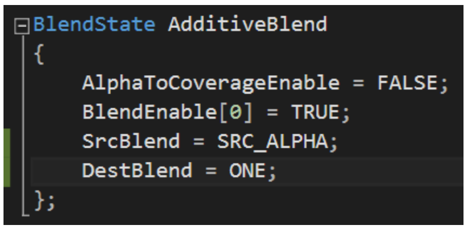
Blending

For all tests, the original color of the first triangle was 0.0, 0.0, 1.0 with an alpha of 1.0. The second triangle had a color of 1.0, 1.0, 0.0 and alpha 0.5. The background color was 0.5, 0.5, 0.5 (the alpha is not used). I will show the results of each mode, along with the blend state values used to create it.

Additive



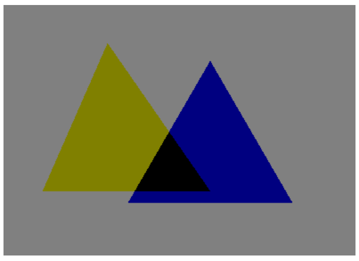
The result is as expected. In the middle, it turns white-ish, due to having 'a bit of every channel' (the triangles together use red, green and blue).



```
BlendState AdditiveBlend
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = TRUE;
    SrcBlend = SRC_ALPHA;
    DestBlend = ONE;
};
```

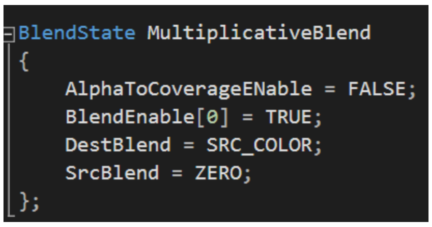
This was used to create the effect. The ONE for the source blend indicates that we retain the original color and simply add the other color to it, multiplied by its alpha. Note that another way of implementing this is by simply adding the colors, disregarding the alpha. That would give similar, but not the same result.

Multiplicative blending



Again the result is what I expected. The triangles become darker, because the background color is a shade of gray, though not white, so each channel's value is reduced. The overlap is an interesting effect, but still as expected. This is because the triangles do not have any matching channels.

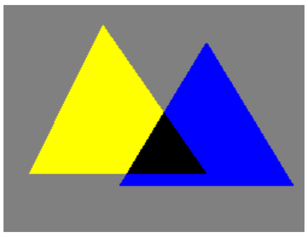
For example, because the yellow triangle has no blue, the blue of the blue triangle will be multiplied by 0. That causes it to become pitch black.



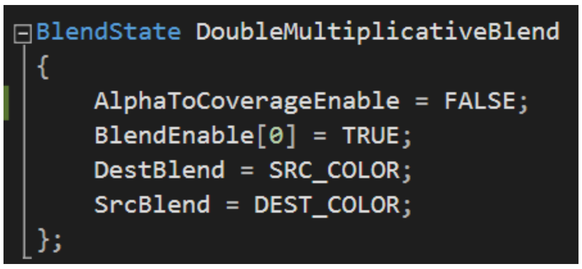
```
BlendState MultiplicativeBlend
{
    AlphaToCoverageENable = FALSE;
    BlendEnable[0] = TRUE;
    DestBlend = SRC_COLOR;
    SrcBlend = ZERO;
};
```

The state makes use of the fact that the factor behind the assignment operator is what is multiplied. This way, we multiply the source by the destination color channels. The source blend is thrown away, since it will already be multiplied at the DestBlend factor.

Double multiplicative blending



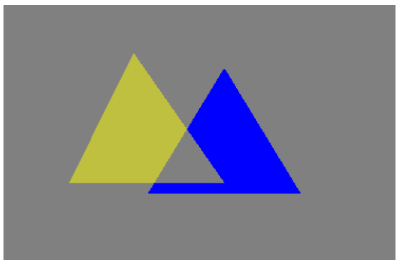
The result is very similar to regular multiplicative blending, with the only difference being that the triangles have a brighter color. This is because we basically double the result of our multiplication, which would always result in a brighter color than with just the multiplication.



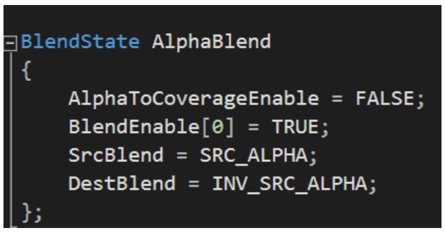
```
BlendState DoubleMultiplicativeBlend
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = TRUE;
    DestBlend = SRC_COLOR;
    SrcBlend = DEST_COLOR;
};
```

The only difference is that we also multiply the source color by the destination, which causes the effect to be ‘doubled’ once the addition of both terms is performed.

Alpha blending



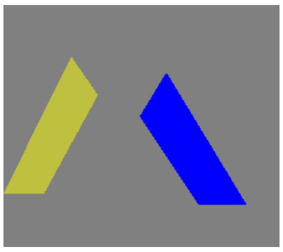
Here we can clearly see the effect of the alpha value of the yellow triangle. It is also clear that the yellow triangle is drawn last. The results are as expected; the yellow somewhat mixes with the blue triangle and is slightly darkened and faded by the background.



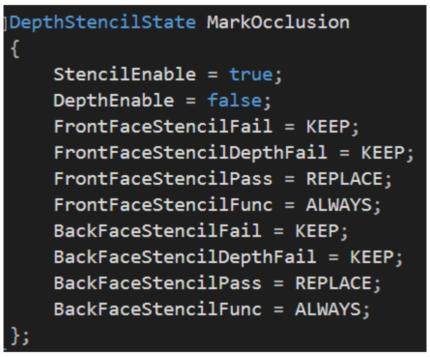
```
BlendState AlphaBlend
{
    AlphaToCoverageEnable = FALSE;
    BlendEnable[0] = TRUE;
    SrcBlend = SRC_ALPHA;
    DestBlend = INV_SRC_ALPHA;
};
```

It should be noted that different to OpenGL, the destination blend is called inverse source alpha. This is simply one minus the source alpha.

Stenciling



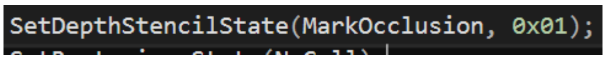
As you can see, there is a triangle in the middle cut out of both, blocking rendering to it. This triangle uses the following depthstencilstate (for which a different shader is used).



```
DepthStencilState MarkOcclusion
{
    StencilEnable = true;
    DepthEnable = false;
    FrontFaceStencilFail = KEEP;
    FrontFaceStencilDepthFail = KEEP;
    FrontFaceStencilPass = REPLACE;
    FrontFaceStencilFunc = ALWAYS;
    BackFaceStencilFail = KEEP;
    BackFaceStencilDepthFail = KEEP;
    BackFaceStencilPass = REPLACE;
    BackFaceStencilFunc = ALWAYS;
};
```

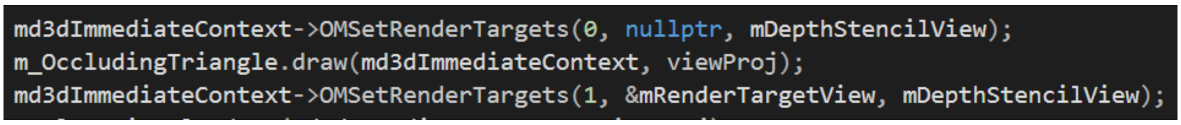
This is to make sure it will 'mark' the rasterized area with the given stencil reference value. Depth is enabled, as we want it to *always* block this part, no matter where we view it from (even from the back). It is also disabled for the other triangles, for obvious reasons regarding blending itself.

We then set it as follows:



```
SetDepthStencilState(MarkOcclusion, 0x01);
```

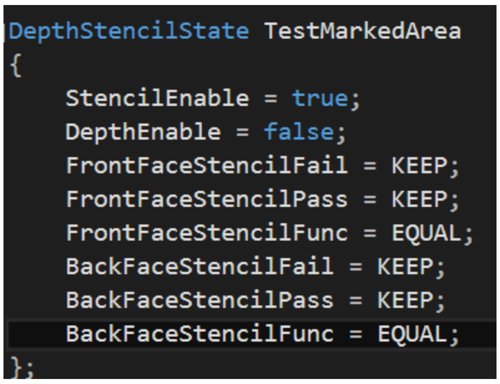
Which makes it mark the rasterized area with 1's. To make sure it does not write any color to the render target, we bind null as render target so that it only updates the z-buffer (if depth testing were to be enabled) and the stencil buffer. A nice addition is that this is also optimized on the GPU side.*



```
md3dImmediateContext->OMSetRenderTargets(0, nullptr, mDepthStencilView);
m_OccludingTriangle.draw(md3dImmediateContext, viewProj);
md3dImmediateContext->OMSetRenderTargets(1, &mRenderTargetView, mDepthStencilView);
```

**If it weren't for D3D warning you every frame in the output log that there is no bound render target*

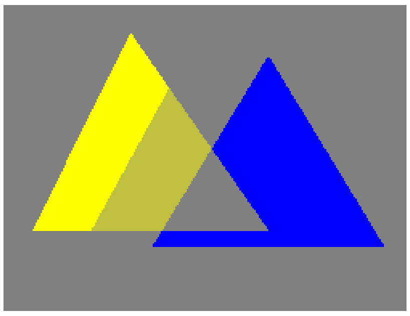
In the shader for the other triangles, we test against this value with the following DepthStencilState.



```
DepthStencilState TestMarkedArea
{
    StencilEnable = true;
    DepthEnable = false;
    FrontFaceStencilFail = KEEP;
    FrontFaceStencilPass = KEEP;
    FrontFaceStencilFunc = EQUAL;
    BackFaceStencilFail = KEEP;
    BackFaceStencilPass = KEEP;
    BackFaceStencilFunc = EQUAL;
};
```

We make sure it never writes a value to the stencil buffer either. Only if it equals the value in the stencil buffer, we allow it to have the pixel shader to draw to that section of the screen. In this case, the reference value is 0, so we allow it to only draw to where the 'occluding triangle' was *not* drawn.

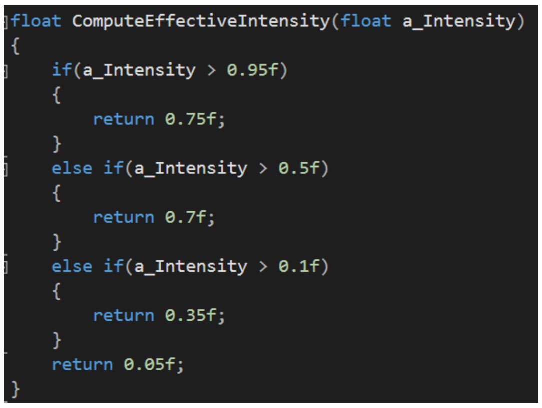
The following is another effect that can easily be achieved by using a second pass. One pass checks for the 0's in the stencil buffer, the other for the 1's, causing it to be partially rendered opaque and partially transparent (using alpha blending).



Lighting

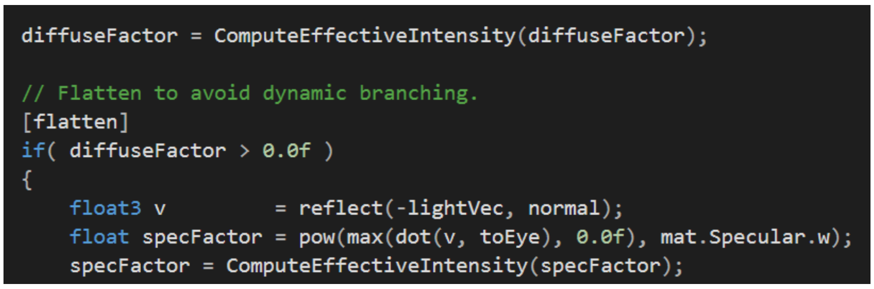
For lighting, I would score a 10 as well. A toon shader has been implemented. There are actually two variants to this. The first is as done regularly with 4 steps, while the other allows you to 'specify' the amount of steps, in other words the amount of banding.

Regular toon shading



```
float ComputeEffectiveIntensity(float a_Intensity)
{
    if(a_Intensity > 0.95f)
    {
        return 0.75f;
    }
    else if(a_Intensity > 0.5f)
    {
        return 0.7f;
    }
    else if(a_Intensity > 0.1f)
    {
        return 0.35f;
    }
    return 0.05f;
}
```

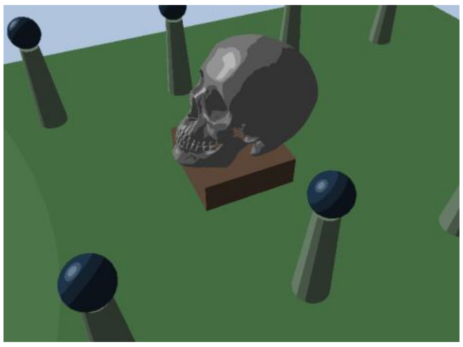
The function above does all the work for regulating the toon shader, or more precisely, the intensity of the lights. The levels are chosen somewhat at random and it is mostly just testing whatever gives decent/good results. This function is called from all the functions in LightHelper.fx as follows:



```
diffuseFactor = ComputeEffectiveIntensity(diffuseFactor);

// Flatten to avoid dynamic branching.
[flatten]
if( diffuseFactor > 0.0f )
{
    float3 v          = reflect(-lightVec, normal);
    float specFactor = pow(max(dot(v, toEye), 0.0f), mat.Specular.w);
    specFactor = ComputeEffectiveIntensity(specFactor);
```

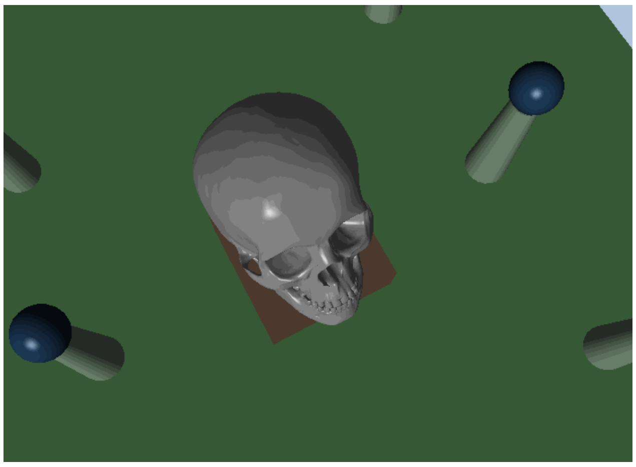
First, the diffuse factor is fed into it. We do this before the diffuse factor test, because otherwise specular might not be calculated when it should be! It actually makes the if statement somewhat useless, since the minimum returned by the 'ComputeEffectiveIntensity' is 0.05.

After that, the same is done once the final specular factor has been calculated. In other words, we are simply 'clamping' the values to appropriate ranges, to limit the amount of steps. This gives the following result:

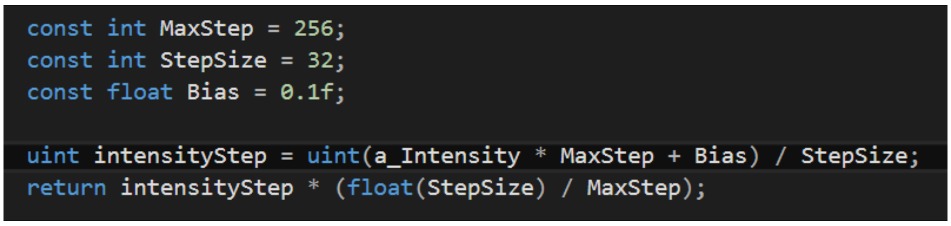


Variable banding

The following image shows the effect when using the variation with adjustable levels of banding.



The result is best seen on the skull itself. This result is achieved by basically creating a step-based formula using integer division.

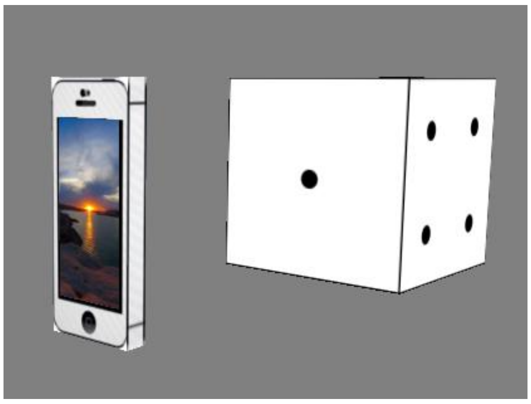


```
const int MaxStep = 256;
const int StepSize = 32;
const float Bias = 0.1f;

uint intensityStep = uint(a_Intensity * MaxStep + Bias) / StepSize;
return intensityStep * (float(StepSize) / MaxStep);
```

We first map the given intensity to the step where we are at (what banding level, like the if comparisons in the regular toon shading algorithm). We add a small bias, because otherwise we would rarely reach the highest step (that would require a given intensity of at least 1.0). In this case, the max step is 256 while the step size, causing 256 / 32 = 8 levels of banding. We then calculate the intensity ‘per step’ and multiply the calculated level/step with the intensity per step.

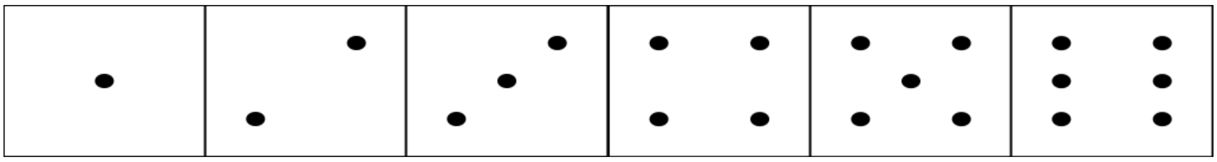
Texturing

For texturing, my score would be an 8, following the rubric. Both a mobile phone with a 'taken picture' can be seen, with a dice in the background using texturing.

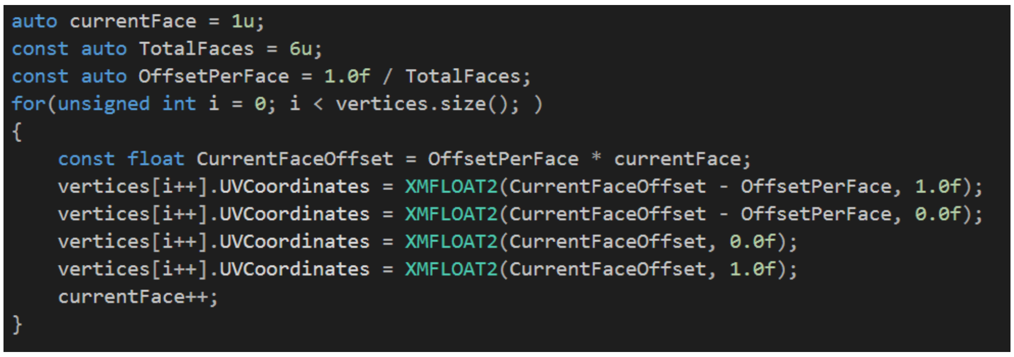


Texturing the dice

The dice texture is as follows:



The UV coordinates are calculated for each vertex based on the 'current face'. Normally, the UV coordinates are within the range of 0 and 1, but based on the current face, I instead map this to the range of where the face in the texture starts and where it ends. This is done as follows:

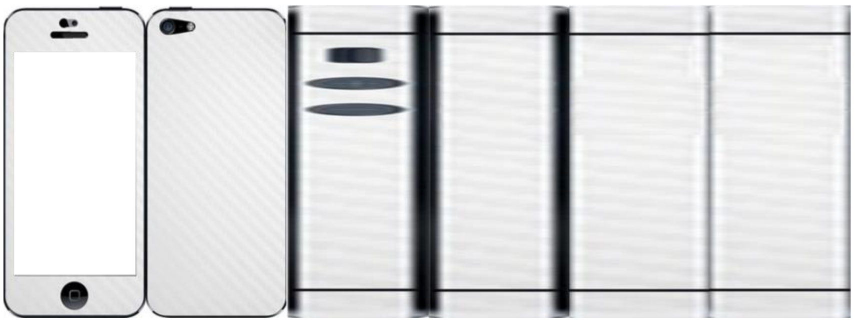


```
auto currentFace = 1u;
const auto TotalFaces = 6u;
const auto OffsetPerFace = 1.0f / TotalFaces;
for(unsigned int i = 0; i < vertices.size(); )
{
    const float CurrentFaceOffset = OffsetPerFace * currentFace;
    vertices[i++].UVCoordinates = XMFLOAT2(CurrentFaceOffset - OffsetPerFace, 1.0f);
    vertices[i++].UVCoordinates = XMFLOAT2(CurrentFaceOffset - OffsetPerFace, 0.0f);
    vertices[i++].UVCoordinates = XMFLOAT2(CurrentFaceOffset, 0.0f);
    vertices[i++].UVCoordinates = XMFLOAT2(CurrentFaceOffset, 1.0f);
    currentFace++;
}
```

Regularly, CurrentFaceOffset would equal the 1.0 boundary of the texture and when OffsetPerFace is subtracted, it would be the same as 0.0 for a regular texture.

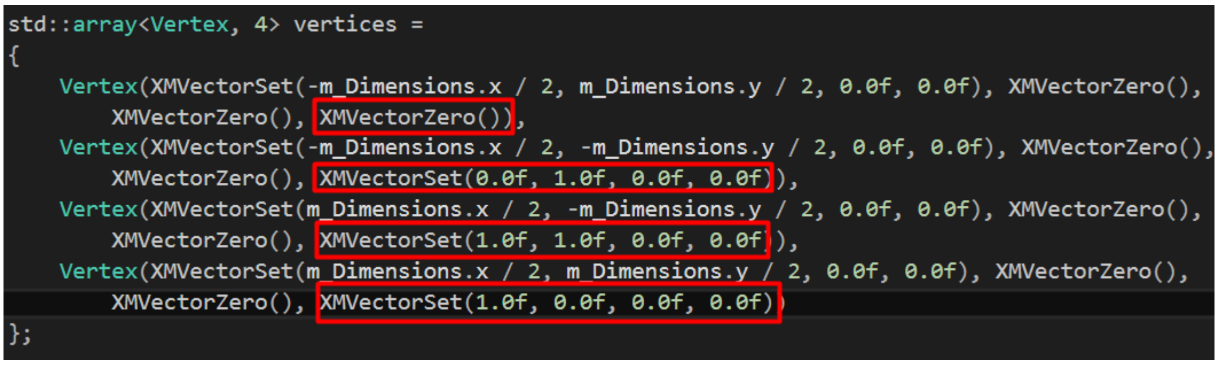
Rendering the phone

The phone itself is rendered in the exact same way as the dice. It is merely a box at different dimensions. The following texture was used:



The reason the sides are stretched out, is because it was easier to assume the general case of each texture being of the same size (width, to be precise).

The display is rendered separately as a quad on top of the phone. The highlighted pieces of code are the UV coordinates used for that vertex. The mapping is fairly simple: it starts bottom left, then top left, top right and finally bottom right.



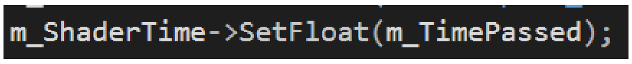
```
std::array<Vertex, 4> vertices =
{
    Vertex(XMVectorSet(-m_Dimensions.x / 2, m_Dimensions.y / 2, 0.0f, 0.0f), XMVectorZero(),
        XMVectorZero(), XMVectorZero()),
    Vertex(XMVectorSet(-m_Dimensions.x / 2, -m_Dimensions.y / 2, 0.0f, 0.0f), XMVectorZero(),
        XMVectorZero(), XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f)),
    Vertex(XMVectorSet(m_Dimensions.x / 2, -m_Dimensions.y / 2, 0.0f, 0.0f), XMVectorZero(),
        XMVectorZero(), XMVectorSet(1.0f, 1.0f, 0.0f, 0.0f)),
    Vertex(XMVectorSet(m_Dimensions.x / 2, m_Dimensions.y / 2, 0.0f, 0.0f), XMVectorZero(),
        XMVectorZero(), XMVectorSet(1.0f, 0.0f, 0.0f, 0.0f))
};
```

Shading

For shading, I would score a 10 as well following the rubric. A shader of moderate complexity (horrible trick), intermediate complexity (flame) and high complexity have been ported to the effects framework. The last shader has approximately 120 lines of code, not counting comments and white lines.

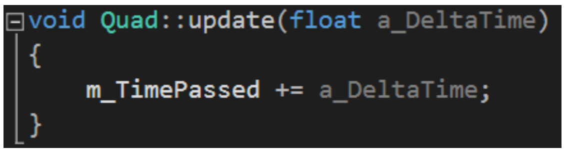
Constant buffer(s)

The shaders from ShaderToy presented here require both the screen or quad resolution to be passed along with the current game time. The time is passed as constant buffer as follows:



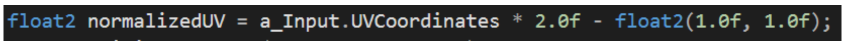
```
m_ShaderTime->SetFloat(m_TimePassed);
```

With the time being calculated as follows:



```
void Quad::update(float a_DeltaTime)
{
    m_TimePassed += a_DeltaTime;
}
```

Resolution is done differently. Since my quad does not match the dimensions of the screen, I modified the code to make it use the interpolated UV coordinates and map that to either a -1, 1 range where necessary (which is for the raymarch based shaders).



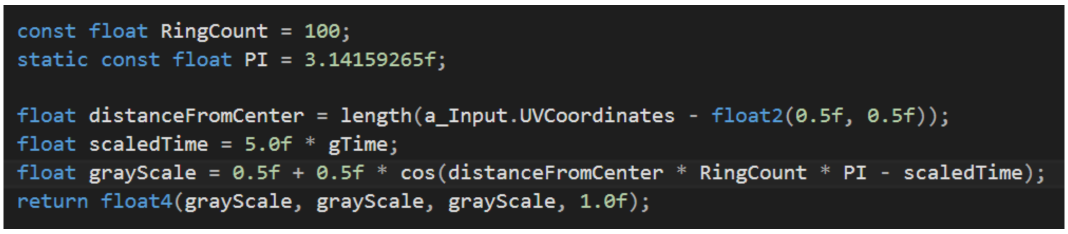
```
float2 normalizedUV = a_Input.UVCoordinates * 2.0f - float2(1.0f, 1.0f);
```

Which is very similar to the process of retrieving normals from a normal map.

Horrible trick

Picture omitted because it is very distracting when reading

The porting was merely a matter of replacing the vector instances with floats as well as passing the game time as constant buffer to the shader. I've also introduced several new variables to making the adjustment of the trick easier.

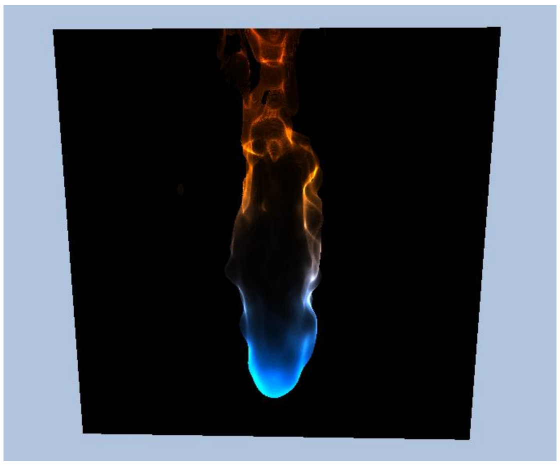


```
const float RingCount = 100;
static const float PI = 3.14159265f;

float distanceFromCenter = length(a_Input.UVCoordinates - float2(0.5f, 0.5f));
float scaledTime = 5.0f * gTime;
float grayScale = 0.5f + 0.5f * cos(distanceFromCenter * RingCount * PI - scaledTime);
return float4(grayScale, grayScale, grayScale, 1.0f);
```

First, it simply calculates the distance to the center and using the cos intrinsic function, maps this to a range of -1, 1. This is based on the distance to the center point, so as we move further, we get close to a boundary and thus closer to the value of white. Since the cos function limits this, we transition back to black and so forth. The ringcount and PI multiplication are a trick based on the properties of the cos graph.

Scaled time is subtracted to make the rings seemingly 'move' to the outer edges. The time was scaled to speed up the effect.

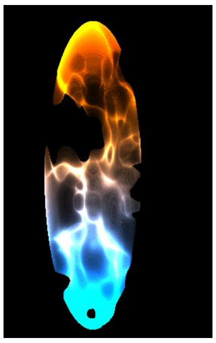
Flame



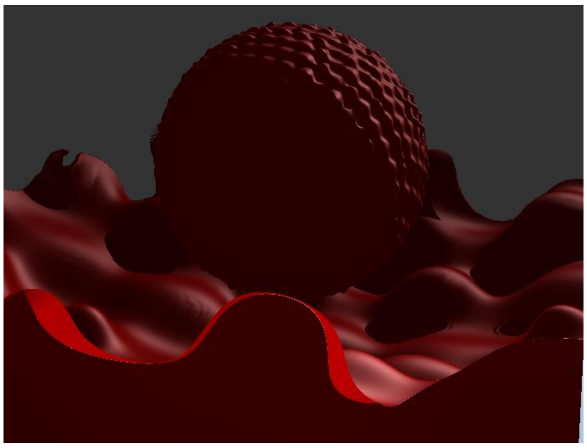
Flame, like horrible trick, mostly consisted of changing the vec*n* variables to float*n*. Additionally, instances of the 'mix' intrinsic function had to be replaced by their HLSL variant, which is 'lerp'. Both perform a linear interpolation between given values.

As already shown, I also had to modify the way it calculates the ray direction for the ray marching algorithm (i.e. calculating the UV coordinates). I won't much get into the details of the algorithm, especially since there are too many 'magic numbers' in the code to be able to elaborate.

The principle of the algorithm is to march into the scene and map a function to a spherical like object (this can also be seen from the bottom of the flame, which looks like a bit like a hemi-sphere. By modifying the way the algorithm performs the marching for the flame slightly, we can see this more accurately.

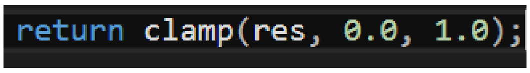


ShaderToy shader



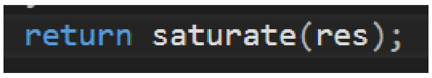
The above shader is the result of the ‘SphereBowl’ shader, which shows an oddly animated sphere (using sine functions to distort its shape) while a fluid-shaped object is moving underneath. Meanwhile, a light travels around the scenery, casting the sphere’s (soft) shadows onto the fluid underneath.

For this shader, most of the porting consisted of mapping the UV coordinates correctly and changing the vec to float types. One additional change was for the following line:



```
return clamp(res, 0.0, 1.0);
```

This operation is available in HLSL too, but can be shortened to:



```
return saturate(res);
```