Tessellation Shader

Yentoff Cuypers

2016-2017 DAE

Graphics Programming 2

INTRO

The purpose of this paper is to learn the basics of an HLSL Tessellation Shader, explain some issues one might encounter and most of all serve an educational purpose.

The shader explained in this paper will allow for tessellating a mesh and applying diffuse, normal and height mapping.

SHADER STAGES

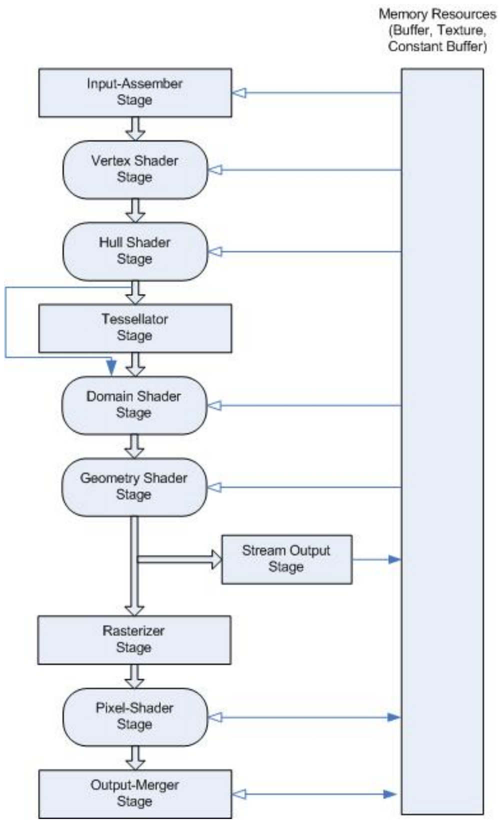


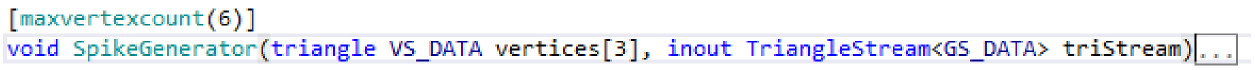
Figure 1. DirectX Shader Pipeline Stages

The above Image shows the different DirectX shader pipeline stages.
The stages we're interested in for this paper are the Hull and Domain Shader. But first I will briefly touch upon the Geometry Shader stage.

Geometry Shader Stage

The main purpose of the geometry shader stage is to change the geometry of the mesh that is passed through. This can be anything, from a single vertex to a replica of the Titanic.

In the geometry shader you can specify what you expect as input and the amount of vertices you will return.



```
[maxvertexcount(6)]
void SpikeGenerator(triangle VS_DATA vertices[3], inout TriangleStream<GS_DATA> triStream)...
```

Figure 2. Geometry Shader parameter list

With this data you can create new vertices, change vertices or even delete vertices and create an entirely new mesh.

A common usage is for grass or hair, which can be created from a single vertex or on an existing mesh.

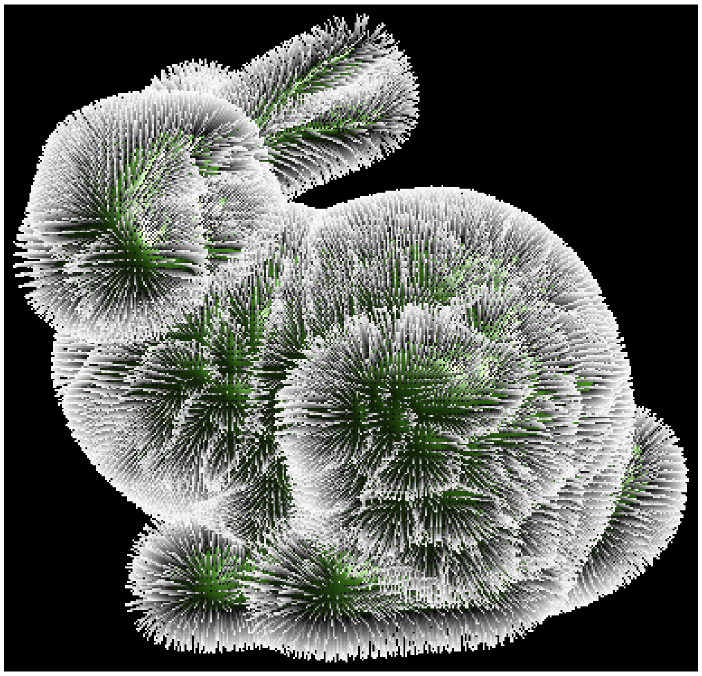


Figure 3. Geometry Shader Example

TESSELLATION STAGES

Unlike all the other shader stages, the tessellation stage consists of three separate stages, of which two are to be implemented by the programmer, namely the Hull and Domain stages.

Hull Shader Stage

The purpose of the hull stage is to tell the actual Tessellator stage how to tessellate the incoming data. The hull stage runs once per 'control point' which is another name for the vertices and calls a 'Patch Constant Function' once per input primitive which in turn returns the tessellation factors.
This factor is what tells the Tessellator how much to subdivide the primitive.

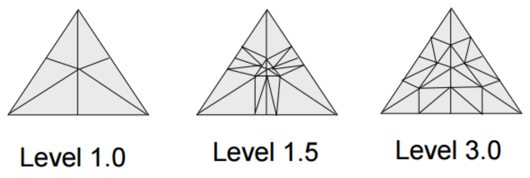


Figure 4. Tessellation Factors

Note, the input topology should be equivalent to the domain and output control points.

```
pDeviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST);

// Called once per control point
[domain("tri")] // indicates a triangle patch (3 verts)
[partitioning("fractional_odd")] // fractional avoids popping
// vertex ordering for the output triangles
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
// name of the patch constant hull shader
[patchconstantfunc("ConstantsHS")]
[maxtessfactor(50.0f)] //hint to the driver – the lower the better
// Pass in the input patch and an index for the control point
HullOut HS(InputPatch<VertexOut, 3> inputPatch, uint uCPID : SV_OutputControlPointID)...
```

Figure 5. Hull Shader Parameters

Domain Shader Stage

After the Tessellator stage has done its work, it passes the newly generated vertices to the Domain stage.

```
// Called once per tessellated vertex
[domain("tri")] // indicates that triangle patches were used
// The original patch is passed in, along with the vertex position in barycentric coordinates,
// and the patch constant phase hull shader output(tessellation factors)
DomainOut DS(ConstantHullPatchOut input, float3 BarycentricCoordinates : SV_DomainLocation,
               const OutputPatch<HullOut, 3> TrianglePatch)
```

Figure 6. Domain Shader Parameters

During the domain stage we can adjust the position of the vertices and calculate final data we'll need in the Pixel Shader or optional Geometry Shader.

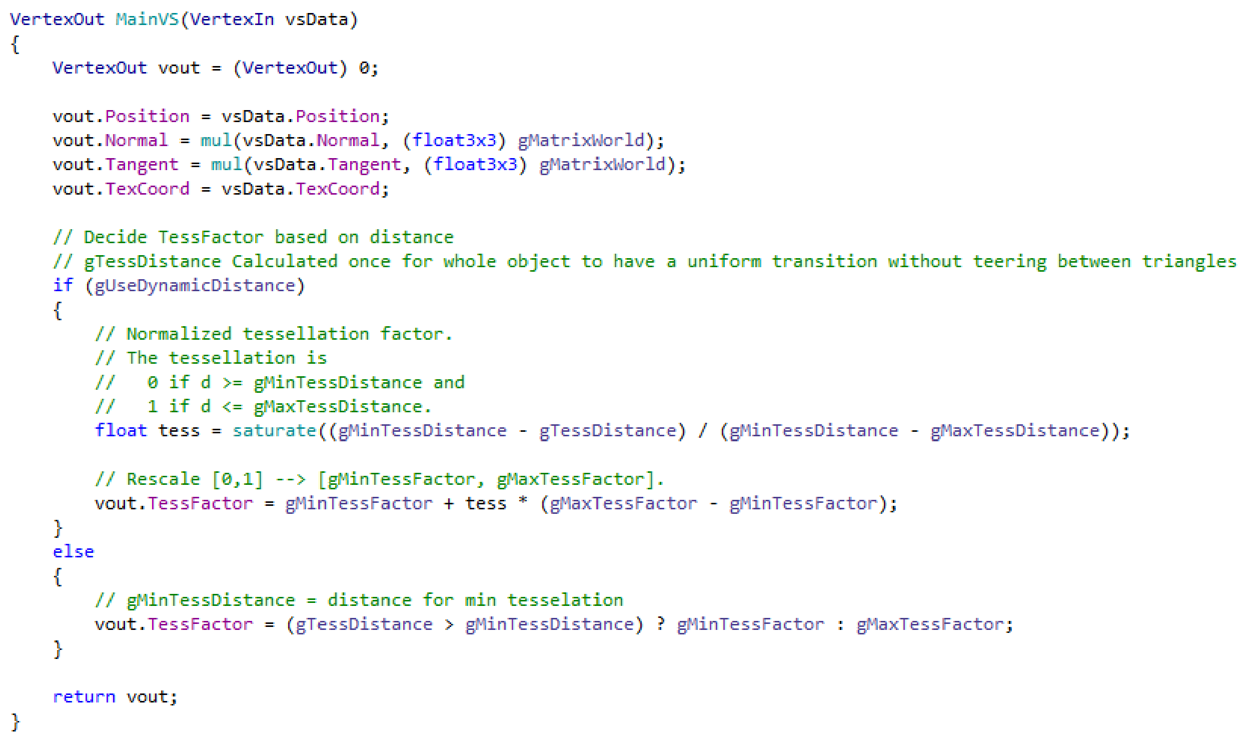
THE CODING

Let’s start with the actual coding and explanations alongside of it.

VERTEX SHADER (VS)

First of we start with a Vertex Shader with following goals:

- Pass through vertex data to HullShader
- Decide a Tessellation Factor based on distance.



```
VertexOut MainVS(VertexIn vsData)
{
    VertexOut vout = (VertexOut) 0;

    vout.Position = vsData.Position;
    vout.Normal = mul(vsData.Normal, (float3x3) gMatrixWorld);
    vout.Tangent = mul(vsData.Tangent, (float3x3) gMatrixWorld);
    vout.TexCoord = vsData.TexCoord;

    // Decide TessFactor based on distance
    // gTessDistance Calculated once for whole object to have a uniform transition without teering between triangles
    if (gUseDynamicDistance)
    {
        // Normalized tessellation factor.
        // The tessellation is
        //   0 if d >= gMinTessDistance and
        //   1 if d <= gMaxTessDistance.
        float tess = saturate((gMinTessDistance - gTessDistance) / (gMinTessDistance - gMaxTessDistance));

        // Rescale [0,1] --> [gMinTessFactor, gMaxTessFactor].
        vout.TessFactor = gMinTessFactor + tess * (gMaxTessFactor - gMinTessFactor);
    }
    else
    {
        // gMinTessDistance = distance for min tesselation
        vout.TessFactor = (gTessDistance > gMinTessDistance) ? gMinTessFactor : gMaxTessFactor;
    }

    return vout;
}
```

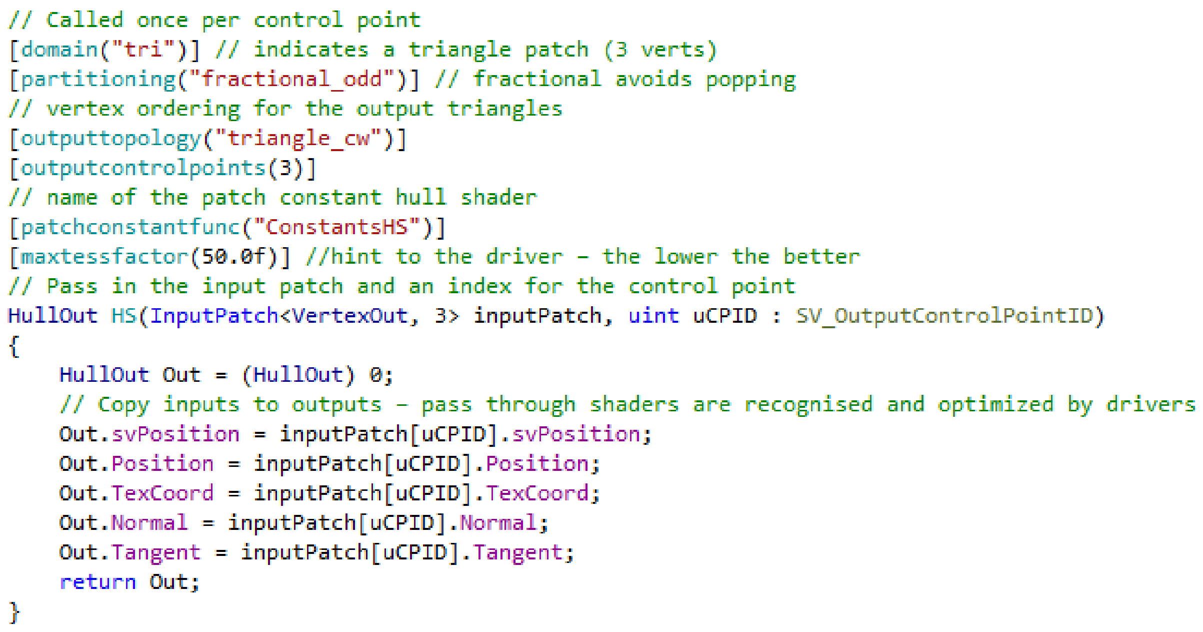
Figure 7. Vertex Shader Code

Unlike usual we simply pass through the Position as we’ll convert it to clipping space later on in the domain shader right before passing it to the pixel shader.

Here we also calculate a Tessellation factor which will determine how much the Tessellator will subdivide the patches.

Hull Shader (HS)

All we need to do here is setup same variables for the tessellator stage and define the patch constant function we want to use. Besides that we simply pass through all the other data for each Control Point ID, which the drivers will recognize and optimize for us.



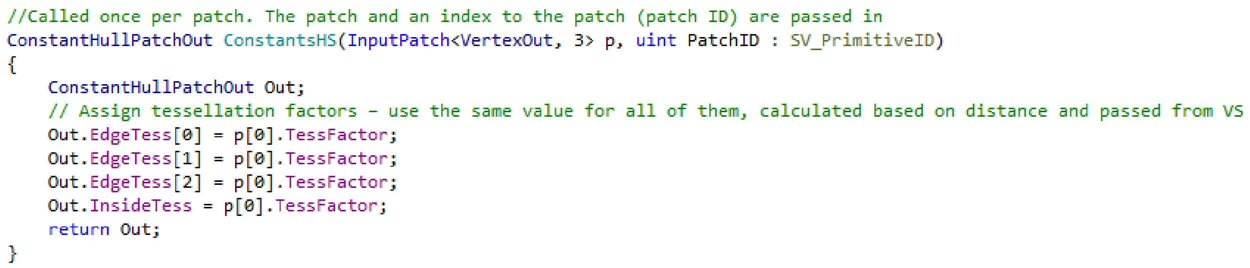
```
// Called once per control point
[domain("tri")] // indicates a triangle patch (3 verts)
[partitioning("fractional_odd")] // fractional avoids popping
// vertex ordering for the output triangles
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
// name of the patch constant hull shader
[patchconstantfunc("ConstantsHS")]
[maxtessfactor(50.0f)] //hint to the driver - the lower the better
// Pass in the input patch and an index for the control point
HullOut HS(InputPatch<VertexOut, 3> inputPatch, uint uCPID : SV_OutputControlPointID)
{
    HullOut Out = (HullOut) 0;
    // Copy inputs to outputs - pass through shaders are recognised and optimized by drivers
    Out.svPosition = inputPatch[uCPID].svPosition;
    Out.Position = inputPatch[uCPID].Position;
    Out.TexCoord = inputPatch[uCPID].TexCoord;
    Out.Normal = inputPatch[uCPID].Normal;
    Out.Tangent = inputPatch[uCPID].Tangent;
    return Out;
}
```

Figure 8. Pass through Hull Shader Code

Patch Constant Function (PCF)

As part of the HS the PCF is called to really determine which tessellation factors you want to use for the inside edge and the outside edges.

We'll keep it simple and give them all the same Tessellation Factor for an even distribution.



```
//Called once per patch. The patch and an index to the patch (patch ID) are passed in
ConstantHullPatchOut ConstantsHS(InputPatch<VertexOut, 3> p, uint PatchID : SV_PrimitiveID)
{
    ConstantHullPatchOut Out;
    // Assign tessellation factors - use the same value for all of them, calculated based on distance and passed from VS
    Out.EdgeTess[0] = p[0].TessFactor;
    Out.EdgeTess[1] = p[0].TessFactor;
    Out.EdgeTess[2] = p[0].TessFactor;
    Out.InsideTess = p[0].TessFactor;
    return Out;
}
```

Figure 9. Patch Constant Function Code

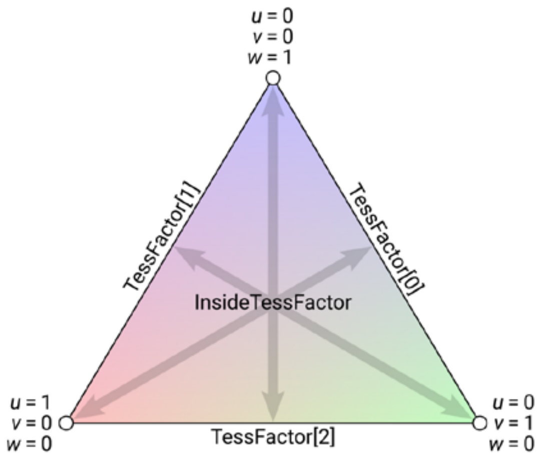


Figure 10. Tri Patch, inside and outside edges

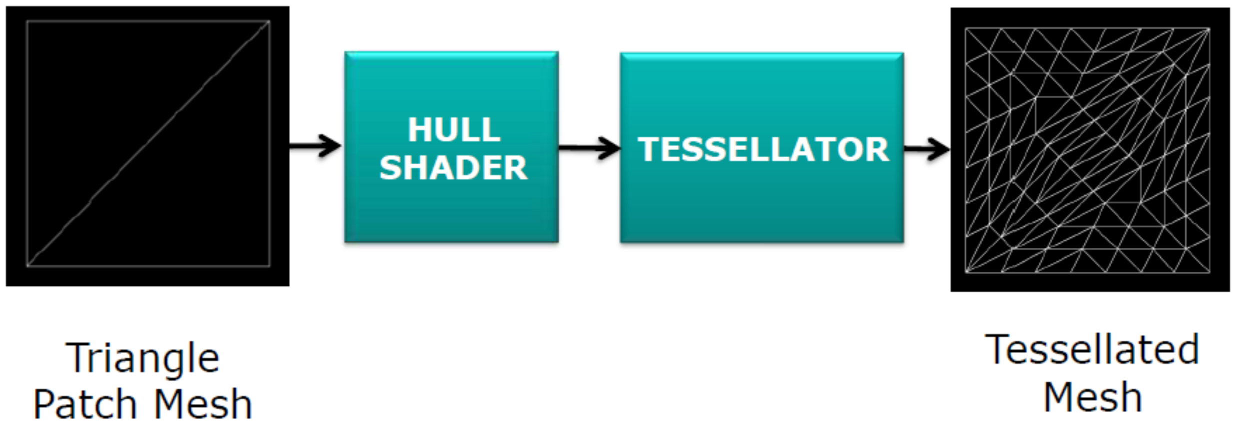
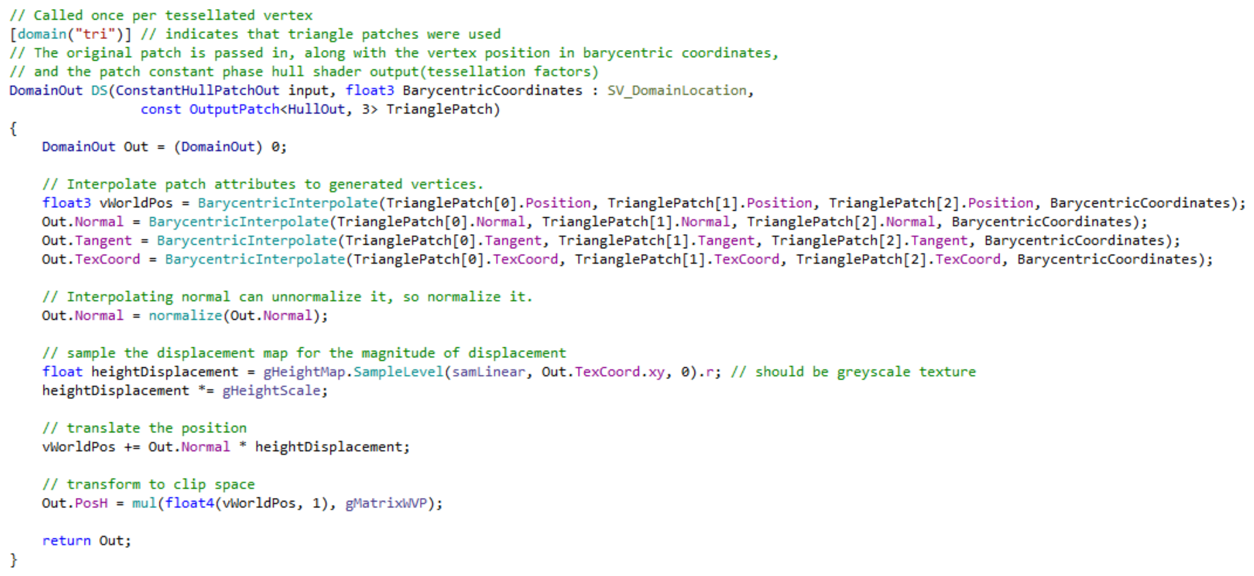


Figure 11. Triangle Patch Mesh to Tessellated Mesh

Domain Shader

Now we can work with our tessellated patch data.



```
// Called once per tessellated vertex
[domain("tri")] // indicates that triangle patches were used
// The original patch is passed in, along with the vertex position in barycentric coordinates,
// and the patch constant phase hull shader output(tessellation factors)
DomainOut DS(ConstantHullPatchOut input, float3 BarycentricCoordinates : SV_DomainLocation,
                const OutputPatch<HullOut, 3> TrianglePatch)
{
    DomainOut Out = (DomainOut) 0;

    // Interpolate patch attributes to generated vertices.
    float3 vWorldPos = BarycentricInterpolate(TrianglePatch[0].Position, TrianglePatch[1].Position, TrianglePatch[2].Position, BarycentricCoordinates);
    Out.Normal = BarycentricInterpolate(TrianglePatch[0].Normal, TrianglePatch[1].Normal, TrianglePatch[2].Normal, BarycentricCoordinates);
    Out.Tangent = BarycentricInterpolate(TrianglePatch[0].Tangent, TrianglePatch[1].Tangent, TrianglePatch[2].Tangent, BarycentricCoordinates);
    Out.TexCoord = BarycentricInterpolate(TrianglePatch[0].TexCoord, TrianglePatch[1].TexCoord, TrianglePatch[2].TexCoord, BarycentricCoordinates);

    // Interpolating normal can unnormalize it, so normalize it.
    Out.Normal = normalize(Out.Normal);

    // sample the displacement map for the magnitude of displacement
    float heightDisplacement = gHeightMap.SampleLevel(samLinear, Out.TexCoord.xy, 0).r; // should be greyscale texture
    heightDisplacement *= gHeightScale;

    // translate the position
    vWorldPos += Out.Normal * heightDisplacement;

    // transform to clip space
    Out.PosH = mul(float4(vWorldPos, 1), gMatrixWVP);

    return Out;
}
```

Figure 12. Domain Shader Code

First we interpolate the 'patch attributes' or the vertex data for the tessellated patch.
Since we're using triangles as data, we need to interpolate the data using barycentric coordinates.

And now is the time for height mapping which effects the geometry. As we have access to our newly generated vertices, we can also manipulate them.

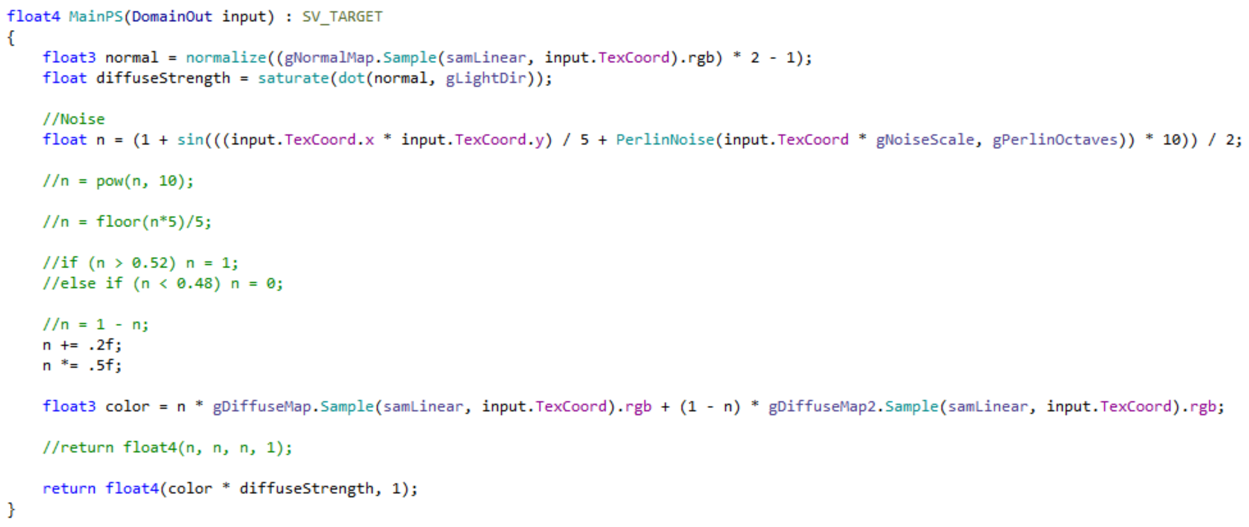
We start by sampling the height map. Since we assume it is a greyscale texture, we can use any channel. As we want to have control over the actual height at runtime, we can use a global height scale value to multiply with our sampled value.

To displace our vertex, we simply add this calculated displacement value multiplied by the normal direction, to our position.

Finally we have to transform the position to the clip space by multiplying it with the WorldViewProjection matrix.

PIXEL SHADER

Finally in the pixel shader we can play with the colors of our final result.



```
float4 MainPS(DomainOut input) : SV_TARGET
{
    float3 normal = normalize((gNormalMap.Sample(samLinear, input.TexCoord).rgb) * 2 - 1);
    float diffuseStrength = saturate(dot(normal, gLightDir));

    //Noise
    float n = (1 + sin(((input.TexCoord.x * input.TexCoord.y) / 5 + PerlinNoise(input.TexCoord * gNoiseScale, gPerlinOctaves)) * 10)) / 2;

    //n = pow(n, 10);

    //n = floor(n*5)/5;

    //if (n > 0.52) n = 1;
    //else if (n < 0.48) n = 0;

    //n = 1 - n;
    n += .2f;
    n *= .5f;

    float3 color = n * gDiffuseMap.Sample(samLinear, input.TexCoord).rgb + (1 - n) * gDiffuseMap2.Sample(samLinear, input.TexCoord).rgb;

    //return float4(n, n, n, 1);

    return float4(color * diffuseStrength, 1);
}
```

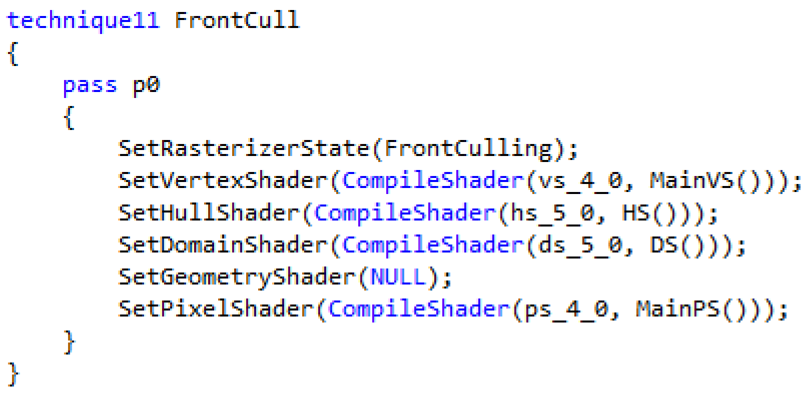
Figure 13. Pixel Shader Code

Considering we are using a normal map which is paired with the height map, we sample the normal from the normal map and use that to calculate the correct lighting info.

To add an extra feature to the shader we generate some Perlin Noise and use this to mask parts with one of two diffuse textures.

TECHNIQUE

After writing all the stages, we can't forget about the most important part of the shader, which is the technique which tells the GPU which stages, rasterizers, blend modes and more to use.



```
technique11 FrontCull
{
    pass p0
    {
        SetRasterizerState(FrontCulling);
        SetVertexShader(CompileShader(vs_4_0, MainVS()));
        SetHullShader(CompileShader(hs_5_0, HS()));
        SetDomainShader(CompileShader(ds_5_0, DS()));
        SetGeometryShader(NULL);
        SetPixelShader(CompileShader(ps_4_0, MainPS()));
    }
}
```

Figure 14. Technique Code

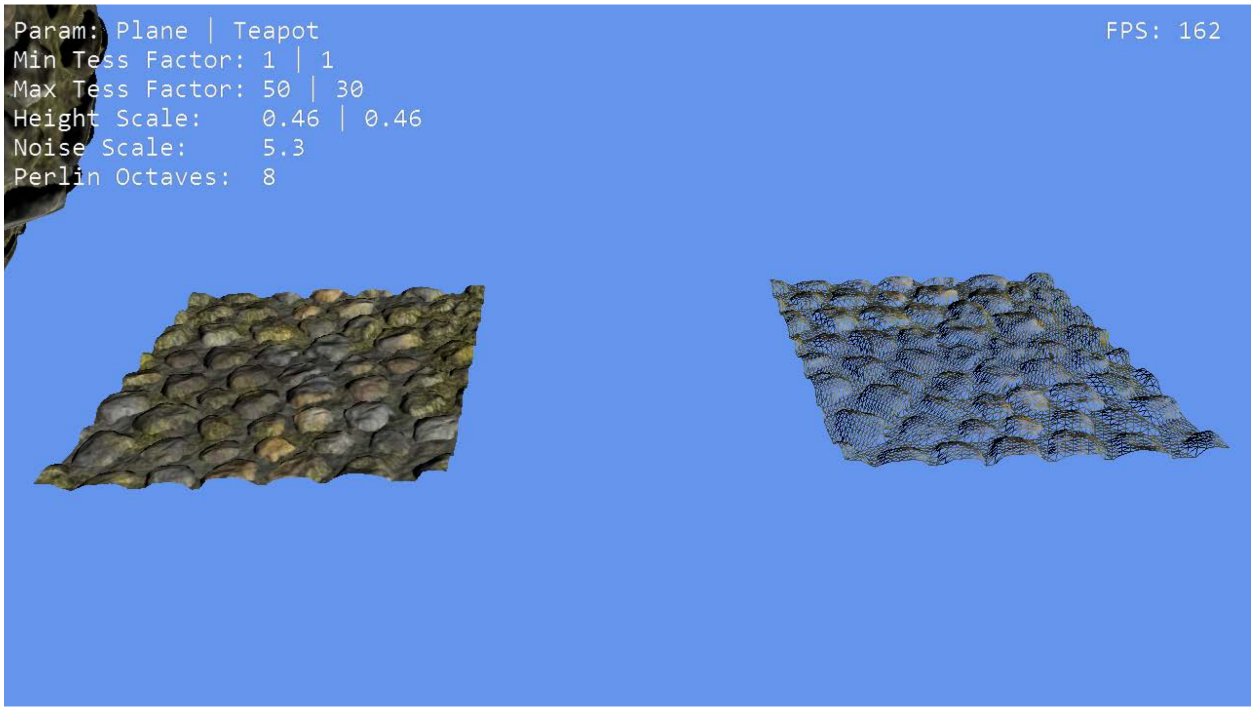
An important thing to keep an eye on when working with multiple shaders which are all using different stages and input topologies, is to set unused stages to NULL, and make sure you are using the correct topology in your code.

CONCLUSION

In making this tessellation shader, I had to research the different stages which at first was a pretty tough task as most sources don't go in depth on the actual code to use in for example the patch constant function.

By splitting up the different stages, testing different values and comparing different sources, I was able to get a decent understanding of the different tessellation stages, their purpose and come up with a tessellation shader with some procedural functionalities and real-time customizations such as dynamic tessellation based on current distance, or using fixed distances.

I also achieved my main goal which was to apply height mapping which changes the geometry and not just faking it with optical illusions. As a result it's also possible to dynamically realise much higher quality details then simple normal mapping or parallax tessellation can do.



```
Param: Plane | Teapot                                        FPS: 162
Min Tess Factor: 1 | 1
Max Tess Factor: 50 | 30
Height Scale:    0.46 | 0.46
Noise Scale:     5.3
Perlin Octaves:  8
```

Figure 15. Result in Overlord Engine

SOURCES

Bilodeau Bill, (2010, March 31), Direct3D 11 Tutorial: Tessellation, Retrieved from http://twvideo01.ubm-us.net/o1/vault/gdc10/slides/Bilodeau_Bill_Direct3D11TutorialTessellation.pdf

Nathan Reed, (2016, December 30), Tessellation Modes Quick Reference, Retrieved from http://reedbeta.com/blog/tess-quick-ref/

Microsoft, Graphics pipeline, Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx

Microsoft, Tessellation Stages, Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/ff476340(v=vs.85).aspx

Microsoft, How To: Design a Hull Shader, Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/ff476339(v=vs.85).aspx

Microsoft, How To: Design a Domain Shader, Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/ff476337(v=vs.85).aspx

Justin Stenning, (2014, September 03), Domain vs Geometry Shader, Retrieved from https://www.gamedev.net/topic/660533-domain-vs-geomtry-shader/?view=findpost&p=5177809