Voxel Shader

Joris Verhelst

2016-2017

Digital Arts & Entertainment

Graphics Programming 2

Intro

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

The shader in this paper will allow for voxelizing a mesh and applying a diffuse texture to it.

Shader Stages

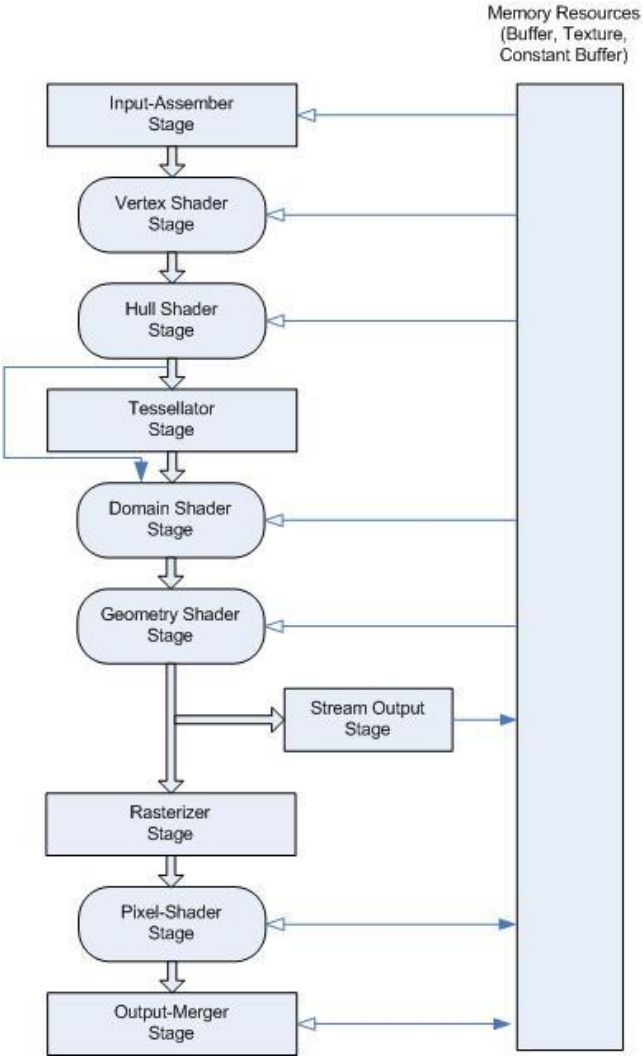

Figure 1. DirectX 11 Shader Pipeline Stages

The above image shows the different DirectX shader pipeline stages.
The stages we're interested in is primarily the Geometry Shader stage and later the Hull and Domain Shader in the Tessellation Stage.

Geometry Shader Stage

The main purpose of the geometry shader stage is to change the geometry of the mesh that is passed through it.

The geometry shader object uses the following syntax

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

Figure 2. Geometry Shader parameter

You have to declare the maximum number of vertices that can be created in the stage.

A primitive datatype to determine the order, with a number between brackets to specify the amount of vertices.
This can be:

- -point[1]
- -line[2]
- -triangle[3]
- -lineadj[4]
- -triangleadj[6]

Declaration of the stream-output object, this streams the data out of the geometry shader into the next stage. this can be a sequence of points, lines or triangles.

With all this data you can add, change or delete vertices and create entirely different geometry.

2 common usages of geometry shaders would be fur/grass and waves.

Figure 3. Geometry Shader Example

Tessellation Stages

The tessellation stage consists of three separate stages, namely the Hull Shader Stage, the Tessellator Stage and the Domain Shader Stage. The Hull and Domain stages are to be implemented by the programmer.

Tessellation

tessellation is a method of breaking down polygons into finer pieces. For example, if you take a square and cut it across its diagonal, you "tessellated" it into 2 triangles. This only impoves realism if the new triangles are used to depict new information.

The most popular way to use these new triangles is displacement mapping. A displacement map is a texture that stores height information, it allows for vertices to be moved up or down based on the height info.

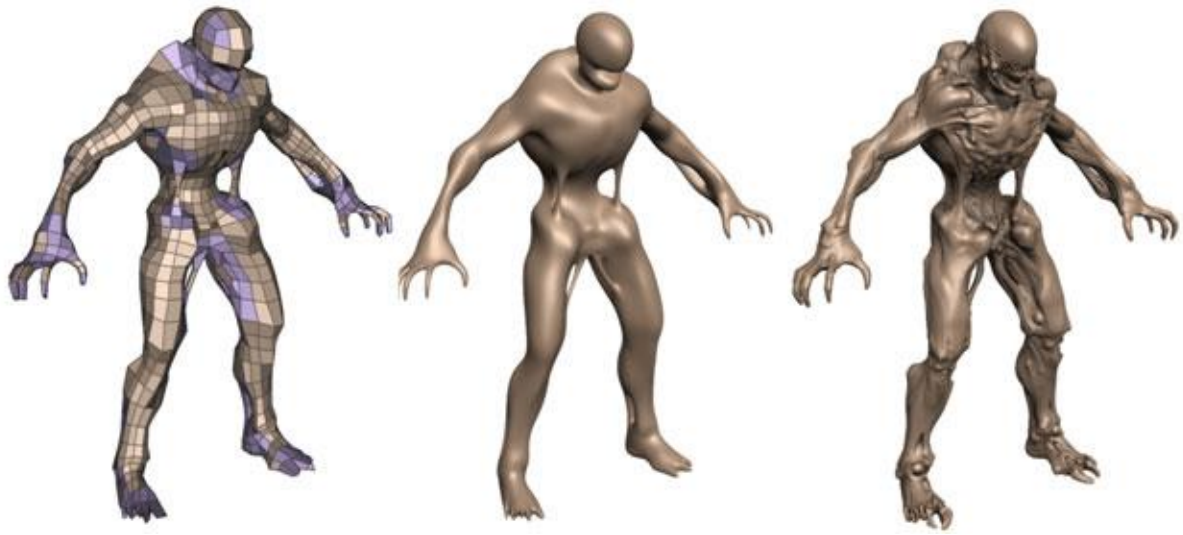

Figure 4. A coarse model(left) after tessellation(middle), the resulting smooth model is then displaced for a realistic looking result(right)

Hull Shader Stage

The purpose of the hull stage is to tell the tessellator stage how to tessellate the incoming data. The shader runs once per 'control point' for a geometry patch and calls a 'Patch Constant Function' once per input primitive. This, in turn, returns the tessellation factors, which is used to tell the Tessellator how much to subdivide the primitive.

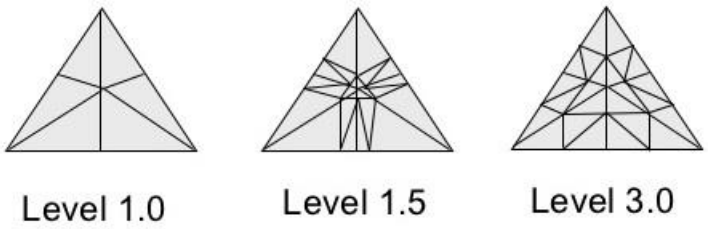

Figure 5. Tessellation Factors

```
[domain("tri")]                          //triangle patch
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]          //ordering output triangles
[outputcontrolpoints(3)]
[patchconstantfunc("ConstantsHS")]       //name constant hull shader
[maxtessfactor(10.0f)]                   //max amount of tessellation
HS_OUT HS(InputPatch<VS_OUT, 3> patch, uint pointId : SV_OutputControlPointID, uint patchId : SV_PrimitiveID)
```

Figure 6. Hull Shader Parameters

-The domain attribute defines the type of patch used: tri, quad or isoline
-The partitioning attribute defines the tessellation scheme to be used.
-Outputtopology: defines the output primitive type for the tessellator.
-Outputcontrolpoints: defines the number of output control points (per thread) that will be created in the hull shader.
-Name of your patch constant function

Tessellator Stage

this is a fixed function, which samples the input patch and generates vertices that divide the patch. This subdividing is according to the tessellation factors supplied by the hull shader and a partitioning scheme, which defines the algorithm used to subdivide the patch.

Domain Shader Stage

After the Tessellator stage has done its work, it sends the new vertices to the domain stage.

During this stage we map the new vertices into their final positions within the patch. Usually the domain shader will interpolate the final vertex value from the control points using barycentric coordinates output by the tessellator.

The output vertices will then be passed along to either the Geometry shader of the Pixel Shader.

```
// Called once per tessellated vertex
[domain("tri")] // indicates that triangle patches were used
PS_IN DS(ConstantHullPatchOut input, float3 BarycentricCoordinates : SV_DomainLocation,
        const OutputPatch<HS_OUT, 3> TrianglePatch)
{
```

Figure 7. Domain Shader Parameters

Coding

Let's start with the actual coding and explaining alongside of it.

Vertex Shader

The vertex shader is very simple and is just passing through data to the Hull shader.

```
//****************
// VERTEX SHADER *
//****************
VS_OUT MainVS(VS_IN vsData)
{

    VS_OUT temp = (VS_OUT)0;
    temp.Position = vsData.Position;
    temp.Normal = vsData.Normal;
    temp.TexCoord = vsData.TexCoord;
    temp.Tangent = vsData.Tangent;

    return temp;
}
```

Figure 8. Vertex Shader Code

Hull Shader

All that needs to happen here is setting up variables for the tessellator stage and defining the patch constant function we want to use. Other than that we can just pass through all the other data for each control point ID.

```
[domain("tri")]                         //triangle patch
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]         //ordering output triangles
[outputcontrolpoints(3)]
[patchconstantfunc("ConstantsHS")]      //name constant hull shader
[maxtessfactor(10.0f)]                  //max amount of tessellation
HS_OUT HS(InputPatch<VS_OUT, 3> patch, uint pointId : SV_OutputControlPointID, uint patchId : SV_PrimitiveID)
{
    HS_OUT output;


    output.Position = patch[pointId].Position;
    output.Normal = patch[pointId].Normal;
    output.Tangent = patch[pointId].Tangent;
    output.TexCoord = patch[pointId].TexCoord;

    return output;
}
```

Figure 9. Pass Through Hull Shader

Patch Constant Function

As part of the Hull Shader, PCF is called to determine which tessellation factors you want to use for the inside and outside edges. For now I will give it a value that can be easily modified.

The edge factor specifies in how many segments each edge gets subdivided into.

The inside factor specifies the number of mesh segments from each edge to the opposite vertex.

```
ConstantHullPatchOut ConstantsHS(InputPatch<VS_OUT, 3> inputPatch, uint patchId : SV_PrimitiveID)
{
    ConstantHullPatchOut output;

    // Set the tessellation factors for the three edges of the triangle.
    output.edges[0] = gEdgeTesselationAmount;
    output.edges[1] = gEdgeTesselationAmount;
    output.edges[2] = gEdgeTesselationAmount;

    output.inside = gInsideTesselationAmount;


    return output;
}
```

Figure 10. Patch Constant Function Code

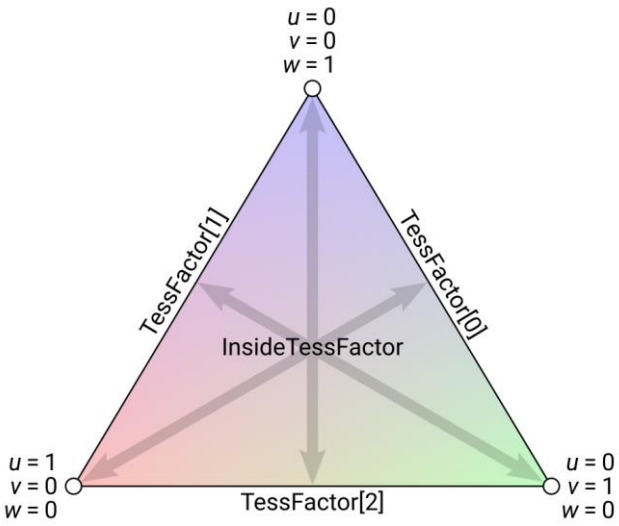

Figure 11. Tri patch, inside and outside tessellation

Domain Shader

Now we work with the patch data right after the Tessellator stage.

```
// Called once per tessellated vertex
[domain("tri")] //triangle patches
PS_IN DS(ConstantHullPatchOut input, float3 BarycentricCoordinates : SV_DomainLocation, const OutputPatch<HS_OUT, 3> TrianglePatch)
{
    PS_IN Out = (PS_IN)0;

    // Interpolate patch attributes to generated vertices.
    float3 WorldPosition = 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);


    // transform to clip space
    Out.Position = mul(float4(WorldPosition, 1), m_MatrixWorld);

    return Out;
}
```

Figure 12. Domain Shader Code

First we interpolate the vertex data or ‘patch attributes’ for the tessellated patch.
This is done using barycentric coordinates because we are using triangles. We then have to transform this position to the clip space my multiplying it with world matrix.

```
float2 BarycentricInterpolate(float2 v0, float2 v1, float2 v2, float3 baryCoord)
{
    return baryCoord.z * v0 + baryCoord.x * v1 + baryCoord.y * v2;
}

float3 BarycentricInterpolate(float3 v0, float3 v1, float3 v2, float3 baryCoord)
{
    return baryCoord.z * v0 + baryCoord.x * v1 + baryCoord.y * v2;
}

float4 BarycentricInterpolate(float4 v0, float4 v1, float4 v2, float3 baryCoord)
{
    return baryCoord.z * v0 + baryCoord.x * v1 + baryCoord.y * v2;
}
```

Figure 13. Calculating Barycentric Coordinates

Geometry Shader (GS)

Now we can finally Voxelize our mesh, First we need to be able to create new vertices, this is done using the following method.
Every vertex has a position, normal and UV coordinate that defines them.

```
void CreateVertex(inout TriangleStream<PS_IN> triStream, float3 pos, float3 normal, float2 texCoord)
{
    //Step 1. Create a GS_DATA object
    PS_IN vertex = (PS_IN)0;
    //Step 2. Transform the position using the WVP Matrix and assign it to (GS_DATA object).Position (Keep in mind: float3 -> float4)
    vertex.Position = mul(float4(pos, 1), m_MatrixWorldViewProj);
    //Step 3. Transform the normal using the World Matrix and assign it to (GS_DATA object).Normal (Only Rotation, No translation!)
    vertex.Normal = mul(normal, (float3x3)m_MatrixWorld);
    //Step 4. Assign texCoord to (GS_DATA object).TexCoord
    vertex.TexCoord = texCoord;
    //Step 5. Append (GS_DATA object) to the TriangleStream parameter (TriangleStream::Append(...))
    triStream.Append(vertex);
}
```

Figure 14. Creating new vertices

```
void CreateCube(float3 pos, float3 normal, float2 texCoord, inout TriangleStream<PS_IN> triStream,float blockWidth){
    float3 origin = pos;
    float3 up = float3(0,0,1);
    float3 pre = float3(0, 1, 0);
    float3 right = float3(1,0,0);
    float3 front = normalize(cross(up, right));

    float2 texHeight = texCoord;


    float3 bottomFrontRight, bottomFrontLeft, bottomBackRight, bottomBackLeft;
    bottomFrontLeft= float3(origin.x - blockWidth /2,origin.y- blockWidth /2,origin.z- blockWidth /2);
    bottomFrontRight = float3(origin.x + blockWidth /2,origin.y- blockWidth /2,origin.z- blockWidth /2);
    bottomBackLeft = float3(origin.x - blockWidth /2,origin.y- blockWidth /2,origin.z+ blockWidth /2);
    bottomBackRight = float3(origin.x + blockWidth /2,origin.y- blockWidth /2,origin.z+ blockWidth /2);

    float3 topFrontRight, topFrontLeft, topBackRight, topBackLeft;
    topFrontLeft = float3(bottomFrontLeft.x,bottomFrontLeft.y + blockWidth,bottomFrontLeft.z);
    topFrontRight = float3(bottomFrontRight.x,bottomFrontRight.y + blockWidth,bottomFrontRight.z);
    topBackLeft = float3(bottomBackLeft.x,bottomBackLeft.y + blockWidth,bottomBackLeft.z);
    topBackRight = float3(bottomBackRight.x,bottomBackRight.y + blockWidth,bottomBackRight.z);

    //Back
    triStream.RestartStrip();
    CreateVertex(triStream, bottomFrontLeft, -up, texCoord);
    CreateVertex(triStream, bottomFrontRight, -up, texCoord);
    CreateVertex(triStream, bottomBackLeft, -up, texCoord);
    CreateVertex(triStream, bottomBackRight, -up, texCoord);
```

Figure 15. Creating vertexes and planes

Creating a cube is essentially just drawing 6 planes around a given vertex, using a blockWidth variable to determine how big the cube is.

In a separate function we first determine where our cube is located on a grid before we call the CreateCube(), in which we first calculate the positions of 4 vertices that define the bottom plane and then the top plane by just adding the block width to the y coordinates. We can then define planes by the 4 vertices that make them up.

The normal of a plane is very simple for this example seeing as the cubes are all aligned the same way. Every vertex of the cube will have the same UV texture coordinate, resulting in a cube with a uniform colour when you apply a diffuse texture to the mesh.

Pixel Shader (PS)

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

```
float4 MainPS(PS_IN input) : SV_TARGET
{
    input.Normal = -normalize(input.Normal);

    //alpha
    float alpha = gTextureDiffuse.Sample(samLinear,input.TexCoord).a;

    //shading
    float diffuseStrength = dot(input.Normal,m_LightDir);
    diffuseStrength = (diffuseStrength+1)/2;
    diffuseStrength=saturate(diffuseStrength);
    diffuseStrength = pow(diffuseStrength,1);

    float3 color = gTextureDiffuse.Sample(samLinear,input.TexCoord).rgb;

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

Figure 16. Pixel Shader Code

This part of the shader is pretty basic seeing as we only use a diffuse texture and basic shading.

Technique

After writing all the stages, we can't forget about actually telling the GPU which stages and rasterizers to use. Note that this is a 'technique11' and not 'technique10' because Hull & Domain shaders are only possible since DirectX 11. Also make sure that you are using the correct topology in your code for your model, in this case :

```
gameContext.pDeviceContext>IASetPrimitiveTopology(D3D10_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

```
technique11 DefaultTechnique
{
    pass Pass1
    {
        SetRasterizerState(FrontCulling);
        SetVertexShader(CompileShader(vs_4_0, MainVS()));
        SetHullShader(CompileShader(hs_5_0, HS()));
        SetDomainShader(CompileShader(ds_5_0,DS()));
        SetGeometryShader(CompileShader(gs_4_0, GeoShader()));
        SetPixelShader(CompileShader(ps_4_0, MainPS()));
    }
}
```

Figure 17. Technique code

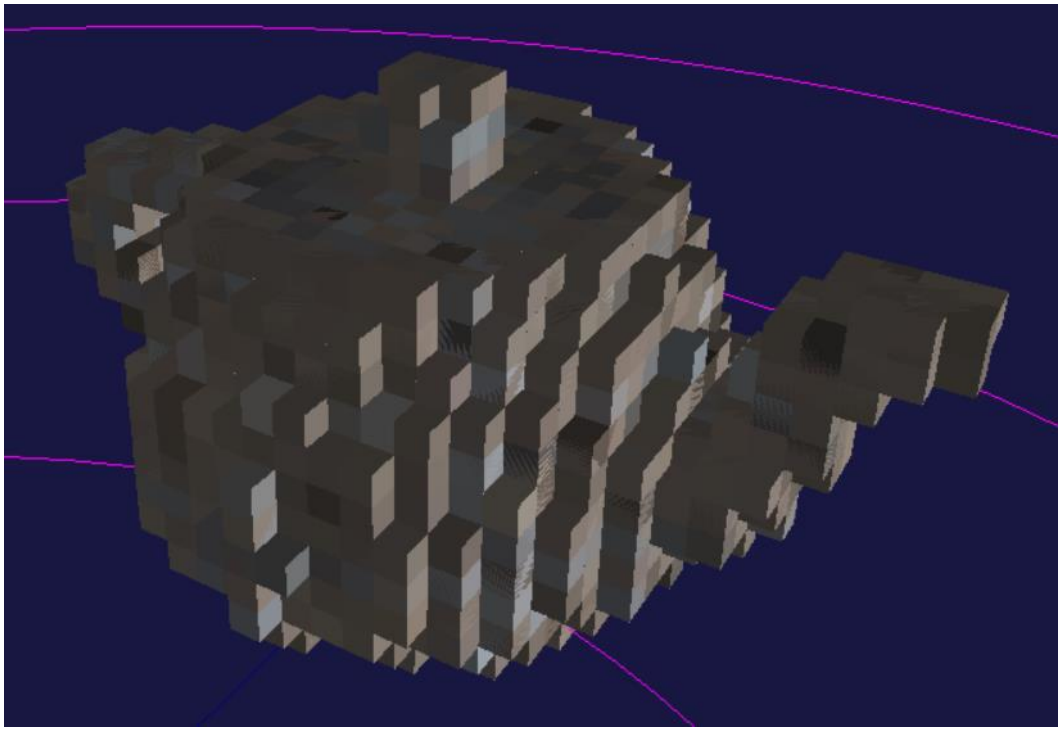

Figure 18. Result in Overlord Engine

Conclusion

In making this shader, I had to research the different stages pertaining the tessellation stages of the DirectX 11 pipeline, at first this was a pretty tough task as most resources don't go very in depth with actual code.

By working on both a geometry and tessellation parts of the shader pipeline, I was able to get a decent understanding of shaders and their different stages and purposes.

Sources

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

Eric Richards, (2013, September 16), Diving into the Tessellation Stages of Direct3D 11 , from http://richardssoftware.net/Home/Post/28

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

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

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

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

Masaya Takeshige, The Basics of GPU Voxelization, from https://developer.nvidia.com/content/basics-gpu-voxelization