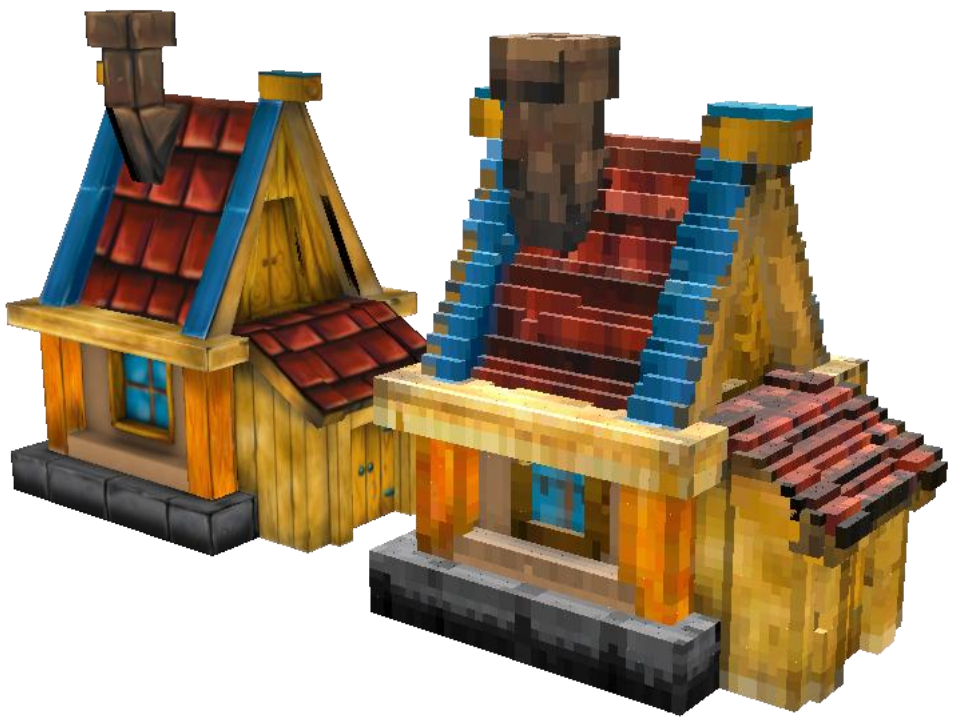
VOXEL SHADER: DOCUMENTATION



PREFACE

The geometry shader is a stage in the graphics rendering pipeline where the GPU generates vertices in order to create extra geometry, specified by the shader.
For this shader stage, I chose to develop an algorithm that would 'voxelise' any 3D model.

A voxel shader essentially recreates the given 3D model with blocks. The shader generates blocks at specific positions, in order to match the original model as much as possible.
The size of the blocks is configurable, and decides the amount of blocks visible and thus the amount of detail in the voxel version of the mesh.

The block color is mapped from the diffuse texture of the base mesh.

VOXEL SHADER ANATOMY: INPUT

SHADER CONSTANTS

Two shader constant buffers to pass on constant values like the matrices and the light direction.

VARIABLES

There is only one variable available, and that's the block size, every mesh will have a different amount of triangles, and thus will allow for different block sizes, but generally every high-poly model will allow a range of 2 (most accuracy and detail) – 10 (least accuracy and detail).

STATES

The rasterizer state is set to back culling, the blendstate is disabled since there is no blending necessary and the texture wrap mode is set to wrap.

DIFFUSE

A diffuse texture, and a variable to determine if it should be used or not. There's also a color diffuse, that's used when there is no texture.

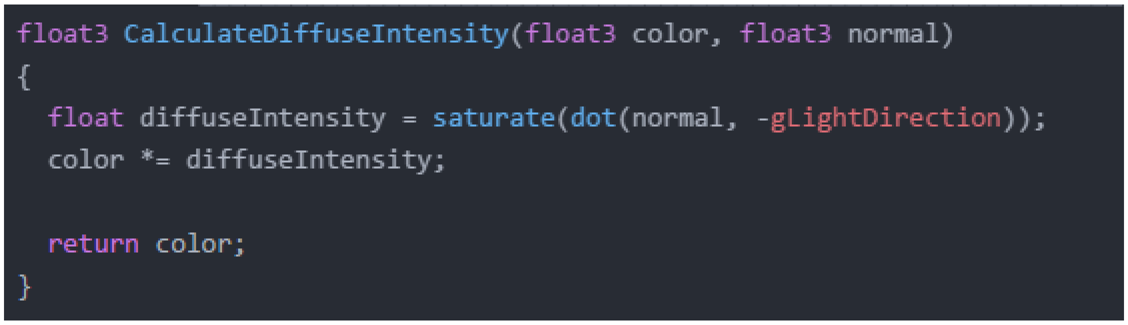
SPECULAR

A variable that defines the specularity color, and the shininess factor.

VOXEL SHADER ANATOMY: FUNCTIONS

DIFFUSE INTENSITY

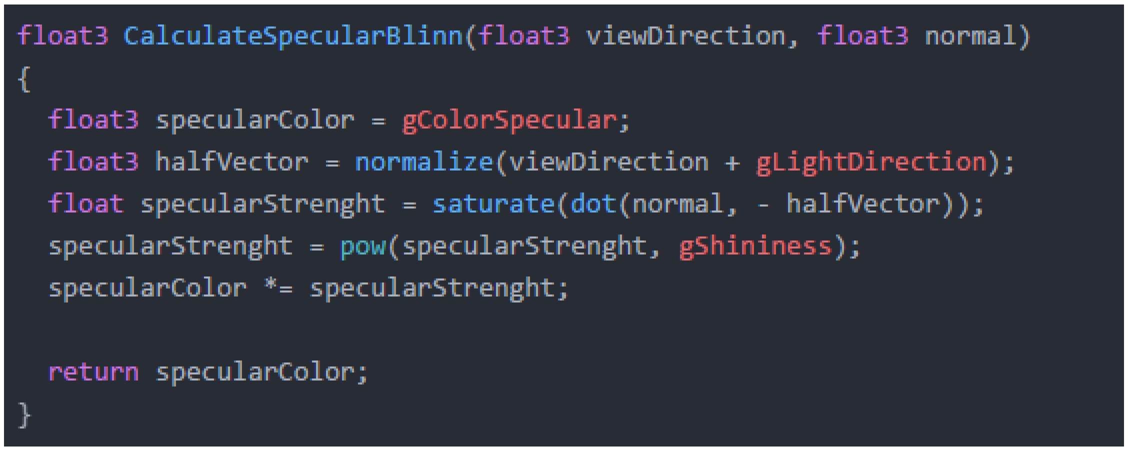
This is a function that will calculate diffuse intensity, the base diffuse color is already calculated in the geometry shader. This function will be used in the pixel shader.
The diffuse intensity is calculated by taking the dot product of the normal and the inversed light direction. The result of this is saturated and multiplied with the diffuse color that was already calculated.



```
float3 CalculateDiffuseIntensity(float3 color, float3 normal)
{
  float diffuseIntensity = saturate(dot(normal, -gLightDirection));
  color *= diffuseIntensity;

  return color;
}
```

SPECULARITY (BLINN)

This function will add specularity according to the blinn specularity model. I used blinn because that seemed to have the best effect on the voxel mesh.
Specularity, according to the blinn model, starts off with a base color. In this case the color is given.
First, a halfvector gets calculated by normalizing two vectors: the view direction and the light direction.
Subsequently, the specularstrength is calculated by saturating the dot product of the normal and the inversed halfvector.
The result, to the power of the shininess factor, then gets multiplied with the original base color.



```
float3 CalculateSpecularBlinn(float3 viewDirection, float3 normal)
{
  float3 specularColor = gColorSpecular;
  float3 halfVector = normalize(viewDirection + gLightDirection);
  float specularStrenght = saturate(dot(normal, - halfVector));
  specularStrenght = pow(specularStrenght, gShininess);
  specularColor *= specularStrenght;

  return specularColor;
}
```

VOXEL SHADER ANATOMY: STRUCTS

VS_DATA

Data about the vertices of the base mesh.
Position, as a float3 (x, y, z).
Normal, as a float3 (x, y, z).
TexCoord, as a float2 (u, v).

GS_DATA

Data about the vertices generated by the geometry shader.
Position, as a float3 (x, y, z).
WorldPosition, as a float4 (x, y, z, w). Note that I had to use the COLOR semantic for WorldPosition, because the POSITION semantic doesn't allow for a float4.
Normal, as a float3 (x, y, z).
Color, as a float3(x, y, z).

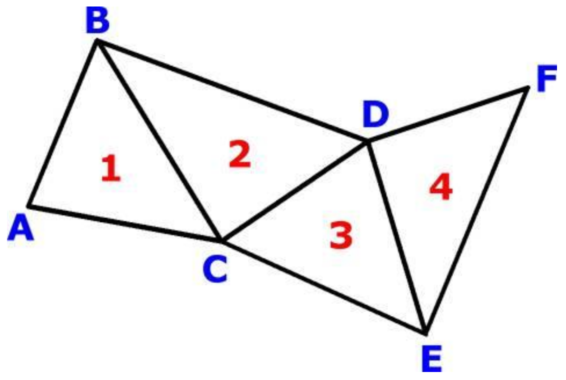
VOXEL SHADER ANATOMY: VERTEX SHADER STAGE

There are no calculations done in the vertex shader, but it's not completely redundant either, it needs to pass on data to the geometry shader.

VOXEL SHADER ANATOMY: GEOMETRY SHADER STAGE

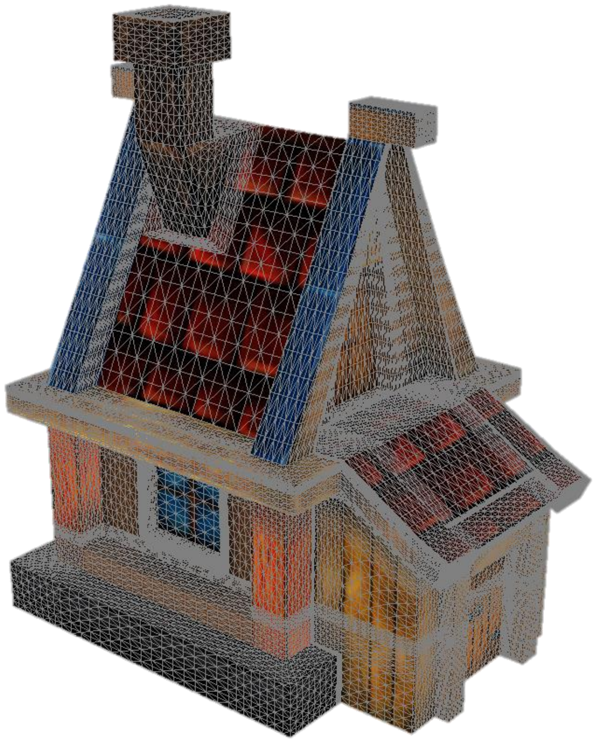
BASIC FUNCTIONALITY

A 3D model consists of vertices, a vertex is essentially a point in 3D space.
It holds data, in this case its position, normal and texture coordinates.
These vertices are clustered in a primitive topology, a way of organizing a group of vertices.
A primitive topology defines how vertices are interpreted and rendered by the pipeline.
Below you can see an example of a primitive topology, the trianglestrip.



The geometry shader works by looping over the vertices of a full primitive, in this case, three vertices to make one triangle.

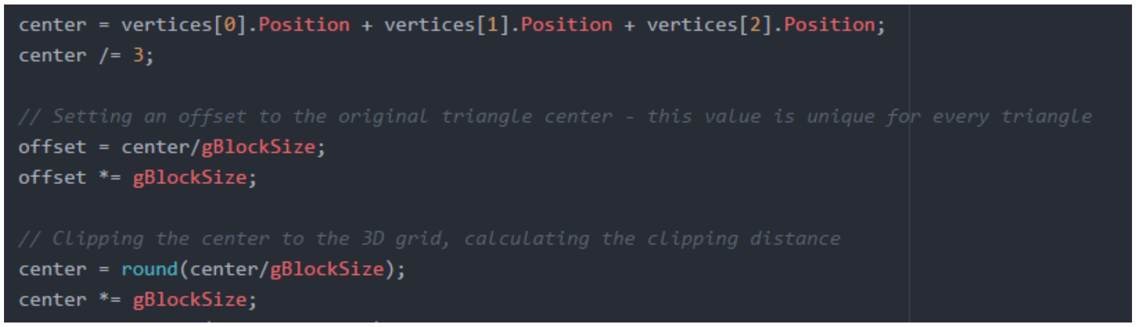
VOXEL SHADER FUNCTIONALITY



My shader works by calculating the center point of each triangle of the base mesh, rounding that value so it gets 'clipped' to a 3D grid and then generating a block with a configurable size on that position.

The color of the block is calculated by taking the average of the color of each vertex.
The color of the vertices either gets mapped off a texture or gets assigned the base diffuse color.

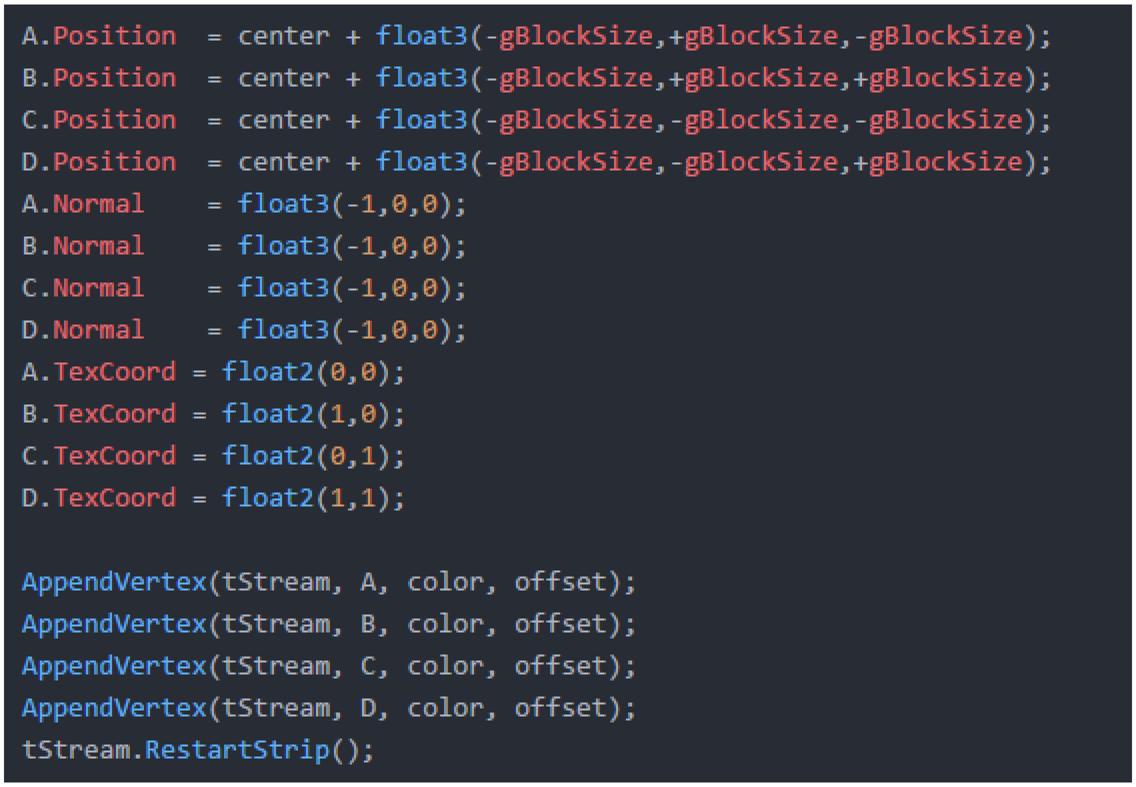
SHADER CODE: CENTER CALCULATION



```
center = vertices[0].Position + vertices[1].Position + vertices[2].Position;
center /= 3;

// Setting an offset to the original triangle center - this value is unique for every triangle
offset = center/gBlockSize;
offset *= gBlockSize;

// Clipping the center to the 3D grid, calculating the clipping distance
center = round(center/gBlockSize);
center *= gBlockSize;
```

First of all, the center of the triangle is required. This is calculated by dividing the sum of all vertex positions by 3.
Before rounding the center position (and, that way, clipping it to the 3D grid), I write the position to a float3 variable 'offset'. This variable will be later used in a way to fix z-fighting.

SHADER CODE: CUBE GENERATION

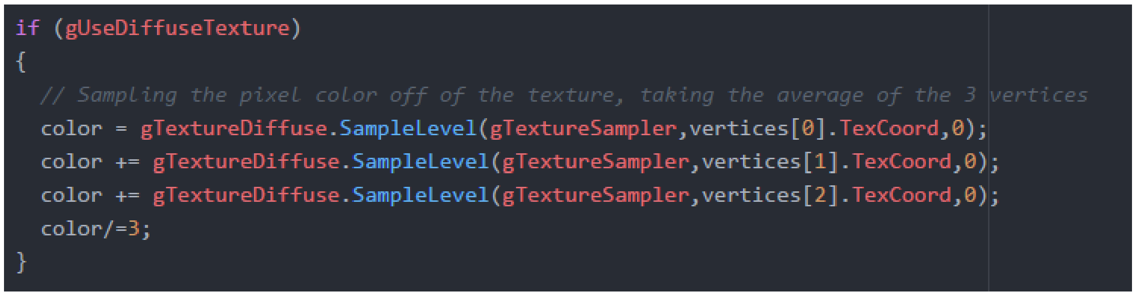


```
A.Position  = center + float3(-gBlockSize,+gBlockSize,-gBlockSize);
B.Position  = center + float3(-gBlockSize,+gBlockSize,+gBlockSize);
C.Position  = center + float3(-gBlockSize,-gBlockSize,-gBlockSize);
D.Position  = center + float3(-gBlockSize,-gBlockSize,+gBlockSize);
A.Normal    = float3(-1,0,0);
B.Normal    = float3(-1,0,0);
C.Normal    = float3(-1,0,0);
D.Normal    = float3(-1,0,0);
A.TexCoord = float2(0,0);
B.TexCoord = float2(1,0);
C.TexCoord = float2(0,1);
D.TexCoord = float2(1,1);

AppendVertex(tStream, A, color, offset);
AppendVertex(tStream, B, color, offset);
AppendVertex(tStream, C, color, offset);
AppendVertex(tStream, D, color, offset);
tStream.RestartStrip();
```

This is the core of the code, where the blocks get generated.
There's 4 VS_DATA objects, each making up one vertex of one side of the block.
So to complete a block, there's 6 of these codeblocks, each to build one quad (or one side of the block).
The position of the vertices is defined both by where on the block the quad is located and where

on the quad the vertex is supposed to be.
The normal are defined by the position of the quad on the block.

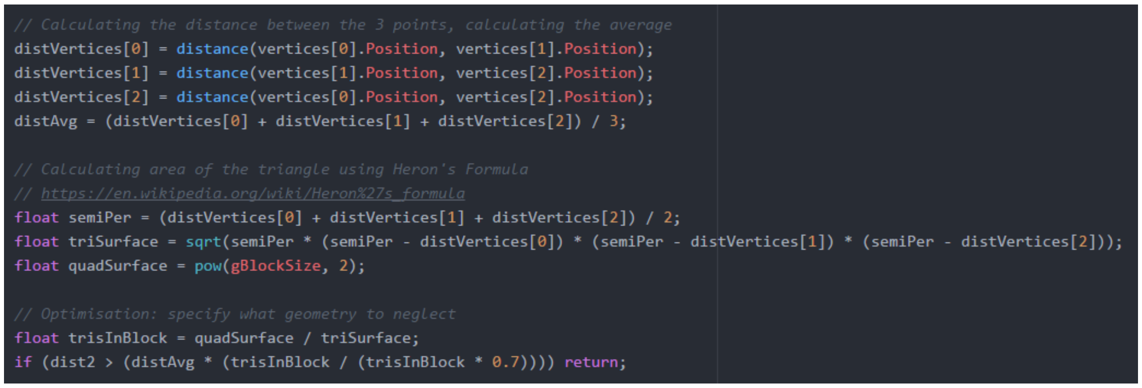
SHADER CODE: COLOR



```
if (gUseDiffuseTexture)
{
  // Sampling the pixel color off of the texture, taking the average of the 3 vertices
  color = gTextureDiffuse.SampleLevel(gTextureSampler,vertices[0].TexCoord,0);
  color += gTextureDiffuse.SampleLevel(gTextureSampler,vertices[1].TexCoord,0);
  color += gTextureDiffuse.SampleLevel(gTextureSampler,vertices[2].TexCoord,0);
  color/=3;
}
```

The way to calculate the color is very similar to that of the position. The color of each vertex is summed and divided by 3. This process will only happen if a diffuse texture is active, if it is not, then the base diffuse color will be used for each block.
Something to note is that gTextureDiffuse.Sample(), is only available in the pixel shader stage, but because this code is executed in the geometry shader stage, I had to use a different way to sample textures.
A list of all texture lookup methods and in which shader stages they work can be found here:
https://msdn.microsoft.com/en-us/library/bb509700

SHADER CODE: OPTIMISATION

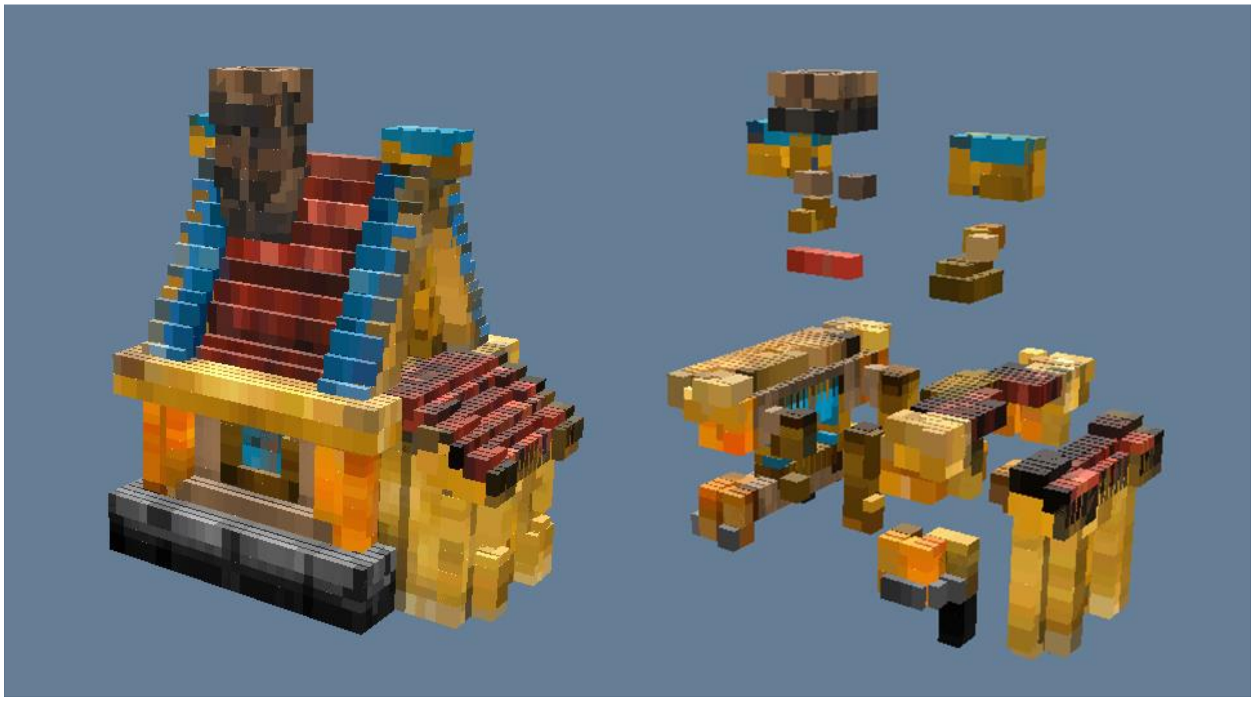


```
// Calculating the distance between the 3 points, calculating the average
distVertices[0] = distance(vertices[0].Position, vertices[1].Position);
distVertices[1] = distance(vertices[1].Position, vertices[2].Position);
distVertices[2] = distance(vertices[0].Position, vertices[2].Position);
distAvg = (distVertices[0] + distVertices[1] + distVertices[2]) / 3;

// Calculating area of the triangle using Heron's Formula
// https://en.wikipedia.org/wiki/Heron%27s_formula
float semiPer = (distVertices[0] + distVertices[1] + distVertices[2]) / 2;
float triSurface = sqrt(semiPer * (semiPer - distVertices[0]) * (semiPer - distVertices[1]) * (semiPer - distVertices[2]));
float quadSurface = pow(gBlockSize, 2);

// Optimisation: specify what geometry to neglect
float trisInBlock = quadSurface / triSurface;
if (dist2 > (distAvg * (trisInBlock / (trisInBlock * 0.7)))) return;
```

Drawing a block for every triangle of the base mesh results in a terribly unoptimised and inefficient shader. In very high poly models this will give a lot of blocks clipped to the same position and thus a lot of unnecessary blocks. These unnecessary blocks are not only inefficient to draw, they also cause z-fighting. More about this later.

Figuring out a way to decide on if a block should be drawn, or the triangle should be discarded was a difficult process. My first idea was to write which 3D grid positions were already taken to a global variable and then just checking that variable for every block. This was shut down pretty quickly because there is no way of writing or altering globals in the geometry shader.

Eventually I decided on an algorithm to calculate the probability of other blocks. If there are a lot of other blocks then don't draw the current one, if there's only one block for this triangle, then draw it. There is still a lot of room for improvement regarding accuracy, but it does isolate a lot of blocks to discard.



On the left you can see the full voxel mesh, on the right the blocks that were discarded. Something to note here is that the bigger the blocks become, the more blocks will be discarded. This is a logical connection between the amount of triangles in the base mesh (which stays the same), and the size of the block (which fluctuates).

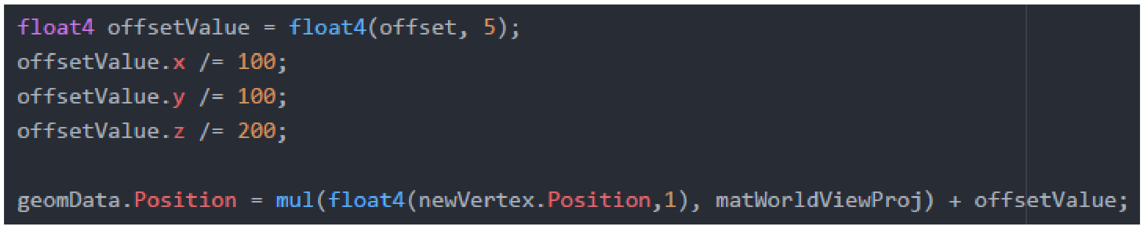
The variables required for the algorithm:

dist2:	Distance between regular center and clipped center
distAvg:	Average distance of all the vertices of the triangle
triSurface:	Area of the triangle, calculated with Heron's Formula https://en.wikipedia.org/wiki/Heron%27s_formula
quadSurface:	Area of the blocks
trisInBlock:	quadSurface / triSurface

The formula:

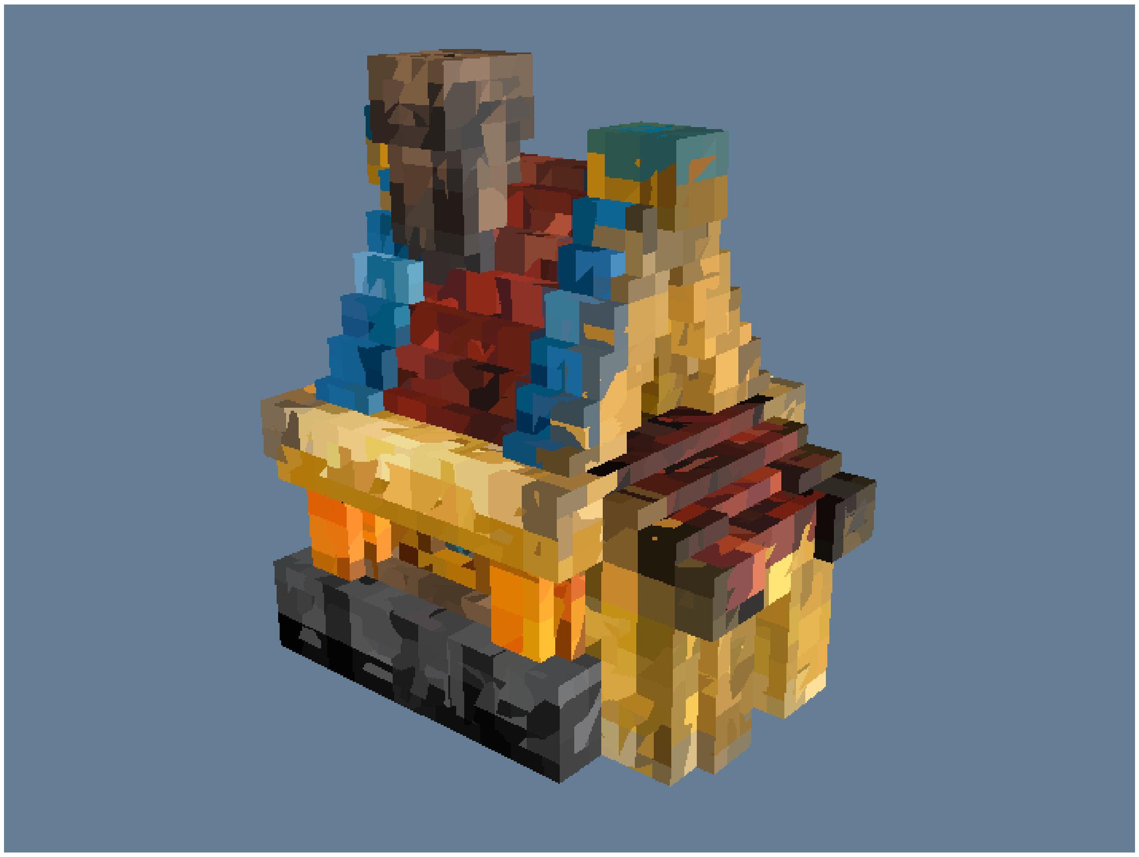
```
if (dist2 > (distAvg * (trisInBlock / (trisInBlock * 0.7)))) return;
```

SHADER CODE: Z-FIGHTING



```
float4 offsetValue = float4(offset, 5);
offsetValue.x /= 100;
offsetValue.y /= 100;
offsetValue.z /= 200;

geomData.Position = mul(float4(newVertex.Position,1), matWorldViewProj) + offsetValue;
```

Z-Fighting occurs when multiple planes are located on the exact same position.
This causes them to 'fight' because they each want to be rendered. This effect, however, is only visible when the planes have different colors or textures.
The result is visible by flickering and an overall terrible look.



I fixed most z-fighting issues by passing on another position variable to my function that writes the new vertices. This position variable is the 'offset' variable explored earlier. It's the calculated center point of the given triangle, but has not been clipped to the grid. This means this position is unique for each triangle.

After I do the worldviewprojection matrix transformation I add this unique position divided by 100 to make it hardly noticeable. It makes sure that the blocks are all on a different, unique position while still positioned on the grid.

VOXEL SHADER ANATOMY – PIXEL SHADER STAGE

The pixel shader starts by normalizing the given normal.
Then, a function is used to calculate the diffuse intensity of the diffuse color calculated in the geometry shader.
Next, the specular color gets calculated with the view direction and the normalized normal using the function described earlier.

The end-result and return value of the pixel shader is the sum of the diffuse color and specular color.

PROJECT FUTURE

I still want to find a better algorithm to optimize the shader, my goal to write one that can successfully filter out all redundant blocks, and only draw one block per clip position.
This will also fix z-fighting, another issue I want to completely take out.

CONCLUSION

So to recap, my shader calculates the center point of each triangle, rounds that position to clip it to a 3D grid, draws a block of configurable size on that position and calculates color based on the original vertices of the triangle. With an optimization algorithm it takes out a lot of redundant blocks and by offsetting the final position after doing to worldviewprojection transformation it takes out most of the z-fighting.

I learned a great deal by developing this shader and I actually really enjoyed it.
I'm not fully satisfied with the end-result, but I am proud of how far I took this shader.

Thank you for reading.