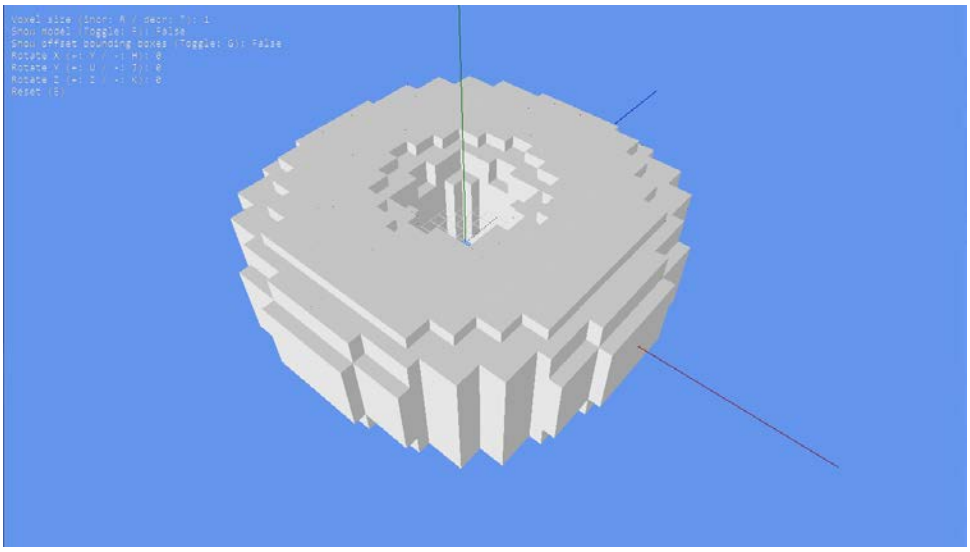
Geometry Shader: Voxelization



Introduction

The goal of the shader is the voxelization of an object, directly to a mesh.

After searching for a couple hours, I found out that the only 2 commonly use techniques are to 1) Convert each primitive into a part of a 3D texture, which will be outputted by the shader and still has to be rendered by another shader, which would mean, that I would have to plug in an extra system into the render pipeline to make it work, but in my case, I want to convert the object directly into a voxelized object, or 2) Instead of creating geometry, the shader will output an image with a raytraced voxel object, again, this is not what I want, so I had to figure out another way of making the shader.

I decided, that instead of truly voxelizing the object in the shader, for which I thought up a possible way, but where the shader was limiting the possibility to implement it (more on this later). I just approximated voxelization using the bounding boxes of each triangle in the object, this also means that an object with a higher density of smaller triangle will yield a better approximation of the truly voxelized object.

Vertex Shader

The vertex shader only has a small purpose, the only operation done by this shader, is converting the vertices to world space, because the voxels will lie on a grid with the size of the

voxels, centered around $(0, 0, 0)$. This calculation is to prevent the object having voxels, rotated on the grid, compared to other voxelized objects.

```
//--------------------------------------------------------------------------------------
// Vertex Shader
//--------------------------------------------------------------------------------------
VS_DATA VS(VS_DATA input)
{
    VS_DATA output = input;
    output.pos = mul(float4(input.pos, 1.f), gWorld).xyz;
    return output;
}
```

Geometry Shader

```
[maxvertexcount(36)]
void GS(triangle VS_DATA vertex[3], inout TriangleStream<GS_DATA> triStream)
{
    VS_DATA vert0 = vertex[0];
    VS_DATA vert1 = vertex[1];
    VS_DATA vert2 = vertex[2];

    if (gShowMesh)
    {
        TransformVertex(triStream, vert0.pos, vertex[0].color, vert0.normal);
        TransformVertex(triStream, vert1.pos, vertex[1].color, vert1.normal);
        TransformVertex(triStream, vert2.pos, vertex[2].color, vert2.normal);
    }
    else
    {
        float4 color = (vert0.color + vert1.color + vert2.color) / .3f;
        float3 minBound, maxBound;
        BoundingBox(vert0.pos, vert1.pos, vert2.pos, minBound, maxBound);

        if (gShowBoundingBoxes)
        {
            GenerateBox(triStream, minBound, maxBound, color);
        }
        else
        {
            ResizeBoundingBox(minBound, maxBound);
            GenerateBox(triStream, minBound, maxBound, color);
        }

    }
}
```

Step One

Step one, alongside the next steps are simple operations, but which result in a good-looking approximation, although this will always be bigger than the actual object.

The best possible fitting bounding box is created by finding the maximum and minimum coordinates on the triangles, because these points will always be on the vertices, this is rather simple. The minimum and maximum bounds can simply be calculated by taking the component wise minimum and maximum of the 3 vertices' positions.

This step is done using the following code:

```
void BoundingBox(float3 v0, float3 v1, float3 v2, out float3 minBound, out float3 maxBound)
{
    minBound = min(min(v0, v1), v2);
    maxBound = max(max(v0, v1), v2);
}
```

Step Two

Once we have the bounding boxes, the next step is to make sure they look like voxels and not like a bunch of randomly sized boxes pasted together.

This is done by aligning both the minimum and maximum bound on a grid, where the division have the size of the voxels we want.

This is done by dividing both bounds by the voxel size. I want to be certain the approximation does not start looking weird with big voxel sizes, so I floor the returned value of the minimum bound and ceil the value of the maximum bound. Because of some issues the standard floor and ceil have, when it comes to negative values, I created custom floor and ceil function that keep in mind the sign of the components.

```
int3 Floor(float3 vec)
{
    int3 ret;
    ret.x = vec.x < 0 ? (int) (vec.x - 1) : (int) vec.x;
    ret.y = vec.y < 0 ? (int) (vec.y - 1) : (int) vec.y;
    ret.z = vec.z < 0 ? (int) (vec.z - 1) : (int) vec.z;
    return ret;
}

int3 Ceil(float3 vec)
{
    int3 ret;
    ret.x = vec.x < 0 ? (int) vec.x : (int) (vec.x + 1);
    ret.y = vec.y < 0 ? (int) vec.y : (int) (vec.y + 1);
    ret.z = vec.z < 0 ? (int) vec.z : (int) (vec.z + 1);
    return ret;
}
```

After they are clamped to a unit grid, the bounds are again multiplied to by the voxel size to fit on the grid defined for the voxels.

Step Three

The third and last step is the convert the calculated bounds into a cuboid that can be rendered. This is done like creating a box, but instead of giving the position and size, we give in the 2 bounds, where the minimum bound represents:
$(\text{position.x} - \text{size.x}, \text{position.y} - \text{size.y}, \text{position.z} - \text{size.z})$, and the maximum bound represents: $(\text{position.x} + \text{size.x}, \text{position.y} + \text{size.y}, \text{position.z} + \text{size.z})$.

```
void GenerateBox(inout TriangleStream<GS_DATA> triStream, float3 minBound, float3 maxBound, float4 color)
{
    //front
    float3 p0 = float3(minBound.x, maxBound.y, minBound.z);
    float3 p1 = float3(minBound.x, minBound.y, minBound.z);
    float3 p2 = float3(maxBound.x, maxBound.y, minBound.z);
    float3 p3 = float3(maxBound.x, minBound.y, minBound.z);

    TransformVertex(triStream, p0, color, float3(0.f, 0.f, -1.f));
    TransformVertex(triStream, p1, color, float3(0.f, 0.f, -1.f));
    TransformVertex(triStream, p2, color, float3(0.f, 0.f, -1.f));
    TransformVertex(triStream, p3, color, float3(0.f, 0.f, -1.f));
    triStream.RestartStrip();

    //back
    p0 = float3(minBound.x, maxBound.y, maxBound.z);
    p1 = float3(minBound.x, minBound.y, maxBound.z);
    p2 = float3(maxBound.x, maxBound.y, maxBound.z);
    p3 = float3(maxBound.x, minBound.y, maxBound.z);

    TransformVertex(triStream, p0, color, float3(0.f, 0.f, 1.f));
    TransformVertex(triStream, p1, color, float3(0.f, 0.f, 1.f));
    TransformVertex(triStream, p2, color, float3(0.f, 0.f, 1.f));
    TransformVertex(triStream, p3, color, float3(0.f, 0.f, 1.f));
    triStream.RestartStrip();

    //left
    p0 = float3(minBound.x, minBound.y, maxBound.z);
    p1 = float3(minBound.x, minBound.y, minBound.z);
    p2 = float3(minBound.x, maxBound.y, maxBound.z);
    p3 = float3(minBound.x, maxBound.y, minBound.z);

    TransformVertex(triStream, p0, color, float3(-1.f, 0.f, 0.f));
    TransformVertex(triStream, p1, color, float3(-1.f, 0.f, 0.f));
    TransformVertex(triStream, p2, color, float3(-1.f, 0.f, 0.f));
    TransformVertex(triStream, p3, color, float3(-1.f, 0.f, 0.f));
    triStream.RestartStrip();

    //right
    p0 = float3(maxBound.x, minBound.y, maxBound.z);
    p1 = float3(maxBound.x, minBound.y, minBound.z);
    p2 = float3(maxBound.x, maxBound.y, maxBound.z);
    p3 = float3(maxBound.x, maxBound.y, minBound.z);

    TransformVertex(triStream, p0, color, float3(1.f, 0.f, 0.f));
    TransformVertex(triStream, p1, color, float3(1.f, 0.f, 0.f));
    TransformVertex(triStream, p2, color, float3(1.f, 0.f, 0.f));
    TransformVertex(triStream, p3, color, float3(1.f, 0.f, 0.f));
    triStream.RestartStrip();

    //bottom
    p0 = float3(minBound.x, minBound.y, maxBound.z);
    p1 = float3(minBound.x, minBound.y, minBound.z);
    p2 = float3(maxBound.x, minBound.y, maxBound.z);
    p3 = float3(maxBound.x, minBound.y, minBound.z);

    TransformVertex(triStream, p0, color, float3(0.f, -1.f, 0.f));
    TransformVertex(triStream, p1, color, float3(0.f, -1.f, 0.f));
    TransformVertex(triStream, p2, color, float3(0.f, -1.f, 0.f));
    TransformVertex(triStream, p3, color, float3(0.f, -1.f, 0.f));
    triStream.RestartStrip();

    //top
    p0 = float3(minBound.x, maxBound.y, maxBound.z);
    p1 = float3(minBound.x, maxBound.y, minBound.z);
    p2 = float3(maxBound.x, maxBound.y, maxBound.z);
    p3 = float3(maxBound.x, maxBound.y, minBound.z);

    TransformVertex(triStream, p0, color, float3(0.f, 1.f, 0.f));
    TransformVertex(triStream, p1, color, float3(0.f, 1.f, 0.f));
    TransformVertex(triStream, p2, color, float3(0.f, 1.f, 0.f));
    TransformVertex(triStream, p3, color, float3(0.f, 1.f, 0.f));
    triStream.RestartStrip();
}
```

During the creation of the box, the vertices still need to be translated into view space, which is done with the following function:

```
void TransformVertex(inout TriangleStream<GS_DATA> triStream, float3 pos, float4 col, float3 normal)
{
    GS_DATA data;
    data.pos = mul(float4(pos, 1.f), gViewProj);
    data.color = col;
    data.normal = normal;
    triStream.Append(data);
}
```

Pixel Shader

Because I have not found a way to implement texturing, the resulting cuboids will be the average value of the triangle they represent. The pixel shader just calculates Lambert shading on the resulting cuboid.

```
//--------------------------------------------------------------------------------------
// Pixel Shader
//--------------------------------------------------------------------------------------
float4 PS(GS_DATA input) : SV_TARGET
{

    float3 color_rgb = input.color.rgb;

    float diffuseStrength = saturate(dot(input.normal, -normalize(gLightDirection)) * .5f + .5f);
    color_rgb = color_rgb * diffuseStrength;

    //return float4(color_rgb, input.color.a);
    return float4(diffuseStrength, diffuseStrength, diffuseStrength, 1.f);
}
```

Afterthoughts

Although what the shader ended up being different than the original concept. It still approximates voxelization close enough, without having to switch to either the creation of a 3d Texture using conservative rasterization, which would still need to be rendered or ray-tracing the voxelized object to a texture generated by the shader.

Originally, I thought that it would be possible to use hlsl shader model 5's new instancing system, but after further reading, it is similar to multiple geometry shaders called on the same input from the vertex shader, instead one being called on the output of the previous one, but even if this was possible, it still would hinder the original concept, because that called on using 1 instance to generate points for each voxel contained in the bounding box of each triangle and in the next instance generating voxels on those points. But this would also have been difficult, because all instances would have the same input, but this could however be solved by each vertex having a value telling if a voxel assigned and doing it in batches of 3 (triangle input of the geometry shader used).