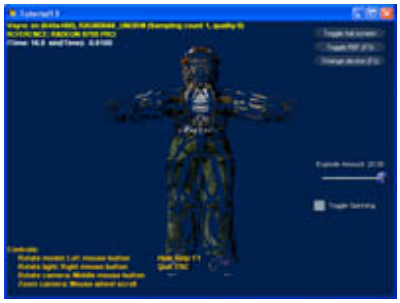
Tutorial 13: Geometry Shaders

Collapse All



Summary

This tutorial will explore a part of the graphics pipeline that has not been touched in the previous tutorials. We will be touching upon some basic geometry shader functionality.

The outcome of this tutorial is that the model will have a second layer extruded from the model. Note that the original model is still preserved at the center.

Source

SDK root\Samples\C++\Direct3D10\Tutorials\Tutorial13

Geometry Shader

The benefit of the geometry shader (GS) is that it allows manipulation of meshes on a per-primitive basis. Instead of running a computation on each vertex individually, there is the option to operate on a per-primitive basis. That is, vertices can be passed in as a single vertex, a line segment (two vertices), or as a triangle (three vertices).

By allowing manipulation on a per-primitive level, new ideas can be approached, and there is more access to data to allow for that. In the tutorial, you will see that we have calculated the normal for the face. By knowing the position of all three vertices, we can find the face normal.

In addition to allowing access to whole primitives, the geometry shader can create new primitives on the fly. Previously in Direct3D, the graphics pipeline was only able to manipulate existing content, and it could amplify or deamplify data. The geometry shader in Direct3D 10 can read in a single primitive (with optional edge-adjacent primitives) and emit zero, one, or multiple primitives based on that.

It is possible to emit a different type of geometry than the input source. For instance, it is possible to read in individual vertices, and generate multiple triangles based on those. This allows a wide range of new ideas to be executed on the graphics pipeline without CPU intervention.

The geometry shader exists between the vertex and the pixel shaders in the graphics pipeline. Since new geometry can potentially be created by the geometry shader, we must ensure that they are also properly transformed to projection space before we pass them off to the pixel shader (PS). This can either be done by the vertex shader (VS) before it enters the geometry shader, or it can be done within the geometry shader itself.

Finally, the output of the GS can be rerouted to a buffer. You can read in a bunch of triangles, generate new geometry, and store them in a new buffer. However, the concept of *stream output* is beyond the scope of tutorials, and it is best shown in the samples found in the SDK.

Many of the samples found in the Direct3D 10 SDK illustrate specific techniques that can be achieved with the geometry shader. If you want a basic sample to get started, you can try ParticlesGS. ParticlesGS simulates and renders a dynamic particle system (creating, exploding, and destroying particles) entirely on the GPU.

Formatting a Geometry Shader

Unlike the VS and the PS, the geometry shader does not necessarily produce a static number of outputs per input. As such, the format to declare the shader is different from the other two.

The first parameter is *maxvertexcount*. This parameter describes the maximum number of vertices that can be output each time that the shader is run. This is followed by the name of the geometry shader, which has

been aptly named *GS*.

The function name is followed by the parameters passed into the function. The first contains the keyword triangle, which specifies that the GS will operate on triangles as input. Following that is the vertex format, and the identifier (with the number signifying the size of the array, 3 for triangles, 2 for line segments). The second parameter is the output format and stream. *TriangleStream* signifies that the output will be in triangles (triangle strips to be exact), then the format is specified in the angled brackets. Finally, the identifier for the stream is denoted.

```
[maxvertexcount(12)]
void GS( triangle GSPS_INPUT input[3], inout TriangleStream<GSPS_INPUT> TriStream )
```

If vertices start being emitted to a TriangleStream, it will assume that they are all linked together as a strip. To end a strip, call **RestartStrip** within the stream. To create a triangle list, you have to make sure that you call **RestartStrip** after every triangle.

Exploding the Model

In this tutorial, we cover the basic functions of the geometry shader. To illustrate this, we will create an explosion effect on our model. This effect is created by extruding each vertex in the direction of that triangle's normal.

Note that a similar effect has been implemented in previous tutorials, whereby we extrude each vertex by its normal, controlled by the *puffiness* slider. This tutorial demonstrates the usage of the full triangle's information to generate the face normal. The difference of using the face normal is that you will see gaps between the exploded triangles. Because vertices on different triangles are shared, they will actually be passed twice into the geometry shader. Moreover, since each time it will extrude it in the normal of the triangle, as opposed to the vertex, the two final vertices may end up in different positions.

Calculating the face normal

To calculate the normal for any plane, we first need two vectors that reside on the plane. Since we are given a triangle, we can subtract any two vertices of the triangle to get the relative vectors. Once we have the vectors, we take the cross product to get the normal. We must also normalize the normal, since we will be scaling it later.

```
//
// Calculate the face normal
//
float3 faceEdgeA = input[1].Pos - input[0].Pos;
float3 faceEdgeB = input[2].Pos - input[0].Pos;
float3 faceNormal = normalize( cross(faceEdgeA, faceEdgeB) );
```

Once we have the face normal, we can extrude each point of the triangle in that direction. To do so, we use a loop, which will step through three times and operate on each vertex. The position of the vertex is extruded by the normal, multiplied by a factor. Then, since the vertex shader has not transformed the vertices to proper projection space, we must also do that in the geometry shader. Finally, once we package the rest of the data, we can append this new vertex to our TriangleStream.

```
for( int v=0; v<3; v++ )
{
    output.Pos = input[v].Pos + float4(faceNormal*Explode,0);
    output.Pos = mul( output.Pos, View );
    output.Pos = mul( output.Pos, Projection );

    output.Norm = input[v].Norm;

    output.Tex = input[v].Tex;

    TriStream.Append( output );
}
```

Once the three vertices have been emitted, we can cut the strip and restart. In this tutorial, we want to extrude each triangle separately, so we end up with a triangle list.

```
TriStream.RestartStrip();
```

This new triangle stream is then sent to the pixel shader, which will operate on this data and draw it to the render target.