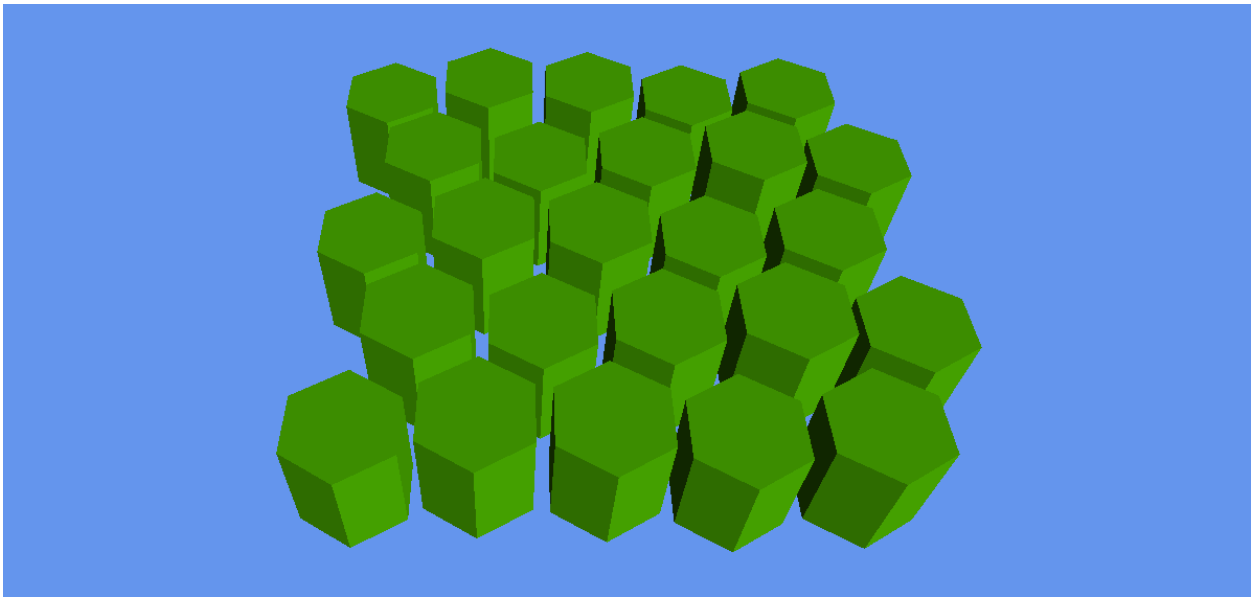
Geometry shader: Hexagon Field



Introduction

In this paper we will take a look at a “sci-fi looking”, moving hexagon field in HLSL.

The different things we will be looking at to see how this hexagon field is made are:

- Creating the hexagonal geometry
- Instantiating the geometry shader
- Calculating the grid
- Making it move

Setting up the geometry shader

First of all, we will need the value of π for our calculations when creating a hexagon. Now unfortunately, HLSL does not have a value of π yet, so we will have to define it ourselves. Well make a global variable for this which I called PI. I made this a float which approximates π with 3.14159265f.

```
float PI = 3.14159265f;
```

Next, we will have to define what the maximum number of vertices is that we will have. We will create one hexagonal tube per geometry shader instance. This means we will have 60 vertices as we'll see later. The way we define this is by typing [maxvertexcount(60)].

After this, we need to define the amount of instances of the geometry shader we will have. The maximum for this is 32 so that's what I used. The reason we instantiate the geometry shader is because there's only a limited amount of vertices that we can draw per instance. If we were to use just one instance, we would only be able to draw one hexagonal tube, but by instantiating we can now make 32 instead of one, giving us a decently sized hexagon field.

The last step of this preparation part is creating our actual geometry shader.

For the shader name, anything will do as long as you remember it for the technique.

I used "HexGenerator". Now the geometry shader needs some parameters.

These parameters are: a primitive type, a stream output object and an unsigned integer to keep track of what instance we are currently on. For the primitive type, I decided to use a point, meaning just one vertex. The reason for this is that I want to be able to apply this shader on any object. If we were to use triangle as primitive type for example, we wouldn't be able to apply the shader on geometry's existing of less than 3 vertices.

Using just a point will allow us to put the shader on any object with at least one vertex. However, putting it on a very complex and high poly geometry would be ill advised because our geometry shader will be called for every single vertex, creating multiple overlapping hexagon fields. It would also be completely pointless because the original geometry gets removed anyway.

For the stream object I used a triangle stream.

```
[maxvertexcount(60)]
[instance(32)]
void HexGenerator(point VS_DATA Vertex[1], inout TriangleStream<GS_DATA> triStream, uint InstanceID : SV_GSInstanceID)
```

Creating the new geometry

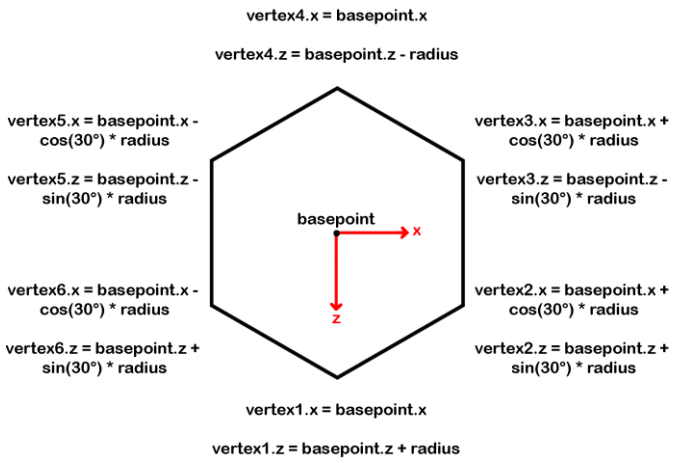
The geometry shader gets rid of the existing geometry unless we tell it not to. We don't want the original geometry, so we can get rid of it. First of all we need a way to create new vertices. For this we make a little function called "CreateVertex" which will append a new vertex with a position, a normal and a color to our triangle stream. This means that our function will need a few parameters, which are: our triangle stream, a float3 for the position, a float3 for the normal and a float4 for the color.

```
void CreateVertex(inout TriangleStream<GS_DATA> triStream, float3 pos, float3 normal, float4 color)
{
    //Step 1. Create a GS_DATA object
    GS_DATA MainGS= (GS_DATA)0;
    //Step 2. Transform the position using the WVP Matrix and assign it to (GS_DATA object).Position (Keep in mind: float3 -> float4)
    MainGS.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!)
    MainGS.Normal = mul(normalize(normal), (float3x3)m_MatrixWorld);
    //Step 4. Assign color to (GS_DATA object).Color
    MainGS.Color = color;
    //Step 5. Append (GS_DATA object) to the TriangleStream parameter
    triStream.Append(MainGS);
}
```

After this, we can start creating our geometry. We will start by defining the positions of our vertices using float3's. We will need 7 positions for the bottom plane and 7 for our top plane (prefix T).

```
float3 basePoint, vert1, vert2, vert3, vert4, vert5, vert6, TbasePoint, Tvert1, Tvert2, Tvert3, Tvert4, Tvert5, Tvert6;
```

We will start with the positions of the vertices of the bottom hexagon. The top and bottom vertex of this hexagon are easy. Just add or subtract the radius from the z value of the position of the basepoint. The other ones are a bit harder. We know that the angle between every point is 360°/ 6 or 60°. We also know that the x-axis splits those angles in half for every vertex except for the top and bottom one. So it is safe to say that the angle between the vertex next to the bottom one (moving counterclockwise) and the x axis is 30°. This means that the x-position of this vertex is the basepoint's x-position plus the cosine of 30° (or PI/6 for radians) times the radius. The z-position is the z-position of the basepoint plus the sine of 30° times the radius. Moving counterclockwise, the x-position of the next vertex will be the same, but for the z-position we need to subtract the cosine of 30° times the radius instead of adding it. Still moving counterclockwise, the next vertex is the top one, for which we already know the position. For the vertex after that, we need to subtract for both the x- and z-position and for the last one we subtract for x but add for z.



```
//1
vert1 = basePoint;
//vert1.y += Height;
vert1.z += m_Radius;

//2
vert2 = basePoint;
vert2.x += cos(PI / 6) * m_Radius;
//vert2.y += Height;
vert2.z += sin(PI / 6) * m_Radius;

//3
vert3 = basePoint;
vert3.x += cos(PI / 6) * m_Radius;
//vert3.y += Height;
vert3.z += -sin(PI / 6) * m_Radius;
```

The y-position of each vertex is equal to the y-position of the basepoint.

This process will be the same for the top plane but only with a different y-position for the height difference. Notice that I haven't said anything about the position of the basepoints yet. This is because these will vary depending on the height of our hexagonal tube, which row of the grid the hexagonal tube is positioned in and it's up and down movement. We will touch on this a bit later.

Now it's time to start adding our vertices. We start by telling our triangle stream that we start a new triangle strip. First we will make the top and bottom hexagon. We can now create the first vertex of our hexagon by calling our CreateVertex function and giving it the triangle stream and the position, the normal and the color of our vertex. For the bottom plane, the normal will be pointing down and for the top plain we will change it to point up. We will first create two vertices with positions on the edge of the hexagon and on the basepoint to create a triangle. After this we tell our triangle stream that we start a new triangle strip again. Create two vertices with positions on the edge of the hexagon again, but we move one position up, meaning we use one of the previous positions and on new one next to it. We also create another vertex at the position of the basepoint. This will create the triangle adjacent to the last one. We repeat this process until we have created six triangles forming a hexagon. In my case I created the opposite triangle every time before creating the adjacent one, but as long as you create all the triangles, the order doesn't really matter. This process is exactly the same for the bottom and top plane.

```
triStream.RestartStrip();

//VERTEX 1
CreateVertex(triStream, vert1, normal, m_Color);

//VERTEX 2
CreateVertex(triStream, vert2, normal, m_Color);

//VERTEX Mid
CreateVertex(triStream, basePoint, normal, m_Color);


//RESTART STRIP
triStream.RestartStrip();

//VERTEX Mid
CreateVertex(triStream, basePoint, normal, m_Color);

//VERTEX 4
CreateVertex(triStream, vert4, normal, m_Color);

//VERTEX 5
CreateVertex(triStream, vert5, normal, m_Color);
```

For the sides, we create a vertex on a position on a corner of the bottom hexagon followed by a vertex on the position on the corresponding corner on the top hexagon. After that we create a vertex on the next corner on the bottom hexagon followed by its corresponding one on the top hexagon. This will create an N-shape which will create two triangles. Calculating the normal for these vertices will be slightly trickier than the ones of the top and bottom plane. To get the normal we first need to find the direction in which the plane that we are creating is facing. We can easily do this by calculating the average between the positions of the two used vertices on the corners of the bottom hexagon and then subtracting the position of the base point of the bottom hexagon from this average. Of course this will not necessarily give us values between zero and one, which is required for a normal vector, so we normalize this resulting vector. This process is repeated for every side of the hexagonal tube. When we're done we'll have created 60 vertices as mentioned before.

```
// Side1

//RESTART STRIP
triStream.RestartStrip();

//Calculate normal
float3 dir = (vert1 + vert2) / 2 - basePoint;
normal = float3(dir.x, 0, dir.z);
normal = normalize(normal);

//VERTEX 1
CreateVertex(triStream, vert1, normal, m_Color);

//VERTEX 2
CreateVertex(triStream, Tvert1, normal, m_Color);

//VERTEX 3
CreateVertex(triStream, vert2, normal, m_Color);

//VERTEX 4
CreateVertex(triStream, Tvert2, normal, m_Color);
```

Creating a grid and movement

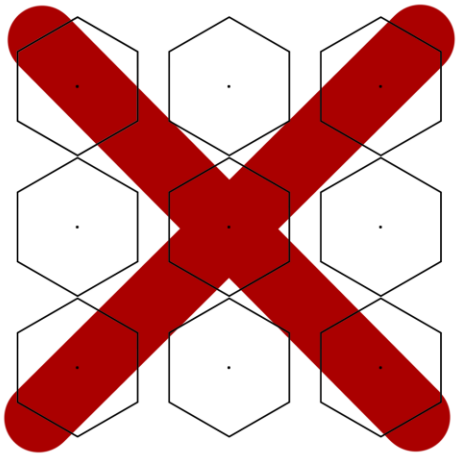
Creating the grid and the movement are together in a chapter because they are both about the position of the base point. If we just set the base points to zero and give them a height, only a single hexagon tube will be visible even though we create 32 of them (one for each instance).

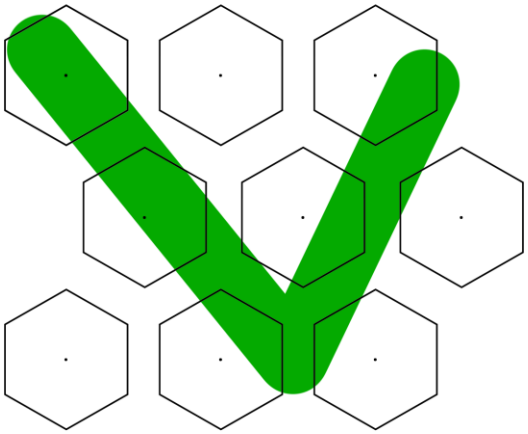
This is because they are all drawn in exactly the same spot. To create the grid we will have to change the x and y positions of the base points each iteration. Note that the x- and z-positions of the top and bottom basepoint are always equal.

To create a grid, we assume that we know the amount of rows and columns of the grid. Note that the amount of rows multiplied by the amount of columns can never be larger than 32 since this is the maximum amount of iterations we can do in HLSL. If this number would be higher, only 32 hexagon tubes will be created.

To know the x-position of the basepoints, we need to know in which column of the grid the hexagon is supposed to be. We can do this by getting the number of the current instance modulo the number of columns (instance nr. % nr. of columns). This number times the radius multiplied by two is the x-position of our basepoints.

There is however a problem with this. The rows of hexagons can't just be underneath each other. They need to fit in the row above, meaning that every even row needs to be shifted the length of the radius along the x-axis.





To do this we first need to know what row the hexagon is supposed to be in. This calculation goes as follows: int(instance nr. / nr. of columns) + 1. The reason this is type casted to an integer is because we need to make sure to get rid of everything after the comma.

Now that we know what row the hexagon is in, we can check whether the row is odd or even, by doing: row nr. % 2. If this value is 0, the row is odd, otherwise it is even. On every even row we add the radius to the x-position another time.

Now that we know the row number of the hexagon, we can calculate its z-position as well, by multiplying this row number with the double of the radius.

All we need to do now is take a look at the height of our hexagonal tubes. This is determined by the y-positions of the basepoints.

We want the possibility to put a bit of variation in our heights, so we will need to specify a maximum height and a minimum height. We will also define a value that will function as a semi-random number. For this semi-random number I use the formula:

(instance nr. % 7) - ((instance nr. % 3) * 1.2f) + ((instance nr. % 5) * 0.3) * ((instance nr. % 5) * 0.2) + ((instance nr. % 4) * 0.4)

The reason I call this semi-random is that there is obviously a pattern in there, but it will look somewhat random.

Now we can calculate the y value as follows:

((minimum height + semi-random * 10) % (maximum height - minimum height)) + minimum height

We only do this if the maximum and minimum heights are different from each other. If they are the same, we just use one of the two as y value.

Note that this is the y-position of the top hexagon. The y-position of the bottom one will be the negative of this value.

The grid is complete and the hexagons have somewhat random looking heights. The only thing left to do now is making them move up and down.

First we make another semi-random number and add this to a time variable. This time variable needs to be a timer and will be updated every frame.

Next we multiply the result with a move speed variable which determines the speed at which the hexagons will be moving.

After that we take the sine of this (cosine would work as well) and multiply this with a “move amount” variable which determines the maximum and minimum height difference of the movement.

Finally, we add the resulting value to the y positions of both our basepoints, which finalizes the geometry shader.

```
basePoint = m_Position;
if ((int(InstanceID / m_Columns) + 1) % 2 == 0)
    basePoint.x += m_Radius * (InstanceID % m_Columns) * 2 + m_Radius;
else
    basePoint.x += m_Radius * (InstanceID % m_Columns) * 2;

if(m_MaxHeight != m_MinHeight)
    basePoint.y -= ((m_MinHeight + semiRand*10) % (m_MaxHeight - m_MinHeight)) + m_MinHeight;
else
    basePoint.y -= m_MaxHeight;

basePoint.y += m_MoveAmount * sin((m_Time + (InstanceID % 5) - ((InstanceID % 2) * 0.5f)) * m_MoveSpeed);

basePoint.z -= m_Radius * (int((InstanceID) / m_Columns) + 1) * 2;
```

Final result