A different take on the grass geometry shader

Introduction

When hearing the term "geometry shader", 2 prominent uses come to mind: particles and vegetation. As there isn't that much variation that can be done on the particle side of things, vegetation still allows for some different approaches.

At first, as an introduction, I will quickly go trough the most commonly used grass shader and explain why I went with a different approach (with the eye on using this in my final project).

Generic grass shader

The most basic and generic grass shader consists of generating a single quad for a patch of grass. Most commonly these are created on the terrain by pushing the tris of the terrain to the geometry shader and generating the pathes there.

In most cases a couple of quads will be generated on a single tris and crossed so the grass looks more dense. (this is done by connecting the edges of the tris and creating planes between them.)

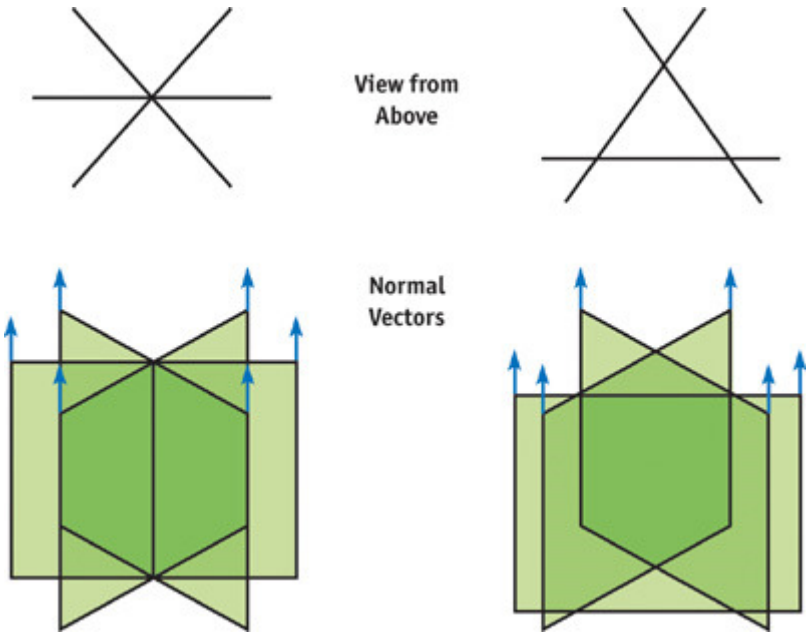

Fig 1. Demonstrates the crossing of grass planes (taken from nvidia website: GPU gems, chapter 7)

The major advantage to this technique is that it is really fast. In most games grass is just in the scene to make it look more realistic and filled up. The player isn't supposed to concentrate on the grass as there is enough interesting stuff happening on the screen.

The only way grass could possibly grab the players attention is when it isn't there (the environment would look dull). Or, and here is the deal breaker, when there is heavy wind influencing the grass.

As there are so few polygons, the grass will skew in a weird way. Just a little bit of skewing is ok. But overdoing it with this technique is definitely something to avoid.

One last techniques that is pretty much always present is bill boarding. Bill boarding is used to avoid that player from looking at the side of a plane as the grass rotates to face the camera.

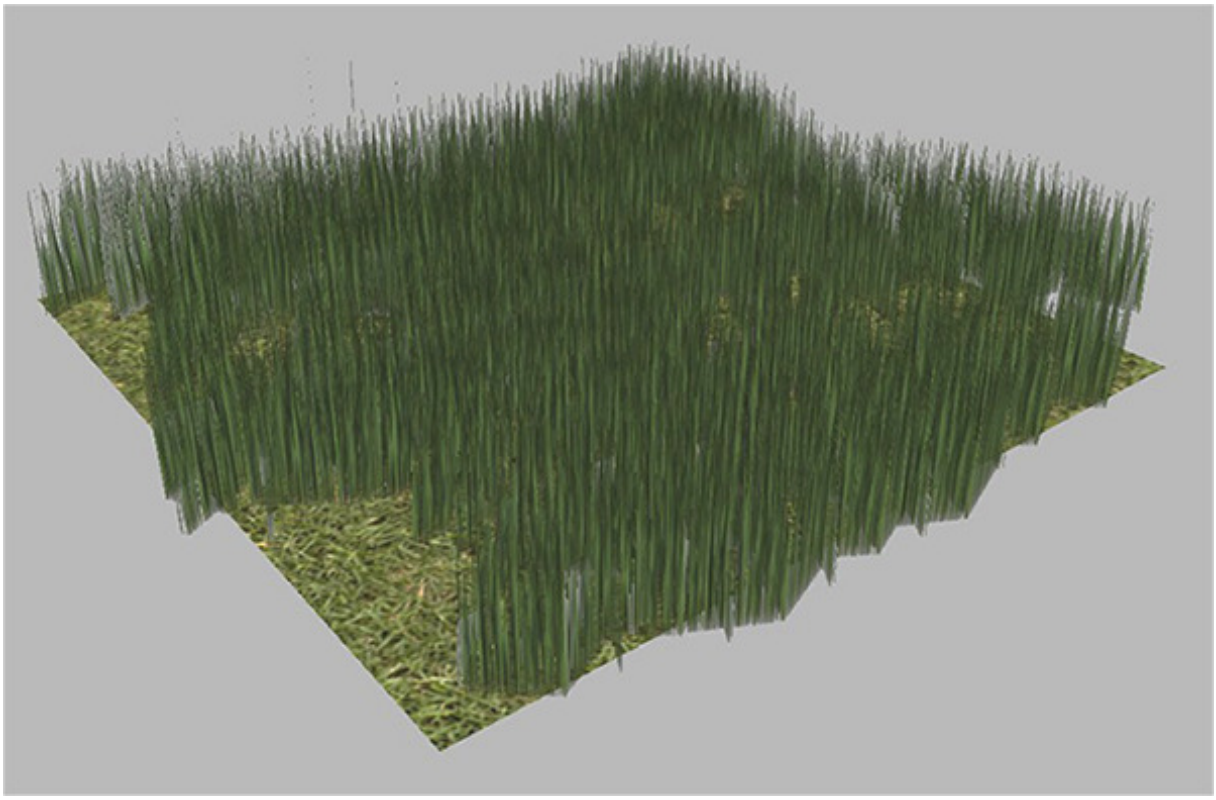

Fig 2. My implementation of this basic grass shader. (Screenshot from fxcomposer)

Fig 3. Grass on the planet Dagobah (screenshot from Star Wars Battlefront 2, 2005)

For full source code of this example please check **GrassShader0.fx** in the Grass1 folder.

The final approach

As mentioned in the previous paragraph, the previous technique doesn't perform very well in 2 occasions.

- There isn't really a lot else on the screen, so the player is naturally more tempted to look at the grass.
- When there is the possibility of heavy wind input.

As my final project features both of these issues, it was time to figure out a different approach.

The key differences between the final shader and the previous one are the following:

- We render every grass blade separately to increase diversity and realism
- Every blade consists of 1-5 quads depending on the LOD, to provide a better way a better base for wind impact.
- The grass positions itself are generated on the CPU instead of on the GPU.

In the following paragraphs we will address each of these points and explain why they were needed.

The blade

Each grass blade is generated from a single root position, and depending on the distance from the player consists of 1-5 quads.

The division is needed in order to avoid skewing and provide a more realistic way of bending the grass when it is influenced by the wind. As the player acts as a wind actor in my final project, it was pretty important to make this look good.

Fig 4. Grass blade division depending on LOD (screenshot by Edward Lee)

The blade itself also uses the bill boarding effect. As it rotates to face the player camera, it is impossible for them to see the side of the plane. Thanks to this effect the grass has volume in all directions even tough it is just a plane.

Grass generation & C++ implentation

It is important to note that the generation for this type of grass is completely different to the first example. Even tough it is entirely possible to generate the positions on the GPU we are greeted by the following error message when we want to generate all the grass blades in a single pass.

" Error Error: error X8000: Validation Error: Declared output vertex count (100) multiplied by the total number of declared scalar components of output data (16) equals 1600. ***This value cannot be greater than 1024****. ..."*

Obviously to generate an grass for an entire tris, we need way more than 100 vertices (as that only grants us 16.6 blades at the highest LOD, keep in mind that 6 vertices are generated per quad as we are not using indices)

The counter this issue there are as far as I know 2 approaches:

- We create a terrain with smaller tris so 10 blades is enough to create a dense tris.
- We use this shader on a separate vertex buffer and generate the grass based on points instead of the tris of the terrain.

In this case I opted for the second solution because the grass is always in the same position during the entire duration of the scene. Plus the shader for the grass would have to work a bit harder as well.

The only big downside to the decision is that it is now impossible to just place this shader on any object and make it look just as dense. You will however get 1 blade of grass per vertex, but that's not really what we are aiming for.

In the C++ implementation this means that we will have to generate a separate vertex buffer for the grass blades depending on the vertices of our terrain.

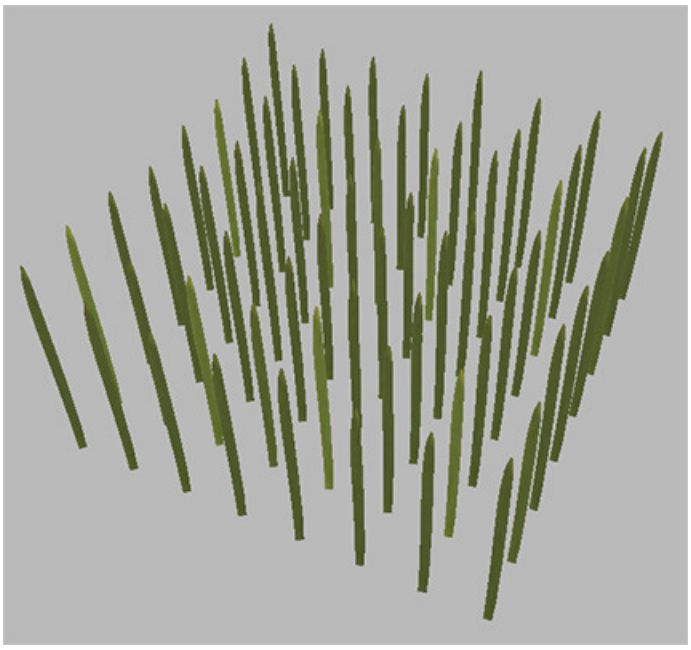

Fig 5.
Shows how the shader reacts to a plane.

(Not really what we are looking for)

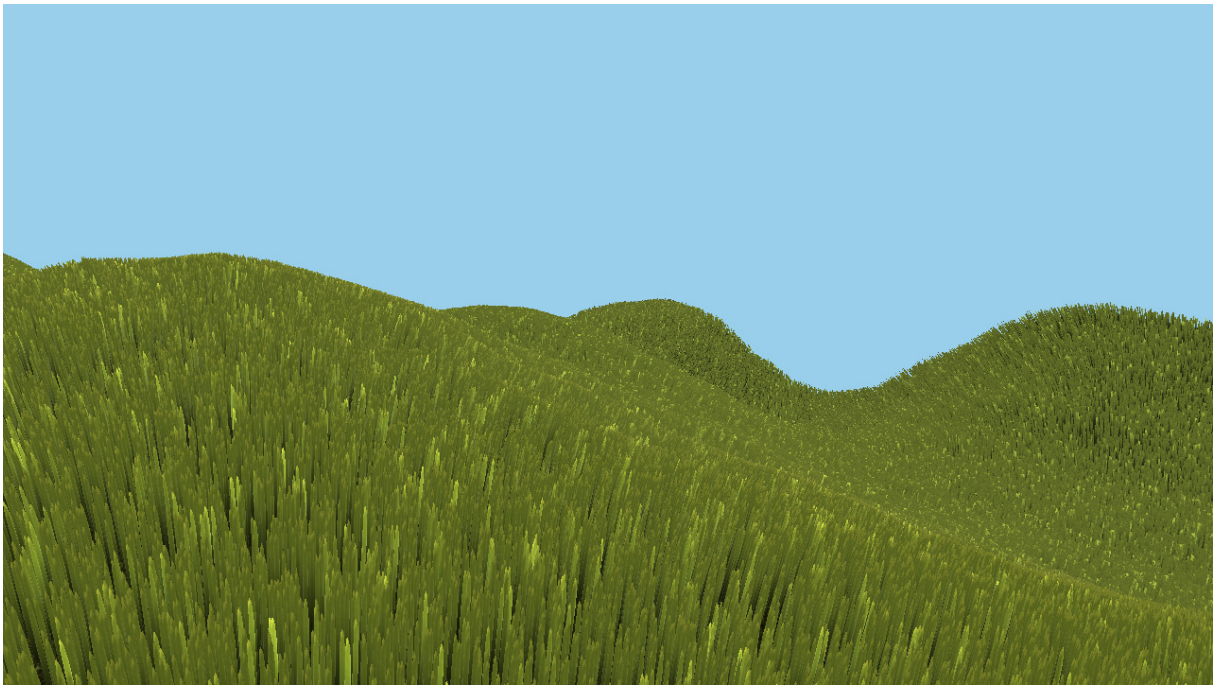

Fig 6. Example in game when using a separate vertex buffer.

Conclusion

Obviously this technique has various downsides. Most importantly there is the speed. In most cases this algorithm simply isn't viable as there are other (more important) things going on in the game. By rendering every blade separately a huge chunk of processing power is lost, and by that point all you have is a field of grass.

The plusses on the other side are also something to consider. By having the ability to alter the properties of every single grass blade it is easy to achieve naturally looking grass. By randomising the height, strength (how much it is influenced by the wind) and colour we can get some nice variation in there. It all comes down to your priorities.

Fig 6. My final project, featuring the grass shader in an environment.

Apendix: Source breakdown

Vertex shader

There's really nothing to see here. As we use a geometry shader all vertices are transformed there.

Geometry shader

The key part of our shader, here we receive a single point and build a blade on top of that position.

It's quite a big function so we'll break it down bit by bit.

```
[maxvertexcount(30)]
void GrassGenerator(point VS_DATA vertex[1], inout TriangleStream<GS_DATA>
triStream)
```

I chose a max vertex count of 30 as it allows for a max lod of 5, 5 quads at 6 vertices each (2 tris, 3 each). You can also see here that I work with point data instead of triangle data.

```
//First thing we do is check if grass actually grows here.
if (dot(vertex[0].Normal, float3(0, 1, 0)) < 0) return;

float3 startPoint, endPoint, startTopPoint, endTopPoint, normal;

//Generate a random number based on the current position
float3 normalPos = normalize(vertex[0].Position);
float3 noiseSample = m_NoiseTexture.SampleLevel(samLinear, normalPos.xy, 0);

//Set some variables
float maxVariation = (m_GrassHeight/5.0f);
float grassHeight = m_GrassHeight - maxVariation + (noiseSample.r *
(maxVariation*2)); //random offset
float3 windDirection = m_WindDirection; //random offset

//Get the start & end position of the root vertices
startPoint = vertex[0].Position;
startPoint.x = vertex[0].Position.x - m_GrassWidth/4;
endPoint = startPoint;
endPoint.x += m_GrassWidth/2;
```

Let's just call this the preparation phase, we setup a bunch of variables to be used later on. This includes doing a sample from a perlin noise map for variation purposes.

Also worth nothing that grass doesn't grow on any surfaces with a normal of more than 90 degrees from an up vector. If this shader was thrown on a sphere, only the top would be covered in grass.

```
//Transform the root vertices to follow the winddirection (all the other vertices will follow)
float4x4 matTranslateBack = CreateTranslationMatrix(float3(0, 0, 0), vertex[0].Position);
float4x4 matTranslate = CreateTranslationMatrix(vertex[0].Position, float3(0, 0, 0));

//billboarding
float3 viewDirection = normalize(vertex[0].Position - m_MatrixViewInverse[3].xyz);

float4x4 matRotate = CreateRotationMatrix(viewDirection);
float4x4 matTransform = float4x4(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1);

matTransform = mul(matTranslate, matRotate);
matTransform = mul(matTransform, matTranslateBack);

startPoint = mul(float4(startPoint, 1.0f), matTransform);
endPoint = mul(float4(endPoint, 1.0f), matTransform);
```

This part is present to rotate our start and endpoints correctly in relation to the players camera. Billboarding in a nutshell. All points generated on the basis of these will follow the same rotation. (startpoint is the bottom left vertex and rightpoint is the bottom right vertex)

```
//Calculate current LOD
float lod = m_MaxLOD;
float distance = length(vertex[0].Position - m_MatrixViewInverse[3].xyz);
float lodScale = 1.0f - (distance/m_MaxViewDistance);

//if there is no wind, then the player doesn't notice the difference between the lod's so
we remove all of them
if (windDirection.x <= 0.1f && windDirection.y <= 0.1f) lodScale = 0.0f;

if (lodScale < 0.0f) lodScale = 0.0f;

lod = (int)(m_MaxLOD * lodScale);
if (lod < 1) lod = 1;
```

Here we perform some calculations for the LOD, the distance and the wind strength is taken in calculation. If there is next to no wind influencing the blade we just use a single quad (as there is pretty much no difference.)

We still cap the minimum at 1 so there is no appearing grass. This choice was made because there is no fog or anything in my scene to avoid grass popping out.

```
//Every level of detail adds an extra quad to the blade
for (int i = 0; i < lod; ++i)
{
//Calculate top position (follow terrain)
startTopPoint = startPoint + (float3(grassHeight/lod, grassHeight/lod, grassHeight/lod) *
vertex[0].Normal);
endTopPoint = endPoint + (float3(grassHeight/lod, grassHeight/lod, grassHeight/lod) *
vertex[0].Normal);

//Wind offset (add dotproduct to check if we are still under a condition)
float direction = vertex[0].Position.xz - m_WindPosition.xz;

float windCoeff = ((i + 1)/lod) * (m_MaxLOD / lod);      //Calculates windstrenght
depending on how close the quad is to the ground
windCoeff *= (1.0f - distance/m_MaxWindDistance) * noiseSample.r; //add a bit of
randomness

if (windCoeff < 0.0f) windCoeff = 0.0f;

startTopPoint.xz += windDirection.xz * windCoeff;
endTopPoint.xz += windDirection.xz * windCoeff;

float2 randomOffset = float2(noiseSample.r, noiseSample.g) * (0.01f * sin(m_Timer));
startTopPoint.xz += float2(noiseSample.r, noiseSample.g) * (0.01f * sin(m_Timer));
endTopPoint.xz += float2(noiseSample.r, noiseSample.g) * (0.01f * sin(m_Timer));

randomOffset = float2(noiseSample.b, noiseSample.r) * (0.05f * sin(0.2 * m_Timer));
startTopPoint.xz += float2(noiseSample.b, noiseSample.r) * (0.05f * sin(0.2 *
m_Timer));
endTopPoint.xz += float2(noiseSample.b, noiseSample.r) * (0.05f * sin(0.2 *
m_Timer));

//Calculate the difference in edge lenght and adjust the Y accordingly & move the
vectices back over the edge vector
//we could just go with a simple formula, but it still generates plane stretching.
//like this (use in paper): startTopPoint.y -= length(windDirection) * windCoeff * 0.5;
float3 edgeDirection = startTopPoint - startPoint;
float edgeLength = length(edgeDirection);
float deltaY = edgeLength - (grassHeight/lod);
edgeDirection = normalize(edgeDirection); //normalize here as we use it to calculate the
lenght above

startTopPoint -= edgeDirection * deltaY;
endTopPoint -= edgeDirection * deltaY;

//Calculate normal (always point in the direction of the vertex)
normal = vertex[0].Normal;

//Restart the strip so we can add another (independent) triangle! (otherwise these will all
be attached!
triStream.RestartStrip();
```

```
//Create the vertices (take a look at texcoords: paper)
float bottomY = 1.0f - (i * (1/lod));

CreateVertex(triStream, startPoint, normal, float2(0.0f, bottomY), vertex[0].Position);
CreateVertex(triStream, endPoint, normal, float2(1.0f, bottomY), vertex[0].Position);
CreateVertex(triStream, startTopPoint, normal, float2(0.0f, bottomY - (1/lod)),
vertex[0].Position);
CreateVertex(triStream, endTopPoint, normal, float2(1.0f, bottomY - (1/lod)),
vertex[0].Position);

startPoint = startTopPoint;
endPoint = endTopPoint;
}
```

This is the final part of the geometry shader, as you can see it's quite long. All we basically do here is generate x number of quads (depending on the lod) on top of eachother.

It features 2 random sines influencing the wind, even when there is no actual wind coming from the scene. This is to avoid the grass from looking 100% static when the player isn't moving.

The wind itself is processed by moving the top vertices back and forth depending on the winddirection. Each vertex starts where the previous ended. It is also important to note that the top quads take more influence from the wind as they are higher above the ground.

The grass blades also remain the exact same size instead of skewing. I could have used a simpler formula to increase speed, but as accuracy is the main goal of this shader I avoided that.

Most other things are very self explanatory trough the comment lines accompanying them.

Sources

Realistic real time grass rendering thesis by Edward Lee
http://illogictree.com/upload/site/LeeRealtimeGrassThesis.pdf

Rendering countless blades of waving grass, NVIDIA
http://http.developer.nvidia.com/GPUGems/gpugems_ch07.html

Prototype grass
http://upvoid.com/devblog/2013/02/prototype-grass/