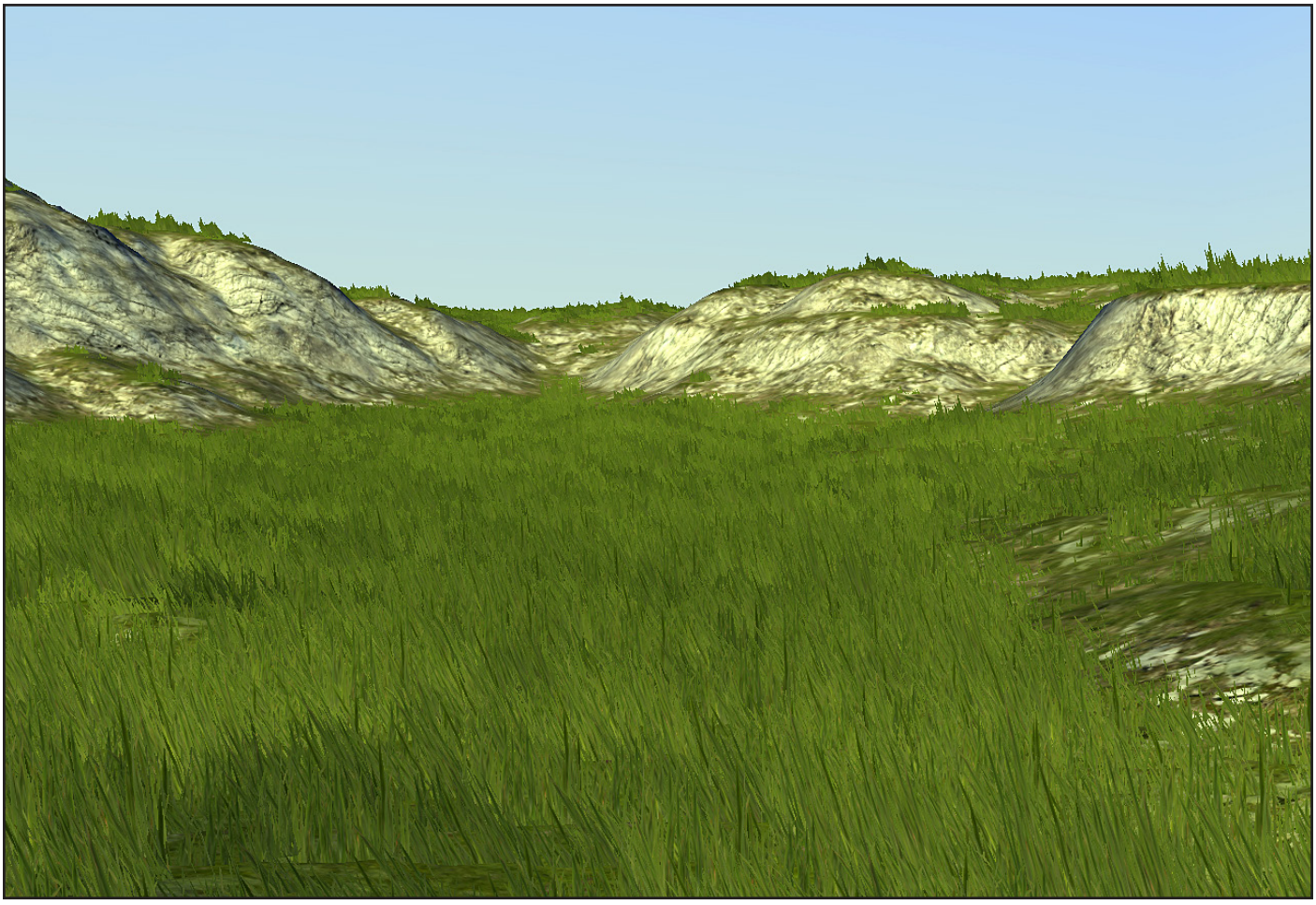
Geometry Shader:
Terrain with grass



Geometry shader

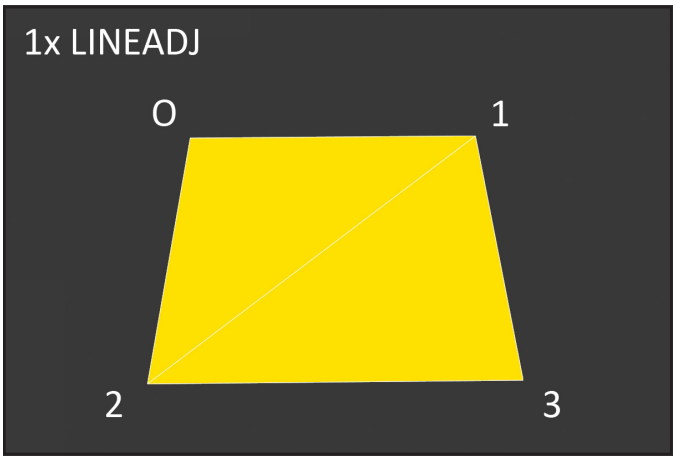
This Directx10 Geometry Shader makes it possible to render terrain with grass in a single drawcall and shader pass. This shader processes the geometry for the terrain itself and creates additional geometry for the grass using a billboarding technique. The shader will only create the geometry if it is necessary.

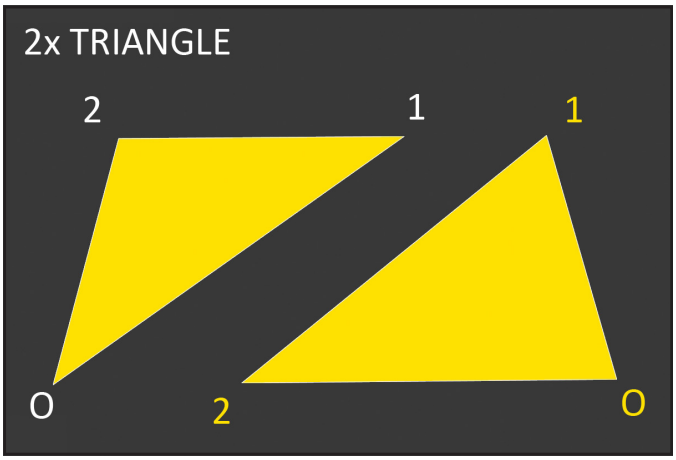
Input

[maxvertexcount(8)]
void GS(lineadj VS_OUT gIn[4], uint primID : SV_PrimitiveID, inout TriangleStream<GS_OUT> triStream)

The input for the geometry shader is a collection of 4 vertices (lineadj). Even though we are not drawing lines, the lineadj structure can still be used because the geometry shader allows us the interpret the data differently. The position, normal vector and texture coordinates for each vertex are passed from the application to the shader.

Because most 3D applications arrange terrain in a grid like structure, it makes sense to process the data per quad instead of per triangle. This also reduces the size of the index buffer (by a third) and the amount of times the geometry shader is triggered (by half). It will also give us more control when placing our billboards later on.



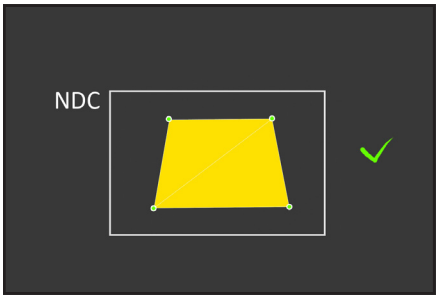


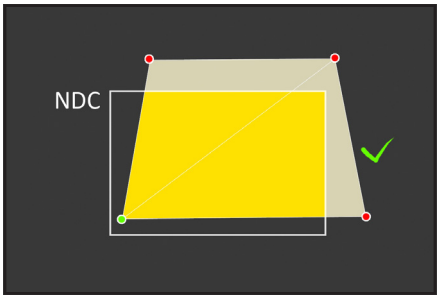
Because my terrain is modifiable, it was initially my goal to calculate the normal vector for each vertex in the geometry shader. This would decrease the size of the dynamic vertex buffer, as it would only need to contain the position and texture coordinates of each vertex. It would also take some workload of the CPU. The downside of this approach was that the normal vectors for each vertex would be calculated every frame on the GPU.

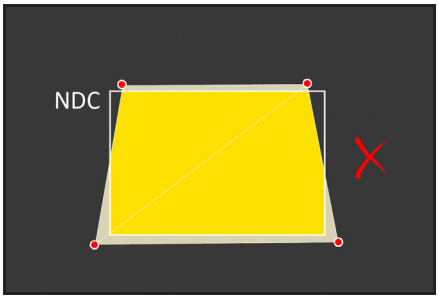
This approach was also unfeasible due to the limitations of the input topology. I was able to calculate the normal vector for each face, but because the geometry shader outputs each quad separately this still resulted in an unsmooth surface. Eventually I opted for calculating the normal vectors of the modified vertices on the CPU.

Culling

In the first step of the geometry shader we check if the input quad is in the viewing frustum. We do this using a simple algorithm. We transform the 4 vertices from local space coordinates to normalized device coordinates (NDC) and check if any of them are in the NDC range (-1 , 1). If so the Geometry shader continues with processing the data. Otherwise no geometry is created.
This algorithm is not completely full proof, some geometry gets culled if the camera is very close to the terrain. This can be avoided by placing the camera at a minimum distance from the terrain or by using a more complex algorithm which checks if points are at opposite sides of the NDC range.
This kind of algorithm would be a little redundant for this application because every quad is more or less the same size and we can control how close the camera gets to the terrain.







Terrain

The next step is rather straight forward. We use the 4 input vertices to create a triangle strip for our terrain quad. We transform the positions from local space to both homogenous clip space (posH) and world space (posW) which we will use later on in the pixel shader to calculate our specular lighting. The normal vectors are also transformed to world space (without translation) and the texture coordinates are just passed through. We also set the Boolean “grass” to false for all vertices. This Boolean will be used by the pixel shader to determine which texture array to sample from. We set it to false, because we want to use the terrain textures instead of the grass textures. After we have appended all 4 vertices for our terrain quad in the correct order we need to restart the triangle strip and create our billboard for the grass.

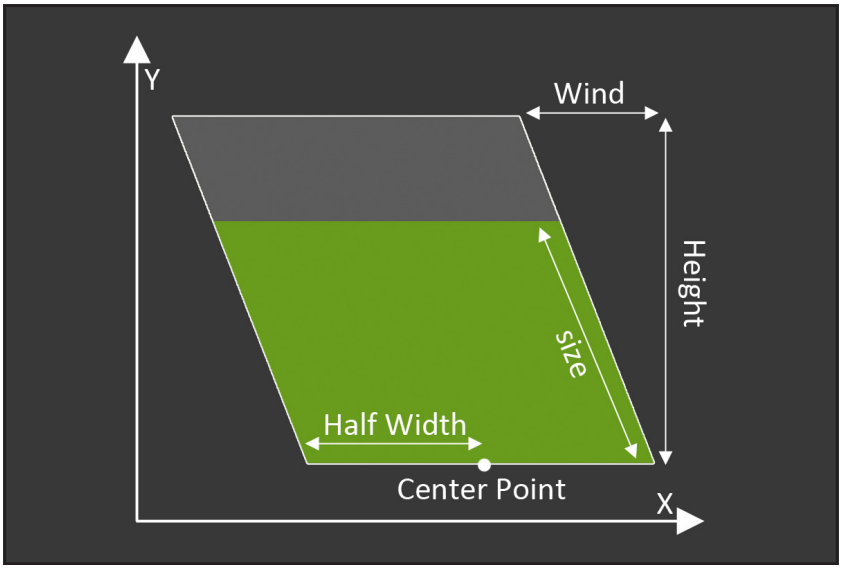
Billboarding

For our terrain we use billboarding for a variety of reasons. The most important one is efficiency. A terrain mesh is quite complex; even a medium sized 256 x 256 terrain already contains 65025 quads or 130050 triangles. If we were to use a shell and fins technique, as with fur, we would never get a decent frame rate, because the geometry of our terrain would need to be copied multiple times. Billboarding allows you to create realistic looking grass with just a single plane per quad. Before resorting to the billboarding technique I also tried to create multiple static planes on top of the terrain quad. I needed a lot of planes on each quad to make it look even slightly convincing. Ideally you would use billboarding in combination with instanced meshes and fade between the two.

Before we let our geometry shader create the billboard we do two additional checks. The first check compares the distance to the billboard with the maximum drawing distance. The second check compares the slope of the terrain on which we will place the billboard with the maximum allowed slope for grass to grow on. If either of these checks fail, no grass geometry is created.

Next up we create 2 pseudo random numbers using the primitiveID which is generated by the input assembler stage. We will use these 2 random numbers to define the characteristics of the grass billboard, such as size, texture, texture coordinates, etc... This is necessary to make the grass look more convincing. If everything was the same size and texture, the use of billboarding would become very obvious. We also calculate the values needed for fading the grass in and out. Grass that is close to the maximum draw distance is scaled down.

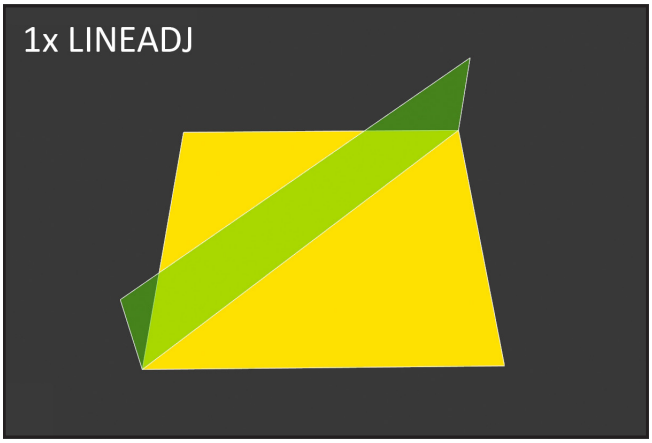
Once we have all these values we can construct the 4 vertices needed for our billboard plane. We build the billboard on the worlds XY plane. The top 2 vertices x coordinates are offset using the gWind variable supplied by the application. This creates the illusion of the grass swaying in the wind. The height is defined using a combination of the previously calculated values. We also need to set the texture coordinates for the billboard plane. The texture is shifted downward on surfaces with a steep slope. On some billboards the U texture coordinates are also switched, which will result in a mirrored texture. This helps us create more variation between the individual billboards.

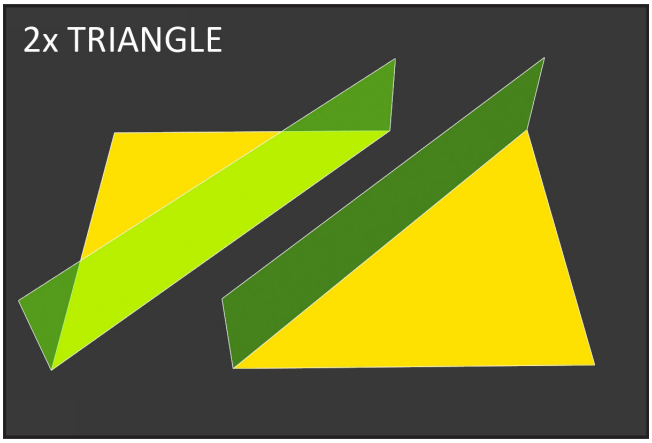


The billboard uses the same normal vector as the surface below. This ensures that the billboard is lit in accordance with the terrain. Because the billboard plane is always facing the camera, the lighting would look unrealistic if we were to use the normal vector of the plane itself. With static geometry a combination of both normal vectors would be the best solution.

Matrix

Every billboard needs a unique world view projection matrix. This matrix changes every time we move the camera or the billboard, so we need to calculate it every frame. The first step for creating our matrix is calculating the position for our billboard. As mentioned before, processing the vertexes per quad instead of per triangle makes this easier. We can find the center of our terrain quad by calculating the midpoint of one of the diagonals and use that as our billboard position. If we were using triangles we would have to make sure that we didn't create any overlapping planes for our billboards, which would require additional calculations. (IMAGE) The rest of the method for constructing the matrix is identical to the one described in Frank D. Luna's book "Introduction to 3D game programming with Directx10", chapter 10, page 287, 291.





Once we have our world view projection matrix, all that is left to do is append our billboard vertices to the triangle stream. We multiply the 4 vertices we have created earlier with the matrix to get our posH. As we won't be calculating specular lighting four our billboards we can just output 0 for posW. We also output the texture coordinates we have defined earlier on. All this data is then past on to the pixel shader, which will calculate diffuse and specular lighting and apply the right textures.