# Grass Renderer

Kedadry Yannis

February 12, 2024

**Abstract**

In this report, we present a grass renderer inspired by the Ghost of Tsushima's GDC conference. The idea is to be able to render a lot of animated 3d grass blades in a very efficient manner using both compute and geometry shaders.

## Introduction

Almost every 3D game needs to render grass at some point. A lot of effort is then made by video game companies in order to make it as realistic as possible while using as less resources as possible to render it. In the early days of computer graphics, impostors and billboards were the state of the art for real-time rendering of grass blades. Nowadays, with the computing power of modern hardware, real 3D geometry can be used to render millions of animated grass blades without having a major impact on performances. In this project, we have tried to replicate some of the techniques used by major video game companies to render grass.

## 1 Method

An important part of the work was put into the creation of the overall rendering pipeline. From this pipeline, we could then use techniques to create both the appearance of the blade and its animation.

### 1.1 Architecture

The first thing we did was cut the world into tiles. Each tile will be filled with multiple blades of grass. A blade of grass is a collection of properties (height, color, width, ...). These properties are built inside a compute shader to increase parallelism. In this compute shader, we fill buffers of fixed size. From the output of the compute shader, we want to create triangles that will look like grass. To do so, we send the buffers filled in the compute shader to the graphics pipeline shaders. In the vertex shader, we use the vertexID to create a vertexData from the buffers. Each of the data represents a grass blade but for now, it is only

a position. To make the blades visible, we use a geometry shader which will generate triangles from the vertexData. For now, all the triangles have the same LOD (a unique triangle see figure 1).

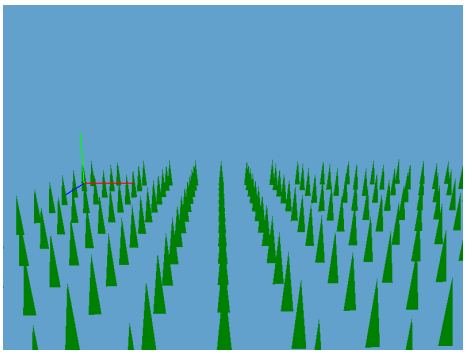


Figure 1: First output of the geometry shader

## 1.2 Blades

From this pipeline architecture, we can now create more interesting blades. First of all, instead of having a regular grass pattern, we want to add some noise to the grass blades. We then have to redefine our way of representing grass blades. From now on, all our grass blades will be located within the same tile. This tile is divided into a regular grid. For every intersection in that grid, we create a clump point (think of it as the center of a grass pattern or as a voronoï vertex). We use a hash function to randomly jitter each of these clump points (we want a hash function and not some random noise so we don't have to store their values but just recalculate them every time in the compute shader). For each of the grass blades we want to draw, we assign them a random position in 2D (see figure 2). We then find the 9 closest intersections in the grid (or less if it is at

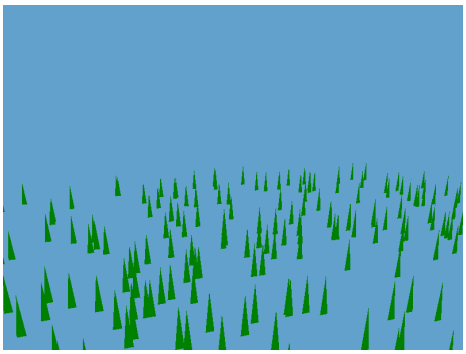


Figure 2: Adding randomness to the positions

a corner, the top, the bottom, the left, or the right of the grid). Using now the jittered position of the previous 9 intersections we find the closest voronoï vertex to the grass blade. For every voronoï cell, we have defined custom information (blade's height, width, orientation, etc...) that we then use to create the vertex data for that blade and send it to the vertex shader. In figure 3, we have assigned a random color for each of the clumps (or voronoi areas). We can see in figure 3 that the z-buffer was not used resulting in a weird order for the triangles so we also fixed that at that stage. Another thing to add is rotation around the center

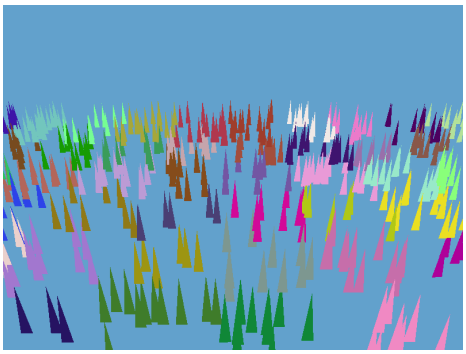


Figure 3: Grass clumps

of the blade (see figure 4). Once again, we do that in the compute shader using a deterministic function given the blade ID, the tile ID, and the clump ID.

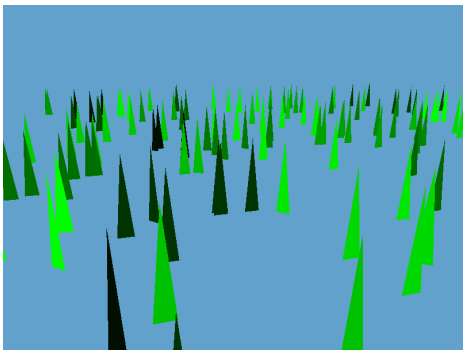


Figure 4: Pseudo-random rotation

An important way to reduce the rendering cost is to use different levels of detail (LOD). In order to see these levels of detail, we added more tiles to render instead of a single one. The first idea is to use the same compute shader for each tile and use the tile IDs in the random generations so they don't all look the same. In figure 5 we have 4 tiles, and the first one is displayed in red. Then, we create two kinds of LODs for the tiles. Some tiles have high LOD and others have low. Grass blades of high LOD are made of 15 vertices and the ones of low

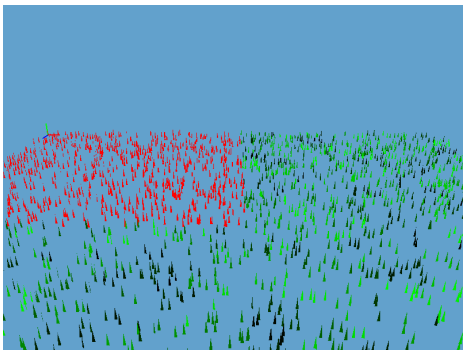


Figure 5: Multiple tiles

LOD are made of 7 vertices (see figure 6) Finally, we update the LOD value of

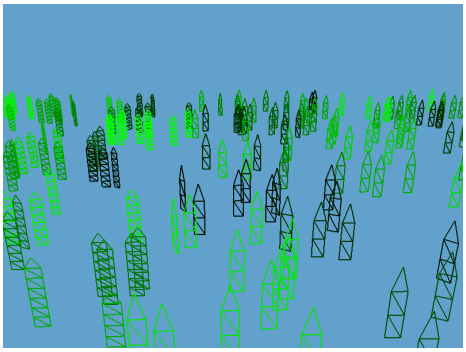
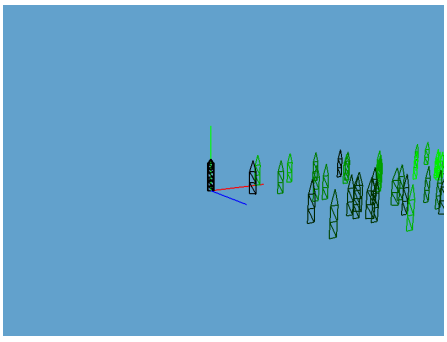
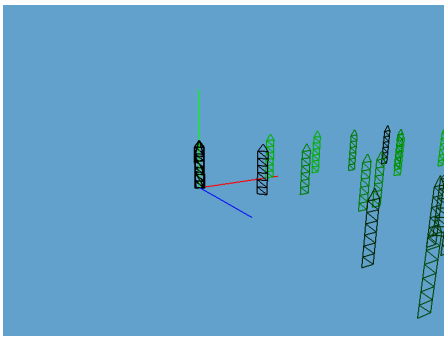


Figure 6: Different LODs

the tile with respect to its distance to the camera's position (see figure 7).



(a) Tile far from the camera



(b) Tile near the camera

Figure 7: Tile LODs

To create the bending effect of the grass blade, we use cubic bezier curves. The curves are made of 3 control points, one at the origin of the blade. One at the tip of the blade which will control the tilt of the blade. One is called the "midpoint" which controls the bending of the curve (see figure 8).

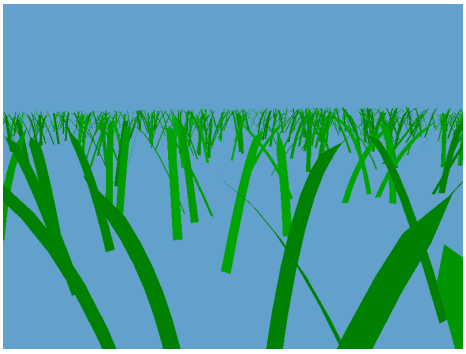


Figure 8: Bent blade

One idea to make the grass look denser without adding any information is to make the blades rotate as much as they can to avoid having their normals perpendicular to the view direction. Indeed, if the normals are orthogonal, it means that the blade will appear super slim and the screen will feel emptier. In their GDC talk, developers of Ghost of Tsushima explained that they have made the blade "fold" toward the camera. Our idea is a bit simpler as we only slightly tilt the rotation of the blade around the y-axis to make it stay in front of the camera more often (see figure 9). We can also manually adjust the effect

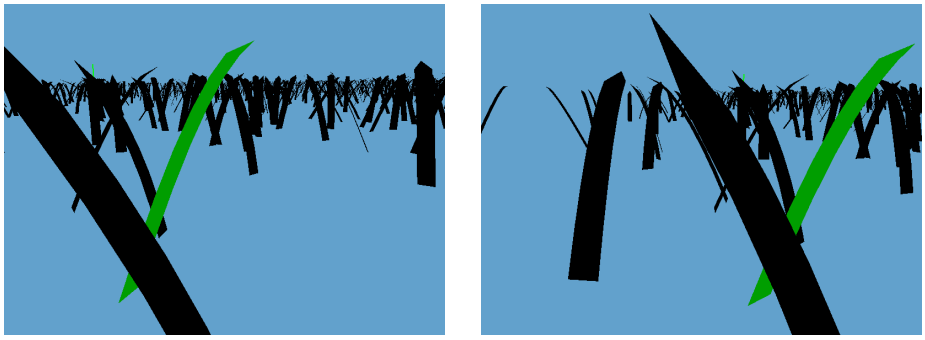


Figure 9: Blade auto rotation

in the geometry shader. However, if the rotation step is too big, the movement will be noticeable and the effect will seem more weird than helpful.

To make the blade look more like real grass, we also tweaked their normals. In the GDC talk, Ghost of Tshushima's developers used an artist-made normal map. Because we are not artists, we have decided to try coding a similar effect.

We start by derivating the bezier curves to get the tangent to the blade at each vertex. By crossing with the tangent on the width of the blade we get the surface normal at each vertex. To give the blades a rounded aspect, we then rotate the normals; to the left for the vertices on the left side of the blade and to the right for the vertices on the right side (see figure **??**).

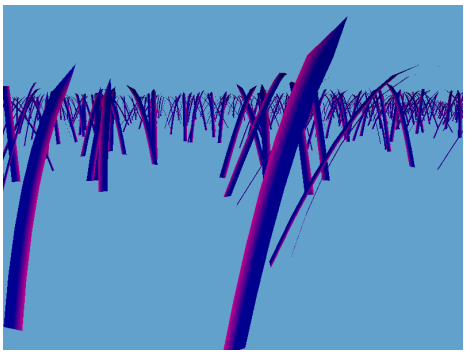


Figure 10: Rounded normals

The blades still look a bit boring. To make things more realistic, we will change how the blade colors are generated. For the blade color, we use a gradient to make colors vary from a dark green color at the bottom to a yellowish color at the tip (see figure 11). The overall color of the blade is also generated using the clump ID.

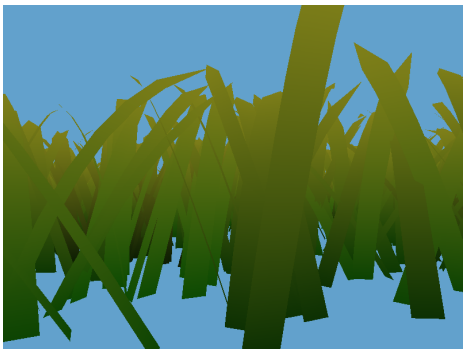


Figure 11: Gradient colors

## 1.3 Animation

Another important part of the renderer is the implementation of a wind system. The wind field used in Ghost of Tsushima was the object of another complete GDC talk. Because we had a limited time, we have decided to go for a simpler solution. We first wanted to create a wind vector field using Perlin noise to

add subtle movement to the grass blades. However, using gradient Perlin noise resulted in a really grid-like field. We can see that in the figure 12 where the output of the noise is used as a grayscale factor. To fix this issue, we used a

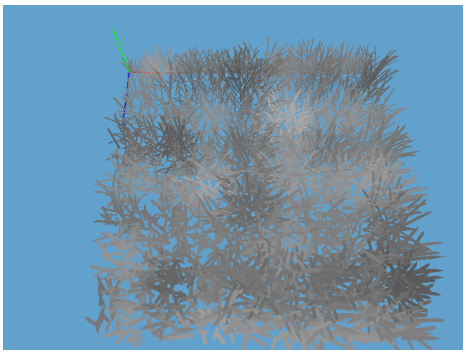


Figure 12: Gradient Perlin noise

more complex Perlin noise called the simplex Perlin noise (see figure 13). The wind is moved using a direction vector and a strength represented as a float. Using this noise we update the control points of the bezier curve to animate the blade. The points are updated using a sine function with a phase and frequency depending on the height of the control points over the height of the blade. This gives the grass some slight changes and gives all the blades a subtle unique aspect. To animate the grass as a whole we use the wind direction to move the

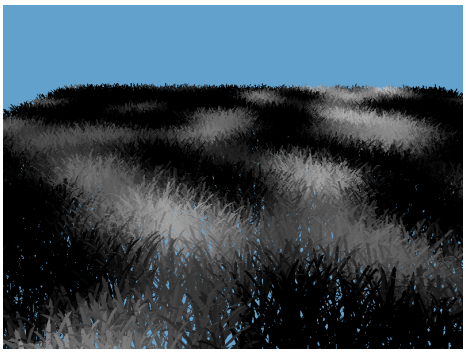


Figure 13: Simplex Perlin noise

vertices of the blade (see figure 14).

## 1.4 Optimization

As visible in some of the previous figures, a lot of blades were stacked at the origin. After a lot of debugging, it appeared that the std430 packing layout rules is doing compiler optimizations and is aligning the stack to $4N$ bytes when using

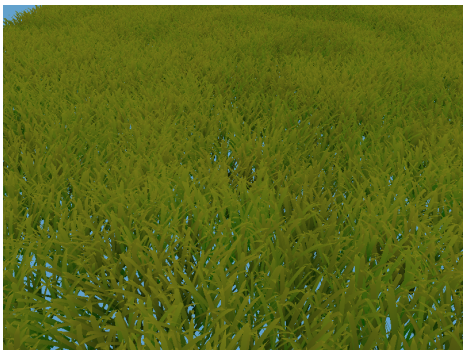


Figure 14: Global wind

elements made of 3 values. The issue was then due to the fact that we were using $vec3$ for both the colors and the positions. This created a shift of one value for each element. For example, with 4 blades of grass set at position $(id, 0, 0)$, I would get:

- position blade 0 = $(0, 0, 0)$,
- position blade 1 = $(0, 1, 0)$,
- position blade 2 = $(0, 0, 2)$,
- position blade 3 = $(0, 0, 0)$

instead of

- position blade 0 = $(0, 0, 0)$,
- position blade 1 = $(1, 0, 0)$,
- position blade 2 = $(2, 0, 0)$,
- position blade 3 = $(3, 0, 0)$

resulting in a lot of blades being at position $(0, 0, 0)$ and with a black color $(0, 0, 0)$.

At first, because of the lot of tiles that were being rendered, we were averaging the 5FPS using a laptop without a dedicated GPU when rendering $20x20$ tiles of 4096 blades each. A basic improvement is to dynamically adjust the number of blades per tile depending on the distance to the camera. Because our grass positions are pseudo-random, the effect doesn't seem weird because the blades are not moving but just disappearing. At first, we were also always sending the grass tiles info to both the compute shader and the graphics pipeline. A second optimization was to only call the shaders for tiles that are within a sphere of a certain radius around the camera. Another optimization is to apply basic frustum culling on the CPU side to avoid sending data from

tiles that are completely outside the camera frustum. Another thing that made the loading time faster was a rethinking of the compute buffers. Indeed, until this point, every tile had its own Vertex Array Object which was really slowing down the initialization. Changing this behavior by having a single instance of the vao shared for every tile drastically improves loading time. One last thing for optimization was the implementation of a parallel pipeline. The idea was to fill computer buffers of the next tile to be rendered while the previous one enters the vertex shader phase. However, because we were using OpenGL and not a more modern and parallelized pipeline (like the Vulkan queues), we had to find a turnaround. The solution was to make buffers $n$ times bigger. Then, we could loop over $n$ tile batches where we filed the same buffers but starting from different indices. By doing so, we can have a single memory barrier after the compute shader phase. However, because all the tiles don't have the same number of blades, we could not have a single call to $drawArrays$. Once again, this would have been a lot easier using the queue calls of the Vulkan's pipeline. With all these optimizations, we went from 5 to around 15 FPS on average to render the same amount of blades using a laptop without a dedicated GPU.

Another issue occurred when trying to implement a basic Blin-Phong Model in the fragment shader. We've realized that the frame rate was reduced by at least half, making the scene really bad. One idea to correct that is to use G-buffers and multiple-pass rendering. However, when trying to create the different textures at the end of the geometry pass (one for the position, one for the color, one for the normal, ...), we realized that outputting multiple textures was as slow as computing the lighting in the fragment shader (see figure 15).

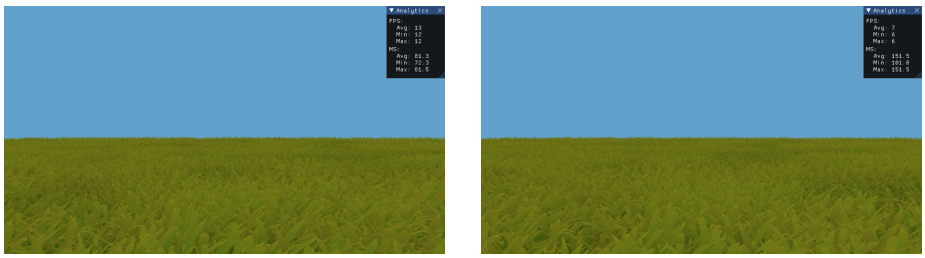


Figure 15: FPS comparison between single and multiple texture output

To fix this issue, we had three different ideas:

- creates a single texture twice as tall and twice as wide so we can store all the information in different parts of the texture
- creates a texture buffer instead of the 3 textures
- uses bit manipulation to fit the necessary information in a single vec4

In the end, we didn't get the time to try any of these solutions.

## 2 Results and conclusion

In the end, we managed to have a real-time grass renderer that produced decent results. However, a lot of improvement can be made.

- To really make the computation faster and avoid too much communication between CPU and GPU, we can add more parallelization and use the Vulkan API for example. Indeed, the Vulkan queue system would automatically handle the parallelized call on the compute shader and the graphics pipeline.
- When trying to implement the Blin-Phong model, we used a point light as our sun. A better approach would be to use a directional light instead.
- Speaking of lighting, we still need to fix the cost issue of the lighting phase by implementing one of the mentioned ideas.
- To make the blades more realistic, we can also use anisotropic brdfs. In real life, grass blades often have small "veins" that could be represented using anisotropic brdfs.
- Finally, we realized afterward that using the geometry shader was suboptimal. Indeed, it would have been better to build the blades in the tesselation shaders. For example, we could then use different inner and outer values in the tesselation control shader to automatically have a way to transition between a high LOD blade and a low LOD blade.

Overall, this was a really fun project to build even if there is still a lot of possible improvement to be done.