# Real-time shader-based rendering of grass

Kin Liu
Manuel Reinfurt

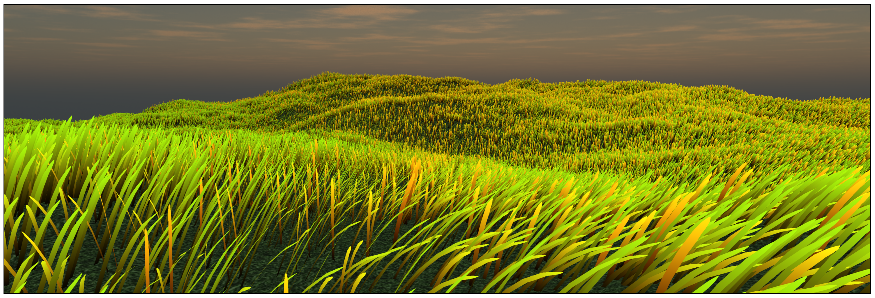

**Figure 1:** *End result*

## Abstract

The grass is the best.

## 1 Introduction

Grass has always been an important factor when it comes to simulate realistic nature scenes. However, due to the sheer amount of individual grass blades and the need to animate them, it is very difficult to implement in real time applications. This paper tries to give an overview of the several techniques that are used in state of the art applications, as well as discussing a geometry-shader based approach in detail.

## 2 Related Work

There are different techniques on how to efficiently and realistically render grass in real time applications. In games, grass is usually simulated by rendering billboards that are laid our in a certain pattern. Animation can be done per vertex, per grass blade or even per cluster. A famous method has been described in GPU Gems 1. While this is a very simple approach that has been used in games for a long time since processing and graphics power is needed elsewhere, it does not yield stunning visuals.

Since DirectX 10, GPUs have been extended with a new shader pipeline - the geometry shader. Using the geometry shader, it is possible to generate grass blades directly on the GPU. This eliminates the CPU as a bottleneck and provides enough power for much more detailed simulations. This paper is based on this approach and combines the work of Eddie Lee (SRC) and ?? (SRC).

The game "Flower", which was released for PlayStation 3, uses this method to render all of its grass. This shows that the approach can be used in real time applications with much success. While the scenes in "Flower" were not really complex and mostly consisted of grass only, the hardware is already very old for todays standards

**Figure 2:** *Game "Flower" for PlayStation 3*

- leaving enough power for more complex scenes which include grass simulation.

### 2.1 The Method

The basic idea is to layout the grass field on the CPU by specifying the root points. This way, the CPU only has to send the position of each single root to the GPU - all other work is done exclusively on the GPU. In order to organize the root points, grass patches are created. Each patch has a certain number of root points and can be controlled individually. The grass patches together then make up the grass grid - which is the grass field.

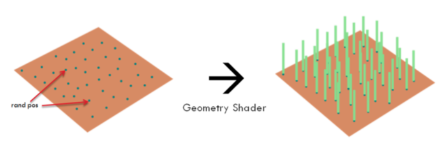


**Figure 3:** *Grass grid visualized*

The grass grid is generated by laying down root points as can be seen in Figure ?. First, a grid of patches is created and each patch generates its own root points. Using a density map Figure ? that is equal the size of the terrain height map, local density spots can be adjusted to control the distribution of the grass. All root points are pushed into a single vertex buffer object and can be, in theory, rendered with a single draw call. However, the patches are used to achieve several things:

**Culling**

A single patch can be visualized by a bounding box, which can then be easily checked against the view frusrum to cull patches outside of it. It is important to take the grass animation into account when creating the bounding box - otherwise grass could be culled when moving into the frustum.

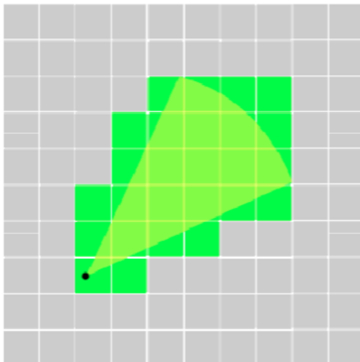

**Figure 4:** *Culling of patches outside the frustum*

**Level of detail**
The distance from the camera to the mid point of a patch can be calculated and used to set the level of detail. This saves a few instructions as level of detail does not have to be calculated within the geometry shader, thus making it possible to use geometry shaders that are pre-compiled with a static vertex count, which will improve performance.

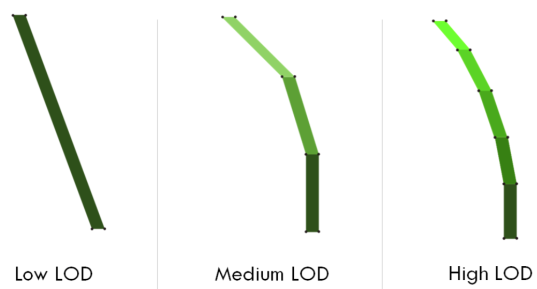


**Figure 5:** *Vertices for level of detail*

**Further individualization**
It's possible to pass options into the shaders for rendering a patch, which can be used for wind animation, grass height or any other property of the grass.

While rendering the patches individually adds overhead compared to doing a single draw call for the vertex buffer object, the performance improvements using the methods above, as well as further individiualization of the scene, more than make up for it.

## 2.2 Terrain

The terrain is generated out of a height map, which was generated by Perlin Noise. We can map the grass root position to our terrain grid to find the matching Y position. It is important to take the density of the grass into account when calculating lighting of the terrain. Using the density map, we can darken areas where grass is very dense - easily simulating ambient occlusion.

[INSERT picture of terrain with grass on it] (TODO KIN)

## 2.3 Generating the blade

The geometry shader needs to be able to generate grass blades with a different level of detail. Instead of writing separate geometry shaders, we decided to write one geometry shader that prodecurally generates the blade. When generating the blade, we want to concentrate the vertices on the top half of the blade to have a smooth animation later on. It is also needed to calculate the UV-coordinate, normals and rotation, as well as vertex displacement for wind, which is described in chapter 4.

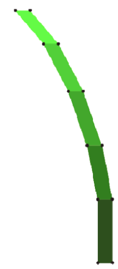

**Figure 6:** *Vertices of the grass blade*

A visual representation of the algorithm can be seen in Figure ? (TODO MANUEL)

## 2.4 Pipeline

To summarize, this is a list of important components in our project and their respective job.

**CPU**
The CPU will generate the root points out of the density map and create a single vertex buffer, that contains all roots. It will also slice the full grass grid into smaller patches, which can be controlled through constant buffers on the GPU.

Since we also have static terrain, the root points will already contain the correct displaced Y position. Additional improvements like culling will also be done on the CPU.

**Vertex shader**
The vertex shader is, in essence, a pass-through shader.

**Geometry shader**
All the hard work is done in the geometry shader. Since the geometry shader gets a single point as an input, it's job is to create a grass blade with a specified number of vertices. The geometry shader will also take care of calculating normals, level of detail, and animation.

**Pixel shader**
To calculate the color, we use the basic Phong BDRF, combined with a texture and a randomized tint.

## 2.5 Flickering

A very prevalent problem while implementing this approach was flickering. Due to the high amount of very thin grass blades, blades were fighting for pixels and a lot of aliasing and flickering could be seen. In order to minimize this effect, several techniques were used:

**Grass blade proportions**
Since grass blades far in the distant can not really be distinguished from each other due to having only a few pixels to be displayed, it is possible to increase the width of the grass

blades. This way, less blades will be fighting for the pixel - having a clear winner in terms of which color to display, leaves less flickering.

**Density Control**
: Since the goal is to minimize the amount of blades fighting for the pixels, the grass density (number of root points in a patch) can be adjusted depending on the distance to the camera. Combined with the grass blade proportions, this leaves very few grass blades that will fill the space in the distance. However, it should be noted that this approach can lead to grass blades ”popping” in and out of the user’s view.

**Downsampling**
: The simplest approach is to increase the render target resolution. Rendering the scene at 4k or 8k resolutions and then downsampling it to Full-HD leaves a stunningly clear and flicker-free image behind. While being simple to implement and having very good results, it has the highest performance cost.

## 3 Randomization

The scene looks very artificial if the grass is laid out with exact distances, when each grass blade has the same tint, height or width. However, it is fairly difficult to generate random numbers on the GPU, which is why we have to use certain tricks to have fairly random grass properties. When laying out the root points, we can generate random positions using the CPU.

Since these root points are given to the geometry shader, and we kow that these positions are random, we can generate a random number between -1 and 1 using the following equation.

$$r = sin(\frac{\pi}{2} * frac(root.x) + \frac{\pi}{2} * frac(root.z)) \tag{1}$$

[INSERT non-randomized and randomized picture]

Using this randomized value, several grass properties can be influenced. Some of those are: width, height, color tint, rotation, wind effect.

## 4 Animation

For animating the grass blades, a simulation of wind is used to achieve a realistic look. In order to maximize the interactivity, the user should be able to influence wind speed and direction.

In general, the wind simulation consists of two components:

**Vertex displacement**
: This components takes care of simulation the effect of wind hitting a single blade, which displaces the mesh.

**Wind field simulation**
: To simulate the effect of wind over a field of grass, a 2D wind field is used.

### 4.1 Vertex displacement

The grass blade vertex displacement depends on two components. The wind that is currently hitting the grass blade affects the displacement - however, if you only use this factor, then the animation will look very shallow and unrealistic. If you watch grass blades reacting to wind in real life, you can see that they don’t smoothly bend in all directions over and over again, instead, they jitter, they oscillate.

The horizontal vertex displacement depending on the wind can easily be implemented by

$$Position.xz = WindVector.xy * WindCoEff \tag{2}$$

where WindVector is the vector of the wind that is hitting the grass blade in the current position. The WindCoEff is a factor that is introduced while generating the grass vertices. This is based upon the idea that the grass blade moves a lot at the top, but does not move at all at the root. Therefore the WindCoEff would be 0 in the root and then increases until it hits 1 at the top.

The vertical vertex displacement is then calculated by

$$Position.y = WindForce * WindCoEff \tag{3}$$

where WindForce is the length of the WindVector. This is needed to prevent the grass blade from stretching. If we displace vertices, we need to make sure that the overall length of the grass blade is still the same afterwards. Otherwise, the grass blade would get longer or shorter depending on the wind.

The oscillation factor can then be computed by

$$LerpCoEff = sin(oscillationStrength*deltaTime+sinSkewCoEff)+1 \tag{4}$$

where the oscillationStrength is just a defined factor, which in our case is 5. The deltaTime is the time in milliseconds that elapsed since the last frame (usual deltaTime used in games). The sinSkewCoEff is derived from the random value we computed in chapter ”Randomization”. It’s used to differentiate the oscillation pattern for each grass blade which makes the scene look much more natural. And then the value we get out of the sin (-1 to 1) is brought between 0 and 1.

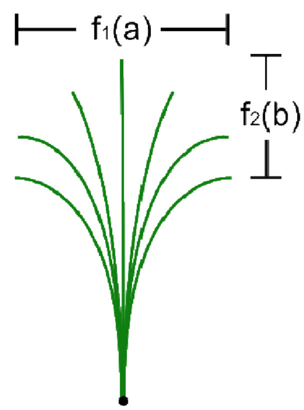


**Figure 7:** *Oscillation range of a grass blade*

This oscillation factor now has to be used to infuence the WindVector that is reponsible for the vertex displacement. This is done using the formula below.

$$WindVector = lerp(WindVector*(1.0-oscillateDelta), WindVector*(1. \tag{5}$$

In essence, this opens up a range depending on the current WindVector and the oscillation constant called oscillateDelta, which in our case is ”0.35”. The final position to take in the range is then defined by our LerpCoEff.

As a sidenote, it can be useful to further randomize the wind by adding a small random direction or force.

### 4.2 Wind field simulation

While the vertex displacement focuses on the impact of wind on a single grass blade indepedently, a system has to be used in order to get all grass blades into the same world - i.e. reacting to the same wind conditions. Since we have a grass field that consists of grass patches, it makes a lot of sense to build a wind field that contains of wind vectors. One wind vector is then assigned to one grass patch and it will be responsible for the wind impact in this grass patch. This wind vector is basically the wind vector that is described in the vertex displacement.

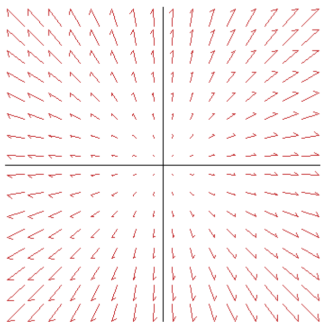

**Figure 8:** *Example wind vector field*

## 5 Benchmarks

We kept randomized values to a minimum to avoid variations in the scene and to have a baseline. All tests were mainly done on the following two systems.

**System 1 (Desktop)**
CPU: Intel Xeon E5-1230v3
RAM: 8GB DDR3-1333
GPU: nVidia GeForce GTX 780

**System 2 (Notebook)**
CPU: Intel Core I7-4980HQ
RAM: 16GB DDR3-1333
GPU: Intel Iris Pro

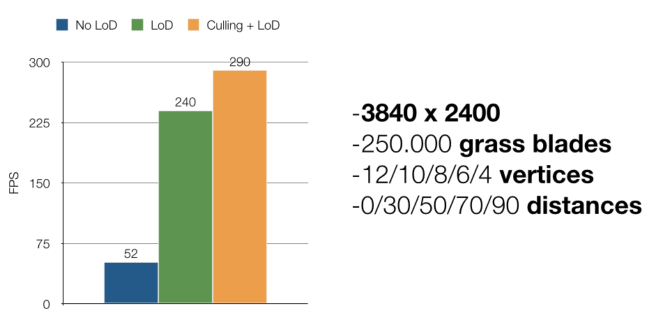


**Figure 9:** *Benchmark on System 1*

Content: Show base FPS without performance improvements, then show level of detail, culling and so on.

## 6 Shortcomings and improvements

As shown in the section ”Benchmarks”, this approach does not need high-end processing power as of todays standards. Using 1 million grass blades, the implementation can be run on integrated graphics cards on Notebooks with 60 FPS.

The main visual problem is flickering. Using native resolution of displays and no anti-aliasing, the thin blades cause a lot of flickering in the distance - even when using the tricks described in the paper to minimize it. However, when rendered on a high PPI screen or by using techniques like Downsampling, flickering vanishes very fast.