Rendering Grass Swaying in the Wind

R. Geleijn (romi.geleijn@hotmail.com)

I. Introduction

This project implements several techniques that create the illusion of wind blowing through a patch of grass. The main focus is on an image-based technique that deforms 2D billboards with grass textures using either trigonometric functions or Perlin Noise. The implementation uses OpenGL version 3.30, with GLAD and GLFW, and C++ version 14 and is run on a laptop with an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz and NVIDIA GeForce GTX 1050 GPU.

Section II gives a short overview of the general techniques that are used to create the effect of wind blowing through grass. Section III focuses on the techniques used in this project specifically to create wind and describes the different components of the project and the theory behind all of this. Followed by this, in section IV, are several screenshots of the project and a few tables that investigate the performance of the approaches that were used. Lastly, section V, summarizes the project and mentions possible improvements that can be made in future projects.

II. Related Work

In general, three different techniques are used to render grass: image-based, geometrical and hybrid techniques [1]. This project focuses on image-based techniques, though an initial set-up of the geometrical approach is also included (a short explanation of this can be found in section III-G1).

The main inspiration for this project is a tutorial on how to create the effect of grass swaying in the wind using Shader Graphs in Unity. None of the steps shown in the tutorial were directly copied, since the actions shown are not easily transferable to OpenGL. However, inspiration was drawn from the concepts described and the techniques used. A further explanation of this technique can be found in section III-H2.

III. Implementation

The following features of the project and the theory behind them will be discussed:

1) Scene Description
2) Project Structure
3) Graphical User Interface
4) Camera
5) Textures
6) Lighting
7) Grass types
8) Wind types

A. Scene Description

The scene consists of two triangles forming a quad symbolizing the ground. This ground is covered in either grass blades or billboards with grass textures, depending on the settings the user is using. Furthermore, a skybox is used to make the scene look more interesting visually and to allow the user to switch between skyboxes that represent day and night. Settings on the billboards can be adjusted in such a way that the illusion of the wind blowing through them will be created.

B. Project Structure

In order to get a general overview of the project, each file is discussed briefly (excluding image files). The project contains a main file with the render loop and a 'primitives' file, where the vertices, indices, colours, normals and UVs for the objects in the scene are defined. Additionally there are files for the SceneObject, Texture, Shader and Camera classes to improve the structure of the project and to abstract away some specific functionalities. There is also a debug file that support debugging OpenGL functions.

Apart from this, the project contains three different shaders.

The general shader is called 'shader' and is used to render the blades and the ground. The vertex 'shader' projects the vertices to the projection space and the fragment 'shader' calculates the light on the objects (for more information read section III-F). No textures are used for the blades or the ground.

The shader for the skybox is called 'skybox' and simply renders the skybox texture on the skybox cube. The skybox is not influences by the lights defined in the scene.

Lastly, the 'billboard' shader, is a little more complicated as the vertices of the billboards are warped to simulate the wind blowing through the grass. How these vertices are warped is explained in more detail in section III-H. The 'billboard' fragment shader calculates the light on the objects similar to the 'shader', except for the fact that a texture for the billboards is also passed that determines the colour of the objects.

C. Graphical User Interface

In order for the user to switch between the different implemented features, the project was extended with a Graphical User Interface (GUI). For this Dear ImGui was used, an Immediate Mode Graphical User interface for C++ with minimal dependencies.

The GUI offers settings for lighting, the skybox, the grass and the wind (see figure 1) that will be explained in the following sections.

D. Camera

The user controlled camera was implemented based on exercise 5.4: [2].

This camera was adjusted in such a way to allow the user to fly, in order to be able to view the grass from above as well.

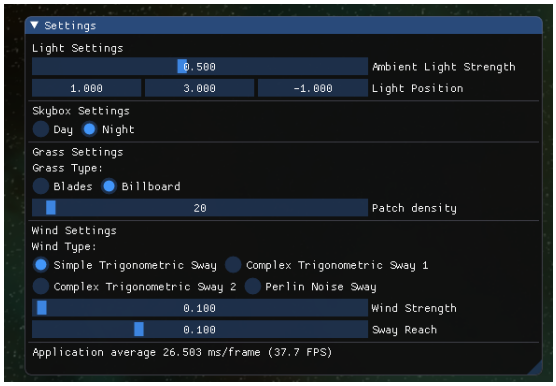

Fig. 1. Screenshot of the GUI inside the project. Can be opened and close by pressing space.

E. Textures

In this project textures are used in two places: the skybox and the billboards. The skybox implementation was inspired by exercise 11 of the Graphics Programming course [3]. Six different images were used, that represent the front, back, left, right, up and down faces of the skybox. These images were mapped to the inside of a cube. The skybox images for day and night were taken from this website and this skybox generator.

The billboard textures were linked at the bottom of the Unity tutorial for swaying grass and can be downloaded here. This texture was mapped to a quad to render multiple grass blades using minimal vertices.

Since the skybox is opaque and the billboards are partially transparent, the skybox is always drawn first. As the transparent billboards intersect each other in such a way (resembling an asterisk shape, see figure 2) that no possible drawing order will result in a correct visual result, pixels that are not visible (or equivalently, that have an Alpha value of below 0.1) are discarded in the 'billboard' fragment shader, using the following line:

```
if(FragColor.a < 0.1) discard;
```

F. Lighting

To increase realism in the grass patch scene, ambient and diffuse light were implemented. Both were implemented using the following tutorial: [4].

Ambient light accounts for the fact that there is always some light in the world, either from a distant light source or the sun or moon [4]. The amount of ambient light that objects receive is determined by the Ambient Light Strength, which can be adjusted using the GUI. When the Ambient Light Strength is increased the fraction that the color of the object is influenced by the color of the light in the scene is increased. This means that the scene becomes lighter when the Ambient Light Strength is increased. Lowering the Ambient Light Strength is an easy way to give the user the impression that it is walking through a meadow at night (which can be further aided by switching to one of the night skyboxes) and the reverse is true for making the user feel like it is day.

Diffuse light "simulates the directional impact a light object has on an object"[4]. In order to calculate the diffuse impact of a light on a fragment, the direction vector between the light source and the fragment position of a pixel of an object needs to be calculated first. After this, the dot product between the normal of the fragment and this direction vector is computed. This value is multiplied by the light's color, resulting in the diffuse component of the light. This means that if an object faces the light source it becomes brighter. The location of the light source can be adjusted in the GUI by changing the coordinates of the light position.

In real life, the intensity of light decreases over distance. In order to include this effect in the project, attenuation was used to decrease the intensity of the ambient and diffuse over distance. Attenuation was implemented according to this tutorial, [5], where the attenuation factor is calculated by dividing 1.0 by a constant term, a linear term multiplied with distance and a quadratic term multiplied with the quadrant of distance. These constants were set to the values recommended in the tutorial (1.0, 0.09 and 0.032 respectively). The attenuation factor is multiplied with the ambient and diffuse component of the light in order to establish the effect of decreasing intensity over distance.

G. Grass Types

For generating grass two types of approaches were taken: rendering blades separately or rendering a collection of grass blades simultaneously using billboards. Both approaches are explained in more detail below.

1) Blades: Rendering blades of grass separately can be useful for applying geometrical techniques for creating wind-effects, which often example higher levels or realism [6].

However, rendering each grass blade separately is a lot more computationally demanding. Even when using the simplest design imaginable for a blade of grass (a single triangle), the FPS already takes a deep dive when hitting 200.000 blades (run on a laptop an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz and NVIDIA GeForce GTX 1050 GPU).

Therefor, the main focus of this project turned to using billboards and wind was only implemented on this grass type (see section III-G2).

When generating blades separately first two triangles, together forming a quad of 10 by 10, is drawn symbolizing the ground. On this ground, separate blades are drawn. The amount of blades that are drawn depends on the grass density, where the grass density represents the amount of blades per triangle. The positions of the blades are chosen using an unbiased function [7], causing the blades to be distributed somewhat uniformly across the triangles. The function, explained by [7], that was used for this is:

$$P = (1 - \sqrt{r_1})A + \sqrt{r_1}(1 - r_2)B + \sqrt{r_1}r_2C$$

Where r_1 and r_2 are random numbers between 0 and 1. A, B and C are the vertices of the triangle and P is a random point somewhere on the triangle. The random numbers are generated using the 'rand()' function from <stdlib.h>.

The blades are also rotated on their X and Y axis randomly, to avoid symmetry and increase realism.

These random positions and rotations are calculated once before entering the render-loop in a function called initPatch().

2) Billboards: In order to reduce the amount of vertices needed to render vegetation, billboarding techniques are often applied [6].

In this project the billboards are place in an asterisk configuration (see figure 2), as recommended by [8] "to ensure good visual quality independent of the current line of sight".

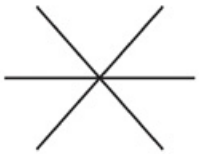

Fig. 2. Asterisk configuration of billboards, viewed from above.

These billboards are placed in the same way as the blades (see section III-G1), on a 10 by 10 patch and given random positions and rotations (in the case of billboards, the random rotation is only applied to the Y-axis).

The texture that is used for the billboards is shown in figure 3.

Fig. 3. Grass texture used on billboards, taken from here.

H. Wind Types

Two approaches for simulating wind were applied, using trigonometric functions, resulting in more symmetric swaying movements, and Perlin Noise. Both are explained below.

1) Trigonometric Functions: Three different trigonometric functions were used in this project to simulate wind. These trigonometric functions distort the top two vertices of the billboards according to specific trigonometric functions, creating the illusion of the blades swaying in the wind, due to the circular motion. The distortion is only applied on the Z-axis and is based on the current position of the vertex, the 'swayReach', the 'windStrength', the time and a random number. The 'swayReach' defines how far the blades sway back and forth and can be adjusted by the user using the GUI. The 'windStrength' can also be altered by the user and determines how quickly the blades sway back and forth by multiplying the 'windStrength' with the time. Causing a high value of 'windStrength' to resemble stronger gusts of wind. In order to ensure that the circular motions aren't too coordinated, a random number is added to the function so that each billboard is in a different phase of the trigonometric function (for example one billboard will swing backwards while the other is swinging forwards, so that it becomes harder to notice that they all have the same pattern of motion).

The 'Simple Trigonometric Function' distort the top two vertices of the billboard using the following function:

```
float z = sin(pos.z + 10
* randomNumber + time) * swayReach;
```

The 'Complex Trigonometric Function 1' and 'Complex Trigonometric Function 2' are more complex, combining several sine and cosine functions and are inspired by example functions from the industry that were used to animate vegetation [9].

2) Perlin Noise: The other approach to simulating wind that was applied in this project is based on Perlin Noise, a type of correlated gradient noise introduced by Ken Perlin [10]. The general idea is that the Perlin Noise function determines the movement of the grass and was inspired by [11]. A precomputed 2D texture was used as the implementation of the Perlin Noise, as this is more computationally efficient than implementing the Perlin Noise directly into the shader [12].

This texture (see figure 4 for an example) is used to determine how the vertices are manipulated. For low values (or the black parts), the vertices are pulled and for high values (or the white parts) the vertices are pushed. In order to determine which vertices has which value inside the texture, the world coordinates of the vertices are mapped to UV coordinates on the texture and the texture is scrolled over time to generate back and forth movement.

The 'windStrength' can be adjusted for this approach as well and functions in the same way as it does for the trigonometric functions. Additionally the 'perlinSampleScale' can be adjusted, which affects the way the world coordinates are mapped to the UV coordinates and allows the user to change this mapping if it leads to undesirable results (for example a sample scale that wraps around the texture and always samples the movement from the same UV coordinate can lead to movement that is too consistent). Lastly, the GUI also allows for two different Perlin textures to be chosen with different patterns of noise.

IV. Results

In this section several screenshots of the project are shown. Additionally, a small comparison in performance between the blades and billboards representation was carried out. These experiments were run on a laptop with an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz and NVIDIA GeForce GTX 1050 GPU.

Table I shows that, as expected, the same amount of grass blades is way more computationally efficient than the same

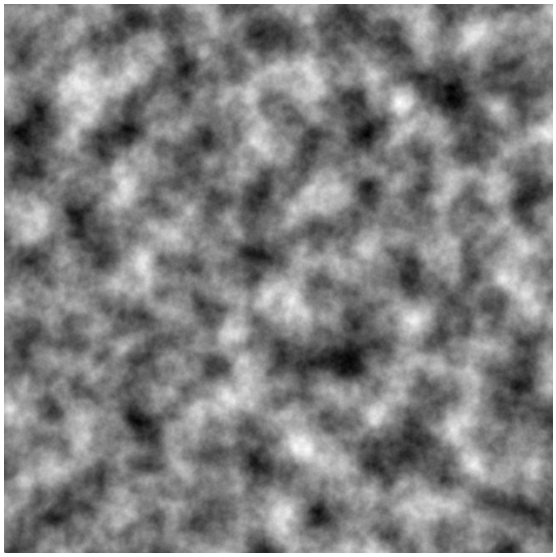

Fig. 4. A texture generated using Perlin Noise, using this online generator.

Density	Blades	Billboards
250	50.0 FPS	23.0 FPS
500	50.0 FPS	11.8 FPS
20.000	2.0 FPS	2.1 FPS

TABLE I
PERFORMANCE COMPARISON BETWEEN BLADES AND BILLBOARDS. NOTE THAT BILLBOARDS ALSO SIMULATE WIND.

amount of billboards. Blades are much more efficient per object (it should be noted that blades are not animated). However, a lot more blades are needed to make a 10 by 10 patch seem covered in grass than billboards.

The blades representation starts being visually passable when the density is upped to about 20.000 (figure 5). However, his is also when performance drops dramatically.

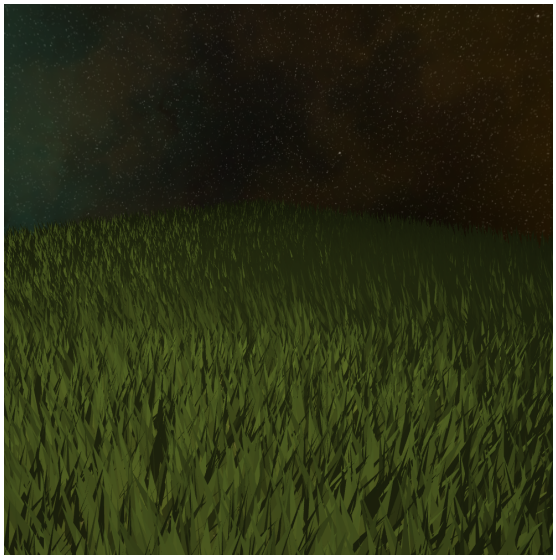

Fig. 5. Screenshot of the blades representation with density set to 20.000 using the night skybox.

The blades representation using a density of 500 (figure 6) has a great FPS rate, but makes the vegetation look very sparse and unrealistic.

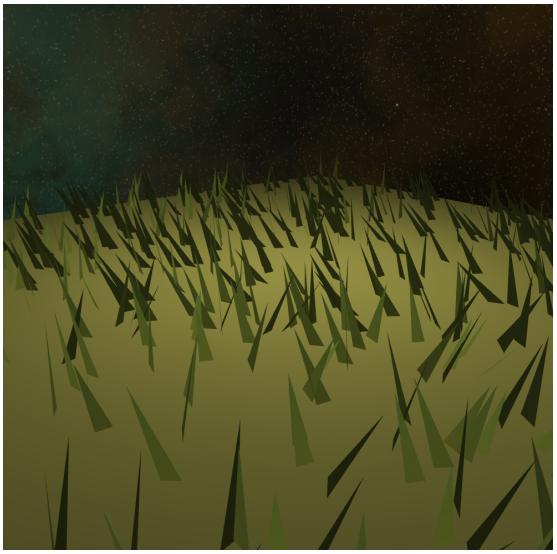

Fig. 6. Screenshot of the blades representation with density set to 500 using the night skybox.

The same density (500) using the billboard representation, however, has great coverage (figure 7), however, and is a lot less computationally demanding than the blades representation with a density of 20.000.

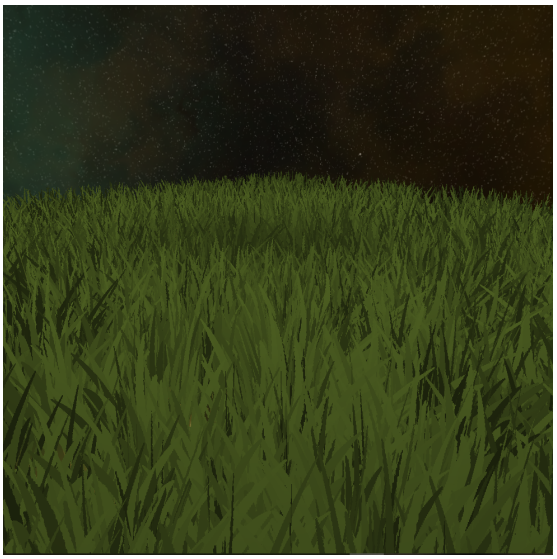

Fig. 7. Screenshot of the billboard representation with density set to 500 using the night skybox and Perlin Noise to simulate the wind.

Even a density of 250 using the billboards representation is acceptable in terms of coverage (figure 8).

V. CONCLUSION & DISCUSSION

In this project grass was rendered using separate blades and billboards. The illusion of the wind blowing through the billboard representation of grass was implemented using trigonometric functions and Perlin Noise.

Possible extensions to this project could be directly implementing the Perlin Noise function in the shader, as this

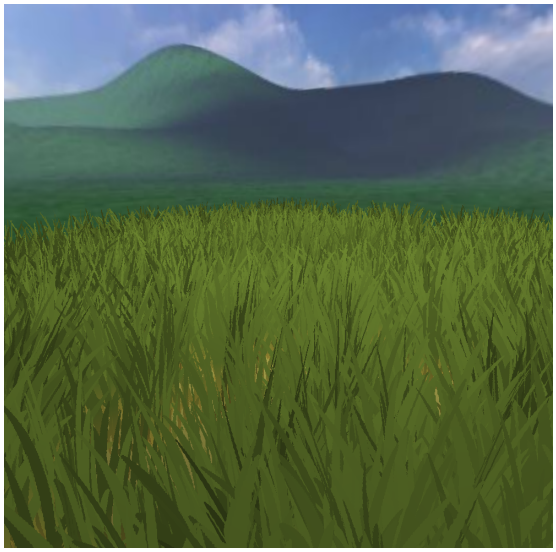

Fig. 8. Screenshot of the billboard representation with density set to 250 using the day skybox and Perlin Noise to simulate the wind.

requires less texture memory and allows for more control of the produced noise [12].

Even though this project achieved a decent amount of grass coverage with acceptable performance, the performance of both the blades and the billboard representation could have been improved by using a technique called instancing. Instancing is a technique used to draw multiple objects using a single render call, in order to minimize the communications between the CPU and GPU [13]. This is useful when a lot of objects have the same vertex data, but different world transformations, which is the case in both the blades and billboard representation.

REFERENCES

[1] K. Jahrmann, ""interactive grass rendering in real time using modern opengl features"; betreuer/in (nen): M. wimmer; institut für computergraphik und algorithmen, 2016; abschlussprüfung: 13.10. 2016."

[2] "GitHub graphicsprogrammingexercises, exercise 5.4," https://github.itu.dk/hend/GraphicsProgrammingExercises/tree/master/exercise_5_solutions/exercise_5_4_s, accessed: 10-02-2020.

[3] "GitHub graphicsprogrammingexercises, exercise 11," https://github.itu.dk/hend/GraphicsProgrammingExercises/tree/master/exercise_11_solutions/exercise_11_3_and_11_4_s, accessed: 10-02-2020.

[4] "Learn OpenGL basic lighting," https://learnopengl.com/Lighting/Basic-Lighting, accessed: 10-02-2020.

[5] "Learn OpenGL light casters," https://learnopengl.com/Lighting/Light-casters, accessed: 10-02-2020.

[6] H. Qiu, L. Chen, J. X. Chen, and Y. Liu, "Dynamic simulation of grass field swaying in wind." *JSW*, vol. 7, no. 2, pp. 431–439, 2012.

[7] R. Osada, T. Funkhouser, B. Chazelle, and D. Dobkin, "Shape distributions," *ACM Transactions on Graphics (TOG)*, vol. 21, no. 4, pp. 807–832, 2002.

[8] "GPUGems chapter 7. rendering countless blades of waving grass," https://developer.nvidia.com/gpugems/gpugems/part-i-natural-effects/chapter-7-rendering-countless-blades-waving-grass, accessed: 10-02-2020.

[9] "GPUGems chapter 6. gpu-generated procedural wind animations for trees," https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-6-gpu-generated-procedural-wind-animations-trees, accessed: 10-02-2020.

[10] K. Perlin, "An image synthesizer," *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[11] "YouTube grass sway in unity - shader graph," https://www.youtube.com/watch?v=L_Bzcw9tqTc, accessed: 10-02-2020.

[12] S. Green, "Implementing improved perlin noise," *GPU Gems*, vol. 2, pp. 409–416, 2005.

[13] "Learn OpenGL instancing," https://learnopengl.com/Advanced-OpenGL/Instancing, accessed: 11-02-2020.