Development of interactive model-based grass tools for open worlds featuring a fluid-simulation-based wind system

Marc Pitarch Dos Santos

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

June 15, 2023

Supervised by: Carlos González Ballester.

To nature lovers

Acknowledgments

First of all, I would like to thank my Final Degree Work supervisor, Carlos González Ballester, for keeping an overview of my progress during the last months.

I would like to thank my friends at Blekerslaan who made me see the world with a brighter light in the Netherlands. My family and friends supported and saw the progress too, so I am really grateful to Víctor, Fátima, Juan Carlos and Carla.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

Abstract

This report shows the process to implement an **optimized model-based grass rendering system** alongside a **fluid simulation based dynamic wind** that is able to **run in open worlds** using Unity and compute shaders. **Compute shaders** are a powerful tool used vastly in AAA titles, so researching about them and trying them out first hand is a must. They offer quick computation on the GPU and are able to manage big streams of data that the CPU can't, so they are ideal for blades of grass or managing 3D textures in an optimized manner. Furthermore, both systems are found in AAA games but there has yet to be an implementation that includes both. Hereby, this project implements a model-based grass rendering system where each blade is independently rendered alongside a fluid simulation that provides deep interaction in open worlds.

Contents

Chapter 1

Introduction

Contents

1.1 Work Motivation

Incorporating natural phenomena in videogames adds a layer of realism to these interactive experiences that makes immersion even more captivating. Moreover, when we factor in the key differentiating aspect of videogames - interactivity - the result is a truly meaningful experience.

As a nature lover and Erasmus participant in late 2022, my time as an exchange student in the Netherlands has really inspired me to work on vegetation and grass. As a matter of fact, I was thinking about this matter for a long time. I had a thought in the back of my mind about creating beautiful grass, and coincidentally, various Youtube videos sparked my interest into implementing a grass and wind system. The first one is a deep dive into the interactive wind and vegetation system inside *God of War (2018)* [6], showing how they use a fluid simulation in order to recreate realistic and interactive wind behavior.

On the other hand, another video from youtuber and technical artist *Acerola* [1]

shows an overview of the utility of compute shaders in grass generation. This was a useful starting point into researching and gathering information on the topic. Furthermore, **compute shaders are a crucial tool in making modern games** as they offer the ability to quickly compute big chunks of arbitrary data, making them flexible for many goals including non-rendering related subjects. This work uses compute shaders to implement complex nature phenomena: grass and wind. This way I intend to learn about this technology and challenge myself to make it run even on mobile devices, **implementing optimization techniques** and profiling the performance.

Another point that motivates the work is the lack of games with fluid simulation wind and blade-model grass. The fluid simulation idea is for wind in games is quite innovative and has only been seen (or at least mentioned) in *God of War* (2018) [6]. On the other hand, single blade grass can be seen across a small number of games such as *Breath of The Wild*[14] or *Ghost of Tsushima*[8].

Although this would make the project blow out of proportion, developing the grass tool and wind system will open the doors to creating a game over the developed technology. As they say, technology determines the development and design of videogames and with the creation of these tools, inspiration would come easily.

1.2 Related Subjects

- **VJ1227** - Game Engines.
- **VJ1221** - Computer Graphics.
- **VJ1216** - 3D Design.
- **VJ1208** - Programming II (Computing).

1.3 Keywords

Computer graphics, vegetation, grass rendering, fluid simulation, compute shaders

1.4 Objectives

The main objective of this work is to implement both systems in an optimized manner within Unity 3D. The main points that should be addressed are:

- **Model Grass Rendering**: create an efficient and flexible model grass rendering system that can be used in open worlds using GPU instancing in order to render thousands of blades of grass. The density of vertices allows a beautiful animation.

- **Grass Design Tools**: development of the necessary tools/parameters to make the generation of the grass as customizable as possible. This includes the ability to place the grass, change color, size and other parameters.
- **Static wind**: implement a basic animation system in a vertex shader in order to modify the position of vertices in the 3D grass model.
- **Fluid Simulation**: implement a 3D grid where a fluid simulation is computed and make it affect the animation though a texture. This works on top of the static wind to create more detailed animation for certain elements.
- **Keeping things optimized**: in the implementation of the forementioned items, compute shaders will be highly involved, so learning about them is unavoidable to keep the systems as optimized as possible. Because we are using GPU instancing, these techniques do not come 'for free' with the engine, so we need to implement them on our own, also giving the chance to learn about them.

Chapter 2

Planning and resources evaluation

Contents

2.1 Planning

In this section Table 2.1 showcases the main tasks that should be completed and the expected amount of time they would take. The tasks are separated in four different categories: **documentation**, **grass development**, **wind development** and **gameplay**. This makes it easier to work organize the tasks independently. It is clear, however that each task also turned into even smaller objectives as the work progressed, fitting the needs detailed in the requirements and system design.

A **Gantt Chart** is also provided, detailing the dependencies and time distribution across the months of work (see Figure 2.1).

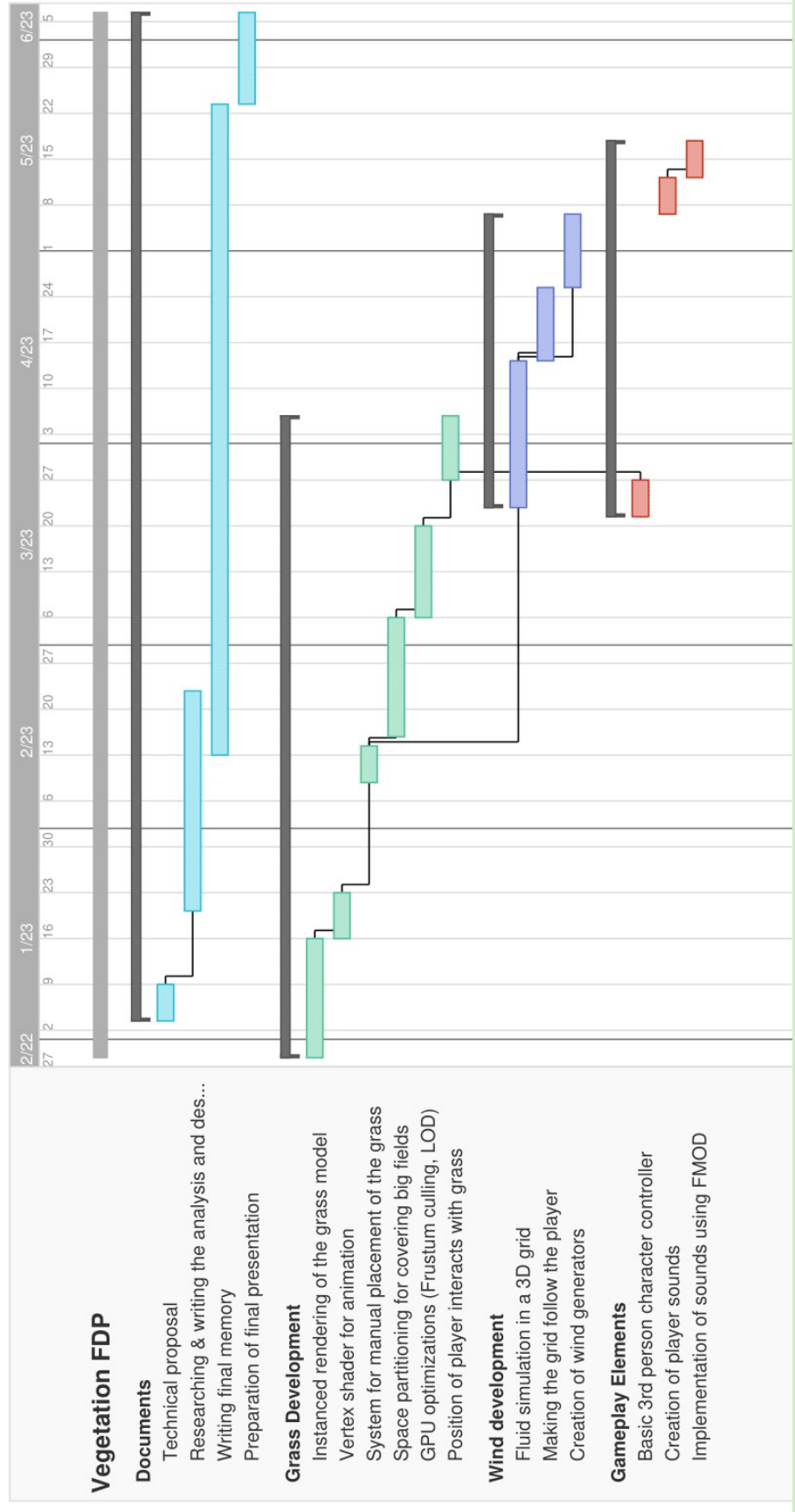

Figure 2.1: Gantt chart

Task	Type	Time (h)
Technical proposal	Documentation	5
Researching & writing the analysis and design document	Documentation	20
Instanced rendering of the grass model	Grass Development	2
Vertex shader for animation	Grass Development	10
Space partitioning for covering big fields	Grass Development	25
GPU optimizations (Frustum culling, LOD)	Grass Development	30
System for manual placement of the grass	Grass Development	15
Basic 3rd person character controller	Gameplay	10
Position of player interacts with grass	Wind Development	20
Fluid simulation in a 3D grid	Wind Development	50
Making the grid follow the player	Wind Development	20
Creation of wind generators	Wind Development	15
Creation of player sounds	Gameplay	10
Implementation of sounds using FMOD	Gameplay	15
Writing final memory	Documentation	40
Preparation of final presentation	Documentation	10
TOTAL		**300**

Table 2.1: Tasks

2.2 Resources Evaluation

For the development of this work the main resources that are needed can be separated into hardware and software.

- **Hardware**: at first, it was not clear whether my personal laptop with would be enough, but it turned out to be capable. The model is a MSI GS65 valued around **1.250 euro**.
  - **CPU**: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz
  - **GPU**: NVIDIA GeForce GTX 1660 Ti - 6GB GDDR6 VRAM
  - **RAM**: 16GB
- **Software**: all of the software that was going to be used could be free of charge, however at certain points photoshop was used, althought it could be swapped for a free alternative.

- **Unity (free)**: the free version of Unity, for personal use and small teams. It was the main rendering platform and the playground where to implement everything that was needed, as it has support for compute shaders.
- **Visual Studio Community 2022 (free)**: as a programming environment, as it comes by default with Unity and the 2022 version is quite easy on the eyes and has useful features.
- **GitHub (free)**: the version control desktop program chosen for safety and ease of use. It helps during the development when doing certain steps which are not safe.
- **Trello (free)**: in order to keep track of the working hours and the remaining tasks at each point in the development, Trello proved to be useful in gathering all the infromation, keeping track of progress and as a hub to everything related to the work.
- **Blender (free)**: the open source 3D suite that enables the creation of any necessary 3D element in the scene, including the grass blades.
- **Photoshop (19,66 euros/month)**: for the creation of certain textures and the mofication of others, it was necessary to make use of an image editing software.
- **Overleaf (free)**: writing the final report was done using the free version of Overleaf, the web LaTeX editor.

If we were talking about a professional environment, the time of the staff in charge of this matter would be part of the cost. However, as proved by the team at *Santa Monica* [6], the tasks of creating a fluid simulation and the rendering and application of the wind are handled by different sections.

So if we took into account that one person is in charge of the project, an approximate of 33 euros/hour is the salary of a junior software engineer in Santa Monica [9], so counting on 300 hours, the cost would be **9.900 euros**. Adding the cost of all hardware and software necesities during 4 months, we end up with a forecasted price of **11.228,64 euros**.

Chapter 3

System Analysis and Design

Contents

3.1 Requirement Analysis

In this section we will go through the requirements for the design of the grass rendering system and the wind simulation within Unity. We should list all the elements that are necessary for the design to satisfy the objectives listed previously.

The system should **render grass blades**, as everything else will be added to this. The tool should be as flexible as possible, introducing **parameters** to modify generation, such as **placement**, **density** and **randomness** in position. Furthermore **length**, **colour** and **bend** of the blades should also be a modifiable part of the system. The **position of the player** should interact with the grass. All of this will be done through the main component that manages the grass inside of the scene and runs the **compute shader** and the blade's **material shader** that controls the rendering and vertex displacement, as part of the rendering pipeline.

On the other hand, for the static wind system, it should be possible to modify its **strength** or overall effect over the grass, **speed** and **direction**. The dynamic wind system needs to be implemented using **fluid simulation** and should feature **gener-**

ators of wind such as directional generators and wake generators which affect the system only when they are moved. This will be done through another **component that manages wind**.

This should all run in **open worlds (min. 500x500 units)**, at a **high framerate (+100fps)** and even on **mobile devices**.

3.1.1 Functional Requirements

In the next section, taking into account the description given above, the functional requirements that define this project will be listed alongside a table explaining its inputs, outputs and behaviour of each element. These are functionalities that should be implemented. The utilities that the ideal system will have and should be developed are the following:

- **GRASS1**. Generate blade positions (see Table 3.1).
- **GRASS2**. Paint a placement texture (see Table 3.2).
- **GRASS3**. Adjust density of grass (see Table 3.3).
- **GRASS4**. Adjust randomness of blade positions (see Table 3.4).
- **GRASS5**. Modify the colour of blades (see Table 3.5).
- **GRASS6**. Modify LOD distance (see Table 3.6).
- **GRASS7**. Modify cutoff distance (see Table 3.7).
- **GRASS8**. Modify bending of the grass through player position (see Table 3.8).
- **WIND1**. Generate wind through a fluid simulation (see Table 3.9).
- **WIND2**. Interact with the grass through wind (see Table 3.1).
- **WIND3**. Create static wind generator (see Table 3.11).
- **WIND4**. Create dynamic wind generators game objects (see Table 3.12).

Input:	Map Size, Heightmap
Output:	Positions buffer
The user inputs the map size and the heightmap corresponding to the terrain that should be filled, the blade positions are calculated through a compute shader	

Table 3.1: Functional requirement «**GRASS1**. Generate blade positions»

Input:	Mouse location
Output:	Texture with the positions of the grass
The user paints directly in the editor where they want the grass to grow. When rendering the grass only in the places where this control texture has alpha, it will be rendered	

Table 3.2: Functional requirement «**GRASS2**. Paint a placement texture»

Input:	Density (grass/unit)
Output:	More positions in the position buffer
The user specifies a density number in the grass main component and the number of blades will change	

Table 3.3: Functional requirement «**GRASS3**. Adjust density of grass»

Input:	Randomness amount
Output:	Position are randomly offset
The user specifies randomness amount int the X and Y axes so the blades' positions look more natural	

Table 3.4: Functional requirement «**GRASS4**. Adjust randomness of blade positions»

Input:	Top color, bottom color
Output:	Blades color change
The user specifies two colors, one for the top of the blade, and another one for the bottom. The fragment shader interpolates between them across the V axis in the UV map	

Table 3.5: Functional requirement «**GRASS5**. Modify the colour of blades»

Input:	LOD distance
Output:	Model is swapped at the specified distance from the camera
The user inputs a distance where they want the higher poly model to change to a lower one in order to save resources	

Table 3.6: Functional requirement «**GRASS6**. Modify LOD distance»

Input:	Cutoff distance
Output:	Blades are not rendered from cutoff distance
The user inputs a distance and the grass will not be rendered from that distance to the camera	

Table 3.7: Functional requirement «**GRASS7**. Modify cutoff distance»

Input:	Player position
Output:	Blades move from position
The user moves the character and its position bends and pushes the blades downwards	

Table 3.8: Functional requirement «**GRASS8**. Modify bending of the grass through player position»

Input:	Volume size, viscosity
Output:	A 3D Texture for X, Y and Z
The user specifies the size of the simulation volume and 3 textures store the velocities of each voxel	

Table 3.9: Functional requirement «**WIND1**. Generate wind through a fluid simulation»

Input:	Wind Strength
Output:	Blades move scaled to strength
The user specifies the strength of the wind so that the movement is stronger	

Table 3.10: Functional requirement «**WIND2**. Interact with the grass through wind»

Input:	Wind direction
Output:	General wind texture
The user specifies the direction of the static wind that affects all blades of grass	

Table 3.11: Functional requirement «**WIND3**. Create static wind generator»

Input:	Wind Motor game object
Output:	Influence over the fluid volume
The user can create game objects that influence the behaviour of the fluid in the simulation, such as a directional motor or a circular motor	

Table 3.12: Functional requirement «**WIND4**. Create dynamic wind generators game objects»

3.1.2 Non-functional Requirements

As stated in the initial description of the system, the non-functional requirements which represent conditions or limitations that the system should accomplish are:

- **NF1**. The grass can be placed in huge areas of terrain, minimum 500x500 units.
- **NF2**. The rendering of the grass and wind simulation must keep a playable high frame rate (+100 fps).
- **NF3**. The overall look should be stylized and compelling.

3.2 System Design

In order for the reader to have a clearer understanding of the working of the systems, two diagrams will be presented in this section.

Firstly, the use case diagram 3.1 showcases how the user can interact with the grass through parameters that modify position, look and displacement of vertices of the blades. Wind is also a part of this system, therefore it can also modify the position of vertices.

Secondly, the activity diagram 3.2 portraits the way the generation of the grass works. The structure of the tiles is a quadtree, where every piece of terrain that has a standalone heightmap should be a independent parent node. The subdivision condition of the quadtree will be further discribed in the next section.

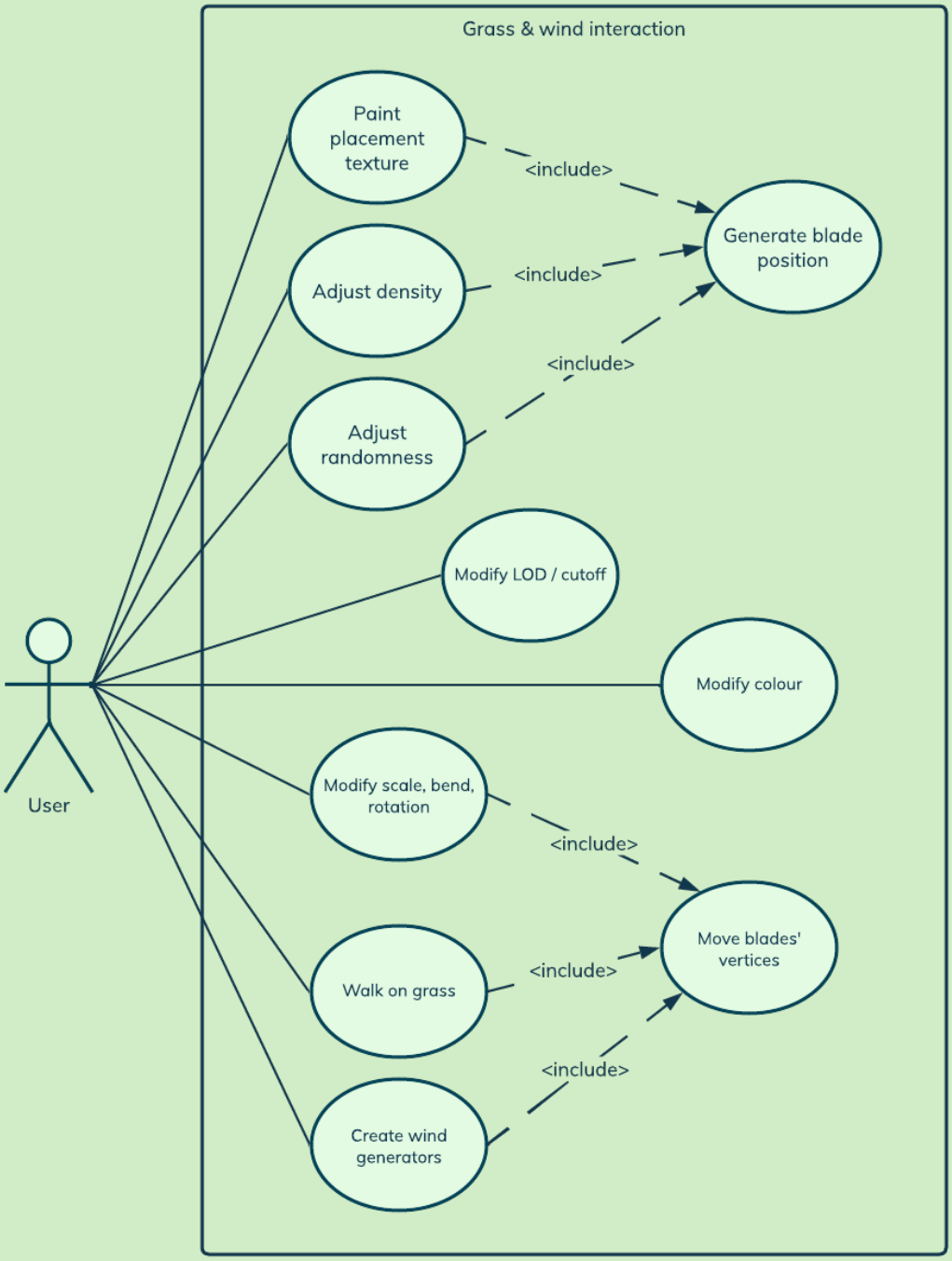

Figure 3.1: Use case diagram

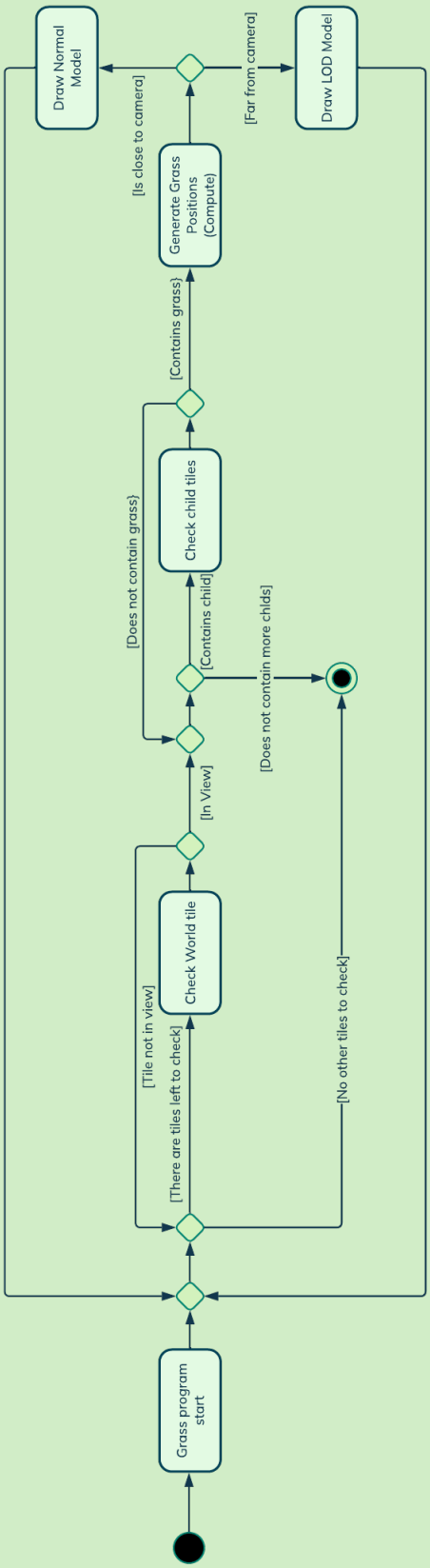

Figure 3.2: Activity diagram

3.3 System Architecture

In this section, the specifics on the tools and structures used to develop the program and hardware requirements of the end user will be detailed.

For the creation of the scripts, **C#** and **Compute Shaders (HLSL)** [26] inside of Unity 2021.3.11f1 are used. Compute shaders are separate from the normal renering pipeline, and enable the programmer to compute any type of data in an efficient manner, so they are perfect for generating lots of positions efficiently. As one of the bottlenecks of rendering so many elements is the action of passing information around the GPU and CPU, **GPU instancing** [20] enables us to render the same object multiple times in an efficient manner.

In order to achieve this, **AppendStructuredBuffer<T>** [19] is used for populating the buffers, as it is a flexible sized compute buffer that will make the programming of LOD and culling easier. These structures enable passing arbitrary data from the CPU to the GPU, that way we can generate the positions of only the blades that will show up on the screen. If we used a normal Structured Buffer, its size should be known beforehand, and as we want to cull positions to be sent to the drawing function during runtime, creating a structured buffer with the desired grass blades would not be straightforward as explained by *Acerola* in his video *'Modern Foliage Rendering'* [2]. Using append structured buffers as seen in the demo created by *Brian* from *UpRoom Games* [7], fixes this problem by providing a felixible sized buffer.

For the vertex and fragment shaders, **Shader Graph**, the visual scripting solution for shaders by Unity is used. This makes the creation and understanding of the shader logic much easier, and almost all the functionality available through scripting can also be done here. With it, the visual appearance and the animation of the grass are implemented.

The partitioning of the world is based on a **region quadtree** [28], in order to make frustum culling faster and to access data in the world, such as placement textures or heightmaps. Thanks to this data structure, we can store all of the information about the grass in the game world efficiently and only access the parts that are important at any certain point, mainly the frustum of the player's camera. The quadtree nodes store: the heightmap corresponding to the area they contain, the placement mask, the materials (one for each LOD), the compute shader that generates the positions and compute buffers for the positions, the LOD positions and the indirect arguments for both (needed for the drawing function). When building the quadtree, the main input is the placement texture, as the tree will only subdivide where there is detail in the placement textures, in other words, where there is grass. The maximum depth is user editable as we don't want regions too small so that the GPU becomes inefficient at generating the positions.

For the **fluid simulation**, compute shaders and 3D textures are used instead of buffers, because as explained by *Rupert Renard* in his GDC talk in 2019 '*Wind Simulation in God of War*' [16], using a separate 3D texture for each axis, proved to be more efficient. The simulation proposed here is closely based on *Jakub Mička*'s implementation in his master's thesis '*Voxel-based fluid simulation in Unity*' [12], changing however, the approach on using textures instead of buffers. This proved to be a solid resource, however it was crucial to understand the holy grail of real-time fluid simulations, *Jos Stam*'s '*Real-time fluid dynamics for games*' [18].

Finally, we can discuss the memory cost of the system. For the grass rendering, each chunk of grass uses 5 buffers. The main one is the grass data where position and scale information is stored using 3 floats for position and 3 floats for scale, the size for each element is: 6 * 4 bytes = **24 bytes**. Then, we have the arguments buffers in order to keep track of how many elements we are rendering - one for each LOD - and uses 5 unsigned integers (4 bytes), adding up to **20 bytes** each (this is shared accross all chunks, so it is insignificant). Finally, two buffers - one for each LOD - store the final culled positions that are used, also **24 bytes** each. If the density of the chunk is 6, meaning that we have 6 blades of grass per unit (enough to make it look dense) and the area of the chunk is 16 meters squared, that means the chunk stores a maximum of: $(16 * 6)^2 =$ **9.216 blades**. As mentioned, we use one buffer for all the data and then two buffers to store the culled position of each LOD, meaning the final storage size of the chunk is: 9.216 * 24 + (9.216 * 24) * 2 = **663.552 bytes** (0.110592 MB). If at any given moment we have an average of 25 visible chunks, the total cost of **VRAM** on the GPU of the grass is **16.588.800 bytes**, or **16,59 MB**. This is a decent cost, and it is achieved thanks to the world partitioning, otherwise, we wouldn't be able to render small chunks and all grass positions would be stored from the beginning, and the size would blow out with the world size. Thanks to the quadtree, the cost of each node much smaller and enables the system to scale infinitely, because we load and free up the chunks dynamically during run-time.

3.4 Visual Style & Scene

The scene will be, at least, a 500x500 units of terrain area filled with grass and vegetation. It will have some hills and mountains and other natural features sculpted with the terrain editor inside of Unity. Furthermore, other types of vegetation such as trees, flowers and bushes can be added. As the wind system is based on a texture that indicates the movement in each axis, interaction with other vegetation is added modifying the shader of the corresponding element's material.
The character is taken from Unity's asset store's '*Starter Assets - Third Person Character Controller*' asset [23], including animations and the basic controller for camera and movement. A stylized approach will be used for everything in the scene. Realism is not a goal here. The style of the grass in *Breath of the Wild* [14] is the main inspiration in looks and the grass in *God of War (2018)* [13] is in behaviour.

Figure 3.3: *Breath of the Wild*'s grass

As we can see in the original game (see Figure 3.3), the grass does react to lighting and it receives but does not cast shadows. We can also see a difference in colours, between different blades but also between the top and the bottom of the blades. What's more, it even changes looks depending on the place it grows. There's also a difference in the height, for example in the transition between places filled with grass and empty planes (see Figure 3.4).

Figure 3.4: Grass height falloff

Figure 3.5: *God of War*'s fluid simulation debug

In terms of the behaviour, as seen in *God of War* (see Figure 3.5), the grass will react to generators of wind in a detailed way. This is done by using the fluid simulation, which mimics the behaviour of wind closely, creating a deep interactivity between the events in the game and the grass.

These are the main references for the visual style of the grass. However in the end, the color variation between blades was not implemented and only one type additional vegetation (a tree) was added.

Chapter 4

Work Development and Results

Contents

In this section, a breakdown of the developed technology is detailed, going over the steps taken to achieve each of the tasks presented beforehand. Some simplified code snippets are added as figures so the reader can follow the explanation at a base level.

4.1 Grass Instancing

The steps taken here follow the overview of *Acerola*'s video '*How Do Games Render So Much Grass?*' [1]. The first objective was to render grass in a plane, using GPU instancing.

As we want to visualize what is happening, we need somewhere to place our grass. A plane and a 2m height cube to have a size reference are added (see Figure 4.1). Then, we want to start generating the positions so... How do we even start? To keep things simple, we can start with the typical grass that is used in the majority

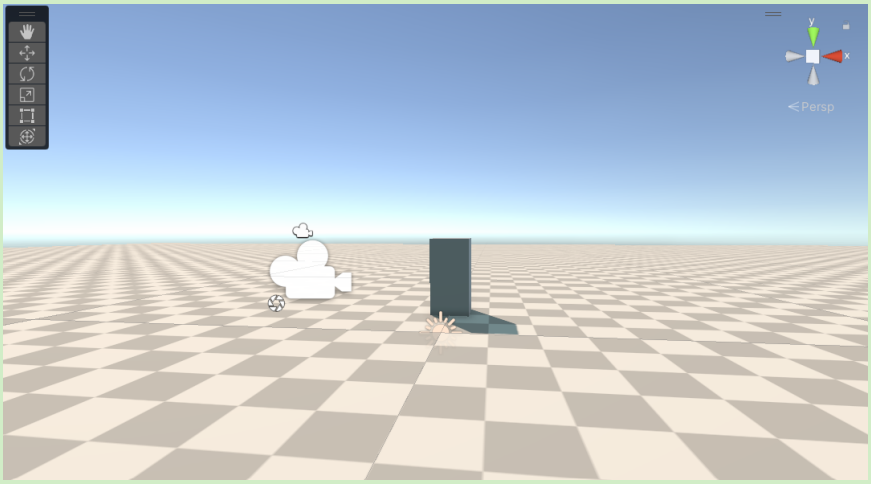

Figure 4.1: Scene's Initial State

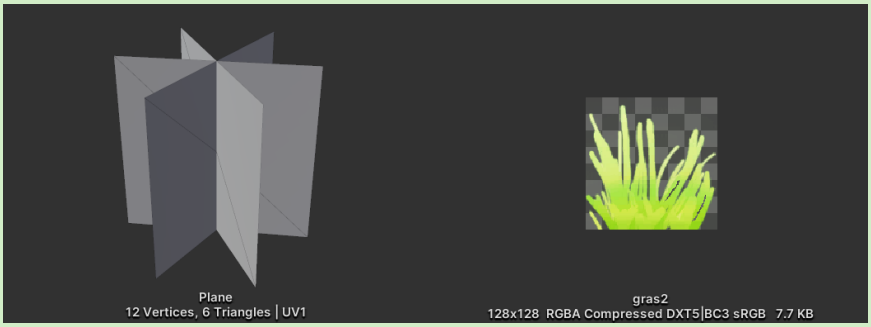

Figure 4.2: Billboard mesh and texture

of games, **billboard grass**. This grass is based on a group of planes positioned in a way that they create a cross section in the middle and using textures to give the impression of dense grass. We can see the model that I first created and its texture, using Blender and Photoshop (see Figure 4.2). After we have the grass and the scene set up, we can start generating positions.

As we have mentioned in order to render the grass using **GPU instancing**, we have to calculate positions, we will use compute shaders as they process data quickly and store the data directly in the GPU. For that purpose the script *GrassMaster.cs* is created. It manages everything related to the generation of grass and the shader. We also need to create a **compute shader** asset using Unity's Project Window. We rename the file *GrassCompute.compute*. In order to utilize the shader we need to describe and understand its elements first (see Figure 4.3).

In a compute shader, we have to create what's called a *kernel* which is not more that a function that will execute whenever we dispatch this shader through a script. This kernel will have a specific size of **thread groups** (up to 3 dimensions) that will

```
1  // Each #kernel tells which function to compile; you can have many kernels
2  #pragma kernel CSMain
3
4  // Create a RenderTexture with enableRandomWrite flag and set it
5  // with cs.SetTexture
6  RWTexture2D<float4> Result;
7
8  [numthreads(8,8,1)]
9  void CSMain (uint3 id : SV_DispatchThreadID)
10 {
11     // TODO: insert actual code here!
12
13     Result[id.xy] = float4(id.x & id.y, (id.x & 15)/15.0, (id.y & 15)/15.0, 0.0);
14 }
15
```

Figure 4.3: Compute shader. In red, kernel; blue, parameters and green group thread size

```
#pragma kernel GrassGenerator


RWStructuredBuffer<float3> _Positions;

[numthreads(8, 8, 1)]
void GrassGenerator (uint3 id : SV_DispatchThreadID) {
    _Positions[id.x, id.y * 8] = float3(id.xy, 0);
}
```

Figure 4.4: Calculating the positions

work in parallel to process whatever we tell it to. Besides, we can pass in buffers and other types of parameters such as floats, vectors, etc.

A first approach we can take is to **generate a position for each thread**. In this case we will cover an area od *128*128 units*. So we can make the *threadgroup* size 8*8, dispatch 16 groups and generate one position for each thread id (see Figure 4.4). The size of the threadgroup can be fiddleded around with to get the best performace possible. Now, in the *GrassMaster* script, we have to hold **a reference of the compute shader**, **create the buffer**, have a **reference of the model and materials** and **dispatch** the compute. In order to create the buffer, we can hold a private attribute in the component and create a *new ComputeBuffer* object. As these objects will allocate memory, it is a good practice to initialize them when the object is enabled and free the memory when the object is disabled (Unity's functions *OnEnable* and *OnDisable*). Next, we have to link up the buffer with the shader, so we use the function *SetBuffer*. Besides, in the grass material we also need to link the

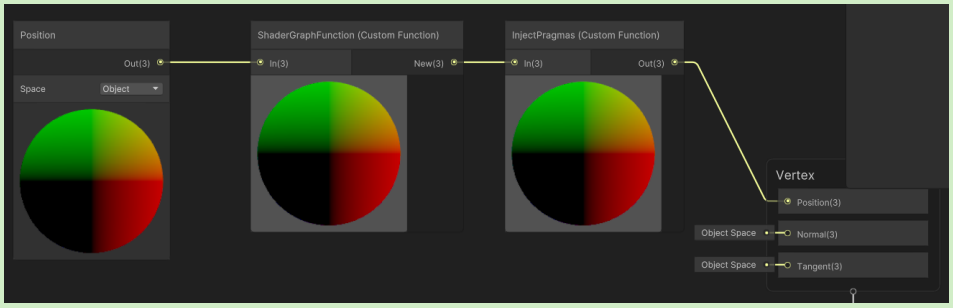

Figure 4.5: Injection in Shader Graph

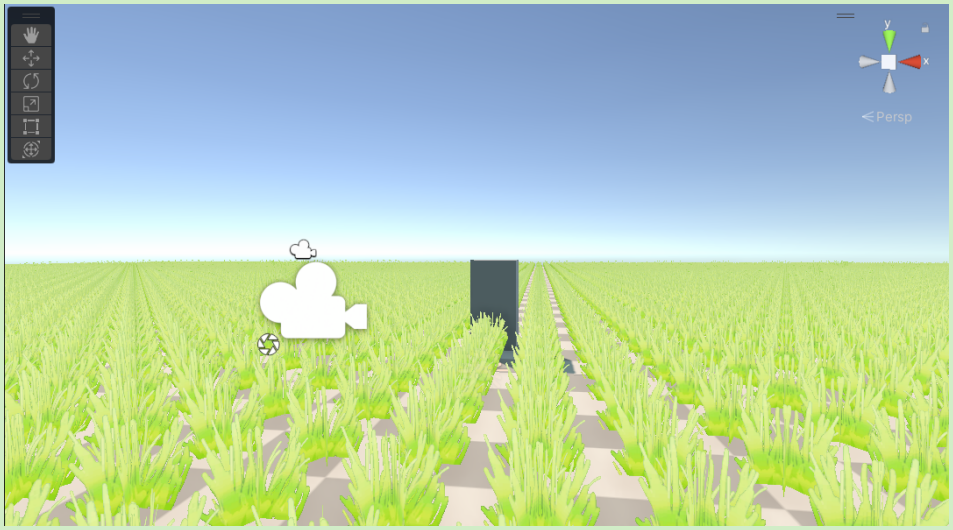

Figure 4.6: First generation result

buffer so it can move the grass where it should be. This also means creating a material shader that is compatible with GPU instancing, which is not default in Shader Graph so we need a quirky workaround. Thanks to the tutorial by *Catlike Coding* on Compute Shaders [5], we see that we can create some custom nodes to 'inject' the necessary code to enable procedural instancing, in other words, to use GPU instancing (see Figure 4.5). This is done in the vertex shader. Finally, we can tell the GPU to render our grass using the model and material that we specified, using the function *Graphics.DrawMeshInstancedIndirect()*[21] one of the options to procedurally draw meshes, in this case it needs an argument buffer that hold information about the mesh and the number of instances that we want to draw..

The result of all of this can be seen in Figure 4.6. We observe that the positions are not natural, so the next step would be to add a **random offset to the location**. Creating new parameters in the compute shader and the script follows the same pattern as before, so we can also add logic to change the size and density always

```
// Random generator
float rand(float2 co)
{
    return(frac(sin(dot(co.xy, float2(12.9898, 78.233))) * 43758.5453)) * 1;
}

void SetPosition (uint3 id)
{
    if (id.x < _Resolution && id.y < _Resolution)
    {
        float Xoffset = rand(id.xy) *_OffsetXAmount;
        float Yoffset = rand(id.xy) *_OffsetYAmount;

        float3 pos =  0.0f;
        pos.x = id.x * _Step - _Size * 0.5 + Xoffset;
        pos.z = id.y * _Step - _Size * 0.5 + Yoffset;


        _Positions[id.x + id.y * _Resolution] = pos;
    }
}
```

Figure 4.7: Random offset

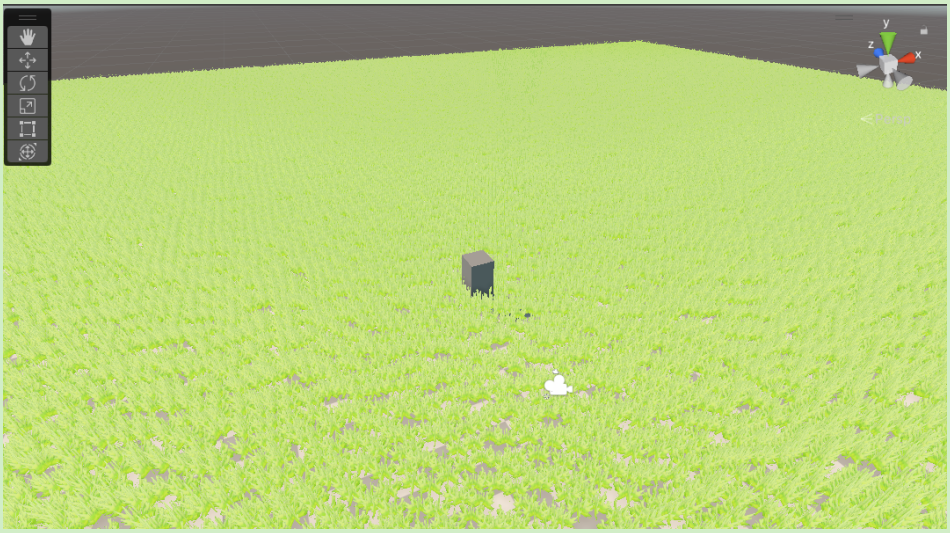

Figure 4.8: Increased density and random offset generation

taking into account the size of the threads, buffers and the previously required set up steps, so no further explanation will be done. We just need to add a random offset when generating the positions, so we can modify the compute shader to do this (see Figure 4.7). Increasing the density and the random offset, the field looks much more natural and appealing (see Figure 4.8).

The next step would be to modify the terrain and have the **grass stick to it**. Thankfully, the Unity Terrain package allows the user to export the heightmap of

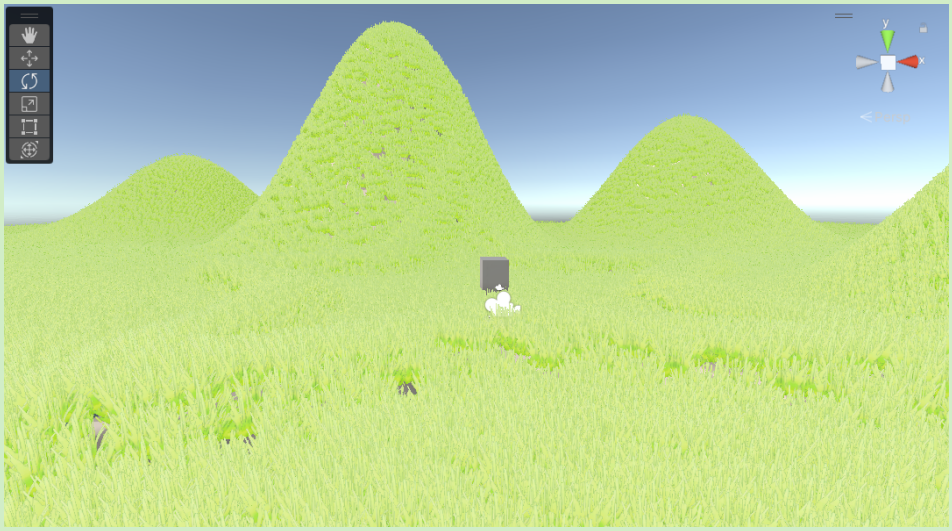

Figure 4.9: Grass sticking to terrain

the generated terrain, so if we calculate the world-space UV, we can sample this texture in the compute shader and offset the position's Y axis to the desired height (see Figure 4.9).

Now that we have a basic system, we can swap the billboard for a **blade model**. First of all we will have to create it. As before, Blender is the chosen tool. The first model that I produced has 9 vertices, in order to have enough detail for an eye catching animation (see Figure 4.10). However, when we simply swap one model for the other, it looks pretty bad. First of all we have to up the density, and second, the **color of the blade** is now missing. One of the ways we can colorize the blade is using a different color for the top and the bottom, faking ambient occlusion. In the fragment shader, the two colors are linearly interpolated across the UV's V axis, so the color changes from bottom to top. The result of this can be seen in Figure 4.11. It is also of crucial importance to mention the direction the vertices' normals should be facing. In the first version of the model, the normals point parallel to the ground and depending on the position of the sun, it can look pretty bad, so in further versions (shown in the next sections), the normals are manually fixed.

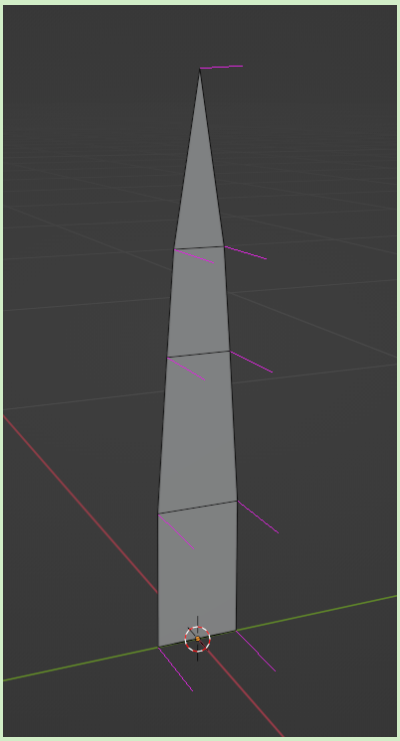

Figure 4.10: First blade model

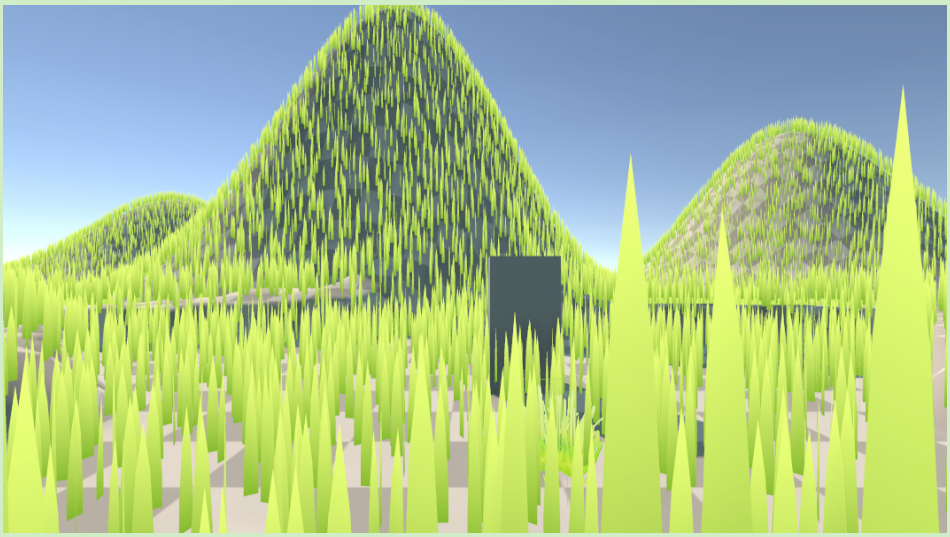

Figure 4.11: Model generation

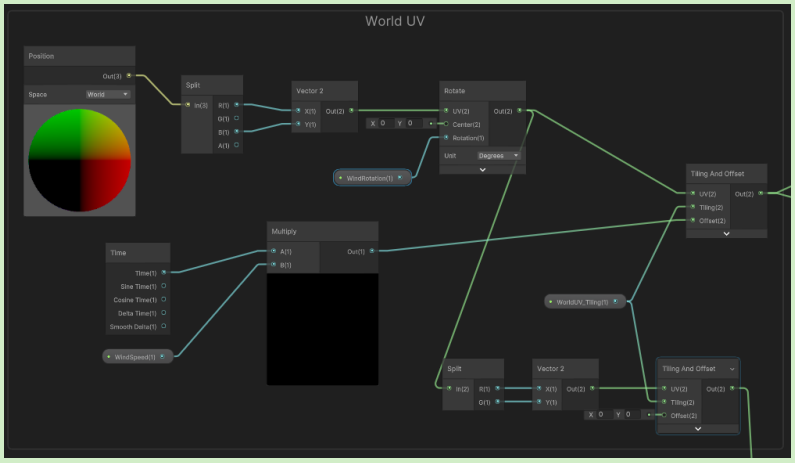

Figure 4.12: Shader Graph world-space UV

4.2 Grass Animation & shape

The next objective is to have animation on the grass. Because we have high density in the vertices, we can really take advantadge of the vertex shader.

One of the many ways we can fake wind is to have a **scrolling noise** [6] texture in the world displace the vertices of our blades. To get this, we first need to calculate the world-space UV in the shader, and then offset (move) it by adding the running time of the scene. In Shader Graph, we use the position X and Z coordinates, rotate them using a wind direction parameter and use a Tiling and Offset node to adjust what we explained. We will also keep a copy without the offset (see Figure 4.12). Then, this UV can be plugged in any texture. As a first instance, we can hook it up to a sine wave to process the UV, as it is the base for more complex wind noises. Eventually it was changed by an implementation of Blender's **distorted noise** as seen in this post in *Stack Exchange*. Furthermore, if we just displaced the vertices, the blades would start to float, so a key point is to mask the displacement of the wind using (again) the V coordinates of the UV map, this way, the base of the blade is not displaced (see Figure 4.13). This translation of vertices actually generates a stretch that is not loyal to reality, but as we have stated, realism is not an objective for this work.

The movement is hard to appreciate in a still image, but in Figure 4.14 we can see a bit of the sway created by the displacement of the noise.

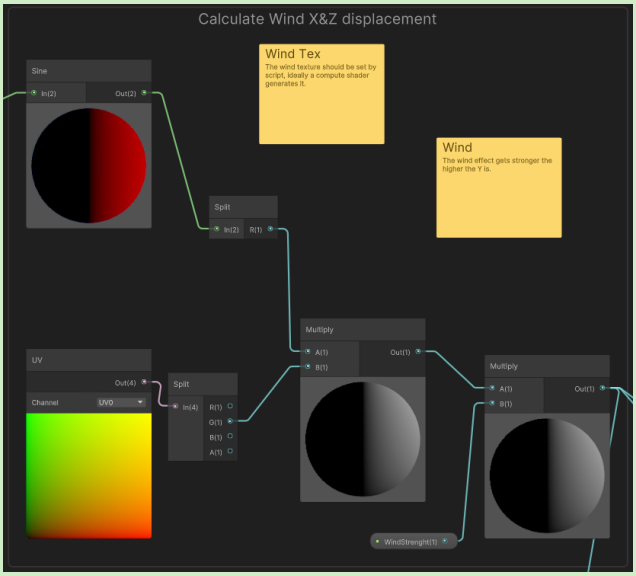

Figure 4.13: Wind inside the shader

Now we have a nice movement, but when we look sideways, we can clearly see the planar nature of the model (see Figure 4.15). One way to fix this is to add a **random Y rotation to each blade**. I decided to do this using another noise texture that will drive the strength of the rotation, effectively giving us random rotation. To perform this correctly, we should use the instance position instead of the vertex's, as it gives a homogeneous rotation throughout the whole blade (see Figure 4.16).

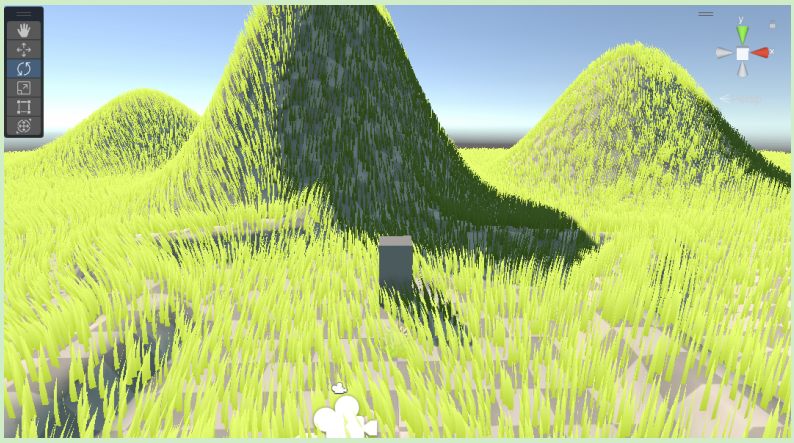

Figure 4.14: First wind generation

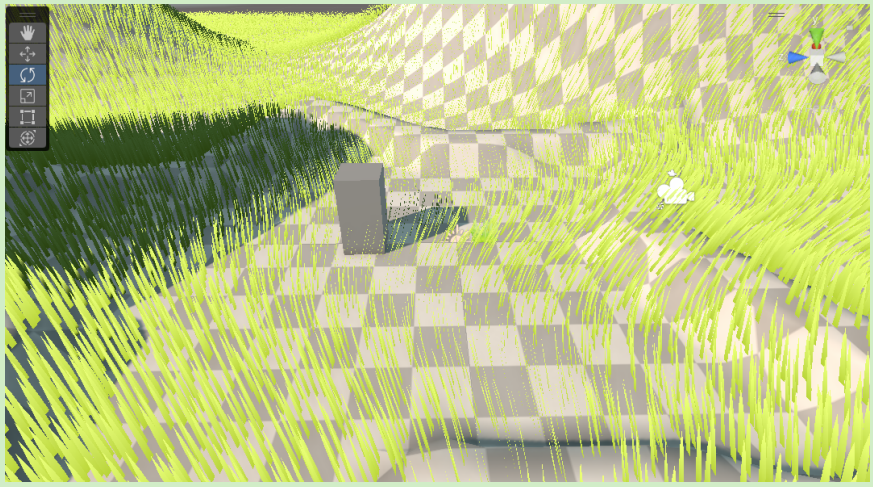

Figure 4.15: Grass field from the side

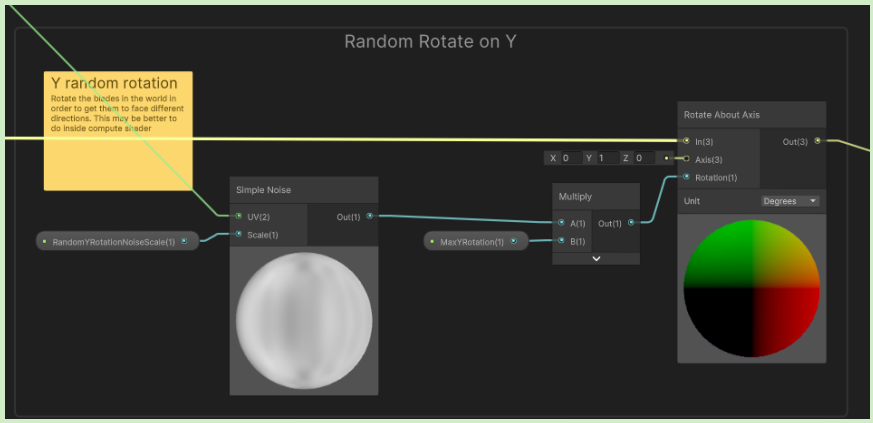

Figure 4.16: Shader random Y calculation

Another matter that should be tackled is the top view. Similarly to the side view, the blades seem thin when looking from the top. We can fix this **bending the blades**. Another section of the shader graph is responsible for this movement. After some iterations, I opted to give the user to chose whether the wind direction drives the bend or the blade just bends around its X axis. The bend strength is randomly assigned to grass blades and it is also modified by the wind strength at that point (see Figure 4.17).

To further improve this we can create **another model** which has a small bend baked in the Y axis. This helps with the feeling of density. Some optimizations to the model were also introduced throughout the whole development, iterating over the number of vertices, shape and normal direction of vertices. The final model can be seen in Figure 4.18.

The final look of the field after bend, more density and additional parameters added to the shader (size randomness, smoothness, detail textures, etc) are applied can be seen in Figure 4.19.

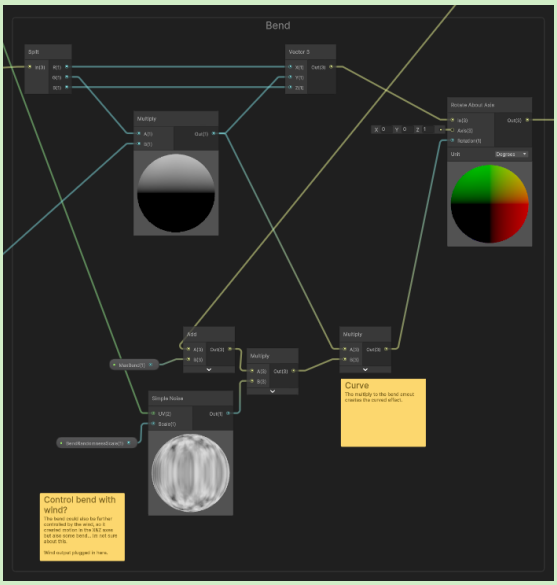

Figure 4.17: Shader bend calculation (only blade's X axis)

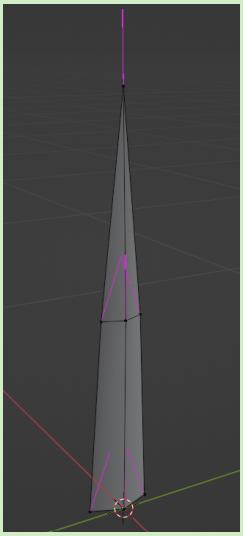

Figure 4.18: Final blade model

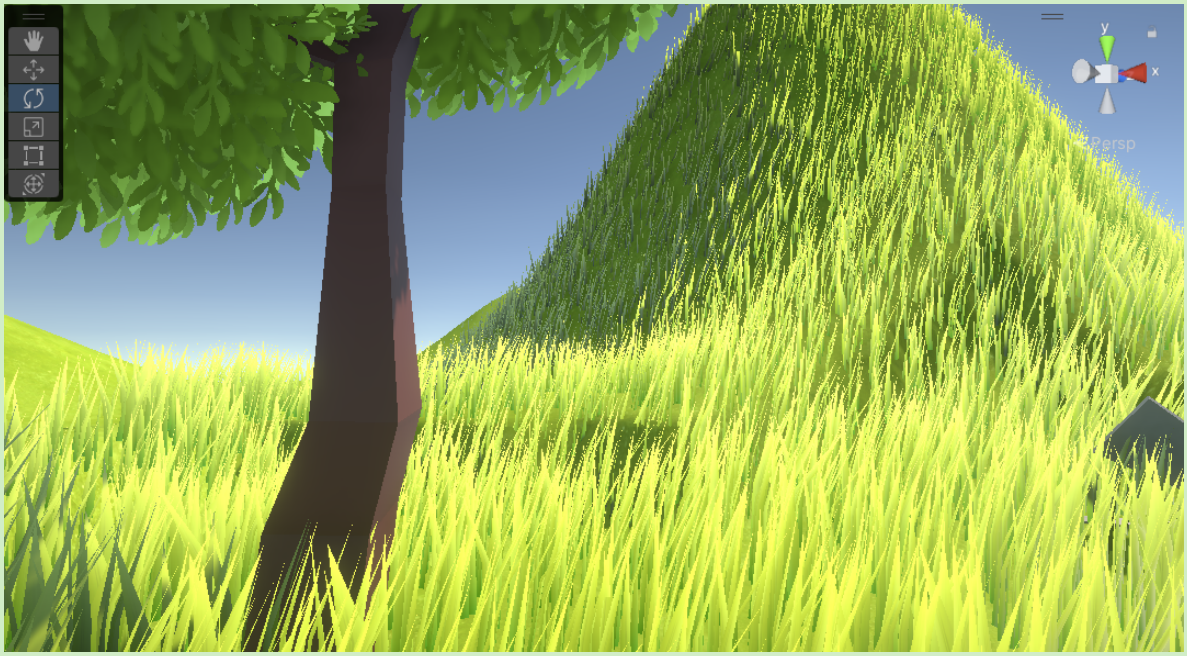

Figure 4.19: Final look of the grass

An additional feature/detail we can implement, is **view dependent parameters**. For example, we can make the bend dependent on the view angle. This means that when the player is looking down, we increase the bend in a subtle manner so that the field always looks as dense as possible. The calculation of the vector is done in the *GrassMaster* script: the parameter *MaxAdditionalBend* is added to the normal bend and its value is scaled by the remapped angle between the view vector and the up axis in Unity (Y axis). An dramatization of the effect can be seen in Figure 4.20.

The full implementation of the *GrassMaster.cs* and the *GrassCompute.compute* scripts can be found in the repository in Appendix A, where we can see the parametrization of all the parameters (also inside Shader Graph).

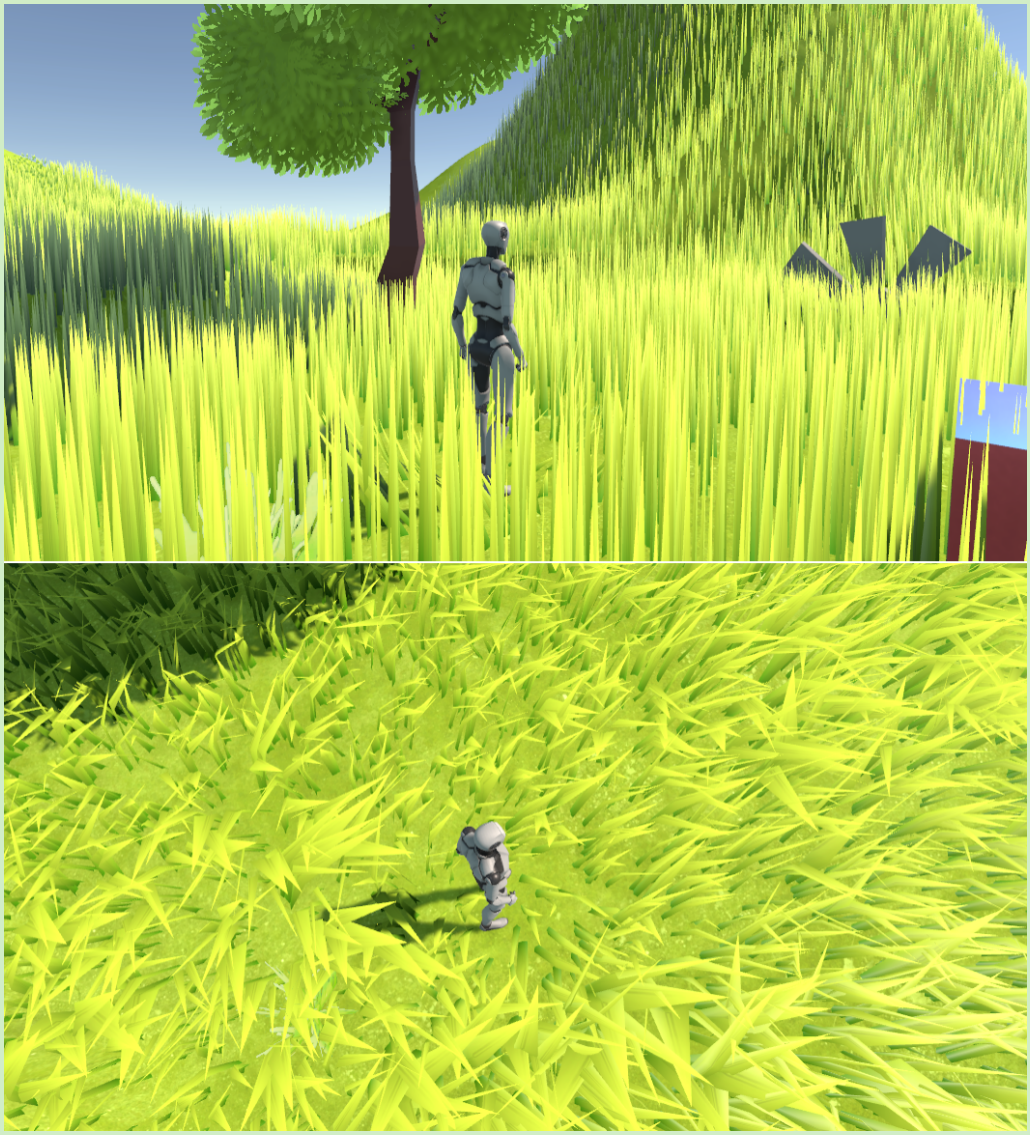

Figure 4.20: Dramatization of view-dependent bend

4.3 Manual Placement of the grass

Approaching a system that enabled the user to paint the grass wherever they want was a challenging problem. From the beginning the idea was to **let the user draw over the terrain in the editor** and the grass would appear in those areas.

The first approach I took was driven by *Minions Art*'s work published in her patreon post [3] where she attatches the scripts of her painters. The editor is really polished and has many functions, however adapting so many parameters to my grass would be challenging and I would not have the freedom to design it as I would want. Besides, her approach uses a **mesh** that stores all positions in its vertices. While this has its advantages, such as placing grass wherever in the world and not depending on the heightmap, it also meant that it has a limit, concretely up to 65.536 vertices [22] (the 32 bit index format enables up to 4 billion vertices, but it is not compatible with all platoforms, so in order to maximize compatibility only the 16-bit buffer is used). We need a huge density as we are working with single blades so generating hundreds of thousands of vertices is not ideal. Morover, displaying these positions in the editor isn't ideal either; at this point in development, the grass was not optimized, so having to constantly render all blades is not an option and displaying a gizmo at each vertex killed the performance in the editor. It was clear that another approach should had to be taken.

Looking at the GDC talk of *Sucker Punch*'s rendering engineer *Eric Wohllaib* [30], we can see that they use a tiling system where each tile holds a **placement texture**. This looked like the proper way to create my own system as it offers flexibility and scalability. The system implemented in this work is divided in 3 scripts. The *TerrainPainterComponent*, the *TerrainPainterEditor* and the *GrassMaskDisplayer*. Let's start with the *TerrainPainterComponent*.

The painter component is the one in charge of all painting behaviour. Here all of the brush parameters and necessary textures are stored. The main attributes are: the **mask texture**, brush strength, texture and size. In order to make it easier to work with, a **custom editor component** (*TerrainPainterEditor*) changes the looks of the component in the inspector and also allows us to create buttons that execute functions in the main painter component (see Figure 4.21). So for example when the user wants to clear the whole position mask, the button *Clear Mask* enables that functionality. It is also crucial to mention that the script will execute in the editor so the *[ExecuteInEditMode]* attribute is stated at the top of the class.

The mask texture is created when the button *Create Texture* is pressed, and takes into account the size of the terrain's internal alphamap textures, which hold data of the applied texture at a certain location in the terrain. Then, using Unity's editor events (*SceneView.duringSceneGui*) we can tell it to do whatever we want. In this case, a raycast will check collision at the mouse position, transform that location to the terrain's local space and then transform it into the texture space. Finally using the

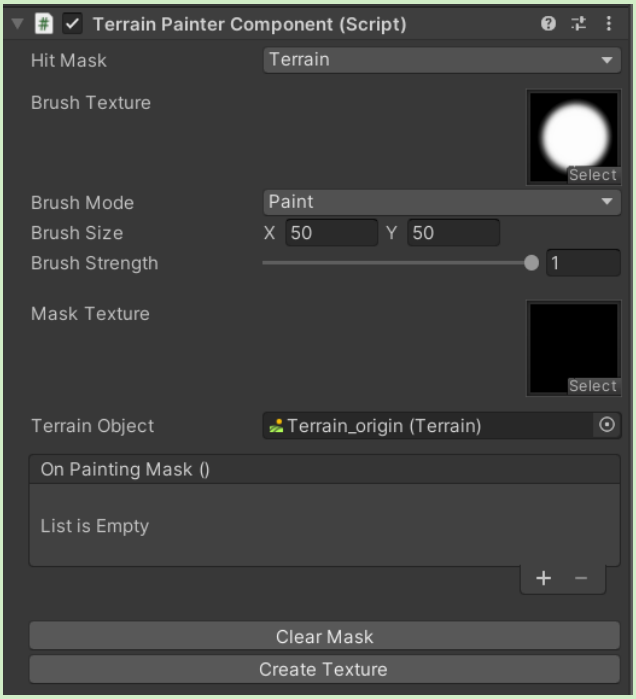

Figure 4.21: Terrain Painter Component in inspector

brush texture, size and strength, we will go through all the pixels of the brush texture and set the pixels' alpha value of the mask texture according to the parameters we set. Another important parameter is the *Brush Mode*, as it allows the user to either paint or erase pixels in the texture.

In order to display the texture so the user has **feedback when using the tool**, the *GrassMaskDisplayer* displays the texture. At first the idea was to interact direcctly with Unity's terrain system. However this proved to be cumbersome, as there is no clear way to edit the splatmaps in a separate way to their own systems in the terrain or use one of the channels differently. So instead we can use a more flexible system using a **decal** object that projects the texture over everything. The *GrassMaskDisplayer* is added to the painter object alongside a child object with the decal. The script moves and scales the decal depending on the size of the terrain. It also links the texture of the mask to the decal's texture. Furthermore, the decal object is only activated when the user has the *GrassPainter* game object selected so no interference is had in other situations. An example of what can be achieved is shown in Figure 4.22.

The final point that should be addressed is **how to actually use the texture**. An important detail that I wanted to add as a way to make the system look more appealing is a kind of **size falloff** at the edges of the grass (as described in Section 3.4). In other words, making the grass shrink as it approaches areas where there is no grass.

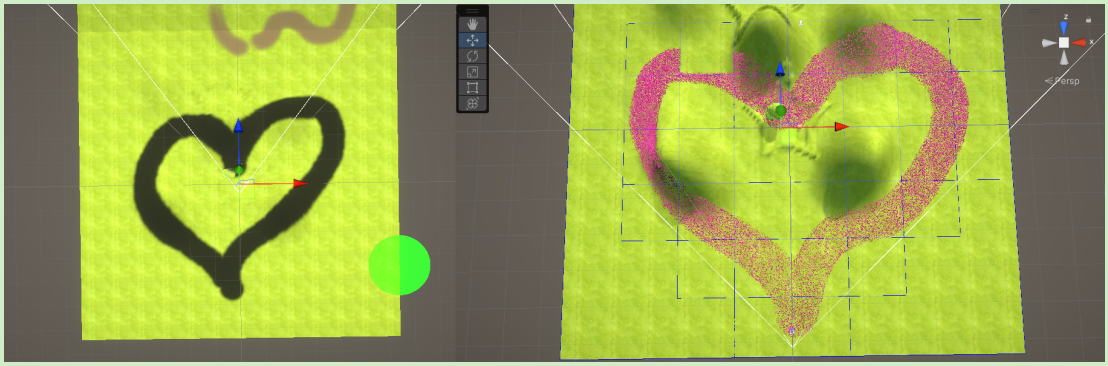

Figure 4.22: Terrain Painter in action

A straightforward way to achieve this is to use the **alpha value to drive the size**. This way, I had to add a new parameters to the compute buffers that held the blade information (which to this point only held position data, but as described in Section 3.3, we stored more information per blade), should be expanded to also hold size data, that way we can read in the compute shader from this texture and not only place grass where the texture has alpha, but also drive the size (see Figure 4.23). Then in the vertex shader, we can tell it to use the size as well as the position.

```
// Data structure for the outputbuffer
struct GrassData
{
    float3 position;
    float3 scale;
};
```

Figure 4.23: New grass data struct for the buffer

Finally we can wrap all of this up into a *prefab*, taking into account the relationship between components and the order in which textures and references are taken. This was a longer process into making a convenient tool, however, there is still room for improvement. To begin with, there is an annoying bug that breaks the control of the camera in the editor until another mouse event is triggered which can be very enfuriating. Another problem is the dependency to Unity's terrain, although it could be easily changed knowing the size of the terrain that would want to be used.

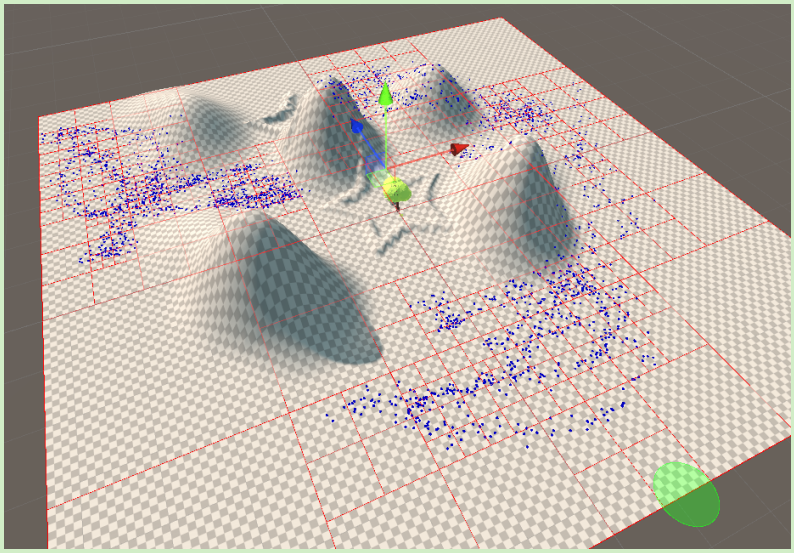

Figure 4.24: First quadtree test

4.4 World Partition

Now that we have a decent system with great looks and customization, we should start thinking about **optimization**. As of right now, the system draws all blades regardless of their position in the world (that means up to hundreds of thousands of blades). Because we are using GPU instancing, the GPU can't perform **frustum culling** for free (deleting all blades outside of the camera's view). Furthermore, if we want to scale the world, it is not a good idea to treat the grass as one big chunk. With these limitations, a great technique that has been at use since the original *Doom* [25] is space partitioning. As we are focusing on open worlds that do not necessarily need verticality and as one of the data structures that called my attention the most in the subject Game Engines, **quadtrees** have been chosen to divide the world into chunks. They offer efficient querying and are not too hard to implement (see Append B for the whole implementation).

First of all, I started gathering information about quadtree implementation and other matters. *Wikipedia* [29] and *The Coding Train*'s series of videos on the matter [24] proved to be very useful when developing a quadtree of my own. The first approach I took was very similar to *The Coding Train's*, creating custom classes for everything that is related to it. This served as a type of practice and learning period to understand how this data structures works and how I could take advantadge of it. When it finally worked (see Figure 4.24), the next step is working on a custom solution for the grass system.

This implementation of the quadtree stores **additional information at each node**, so that during runtime it can be accessed, as explained in Section 3.3. The main

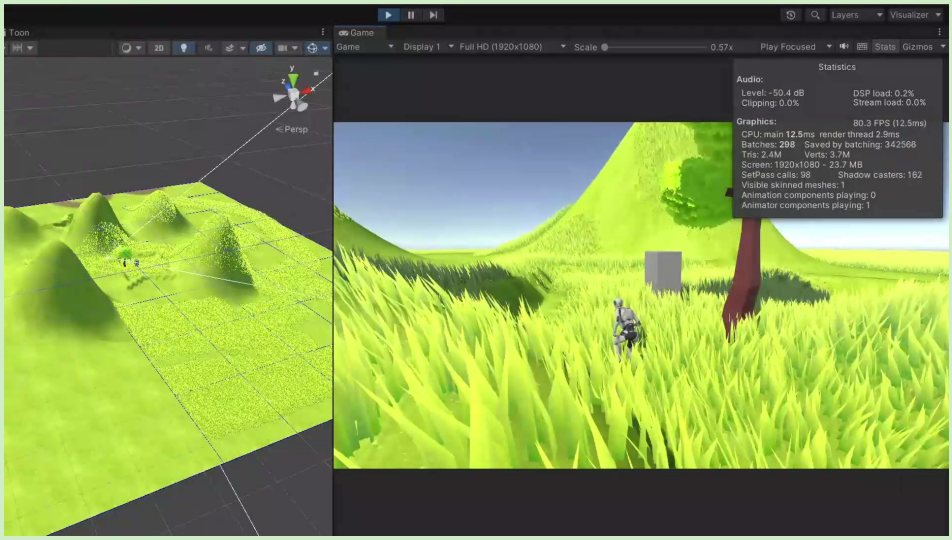

Figure 4.25: Quadtree culling

methods are: *Build()*, *SubdivideTexture()* and *TestFrustum()*.

Each quadtree is created by the *GrassMaster* at the start of the first frame, and it will create a parent quadtree for each *GrassPainter* game object (each chunk of terrain). Then, the *Build()* function is in charge of creating all of the tree, subdividing the mask texture only where it needs to (where there is detail). *SubdivideTexture()* creates smaller copies of the mask texture and the heightmap for each node that needs it. The condition to keep subdividing is given by the method *GrassTexture-ContainsAlpha()*, if a node does not have more information about the position, it is not subdivided further. If it has information, it is subdivided into four smaller chunks that cover the same area and the same test is done for them.

Now that we have created a quadtree, we can start **testing the frustum** against it. In the *Update()* function (executed each frame) of the grass master, we check the quadtrees for the visible nodes. This is done through the *TestFrustum()* function, which revursively checks if a node of the quadtree is inside the frustum, if it contains grass and if it's whithin the *quadtreeCutoffDistance*, a parameter that helps the user optimize the system. The nodes that are visible, are inserted to a list. Then this list is compared to the same list in the previous frame, and all the nodes that are not visible anymore, free up the memory allocated by their buffers. Then, for all visible nodes, if they have not been initialized yet, we allocate memory for their buffers and dispatch the positions compute shader. This way, we have a dynamic system that at any point only uses the memory that is necessary in exchange of generating positions each frame. This is great for performance, as only the grass near the player is rendered and the quadtree enables to query the chunks efficiently. The chunking of the grass can clearly be seen in Figure 4.25.

4.5 GPU Optimizations

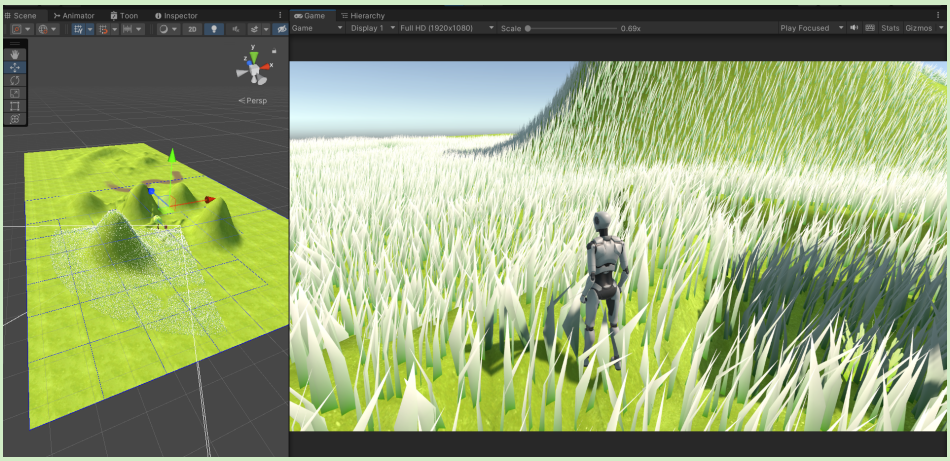

Figure 4.26: Gpu Optimizations

Even though using quadtrees for a low detail frustum culling method is quite optimized, we can really get a performance boost by culling more accurately to the **frustum** and even adding **LOD (level of detail)**. As we are using GPU instancing, none of these optimizations come 'for free' with the engine, so we have to implement them on our own.

For this goal, a **new compute shader** is in charge of the frustum culling. In this shader, we perform frustum culling the same way as *Acerola*'s his implementation found in GitHub [10], transforming the position to clipspace. We can even add a parameter for **distance culling**, which will eliminate grass positions farther away from that distance to the camera. This shader is executed at each visible chunk, and takes the original grass data buffer to cull the positions. A new buffer is generated, which is the actual one we will use to draw, update the arguments for the chunk and as the buffer for the material of the chunk (instead of the buffer we were using previously, finally in line with what was described in Section 4.1).

For **Level Of Detail** a new model had to be created. This is very simple task as we only need one quad with the correct normals (see Figure 4.27). As for the code structure, we always keep two buffers, one for the normal positions, and another one for the LOD positions. We can make use of the culling shader for this matter, as it already checks distances and the frustum. So at the end of the shader, we make one last comparison checking whether the distance to the camera is bigger than the LOD_DISTANCE parameter and appending the grass data to the main buffer or the LOD buffer. This way, in the end we have 3 different buffers at each node of the quadtree: one for the total data, one for the culled normal blades, and another one for the culled LOD blades. We call *Graphics.DrawMeshInstancedIndirect()* twice,

once for normal blades and another one for the LOD blades.

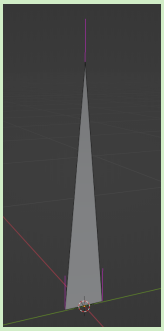

Figure 4.27: LOD Model

This process seems pretty straightforward now, as everything that has been described before fits everything needed for the GPU optimizations. However, during development it was not as easy and many of the forementioned design decision had to be changed in order to suit these GPU optimizations. Mainly, the use of AppendStructured buffers, the use of argument buffers that store information about the mesh and number of instances that will be rendered and the choice of using the drawing function *Graphics.DrawMeshInstancedIndirect()*, which was the one that worked best for my case. Things were, many times, discovered by experimentation and trial and error as it is hard to find documentation on these matters, even more so in Unity. This resulted in a frustating experience that took a longer time than this section evokes.

4.6 Player interaction

Finally, our grass is capable of running even in mobile devices. That is quite an achivement. However, when adding a character, moving around the environment feels quite dull and as we said in the beginning, videogames can offer something more to us. That means it is time to add some **player interaction**. In this case, the obvious thing to do is make the **player position displace the position of the grass**.

In order to do this, *Feeley*'s video on *God of War*'s vegetation [6] mentions how other games approach the subject. More specifically, he mentions how Uncharted uses a normal texture that follows the player to move the grass away from the player, according to the direction the normal texture stores. This looked like a nice idea so I applied the same concept.

First of all we need a **2D circle with the outfacing normals**. For this, I took an image from *Stack Overflow* [15], which has the perfect shape and directions we want. Now we need a system to tell the grass to move wherever the player is moving. This is done using a **render texture** as it can be seen in many tutorials, including one about snow displacement [17]. However all of these tutorials miss one thing, and that is to make the system **infinitely scalable**.

The system works using a **camera** that renders from the **top view** to the bottom to a **custom render texture** (see Figure 4.28). It will only render the normal textures that we want. This texture is then used to drive whatever we need inside a shader. As we can deduce, this will only work in the region where the camera is rendering. I was always curious to make this work on an **infinite scale** so I took this as an excercise to find the solution. What we can do is make the camera follow the player. Only

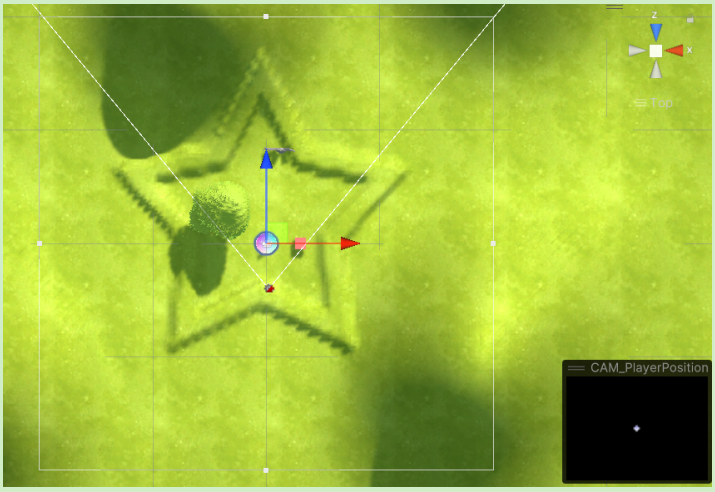

Figure 4.28: Player position system

doing this would not have any effect, so the next step is to in the shader, subtract the position of the player from the world coordinates. This way we align the blade's world UV with the player at all times. This is useful if we want to maintain some type of trail or keep some change behind the player. In our case, the camera follows the player, who has a plane attached with the texture of the normal circle, and these normals will push the vertices away from it. The effect can be seen in Figure 4.29.

Figure 4.29: Player position interaction

In the end, however, a **trail renderer from a particle system was also added, so the path the player walks through is left behind in the grass** (see Figure 4.30). This approach is closer to what *Feeley* describes for *God of War*. The particle properties of the system make the trail left behind come up slowly, so the effect is quite pleasant. There is, unfortunately a small glitch in the movement, due to the nature of Unity's trail renderer, where the end of the trail cuts off abruptly. This could be fixed in future revisions. There is one caviat to this system with the render texture though; we can only send vertical data of a certain amount of elements, because we need to take into account their heights if we don't want ghosts to be running on our grass. For that reason only the player's Y position is taken into account for the displacement of grass, masking the center of the render texture when the player is off the ground.

We can see a debug of the effect in Figure 4.31, where we can also appreciate how other objects can be added to the system.

Figure 4.30: Player position trail

Figure 4.31: Player position trail

4.7 Fluid Simulation

The final and greaest detail and one of the reasons this grass system is created in the first place, is to add **dynamic wind**.

As explained in both videos about *God of War*'s dynamic wind system [[16], [6]], a **fluid simulation** inside a **3D grid** is the backbone of the effect. Implementing a real-time fluid simulation is quite a challenge and so a lot of time went into researching about it and what options would be avaliable for me and my knowledge. The majority of papers on the problem, go deep into the mathematical structure of the algorithm. However, they also proved a high quality resource into grasping and getting an overview on how the simulation works. The most useful sources were *Mike Ash*'s '*Fluid Simulation for Dummies*' [4], NVIDIA's '*GPU Gems*' book entry on fluid simulation [11] and the foundation of real-time dynamic fluid's, *Jos Stam*'s paper [18]. Spending time trying to understand them was good, however, I never fully undertood them, so the pseudocode implementation in *Jakub's* master thesis proved to be really useful. The implementation is closely based on his. Nevertheless, as already stated in previous sections, a change was introduced when analizing *Ruper Renard*'s GDC talk on his implementation for *God of War* [16]. There, the team uses three **3D textures** instead of buffers, as in the end they proved to be faster. In the end it worked and got to use 3D textures for the first time making it a fruitful experience.

The implementation uses 2 sets of manually double-buffered single-channel textures: **velocity sources** and **velocities**. They both have copies for each axis and another copy for double-buffering. Double buffering is a technique used to reassure parallel tasks in the GPU do not interfere with a texture as we are reading and writing to it [27].

3 main steps are taken in the simulation: **adding forces**, **advection**, and **diffusion**. *Adding forces* modifies the sources textures. *Advection* is a property of water which moves the velocities and whatever is inside of it around. And finally *diffusion* 'dilutes' the contents around so that the state of the whole body is homogeneous. The implementation of each step will be omitted as it is quite complex and requires a lot of mathematical understanding that can be found in the references. Each of this steps is performed using chained compute shaders in order to maintain a tidy workspace. The result of these steps can be seen in Figure 4.32.

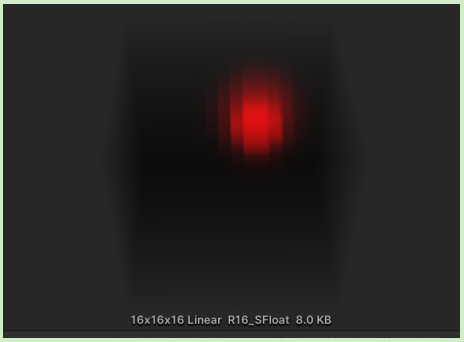

Figure 4.32: Velocity 3D texture in Z axis

A **3D voxel grid** in which each cell is 1x1x1 units is used as the simulation vol-

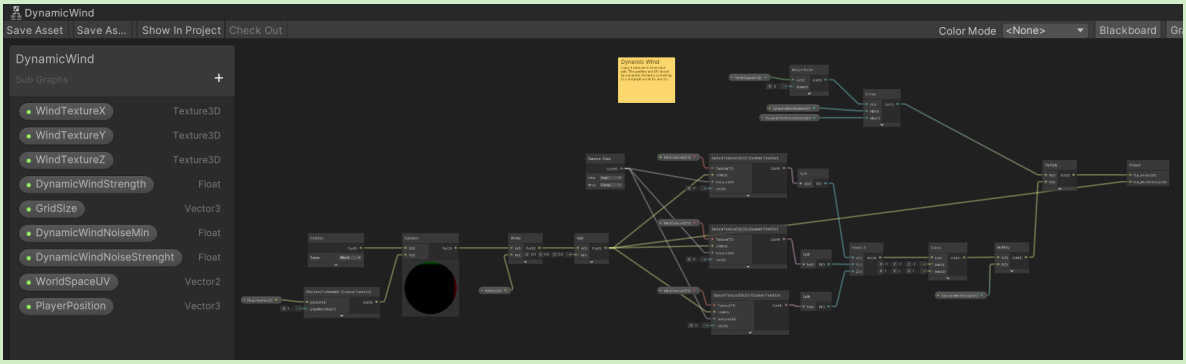

Figure 4.33: Vertex shader: dynamic wind subgraph

ume. It gives enough detail around the player and can be managed to be kept performant. For now, only a 16x16x16 grid has been tested, but as seen in God of War a 32x16x32 grid would be enough for far reaching objects. It is important to note, that it also should follow the player anywhere in the world, in a similar way to the position of the player affects the grass. However, it is more challenging here as we also have to take into account the position of the wind generators and the state of the fluid when we move. For the system follows the player although it **breaks** everytime the grid has to move, where we loose all data. This should be addressed in the future.

A topic we just flew by are **wind generators or motors**. In order to create wind and let the player interact with the simulation we need motors that add forces to the wind. In this case, only a directional motor is implemented. It adds a constant force in one direction but as seen in *God of War* there are plenty of possibilities to create motors with different effects. The motors are in world-space so converting to the grid's local position is needed when calculating their effect. They also have to be taken into account when moving the grid.

Finally, when we have the simulation working, we should add it to the **vertex shader** of whatever element we want to be affected by dynamic wind. In our case, the first element is the grass. Here, after all the calculations for the UV coordinates after the displacement of the grid in the world (similar to position), we sample from each of the velocities on each axis, clamp its value and add displacement to the vertex. We can also use parameters to scale the strength. An additional noise texture is scrolled on the output of the combined axis to give high frequency detail. Finally, this displacement is added to the static wind described in the animation Section 4.2. Everything in the graph that calculates the dynamic wind can be packed into a subgraph in order to keep a tidier shader graph (see Figure 4.33).

The final effect can be seen in Figures 4.34 and 4.35 where we can see that using the same node, we can apply dynamic wind to the tree model that I added to the scene, which uses another custom shader.

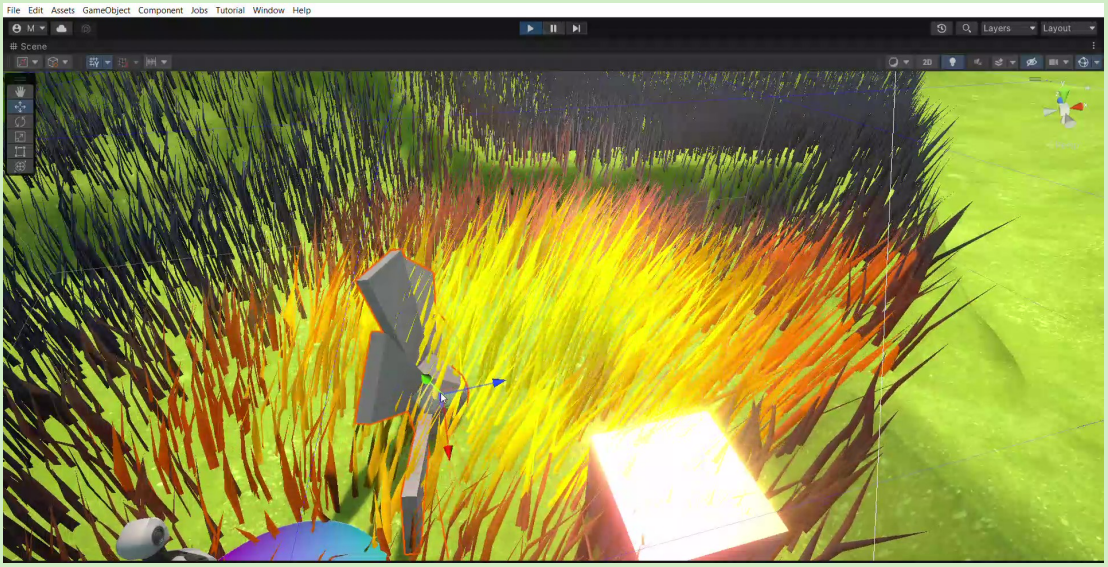

Figure 4.34: Fluid simulation debug on grass

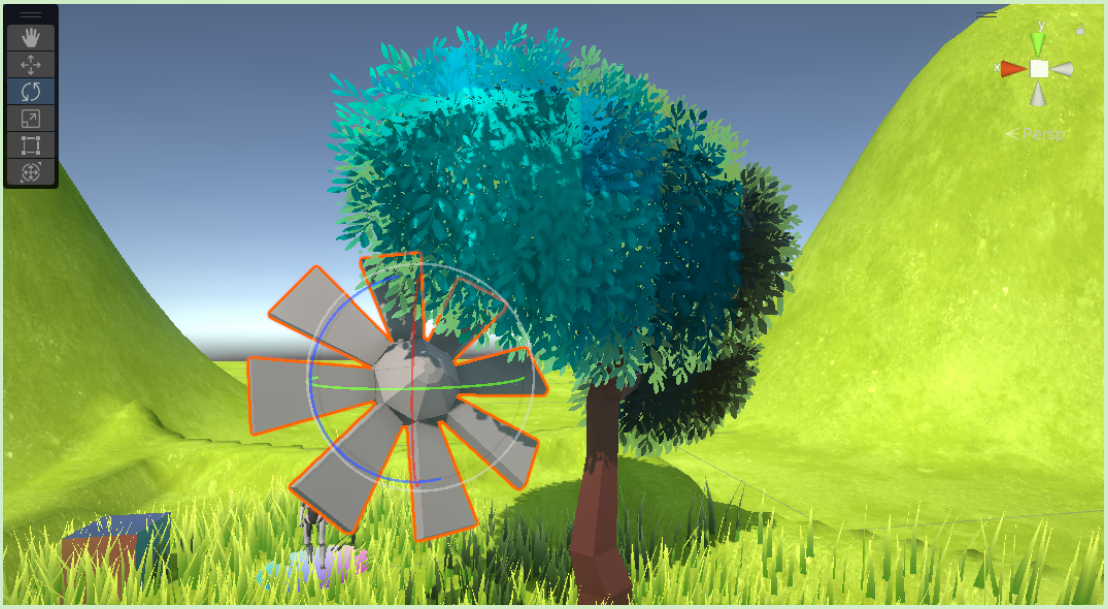

Figure 4.35: Fluid simulation debug on a tree

4.8 Results

In this section, we will outline the things that were accomplished, complimenting the information given in the past sections where each objective was explained.

First of all we can **compare the original table** 2.1 with the **times that were actually put in the work** 4.1. In red, the ones that took longer than expected, in blue, the ones that took less than expected. The comparison shows us that the grass related tasks were even harder than expected and many of the things that were created during development had to be tweaked many times, this resulted in an increase in their duration. The tasks related to gameplay were given a lower priority, changed by simpler and faster approaches. The wind system also took a long time, and as will be discussed promptly, it should still be in development. Finally, the documentation side, was pretty on point, although some more work than expected had to be put in.

Task	Type	Time (h)	True time (h)
Technical proposal	Documentation	5	5
Researching & writing the analysis and design document	Documentation	20	20
Instanced rendering of the grass model	Grass	20	27
Vertex shader for animation	Grass	10	25
Space partitioning for covering big fields	Grass	25	27
GPU optimizations (Frustum culling, LOD)	Grass	30	33
System for manual placement of the grass	Grass	15	17
Basic 3rd person character controller	Gameplay	10	5
Position of player interacts with grass	Grass	20	22
Fluid simulation in a 3D grid	Wind	50	52
Making the grid follow the player	Wind	20	26
Creation of wind generators	Wind	15	5
Creation of player sounds (discarded)	Gameplay	10	0
Implementation of sounds using FMOD (discarded)	Gameplay	15	0
Writing final memory	Documentation	40	44
Preparation of final presentation (expected)	Documentation	10	15
TOTAL		**300**	**313**

Table 4.1: Tasks - True times

	No optimization (fps)	+ Quadtree culling (fps)	+ GPU optimization (fps)
Editor	75	170	150
PC Build	85	110	140
Android Build	-	15	25

Table 4.2: Performance comparison between platforms and optimizations

All of the objectives were achieved in various degrees of completition. We have a flexible grass rendering system, a useful tool that enables customization, static and dynamic wind systems and everything is well optimized. All of them do still need some work to get right, as there are some big problems in some of the systems that make the tools not so user friendly, but it can be worked on. Mainly, the placement tool bugs the input in the editor and the user cannot move around the map. Also, the grid of the wind system does not move correctly across the world, so this should also be fixed.

The performance of the system can be analyzed by **testing in different platforms** with different optimizations. Table 4.2 shows the comparison between frame speeds. In these examples, the editor and PC build use the exact same parameters and include the fluid simulation, the Android build uses the same density but the culling of quadtrees and cutoff is closer to the camera and the wind simulation is omitted. The size of the terrain is 128*256 (to make a fair test with the version without optimization).

Firstly we can observe that, with **no type of optimization**, the performance is not ideal, hovering around *70 to 90* frames per second in both the build and the editor (see Figure 4.36). As there is no type of customization other than rendering less grass (which is not what we want), an Android device is not even capable of running the app.

In the **editor**, with the computer described in Section 2.2, the performance is *170 fps* average using only quadtree culling. When we add tighter frustum culling, LOD and cutoff, the performance is reduced by an average of 20 frames per second, around *150 fps* as seen in Figure 4.37.

On the other hand, for the **PC build**, there is an increase in performance when adding more optimizations, which is what would be expected (see Figure 4.38).

Looking at the **Android build**, we can see an average of *25 fps*, as seen in Figure 4.39, when GPU optimizations are added, otherwise the game becomes barely playable at an average of *15 fps*.

It is interesting to observe that, in the editor, the performance takes a hit when using gpu optimizations, but andoid benefits greatly. We have to note that while the optimizations can be expensive, in weaker graphical computers and mobile devices where drawing is even more expensive, reducing draw calls avoids the bottleneck of GPU power.

Finally, we can assure that the tools are reusable and customizable, and I will definitely use them in the future as an asset for games. If all of the rough edges were smoothed out, it could even be **sold as an asset in Unity's asset store**. The whole project can be found in GitHub and a build is also available to download in Append A.

All of the parameters available to the user are shown in Figure 4.40. Some examples of the looks that can be achieved with the grass tools are presented in Figures 4.41, 4.42, 4.43, 4.44, 4.45, 4.46 and 4.47. It is interesting to note that the bike shown in Figure 4.47 is also modelled for this project.

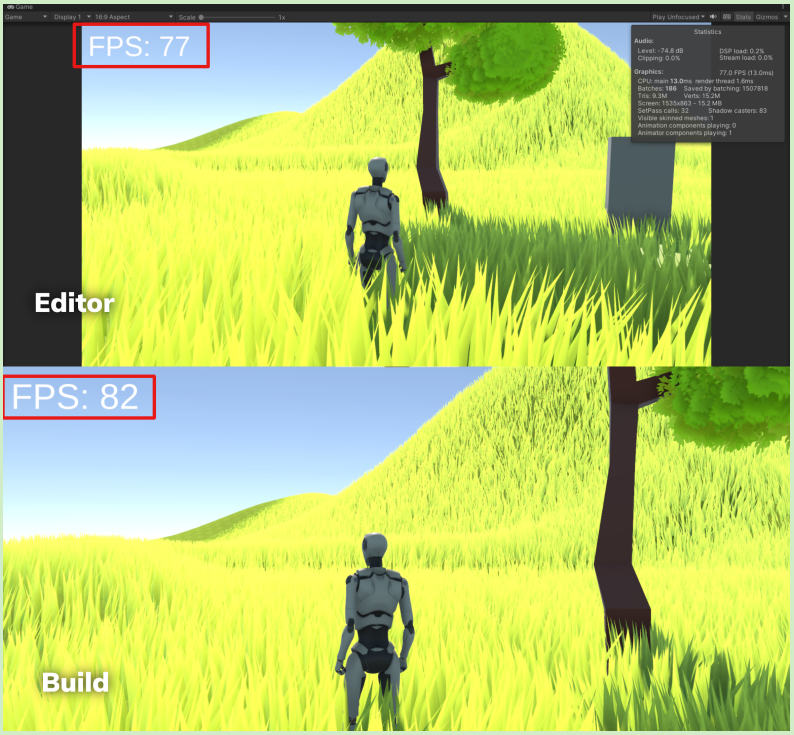

Figure 4.36: Comparison between editor and build without optimization

Figure 4.37: Editor optimizations comparison

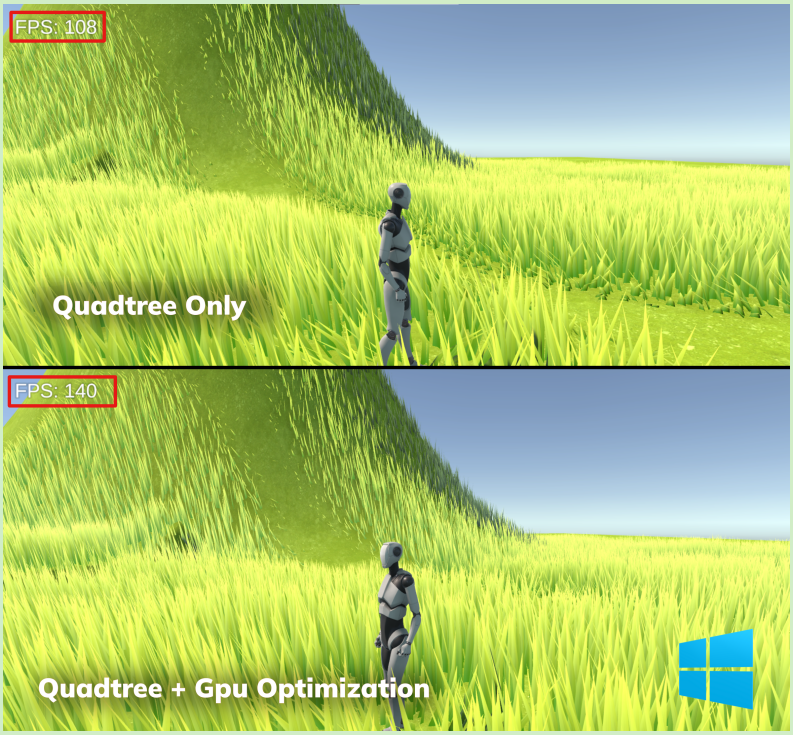

Figure 4.38: PC build optimizations comparison

Figure 4.39: Android optimizations comparison

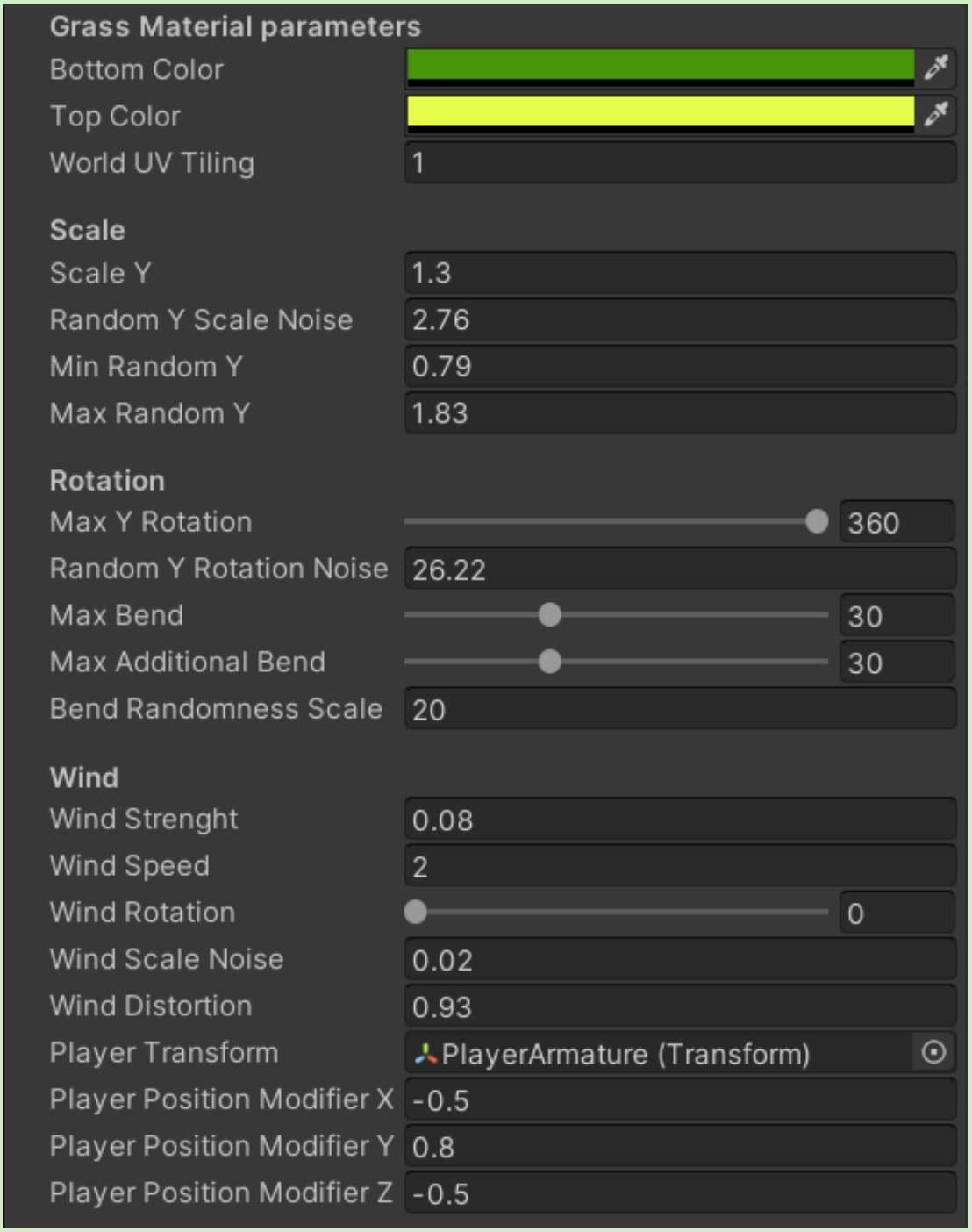

Figure 4.40: Default grass parameters

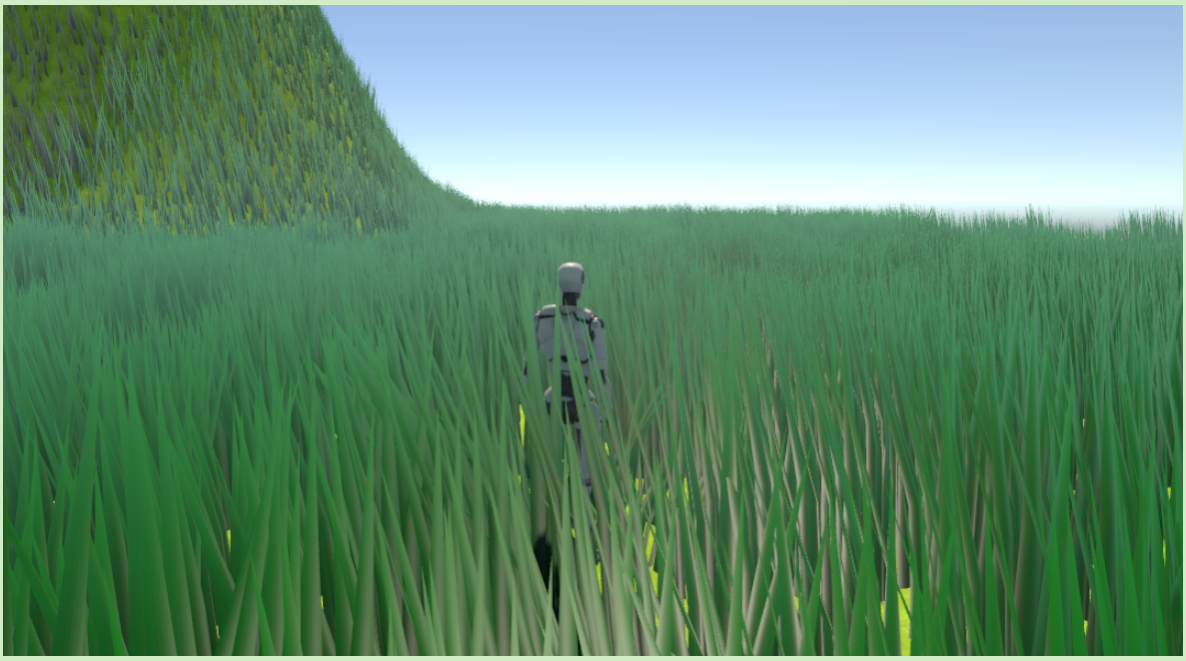

Figure 4.41: Tall grass, realistic colour

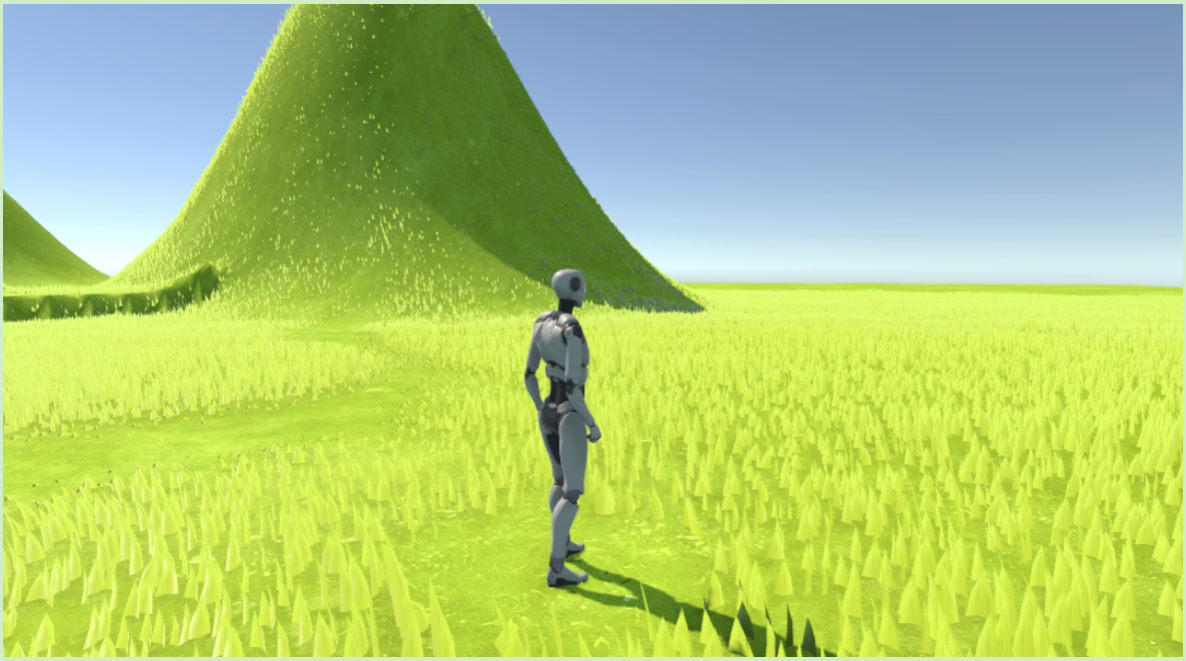

Figure 4.42: Short grass

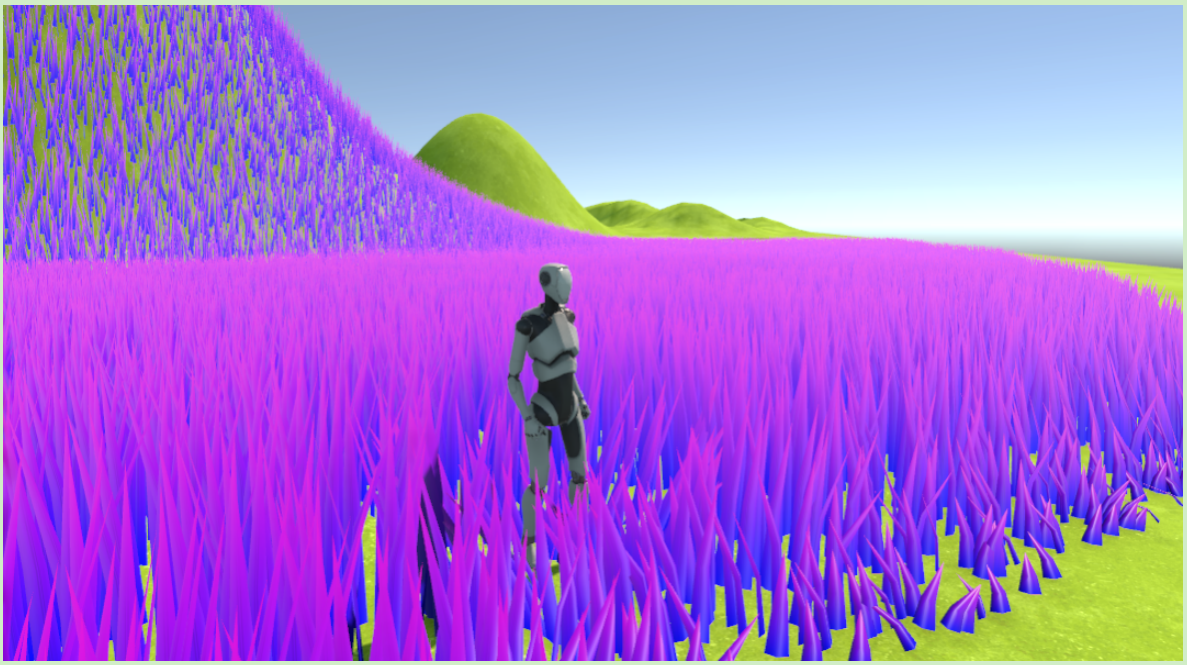

Figure 4.43: Stylized purple/blue grass

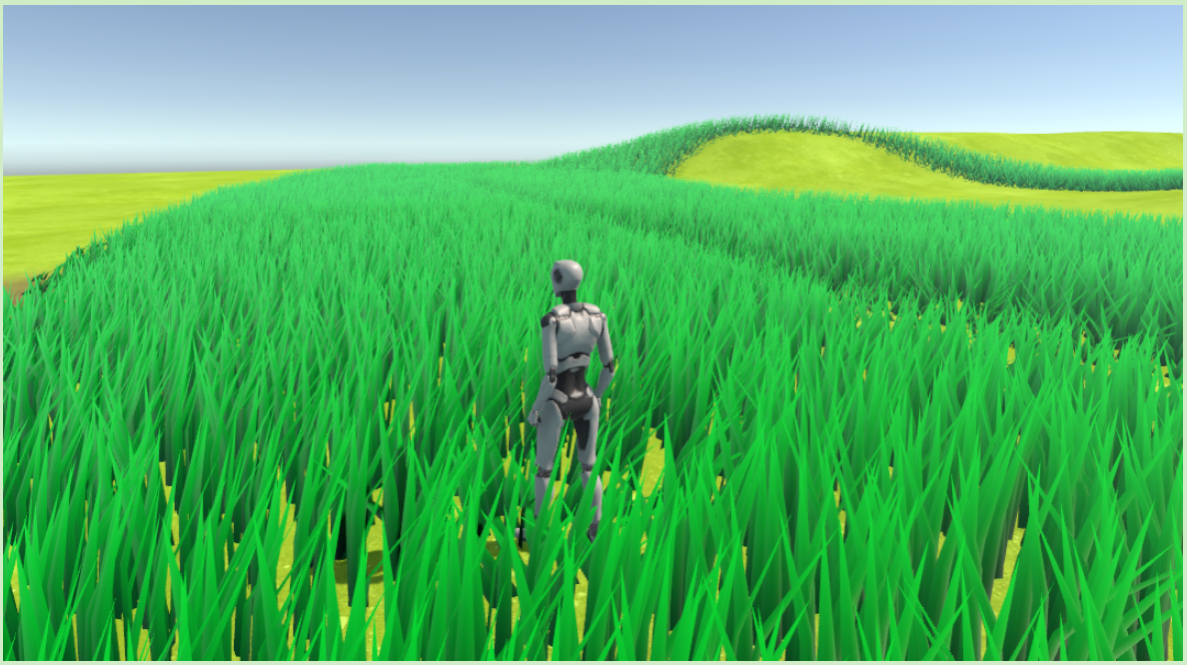

Figure 4.44: Normal height, strong random bend, path placement

Figure 4.45: Yellow grass, small bend

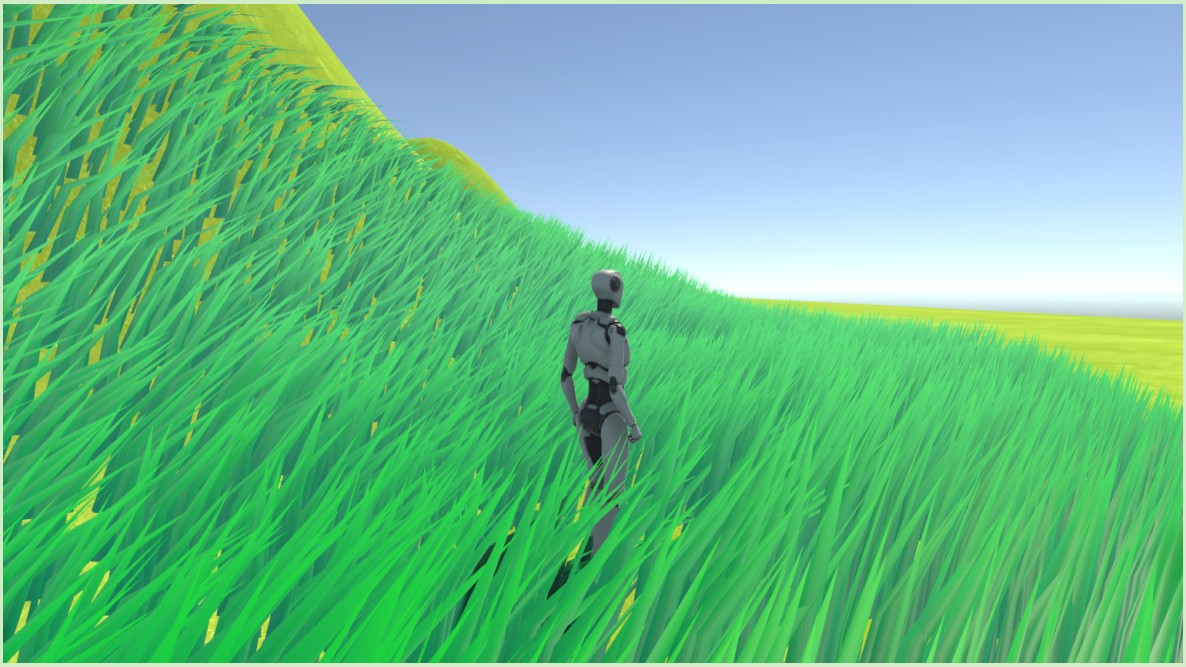

Figure 4.46: Bend in the direction of the wind

Figure 4.47: Artistic composition and object 'collision'

Chapter 5

Conclusions and Future Work

Contents

5.1 Conclusions

Very hard work from my side had to be put inside this project, I've been working on it since the end of 2022. It's incredible to think that so much effort has been put in and yet there are still so many things to fix and polish in order to have a solid system. The knowledge gained in the degree was useful up to some point, as many of the things that were discussed here are never touched upon. It was useful to know concepts and ideas that then I could research on and implement, but there is still so much to learn. And that is one of the things that I take with me from this experience, you can never stop learning.

Even if it will not be perfect anytime soon I am proud of this and will be showing it off in games and other types of media such as videoclips and such for my musical persona *Pitarcus*. Besides, it is also a really eye catching project to show off, specially in the technical art department, a role I'm very interested in for my future professional career.

5.2 Future work

Even though the system is pretty well optimized, we have to be aware that the vertex shader is very expensive right now. Generating noises for every vertex for each blade of grass is not fast at all. A really simple trick that would fix this is to **precompute the noises**, as in using textures instead of generative noises. This way, the system would gain quite some speed. Besides, time should be spent fixing some **bugs** involving the disappearance of chunks at seemingly random moments and the blocky movement of the fluid simulation when following the player.

As mentioned in the previous section, if the bugs were fixed, the **tools could be sold as an asset in Unity's Asset Store** as they would make for an eye catching system that many people would want to put their hands on. **Additional parameters** could be added such as dynamic wind strength and even more flexibility in the fragment shader to affect the looks of the grass with specular color or even transulency for example. Some **other types of vegetation** should also be generated to create a less monotonous field of grass. This could include small leaves, clumps of flowers, bushes, etc.

Other types of natural phenomena could be developed over the fluid simulation, such as water or smoke. These would need to have a custom voxel renderer so the user can visualize what appears on screen. Water is a very complex matter although it also interests me.

As mentioned in the work motivation, I would love to **develop a game using this technology**. I've been gathering ideas for a game where the player takes on the role of the princess of a nature-based human-like race who wakes up in a desolated kingdom and she will have to bring back the life of all vegetation. It is quite a long project and probably will need fixes, reworks and extensions of the systems created here but I will push towards this idea. A small team should probably be gathered for this, as I discovered through the game I created in *Game Engines*, the whole process is quite hard, but I love it and learning all about it makes me closer to being a game director.

Bibliography

[1] Acerola. How do games render so much grass? https://youtu.be/Y0Ko0kvwfgA. Accessed: 2023-04-28.

[2] Acerola. Modern foliage rendering. https://youtu.be/jw00MbIJcrk?t=370. Accessed: 2023-04-30.

[3] Minions Art. Grass update part 1, painting tool improvements. https://www.patreon.com/posts/61761853. Accessed: 2023-05-02.

[4] Mike Ash. Fluid simulation for dummies. https://mikeash.com/pyblog/fluid-simulation-for-dummies.html. Accessed: 2023-05-04.

[5] Catlike Coding. Compute shaders. https://catlikecoding.com/unity/tutorials/basics/compute-shaders/. Accessed: 2023-05-02.

[6] Sean Feeley. Interactive wind and vegetation in 'god of war'. https://youtu.be/MKX45_riWQA. Accessed: 2023-04-28.

[7] Brian UpRoom Games founder. Procedural terrain in unity. https://www.uproomgames.com/dev-log/procedural-terrain. Accessed: 2023-04-30.

[8] Sony Games. Ghost of tsushima web page. https://www.playstation.com/es-es/games/ghost-of-tsushima/. Accessed: 2023-05-16.

[9] Glassdoor. Salary details for a software developer at santa monica studio. https://www.glassdoor.com/Salary/Santa-Monica-Studio-Software-Developer-Salaries-E716413_D_KO20,38.htm?selectedLocationString=N%2C1&filter.jobTitleExact= Accessed: 2023-06-14.

[10] Garrett Gunnell. Grass. https://github.com/GarrettGunnell/Grass. Accessed: 2023-05-04.

[11] Mark J. Harris. Chapter 38. fast fluid dynamics simulation on the gpu. https://developer.nvidia.com/gpugems/gpugems/part-vi-beyond-triangles/chapter-38-fast-fluid-dynamics-simulation-gpu. Accessed: 2023-05-04.

[12] JAKUB MIČKA. Voxel-based fluid simulation in unity.

[13] Santa Monica. God of war. https://www.playstation.com/es-es/games/god-of-war/. Accessed: 2023-05-02.

[14] Nintendo. Breath of the wild. https://www.zelda.com/breath-of-the-wild/es/. Accessed: 2023-05-02.

[15] Rabbid76. Drawing a sphere normal map in the fragment shader. https://stackoverflow.com/questions/53271461/drawing-a-sphere-normal-map-in-the-fragment-shader. Accessed: 2023-05-04.

[16] Rupert Renard. Wind simulation in god of war. https://youtu.be/dDgyBKkSf7A?t=297. Accessed: 2023-04-30.

[17] Daniel Santalla. Interactive snow in unity. https://twitter.com/danielsantalla/status/1391135820229222401. Accessed: 2023-05-04.

[18] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, volume 18, page 25, 2003.

[19] Unity Technologies. Computebuffertype.append. https://docs.unity3d.com/ScriptReference/ComputeBufferType.Append.html. Accessed: 2023-05-02.

[20] Unity Technologies. Gpu instancing. https://docs.unity3d.com/Manual/GPUInstancing.html. Accessed: 2023-05-02.

[21] Unity Technologies. Graphics.drawmeshinstancedindirect. https://docs.unity3d.com/ScriptReference/Graphics.DrawMeshInstancedIndirect.html. Accessed: 2023-05-03.

[22] Unity Technologies. Mesh.indexformat. https://docs.unity3d.com/ScriptReference/Mesh-indexFormat.html. Accessed: 2023-05-02.

[23] Unity Technologies. Starter assets - third person character controller. https://assetstore.unity.com/packages/essentials/starter-assets-third-person-character-controller-196526. Accessed: 2023-05-01.

[24] The Coding Train. Coding challenge #98.1: Quadtree - part 1. https://youtu.be/OJxEcs0w_kE. Accessed: 2023-05-03.

[25] Doom Wiki. Doom rendering. https://doomwiki.org/wiki/Doom_rendering_engine. Accessed: 2023-05-03.

[26] OpenGL Wiki. Compute shader. https://www.khronos.org/opengl/wiki/Compute_Shader. Accessed: 2023-05-02.

[27] Wikipedia. Buffer múltiple. https://es.wikipedia.org/wiki/Buffer_m Accessed: 2023-05-04.

[28] Wikipedia. Quadtree. https://en.wikipedia.org/wiki/Quadtree. Accessed: 2023-04-30.

[29] Wikipedia. Quadtree. https://en.wikipedia.org/wiki/Quadtree. Accessed: 2023-05-03.

[30] Eric Wohllaib. Procedural grass in 'ghost of tsushima'. https://youtu.be/Ibe1JBF5i5Y. Accessed: 2023-05-02.

Appendix A

Important links

Repository: https://github.com/Pitarcus/Grass-FDP-2023
Build: https://drive.google.com/file/d/1toeKujmMQOrkvTsd9eYBCclsO_FRemsk/view?usp=share_link

Appendix B

Source code

Quadtree Implementation

Listing B.1: Quadtree for grass implementation

```
// ----------- GRASS QUADTREE ----------------
public class GrassQuadtree : IEquatable<GrassQuadtree>
{
    public AABB boundary;
    public int maxDepth;
    public int currentDepth;
    public bool containsGrass;
    public bool hasBeenSet;

    public Texture2D grassMask;
    public Texture2D heightMap;
    public Material material;
    public Material materialLOD;
    public ComputeShader grassCompute;
    public uint numberOfGrassBlades; // Max number of visible blades in the
        node
    public uint numberOfInstances;  // Actual number of instances

    public ComputeBuffer grassDataBuffer;   // Fist data buffer with all the
        positions
    public ComputeBuffer culledGrassDataBuffer;
    public ComputeBuffer culledGrassDataBufferLOD;
    public ComputeBuffer argsBuffer;
    public ComputeBuffer argsLODBuffer;

    public GrassQuadtree northWest = null;
    public GrassQuadtree northEast = null;
    public GrassQuadtree southWest = null;
```

```
27      public GrassQuadtree southEast = null;
28
29      public bool subdivided = false;
30
31      public GrassQuadtree(AABB boundary, int currentDepth, int maxDepth,
            Texture2D grassMask, Texture2D heighMap)
32      {
33          this.boundary = boundary;
34
35          this.maxDepth = maxDepth;
36
37          this.currentDepth = currentDepth;
38
39          this.grassMask = grassMask;
40
41          this.heightMap = heighMap;
42      }
43
44      public void Subdivide()
45      {
46          float x = boundary.p.x;
47          float y = boundary.p.y;
48          float w = boundary.halfDimension / 2;   // Half the width of the parent
49
50
51          AABB nw = new AABB(x - w, y + w, w);
52          northWest = new GrassQuadtree(nw, currentDepth + 1, maxDepth,
                SubdivideTexture(grassMask, false, true, true), SubdivideTexture(
                heightMap, false, true, false));
53
54          AABB ne = new AABB(x + w, y + w, w);
55          northEast = new GrassQuadtree(ne, currentDepth + 1, maxDepth,
                SubdivideTexture(grassMask, true, true, true), SubdivideTexture(
                heightMap, true, true, false));
56
57          AABB sw = new AABB(x - w, y - w, w);
58          southWest = new GrassQuadtree(sw, currentDepth + 1, maxDepth,
                SubdivideTexture(grassMask, false, false, true), SubdivideTexture(
                heightMap, false, false, false));
59
60          AABB se = new AABB(x + w, y - w, w);
61          southEast = new GrassQuadtree(se, currentDepth + 1, maxDepth,
                SubdivideTexture(grassMask, true, false, true), SubdivideTexture(
                heightMap, true, false, false));
62
63          subdivided = true;
64      }
65
66      private Texture2D SubdivideTexture(Texture2D texture, bool positiveX, bool
            positiveY, bool isPosition)
67      {
68          Texture2D resultTexture;
69          if (isPosition)
70          {
```

```
71              resultTexture = new Texture2D(texture.width / 2, texture.height /
                    2, TextureFormat.RGBA32, false);
72              resultTexture.wrapMode = TextureWrapMode.Clamp;
73              resultTexture.filterMode = FilterMode.Bilinear;
74          }
75          else
76          {
77              resultTexture = new Texture2D(texture.width / 2, texture.height /
                    2, TextureFormat.R16, false);
78              resultTexture.wrapMode = TextureWrapMode.Clamp;
79              resultTexture.filterMode = FilterMode.Bilinear;
80          }
81
82          int startX, startY;
83
84          if (positiveX)
85              startX = texture.width/2;
86          else
87              startX = 0;
88
89          if (positiveY)
90              startY = texture.height / 2;
91          else
92              startY = 0;
93
94          for(int y = startY; y < startY + texture.height / 2; y++)
95          {
96              for(int x = startX; x < startX + texture.width / 2; x++)
97              {
98                  resultTexture.SetPixel(x % (texture.width / 2), y % (texture.
                        height / 2), texture.GetPixel(x, y));
99              }
100         }
101
102         resultTexture.Apply();
103
104         return resultTexture;
105     }
106
107     private bool GrassTextureContainsAlpha()
108     {
109         for (int y = 0; y < grassMask.height; y++) // Loop through the size of
                the mask
110         {
111             for (int x = 0; x < grassMask.width; x++)
112             {
113                 Color currentPixel = grassMask.GetPixel(x, y);
114                 if (currentPixel.a > 0.1f)
115                 {
116                     containsGrass = true;
117                     return true;
118                 }
119             }
120         }
```

```
121         containsGrass = false;
122         return false;
123     }
124 
125     // Subdivide the whole quadtree at the same time taking into account the
        max depth
126     public void Build()
127     {
128         // Only keep subdividing if there is alpha (grass) in the texture - The
              last nodes should also test to get the correct bool OMFG!!!!!!!!!
129         if (!GrassTextureContainsAlpha())
130         {
131             return;
132         }
133         if (currentDepth < maxDepth - 1)
134         {
135             this.Subdivide();
136 
137             northEast.Build();
138             northWest.Build();
139             southEast.Build();
140             southWest.Build();
141         }
142     }
143 
144     // Test the frustum against a quadtree, only visible quadtrees with grass
        will appear
145     public bool TestFrustum(Vector3 cameraPosition, float leafCutoffDistance,
        float quadtreeCutoffDistance, Plane[] frustum, ref List<GrassQuadtree>
        validQuadtrees)
146     {
147         if(!boundary.IsOnFrustum(frustum))
148         {
149             return false;
150         }
151 
152         if (!containsGrass)
153         {
154             return false;
155         }
156         if (Vector3.Distance(cameraPosition, new Vector3(boundary.p.x, 10,
            boundary.p.y)) > quadtreeCutoffDistance)
157         {
158             return false;
159         }
160 
161 
162         // Quadtree is in frustum, in distance && contains grass
163         if (subdivided)
164         {
165             if (northWest.TestFrustum(cameraPosition, leafCutoffDistance,
                quadtreeCutoffDistance, frustum, ref validQuadtrees) |
166                 northEast.TestFrustum(cameraPosition, leafCutoffDistance,
                    quadtreeCutoffDistance, frustum, ref validQuadtrees) |
```

```
167                 southEast.TestFrustum(cameraPosition, leafCutoffDistance,
                        quadtreeCutoffDistance, frustum, ref validQuadtrees) |
168                 southWest.TestFrustum(cameraPosition, leafCutoffDistance,
                        quadtreeCutoffDistance, frustum, ref validQuadtrees))
169             {
170                 return false;
171             }
172             else
173             {
174                 if (Vector3.Distance(cameraPosition, new Vector3(boundary.p.x,
                        10, boundary.p.y)) > leafCutoffDistance)
175                 {
176                     return false;
177                 }
178                 validQuadtrees.Add(this);
179                 return true;
180             }
181         }
182         else
183         {
184             if (Vector3.Distance(cameraPosition, new Vector3(boundary.p.x, 10,
                    boundary.p.y)) > leafCutoffDistance)
185             {
186                 return false;
187             }
188             validQuadtrees.Add(this);
189             return true;
190         }
191     }
192
193     public bool Equals(GrassQuadtree other)
194     {
195         return this.boundary.p.x == other.boundary.p.x && this.boundary.p.y ==
                other.boundary.p.y;
196     }
197 }
```