Building a Grass Shader in Unity Shaderlab

ROSKILDE UNIVERSITY

6. SEMESTER, SPRING 2019

GROUP NUMBER: S1925626169

SUPERVISOR: FINN GUSTAFSSON

AUTHORS:

Michael Piepgras Neergaard	59729	mipine@ruc.dk
Bertram Pelle Metcalf	59747	bpellem@ruc.dk
Alexander Varnich Hansen	59762	varnich@ruc.dk
Jeppe Stougaard Faber	59774	jsfaber@ruc.dk
Nils Müllenborn	61462	nimu@ruc.dk

Abstract

This paper details an implementation of a 'grass' shader that attempts to imitate grass-like terrain, using real-time rendering. The grass shader was implemented using various shaders in the Direct3D 11 rendering pipeline. Three distinct approaches to this was explored, prototyped, and compared with the purpose of examining their individual advantages and disadvantages.

These approaches are as follows:

- A model based approach using a vertex and fragment shader to colour pre-defined, 3D grass models.
- A plane based approach utilizing a fragment shader to apply a 2D grass texture on to three intersecting planes.
- A point based approach using a geometry shader to generate tetrahedral blades of grass based on a simple mesh of individual points.

A test was performed to gauge the overall performance of each prototype by measuring the generated frames per second(FPS) under similar circumstances.

The point-based approach was chosen for further development, and laid the basis for our current implementation.

The final implementation iterated upon the point-based prototype and added a 'wind'-effect that made the grass appear to sway in the wind. The wind-effect was implemented by rotating the tip of each blade of grass around its center. The rotation was implemented using *Rotation Matrices*, although *Quaternion Rotation* was also tested, but it proved ineffective due to the structure of our implementation.

Lastly a next-iteration is proposed for our grass shader that seeks to implement distance-based, dynamic Level-Of-Detail tessellation.

Contents

1 Introduction

Computer-generated graphics are essential to digital media as we know it today. Many big, block-buster movies rely heavily on digital image-processing to provide evermore impressive experiences in cinemas. Video games as a media is almost entirely reliant on computer graphics; the modern understanding of a 'game' has come to refer to experiences which 'worlds' are entirely comprised of such graphical representation. There's just one problem with this reliance: Generating images is an extremely computationally expensive process. A single image on a modern computer screen is comprised of millions of pixels, whose position and color must be calculated individually from the mathematical description of that image. Combine this with the fact that modern games expect at least 30 images to be displayed every second, and it becomes clear that conventional computation is not suited for this task.

Various methods have been used to alleviate this computation, but none have been more effective than the use of dedicated graphics hardware like the Graphics Processing Unit(GPU) in modern computer architecture: Through the power of parallel processing, the GPU can manage the large operations that computer graphics require, in a fraction of the time that traditional processing methods would take.

Of course, the images that a GPU generates must be influenced by other parts of the computer. The interaction between the Central Processing Unit(CPU) and the GPU is governed by programs called Graphics Application Programming Interfaces (Graphics API's). The main purpose of a Graphics API is to simplify the process of communicating with the GPU itself, but Graphics API's also implement a variety of other functionalities, one of the most prominent being Shader programs.

Shader programs or *Shaders* are programs that can be used to influence the image-creation process of the GPU, and are used to great effect in many productions that utilize digital graphics manipulation.

In this project, we explore the field of shaders and how they can be utilized to generate grass terrains. To do this, we will implement so-called 'grass' shaders, that attempts to imitate real grass. Through these examples we will explore various topics relating to shaders in general, as well as present the decisions and considerations that we have made in our implementation process.

1.1 Problem statement

How can we use shaders to generate terrains of grass in the Unity 5 game engine, for use in a video-game context?

- Which shader types can be used for this and how?
- How can we improve the practicality of the terrain generation?
- Which features can we implement to make the grass terrain more closely imitate grass?

1.2 Scope

This study can be used as a written introduction to shader programming. The project is limited to the following scope:

- A minimum understanding of computer science terminology will be required to fully grasp this study.
- It is not necessary to understand the underlying hardware architecture of shaders, to understand shaders. Therefore it will not be part of this study. There will only be used common hardware terminology during the tests.
- The rendering pipeline is a very complex subject. However a detailed knowledge hereof is not necessary when trying to understand the basics of shaders. Therefore this study will be limited to explaining the rendering pipeline at a conceptual level and describe only the parts which are most important to shaders.
- While a large part of shader functionality is standardized, many parts of the rendering pipeline remain API-specific. Therefore this study is limited to the Direct3D 11 API and also only hardware which supports the same API.
- We have chosen to use Unity as the application to program our shaders in. This is because Unity delivers a functional and versatile environment to program and develop shader programs in. Unity has a built-in tool to work with shaders called ShaderLab, that lets the user program the different types of shaders. This lets the user easily access the shader programs while not having to cope with the setup required to execute and view the shader programs.
- Additionally to the previous point, while Unity is a cross-platform application and it works on most operating systems, there still remain cross-platform issues and therefore this study is limited specifically to working with Unity 5 on Windows 10.
- Finally while *compute* shaders are technically part of the rendering pipeline, they will not be included in this study since they are too broad a subject.

2 | The Rendering Pipeline

Before we dive into explaining in detail, we must ensure a certain level of familiarity with some central concepts of the rendering pipeline. This chapter is dedicated to an explanation of these.

We start off by outlining some basic terminology and principles of graphics processing, this includes rendering, pipelines in software development and graphics API's. Following that, we consider the fundamentals of a rendering pipeline and the major concepts required to understand how the rendering pipeline works.

2.1 What is 'Rendering'?

A central term in any type of graphics processing is 'rendering'. In simple terms, rendering is the process of creating images from sets of instructions. Rendering is what modern GPU's are created to do.

In general, we distinguish between two types of rendering: real-time rendering and pre-rendering.

1. Real-time rendering is exactly what it sounds like; rendering done in real time. This type of rendering is the prime method of rendering interactive graphics and games. Images are generated and displayed on the fly and with very high frequency. Rendering is a complicated task and is, potentially, very resource-demanding. As a result, real-time rendering is usually carried out on the GPU and must always take into account the computational complexity of its implementation. [Sli19]
2. Pre-rendering, on the other hand, is a rendering technique that relies on pre-generating images before they are displayed. This technique is most often used in movie-productions where no real-time interaction occurs with the graphics themselves and images can be planned out ahead of time. Pre-rendering is best suited for situations where time is not a concern, and as a result, allows for more computational power to be put into each image generated which in turn allows for more complex effects. Because of the potential complexity of its operations and the lack of time constraints, pre-rendering is usually carried out by the CPU rather than the GPU. [Sli19]

2.2 What is a 'Pipeline'?

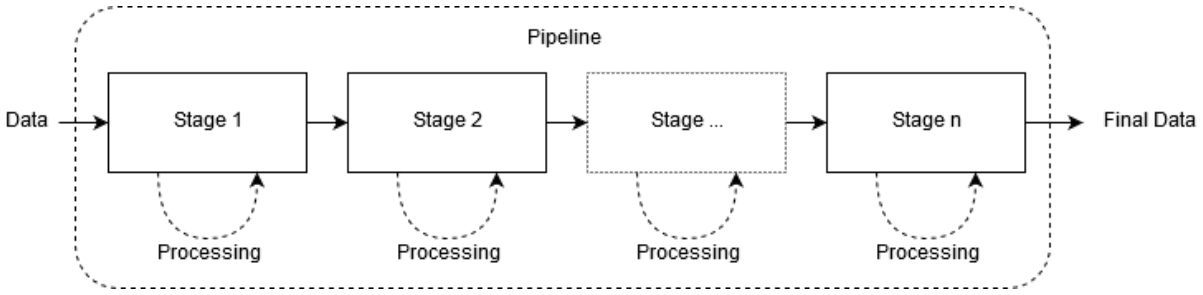


Figure 2.1: Overview of the pipeline design pattern.

A pipeline is a classic software design pattern that centers around the processing of data. A pipeline consists of a series of processing elements, called *steps* or *stages*, each with their own responsibilities and capabilities in handling the data they receive. Data is originally passed to the first stage in the series and is then sequentially taken through each stage in the pipeline. Each stage can process the data it receives in a unique way, and are sometimes made to change the data it receives, in either its contents or its form. In this way, complex tasks can be broken up into less complex, individual steps. A good analogy to the pipeline design pattern would be a car assembly line, constructing a car from basic car parts, through a series of isolated steps. [VBT95]

The rendering pipeline is simply a specialized example of this concept, with the purpose of drawing images.

2.3 What are Graphics API's?

Direct3D, OpenGL and Vulkan are names, often mentioned in graphics-rendering contexts. These are the names of three very popular Graphics API's in modern computing. Graphics API's are programs that communicate with GPU drivers to provide rendering-capabilities to computer programs. By accessing functionalities in these graphics API's, a program becomes capable of interacting with the GPU, most often with the purpose of drawing images to the computer screen.[Dev18]

2.4 The Rendering Pipeline

A rendering pipeline is a conceptual model description of the processes that a Graphics API uses to perform rendering operations.[Mic18g] Each Graphics API implements different processes, structured in different ways, but some elements of the pipeline can be somewhat generalized. This section is meant to provide a generalized outline of the rendering pipeline and an explanation of some essential concepts and terminology relating to the rendering process.

2.4.1 Overview

The rendering pipeline produces images by processing mathematical descriptions of objects. Objects are described in different coordinate systems by collections of vertices and edges, called *primitives*. These primitives are passed to the rendering pipeline, where they are projected onto a plane called the *viewport*. The viewport-projection is then approximated into a grid of squares, called fragments, by a process called *rasterization*. Finally, the rasterized projection, called a *frame*, is sent to the screen to be displayed.

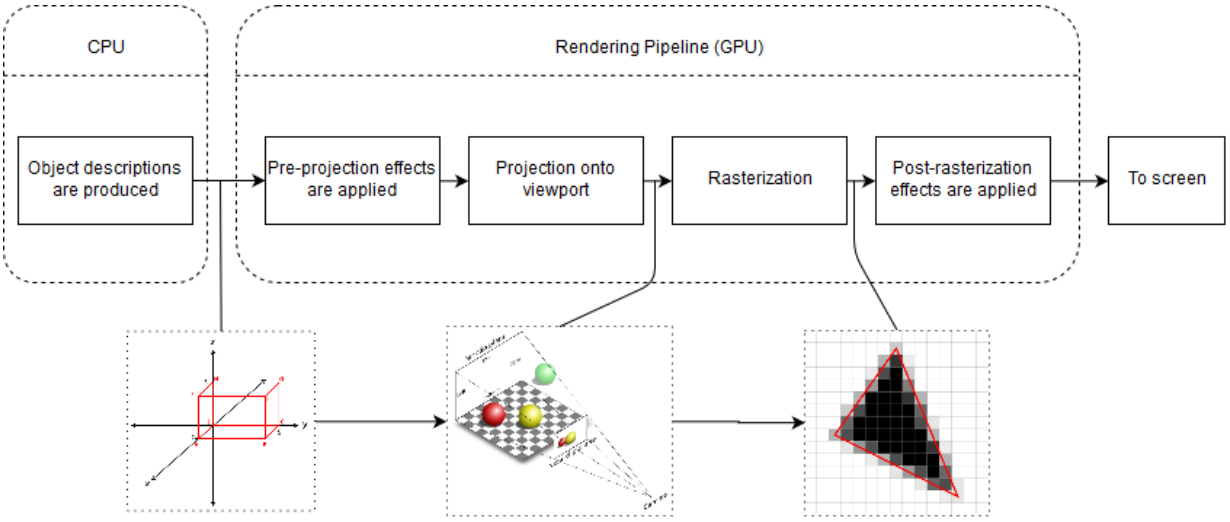


Figure 2.2: Very abstract overview of a general rendering pipeline. The program on the CPU produces object descriptions and passes them on to the GPU, which applies various effects to the object descriptions, projects them onto the viewport and rasterizes the projection. The rasterized image is then sent to the computer screen.

In between the three main stages of Primitive-generation, rasterization, and output to the screen lies 'effect-application' stages. This will be covered later, but for now it is important to remember that the primitives that make up object-descriptions and the fragments that make up the final image, are modified during these stages.

2.4.2 Primitives and Winding Order

Primitives is the common term for the basic shapes that the rendering pipeline handles during the pre-rasterization stage. The four basic primitives are: Points, Lines, Triangles and Quads (rectangles). [Scr15a]

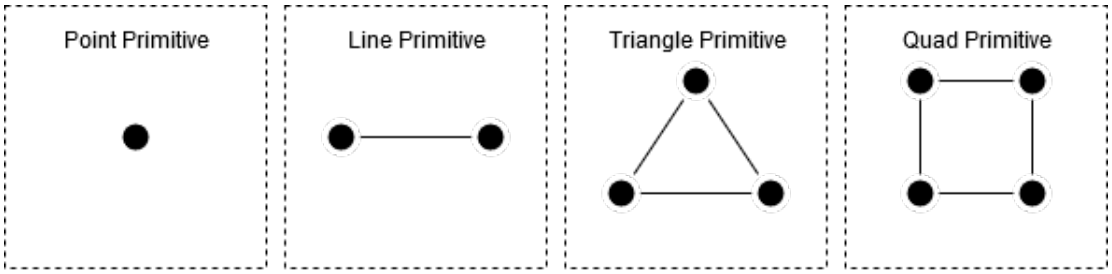


Figure 2.3: The four basic primitives. Vertices are illustrated as black circles, Edges are illustrated as the lines in-between them. Multiples of these are combined together to form the object-descriptions that the rendering pipeline handles.

All primitives are comprised of *Vertices*, which are points in space, and *Edges*, which are lines in-between vertices. A Point-primitive consists of a single vertex in space. A Line-primitive consists of two vertices connected by a single edge. A Triangle-primitive consists of three vertices connected by three edges. A Quad-primitive follows the same principle, but with four vertices. Primitives can have any number of vertices, but the previous four are the most commonly named.

The vertices of a primitive can be placed anywhere in relation to one-another. This is important, because it allows objects to be described by multiple triangles with different shapes, which in turn allows one to define very complex shapes.

All primitives with three or more vertices are also called polygon primitives (or simply polygons) and have an attached *face*. A face is merely a closed set of three or more edges. The front-face of a polygon defines which side of the primitive that is the front. This distinction is important due to some commonly-used optimization techniques that only draws the front of the primitive.[Gro16] The front-face is determined by the winding order of a primitive, which is simply the order in which the vertices of that primitive is defined. Winding order can either be clockwise or counter-clockwise.

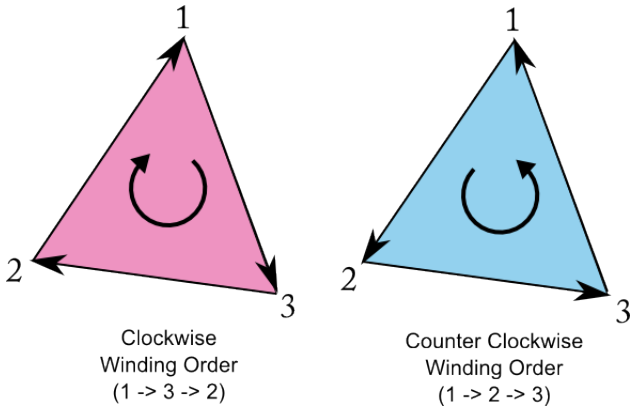


Figure 2.4: Image source: [Mic16]. Two front-facing triangle primitives with clockwise (left) and counter-clockwise (right) winding order. The vertices of the clockwise triangle is defined in the order 1 3 2, and the vertices of the counter-clockwise triangle is defined in the order 1 2 3.

2.4.3 Meshes

Meshes are simply collections of one or more primitives. In other words, meshes are data collections of vertices, edges and faces. Mesh-data is the main type of data that is passed to the GPU and are used to define the geometry of objects. Meshes are defined differently dependent on their implementation; The structure of the mesh-data is optimized differently in different applications, so meshes are not always cross-compatible.[Scr15a]

An important note about meshes is that the primitives that define them do not have to be adjacent to each other.

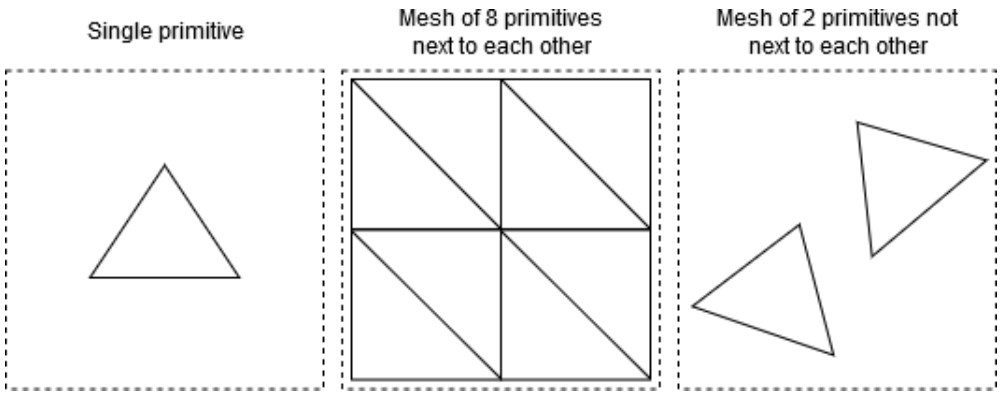


Figure 2.5: An illustration of meshes with different structures. Note that the rightmost mesh is comprised of multiple primitives, but these primitives are not adjacent to each other.

Meshes consisting of polygon-primitives are called polygon meshes. Polygon meshes are, by far, the most common mesh type for describing objects, since these are the only types of meshes that include faces.

2.4.4 The Viewport

The viewport is some defined plane in 3D space, that the vertices of objects passed into the rendering pipeline are projected onto. This projection is called the viewport transform. The resulting projected vertices are sent to the rasterizing stage, where they are used to compose an image. [Mar02]

The viewport is given a size, which is then used to define which vertices are projected onto the plane. The viewport acts much like a camera that defines what can be 'seen' and what cannot be seen. By moving the viewport around, you can achieve different perspectives of the same objects.

2.4.5 Coordinate Systems

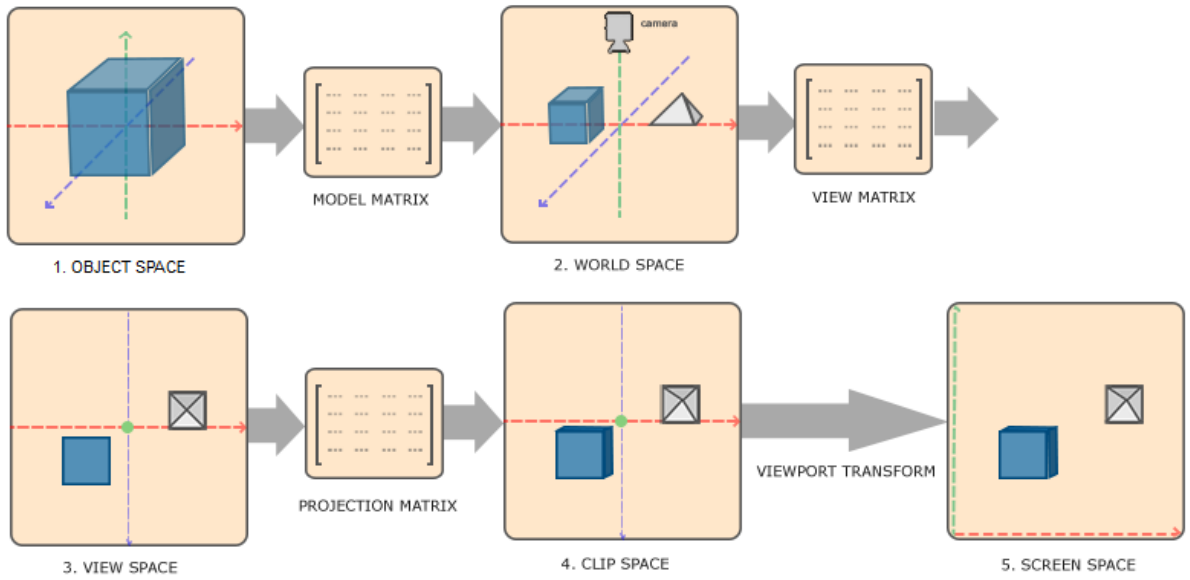


Figure 2.6: Image source: [Joe14a] An overview of the different coordinate systems used in the rendering pipeline. Vertices are translated between each coordinate system using transformation matrices.

For ease of use, the positions of individual primitives and their vertices can be viewed in different coordinate systems. By projecting the coordinates of their vertices, the programmer can use the different coordinate systems to make it easier to work with transformations on the primitives that they're handling. Some coordinate projections are also required in the graphics pipeline. In general, we distinguish between 6 different coordinate systems [Wik18]:

1. **Object space**; A coordinate system used for describing primitives in relation to the object that they are a part of. The origin of the coordinate system is the local center of the described object.
2. **World space**; Defines a coordinate system used for describing objects and their primitives in relation to other objects. The origin of world space is the global origin of the 'world', which is identical for all objects.
3. **View space**; defines a 2-dimensional coordinate system that maps vertices in world space to their corresponding position on the viewport-plane. The origin of view space is the center of the viewport.

4. **Clip space**; The purpose of clip space projection is to filter out any vertices that exist outside the defined viewport size. For this reason, clip space is merely a normalization of positions with respect to the defined viewport size. Mapping to clip space translates each coordinate within the defined viewport to a number between -1, and 1. All other coordinates follow the same translation, such that any vertices, mapped outside the viewport possess coordinates that are either larger than 1, or less than -1. [Joe14b] The origin of clip space is the center of the viewport.
5. **Normalized Device Coordinates** or *NDC*; is a subset of clip space, where any vertices outside the viewport are discarded - i.e. any vertices with coordinates larger than 1, or less than -1. The origin of NDC space is the center of the viewport.
6. **Screen space**; Takes vertices in NDC space and maps them to their corresponding positions on the image to-be-drawn. The size of this image is defined in pixels by settings inside of the Graphics API itself and can be changed to match the physical computer screen that the GPU outputs to.

The various stages of the rendering pipeline expect coordinates to be in different coordinate systems, so vertices are translated using transformation matrices during the rendering process.

2.4.6 Rasterization & Fragments

Figure 2.7: Image source: [Sto14]. Rasterization of a triangle. The triangle outlined in red is projected onto the viewport, and the rasterization process attempts to approximate this shape by generating fragments (the green squares).

Rasterization is the process of creating representations of the objects described in the viewport, that can be shown on the physical computer screen. These representations are directly equivalent to pixel-positions on computer screens, and describe which color each pixel of the final image should display. Rasterization is, in short, a process of approximating the primitives on the viewport into discrete points, called *fragments*, which can be directly mapped to pixels on a screen. The approximated image, output after the rasterization-process, is known as a *raster image*.[Scr15b]

Various implementations of rasterization exists, which result in different outcomes and/or more efficient processing. For the purpose of this paper it is not necessary to elaborate these.

2.4.7 Textures and UV-coordinates

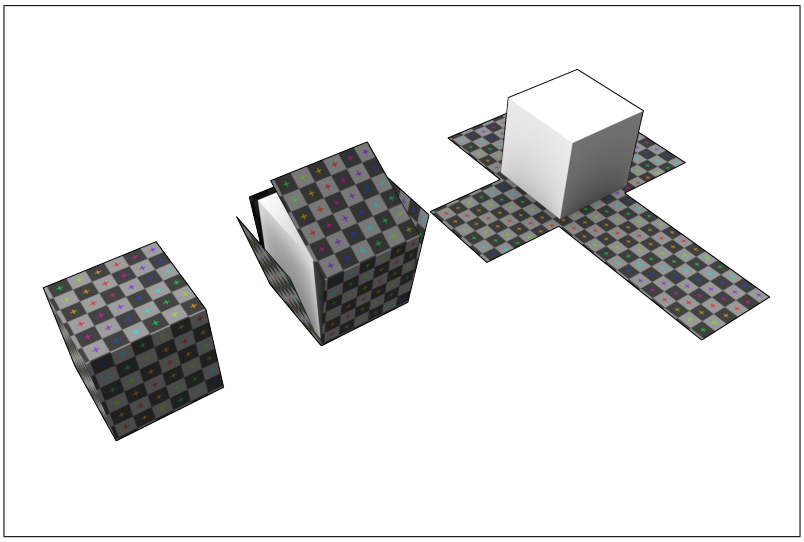


Figure 2.8: Image source: [Zep08]. Visual representation of a 3-dimensional object being unwrapped, part of the method used to generate UV-coordinates for textures.

Textures are the second data type available in the rendering pipeline. They use UV-mapping to map an image correctly onto an object. The UV-map is a 2-dimensional coordinate system that uses the variables u and v to represent the axes, similar to x and y in the traditional coordinate system. The UV-coordinates are simply called so because the variable names x and y are already used to describe the object mesh.

UV-coordinates, also called texture coordinates, describe the location of the each pixel in a texture that can be matched to a location on an object. To create UV-coordinates, the mesh of a 3-dimensional object is transferred to a 2-dimensional surface, this process is called unwrapping. Once unwrapping has been completed the image that is to be used is projected onto the flattened object mesh. This creates sets of coordinates in a 2-dimensional plane that enables an image to be projected pixel by pixel onto an object without stretching the image.[Uni18a][Uni17]

3 | Shaders

Now that the basic concepts of the rendering pipeline have been introduced, we can dive into shaders themselves. This chapter is dedicated to explaining shaders, the concepts surrounding shader programming, and how shaders fit into the rendering pipeline.

3.1 The Programmable Rendering Pipeline

As mentioned in section 2.4.1, the rendering pipeline is an abstract model of the series of sequential operations that a Graphics API uses to 'render' images to the screen. The Programmable Rendering Pipepline (PRP) is a modern evolution of this concept, and a successor to the so-called 'fixed-function' pipeline. The PRP introduces a number of intermediary processing steps, called *shader stages*, that allows graphics programmers to directly influence the data moving through the rendering pipeline.[Gał14]

Each stage takes a series of instructions called *shader programs* or simply *shaders* and executes these on the data passed from the previous stage. Shaders are written in specialized programming languages called *shading languages* and can be defined to produce various kinds of effects.

Every shader stage in the PRP handles different types of data, in different stages of the rendering process and are given their own constraints. Shaders must adhere to the constraints of the stage that they are written for. Examples of constraints are 'the output of this shader must be in the form of a single color' or 'The output of this shader must be a single vertex position in world space'.

The shader stages utilize parallel processing to execute shader instructions on every piece of data they are given, individually: e.g. a shader in the vertex shader stage is run once for every single vertex in the mesh it is provided. Additionally, some stages are optional, i.e. skipped unless a shader has been defined for the stage.

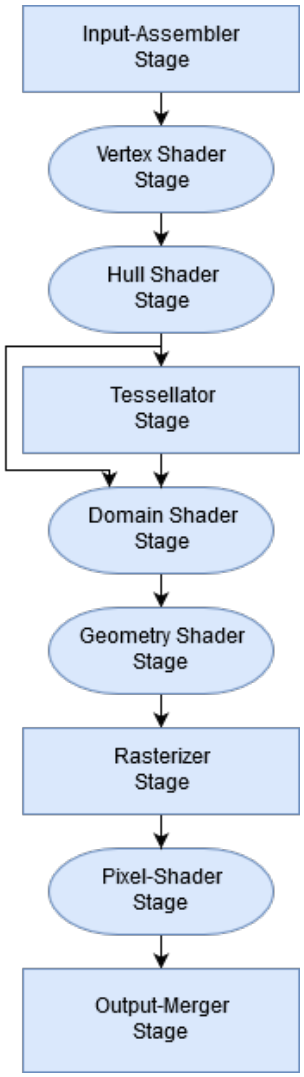


Figure 3.1: Simplified version of image from: [Mic18g]. Overview of the programmable rendering pipeline in Direct3D 11.

3.2 Shading Languages

Shading languages are specialized programming languages with built-in functionality and commonly defined variables that make it easier to implement shader functionality.[Lov05]

Each graphics API define support for different shading languages - some of which are proprietary to the API itself, some of which might be shared amongst several API's. Three important examples of shading languages are:

- *High Level Shading Language* (HLSL) - A shading language, developed specifically for the Direct3D API.
- *OpenGL Shading Language* (GLSL) - A shading language, developed specifically for the OpenGL API.
- CG, short for *C for Graphics* is a shading language developed by the graphics hardware manufacturer *Nvidia*. This shading language is supported by a wide variety of different Graphics API's, including OpenGL and Direct3D.

All of the above shading languages are based on standard C syntax.

3.3 The different types of shaders

Shaders are small, low-level programs that are compiled and run at specific shader stages in the rendering pipeline. Shaders are most often not compiled at runtime, but are rather pre-compiled for later use, when the rest of the program is compiled.[Mic18i]

The different types of shader are named after the shader stage they are defined for, e.g. a shader in the fragment shader stage is simply called a *fragment shader*, a shader in the geometry shader stage is called a *geometry shader*, etc.

While each shader has specific functionality and core purposes, additional computation is possible in each shader stage; Various values may need to be computed to achieve the wanted 'effect'. As such, these values may be calculated in each shader stage, with no limitations other than the minimum requirements for each individual shader. Direct3D supports this functionality by carrying user-defined data-structures between each shader stage, where values can be defined and placed as needed.

The rest of this chapter is dedicated to explaining every shader type in the Direct3D 11 rendering pipeline. We'll start with the most common types of shaders, namely vertex and pixel shaders, and use these as a basis to lead into the more complex types of shaders.

3.3.1 Vertex shaders

The *vertex shader* stage is the first programmable step in the rendering pipeline and is placed immediately after the input-assembler stage. Vertex shaders are programs that act on individual vertices in meshes. A vertex shader takes a single vertex as input and has access to a number of vertex attributes. These attributes contain information such as the index of the vertex in the mesh data set, the normal vector of the vertex and many others.[Mic18e]

A vertex shader is run once for each individual vertex and can manipulate the position and texture-coordinates of existing vertices. The vertex shader is a non-optional stage of the rendering pipeline and must be defined for it to work. If no vertex processing is needed, one can define a pass-through shader, that simply converts vertex positions to their proper space. [Mic18h]

The most defining limitation of vertex shaders is that they must maintain a 1:1 input / output ratio - i.e. vertex shaders must take exactly one vertex position as their input, and are required to output exactly one vertex position as their output. This, in turn, means that vertex shaders are unable to delete or create vertices in the mesh.[Mic18h]

The vertex shader stage is generally regarded as the most optimal place to compute custom values for use later in the pipeline. This is due to the fact that the vertex shader stage is run the least times of all shader stages (only once for each vertex in the original mesh), and as a result, takes the least processing resources.

If no further vertex processing stages have been defined after the vertex shader stage, the pipeline jumps directly to the rasterizer stage, which includes clipping the vertex positions to the viewport (See section 2.4.5). In this case, the vertex shader is expected to output vertex positions in world-space, since this is what the rasterization stage expects as input.[Mic18d]

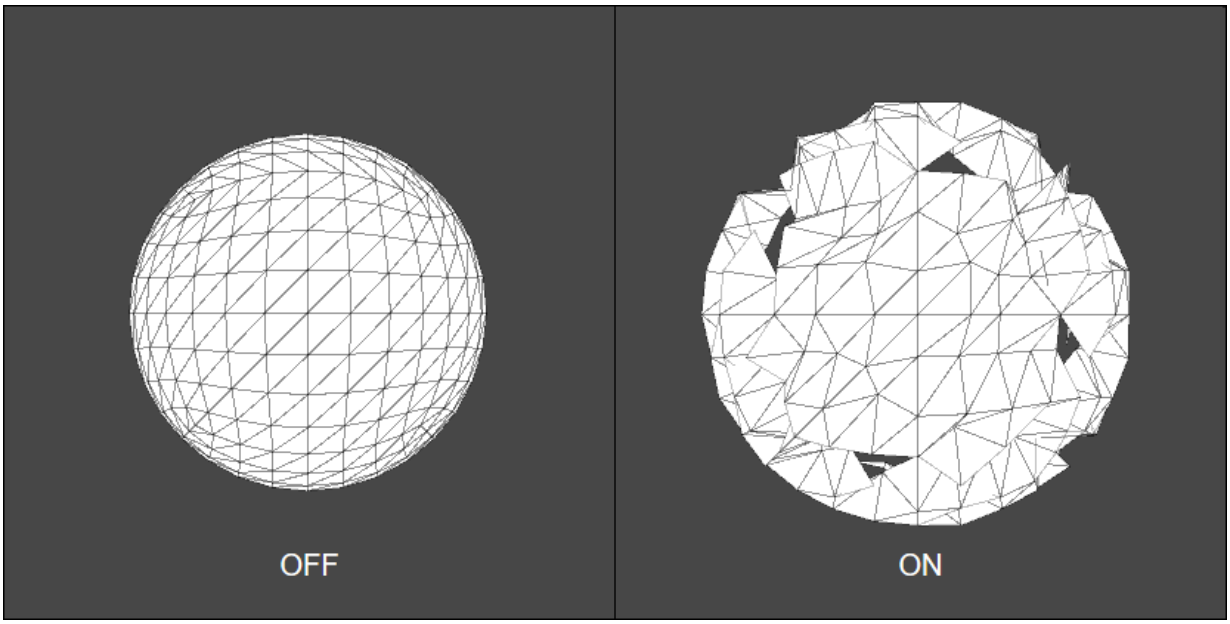


Figure 3.2: Example of a vertex shader applied to a spherical mesh in Unity. The black lines outlines each primitive in the mesh. On the left side we see the mesh without the shader applied, on the right - the mesh with the shader applied. This particular shader extends the position of every other vertex in the mesh along the normal of that vertex. The code for this shader can be found in appendix A.1

3.3.2 Pixel Shader

The *pixel shader* stage, sometimes called the *fragment shader* stage, is the last programmable stage in the rendering pipeline. It is followed by the output-merger stage and acts as an extra layer of image processing immediatly after the rasterizer stage. Pixel shaders are programs that act on the individual fragments of a raster image to influence the final colors of each pixel.

Pixel shaders can be used to implement various post-processing effects such as colour correction and lighting. The rasterizer stage invokes a pixel shader once for every pixel covered by a primitive. For this reason, the pixel shader stage is a non-optional shader stage. Nonetheless, it is still possible to bypass any image processing in this shader stage by explicitly defining a NULL shader, which is simply skipped in the rasterizer stage.[Mic18c]

In Direct3D 11, a pixel shader takes a maximum of 32, 32-bit, 4-component input values and can output up to 8, 32-bit, 4-component colors in RGBA format, or no color if the pixel is discarded. The input data of pixel shaders include: various vertex attributes that have been interpolated in the rasterizer stage, per-primitive constant values (as defined by the rendering pipeline itself) and user-defined constants and values from other shader stages. [Mic18c]

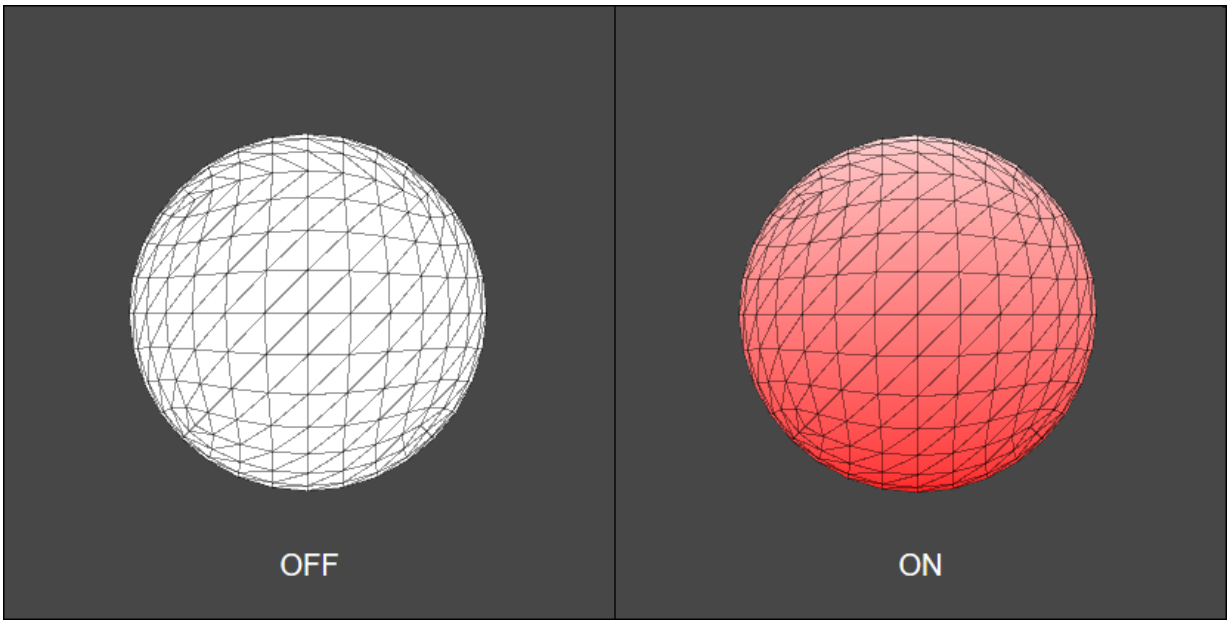


Figure 3.3: Example of a pixel shader applied to a spherical mesh in Unity. The black lines on the mesh itself, outline each primitive in the mesh. On the left side we see the mesh without the shader applied, on the right - the mesh with the shader applied. This shader colors each pixel based on its position in respect to the mesh, creating a smooth transition between red and white. The code for this shader can be found in appendix A.2

3.3.3 Geometry shaders

The *geometry shader* stage is an optional stage in the rendering pipeline that is placed immediately after the domain shader stage. Geometry shaders are run once for every primitive passed down, or generated earlier in the pipeline (See section 3.3.4).

Unlike vertex shaders, which operate on individual vertices, geometry shaders take the vertices of entire primitives as input, and can output vertices. Geometry shaders must output data one vertex at a time and have the capabilities to discard and create vertices on the fly. Most importantly,

a geometry shader can define entirely new primitives and faces by linking existing vertices as well as vertices it creates itself. Each geometry shader can output only one type of primitive as its output. This type is declared in the shader itself, along with the maximum number of output vertices.[Mic18b]

If, at the end of an instance, a geometry shader has output an incomplete primitive (that is, a primitive with less vertices than its primitive type requires), that primitive is completely discarded before it is inserted into the mesh. [Mic18b]

Geometry shaders have access to vertex data of the edge-adjacent primitives of the primitive they are processing, which allows for the implementation of adaptive algorithms. [Mic18b]

The geometry shader is immediately followed by the rasterizer stage, which means vertex positions must be output in clip-space.

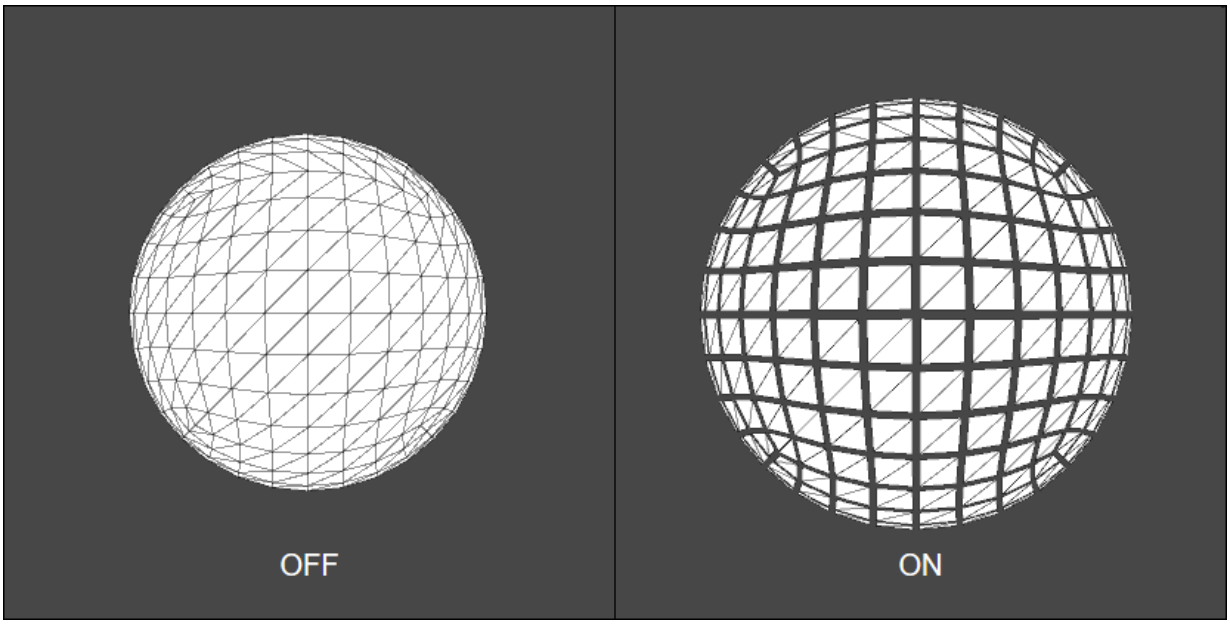


Figure 3.4: Example of a geometry shader applied to a spherical mesh in Unity. The black lines on the mesh itself, outlines each primitive in the mesh. On the left side we see the mesh without the shader applied, on the right - the mesh with the shader applied. This shader 'redefines' the mesh by creating new primitives from the original primitive positions: Each primitive is extended along the normal of the old primitive. The code for this shader can be found in appendix A.3

3.3.4 Tessellation stages

An optional step in the rendering pipeline is the implementation of tessellation, which in essence is the process of subdividing primitives into several, smaller primitives and thereby creating more detail. In other words, it tiles, or breaks-up, a low-detail mesh so that it may gain more vertices, which may then be displaced.

In Figure 3.5 you'll see tessellation performed on a low-detail mesh to produce a higher-detail mesh. The level of detail added to this specific mesh is arbitrary, but the amount of tessellation that can be done is constrained to a factor of 1 to 64. [Mic18a] The process of tessellating in itself does not change the shape of the object being tessellated, but must be followed by a displacement of the newly created vertices to have a visual impact.

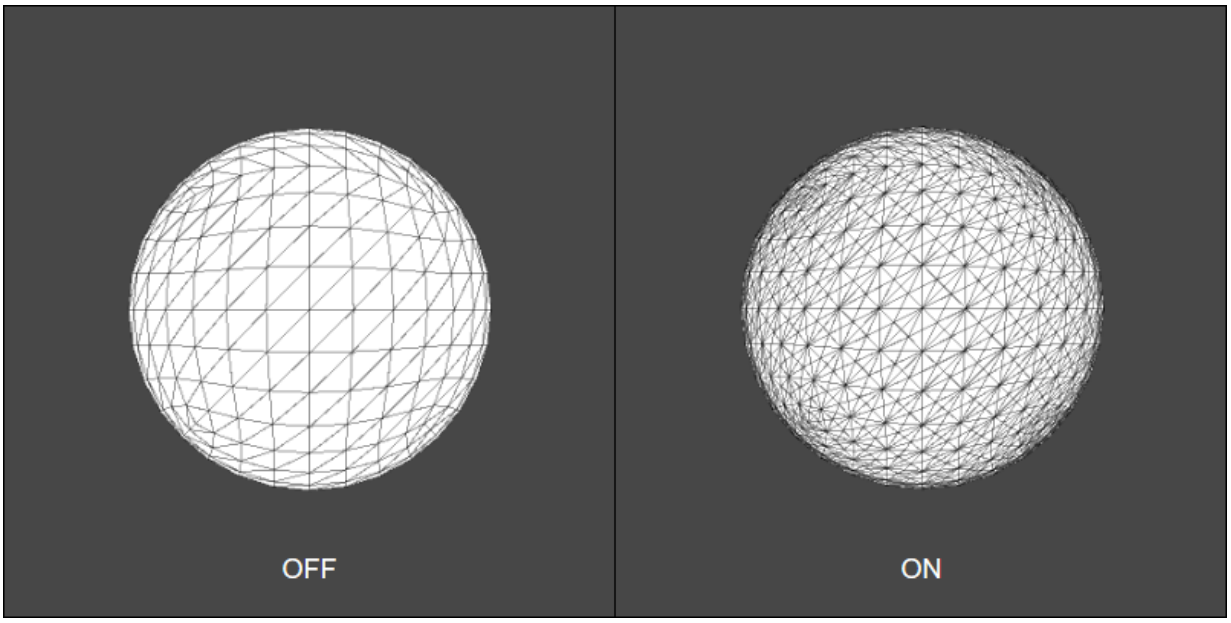


Figure 3.5: Example of tessellation being used on a sphere. Notice how each primitive has been subdivided into several, smaller primitives. Usually, this process would be followed by a displacement of the vertices in the newly tessellated mesh, to change its shape. The code for this example can be found in appendix A.4

The performance-related benefits[Mic18a] of using tessellation are:

- Saves a lot of memory bandwidth, by allowing for higher-detail models to be 'generated' on-the-fly from low-detail models.
- Supports displacement mapping, which is often used to reach a stunning amount of details.
- Supports scalable rendering techniques, such as continuous or distance based Level-Of-Detail which can be calculated on-the-fly.
- If implemented correctly, it can lead to very large performance improvements, as many of the demanding tasks, which are handled by the CPU, can be done by the GPU instead.
- Assets may be created in low detail and tessellated later.

To use tessellation, the following three pipeline stages must be implemented: [Mic18a]

- *Hull-Shader* stage. A programmable shader stage that will be invoked once per specified patch (polygon-primitive). The hull shader specifies which patches should be tessellated and how they should they be tessellated.
- *Tessellator* stage. A non-programmable stage that performs tessellation on the patches defined by the hull shader.
- *Domain-Shader stage.* A programmable shader stage that calculates the vertex position corresponding to a displacement method - essentially moving the newly created vertices from the tesselator stage.

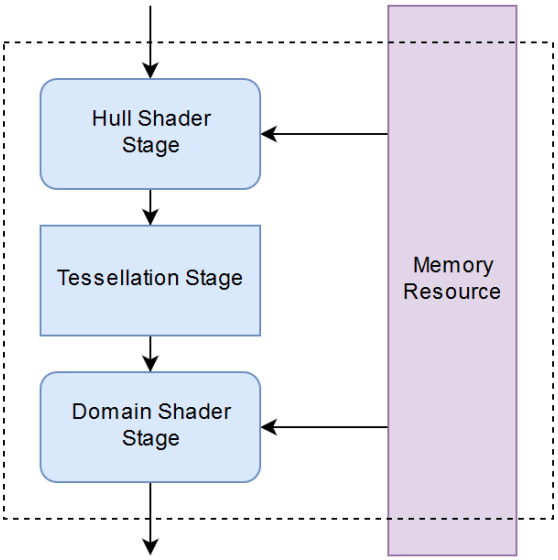


Figure 3.6: The three stages that together perform tessellation, notice how the Tessellator Stage does not read from memory.

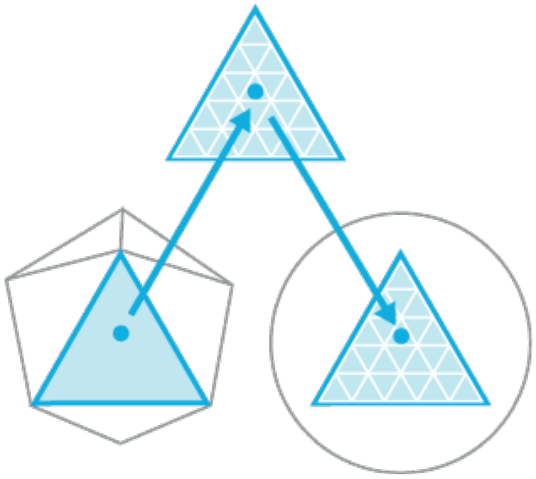


Figure 3.7: This illustration[Mic18f] represents what happens at every tessellation stage, in broad strokes; a patch is selected, that patch is tessellated and lastly displaced to increase detail.

Hull Shader Stage

Invoked once per patch (polygon-primitive). The hull shader takes *one* input and gives *two* outputs. Its input are the *control points* that makes up a patch[Mic18a]. Control Points are essentially the vertices that define the shape of a single patch. The two outputs are the same control points and a list of settings that configure the tessellation process, called *patch constants*. Examples of patch constants are: the tessellation factor, the number of control points, the patch face type (tris/quads/etc), and the partitioning type to use when tessellating.[Mic18a]

The hull shader operates in two parallel phases [Mic18a], the *control-point phase* and the *patch-constant phase*.

- The **control-point phase** is run once for every control-point, reading the control point for a patch and supplying a control point output.
- The **patch-constant phase** is run once per patch to generate tessellation factors or other per-patch constants.

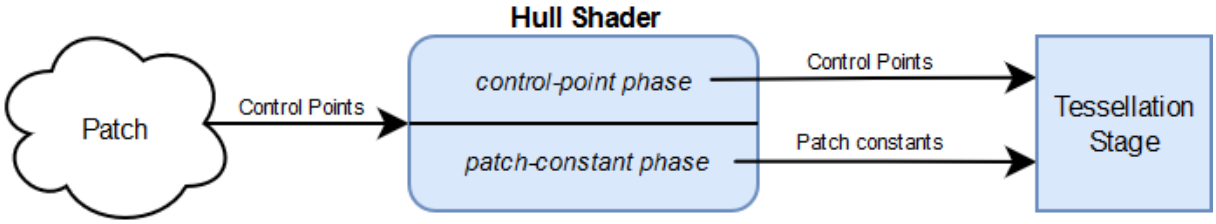


Figure 3.8: The hull shader starts by fetching a patch from memory and then in parallel; fetching and processing all control points and all patch constants (tessellation factor, partitioning type etc.) for the tessellator stage.

Tessellator Stage

Creating a hull shader initiates the tessellator which is a configurable, but not programmable, pipeline stage. The tessellator also operates once per patch applying the patch constants provided by the hull shader and tiling a canonical (coordinate) domain, using barycentric coordinates (a normalized system used for easier processing). The tessellator outputs the newly created vertex positions and corresponding UV-coordinates, to the domain-shader stage. [Mic18a]

Internally, the tessellator operates in two phases, these are not run in parallel:

1. The first phase processes the input patch constants.
2. The second phase generates additional primitives based on a set configuration. This is the main phase where all tessellation is done.

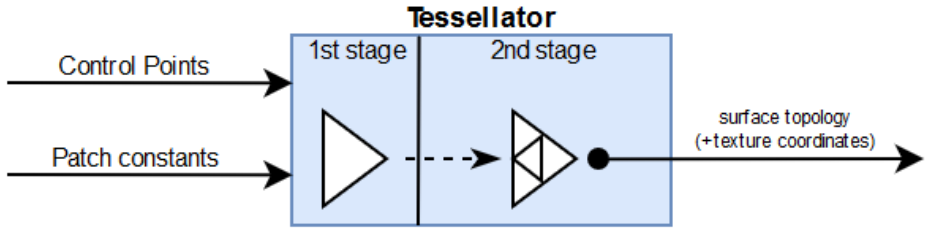


Figure 3.9: The tessellator constructs the simple input patch in the first stage and then, in the second stage, applies the configured tessellation type to the patch and output the corresponding uvw coordinates and higher-detail surface topology.

Domain Shader Stage

A domain shader is run once for each vertex that is output by the tessellator stage. It uses these, along with the control points and constants from the hull shader, to calculate the final vertex position of a subdivided point in the output patch. This means that the domain program produces more vertices than we started out with (it has been tessellated). [Mic18a]

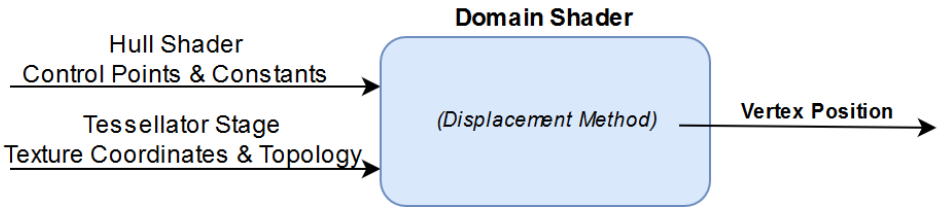


Figure 3.10: The domain shader takes the output of the tessellator stage and hull shader and calculates the correct displacement of the vertices generated by the tessellator, according to the hull shaders patch and the specified tessellation configuration.

The position of these new vertexes is usually displaced along some vector, otherwise the tessellation serves no purpose. The distance with which this vertex is displaced, is determined by the displacement method that is implemented, as seen in Figure 3.10. A common method used for as the displacement method is *displacement mapping*, which simply displaces the vertices along a vector according to a 2D displacement map, also know as height map.

3.4 Shaders in Unity

Working with a Game Development Engine like Unity, usually means that many of the features that are commonly used in games have some level of support accompanying them.

Especially in this case it's worth talking about how Unity interacts with shaders and how it can be used to produce grass shaders.

- Unity can handle many shader functionalities at its UI level, such as importing textures for use in shaders.
- Unity supports the creation of shaders using a proprietary markup language called ShaderLab.
- ShaderLab allows programmers to define shaders as atomic methods with a mixture of HLSL and CG features.
- ShaderLab handles a lot of the technical parts of writing shaders for applications, which is why we have been glancing over these for the most part; the complicated details are not necessary to understand our grass shader implementations.
- ShaderLab also makes it easy to interact with shaders at runtime through scripts, which will be a significant part of our further implementation.
- Unity uses so-called *materials* to apply shaders (among other things) to meshes.

Additionally Unity has its own implementation of a shader, named the *surface shader*. The surface shader is Unity's implementation of a multi-functional shader 'wrapper', which handles all the underlying shader stages. This can be used to quickly make detailed materials, with many aspects being handled automatically.

The surface shader has many specific functionalities, which will not be explained further. but when developing shaders in Unity, it's likely that it will be utilized. [Mic17]

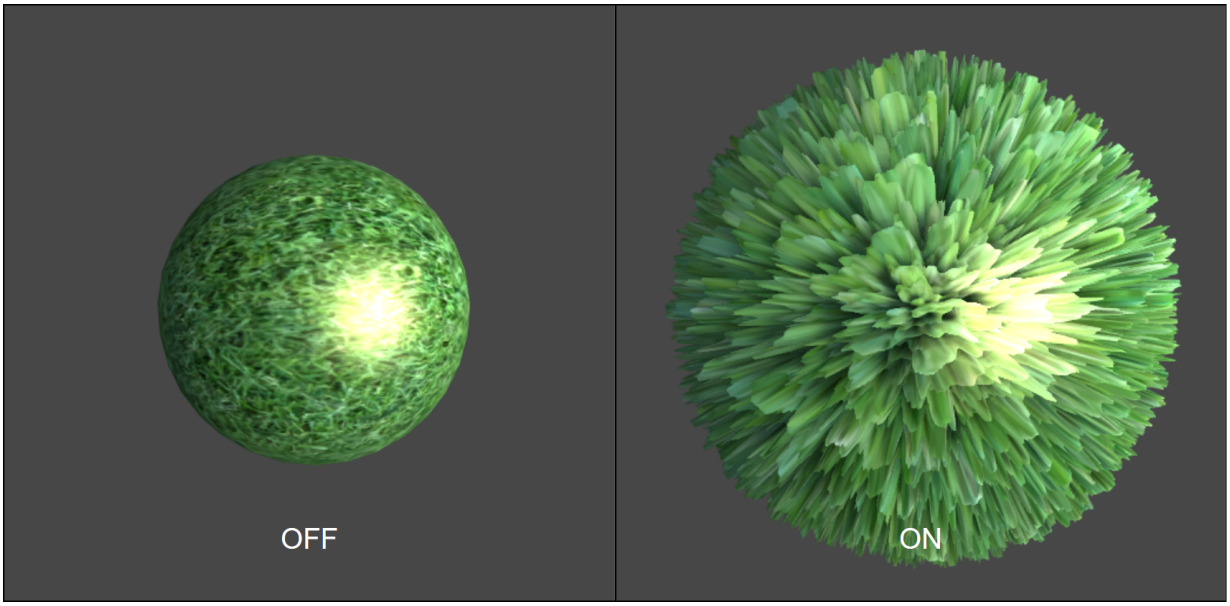


Figure 3.11: Surface shader used to implement tessellation with a displacement- and uv mapping. Notice the lighting upon the two spheres, which is also handled by the surface shader. The code for this shader can be found in Appendix A.5

4 | Implementation & Design

This chapter is dedicated to explaining the implementation of our grass shader and the various design choices we made underway.

4.1 Pre-rendering or real-time rendering?

The purpose of this project is to write a program that can generate grass terrain with the capabilities of being used in a video game context. A central question in the process of conceptualizing, was the question of whether we should make an implementation using pre-rendering or real-time rendering techniques. The answer was quite obvious:

Since video games are programs that often require real-time interaction with the user, it would make little sense to utilize processes that needed to be heavily processed before they can be used. As such, the only other option was real-time rendering.

This is also the reason why we chose to imitate grass, rather than create a perfect simulation of grass; simulating grass uses a lot of resources and is hard, if not impossible, to implement efficiently in real-time rendering.

4.2 Three approaches to grass shaders

After researching shaders and real-time rendering, we set out to decide which approach to grass rendering we would like to implement. In our studies, we encountered 3 basic approaches to generating grass terrain, each with their own own weaknesses and advantages. We made a prototype for each approach, with the purpose of choosing a single one to continue development on. This section is dedicated to explaining these three prototypes, our reflections about them, and which one we decided to continue with.

4.2.1 Model-based

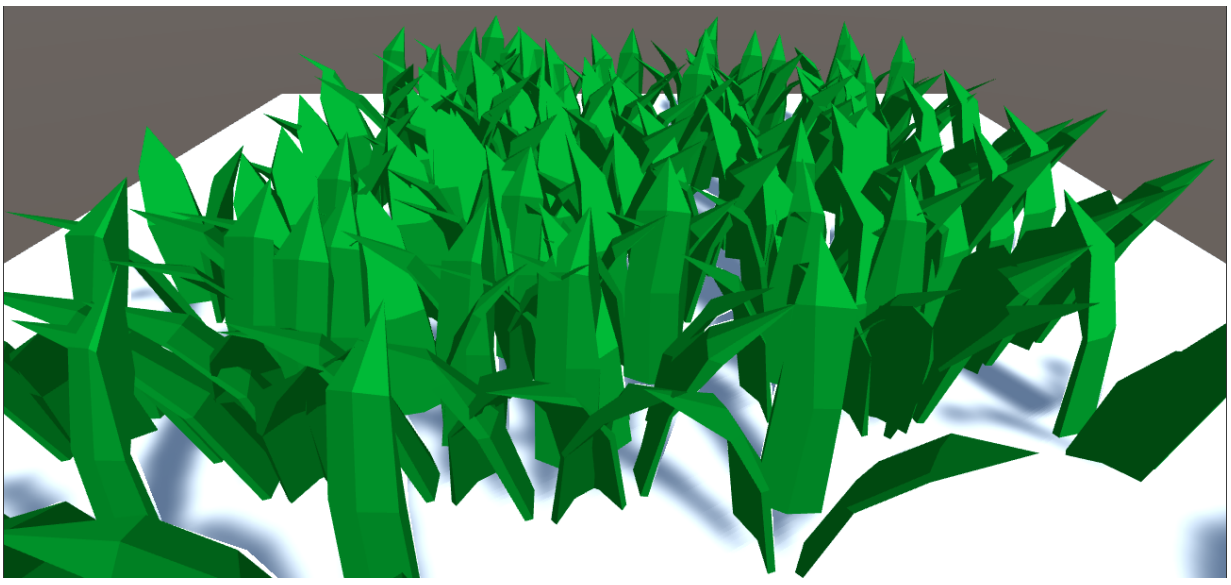


Figure 4.1: Screenshot of a Unity program running the model-based shader on multiple models

The model-based grass implementation is based on placing pre-defined grass models around a terrain. To imitate a field of grass, an amount of grass models are placed randomly about an area. This placement is done with a simple Unity script that instances objects around the defined area.

Our implementation uses a simple grass model comprised of three individual blades of grass. This model works well for illustrative purposes, but it could be replaced with other, potentially larger models.

The model-based prototype uses a relatively simple pixel shader to color the given model. The shaders emulate the appearance of shadows by applying different brightness levels of a defined color to the model. The pixel shader combines a texture, some adjustable parameters and a pre-defined color, to assign brightness-levels to different areas of each blade of grass. The specified texture is encoded with the different levels of brightness that should be applied. The pixel shader simply samples the texture and applies the color at the sampled UV-position as a layover on top of the specified color. The resulting effect makes the object look like it casts a shadow on itself.

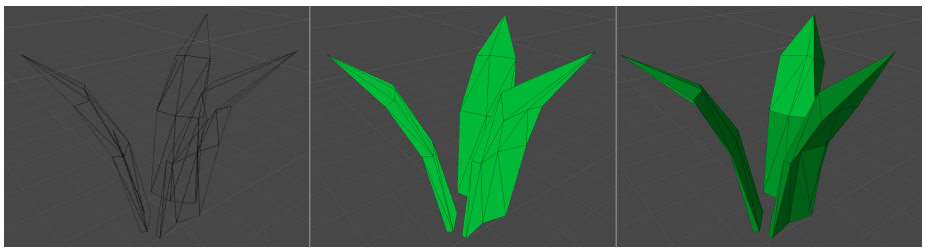


Figure 4.2: Image of single grass model colored at different stages of shader application: On the left, we see the model with no shader applied. In the middle, we see the model, colored with the specified color. And on the right, we see the full shader at work; applying 5 different brightnesses of the specified color by sampling the brightness-texture.

Advantages

- It is easy to make a relatively detailed model of a grass blade and apply a simple shader to it.
- The shader can be applied on a variety of grass models, making it easy to emulate a field of varying types of grass and it is therefore also easy to change the look of the grass.

Disadvantages

- Storing a separate model for each cluster of grass can result in a lot of memory overhead. Methods would have to be implemented to dynamically load and de-load models based on which models are in view for this implementation to be viable for large-scale use.
- Unity has poor support for placing and removing grass objects, so it is harder to efficiently generate large amounts of grass through scripts.

4.2.2 Plane-based

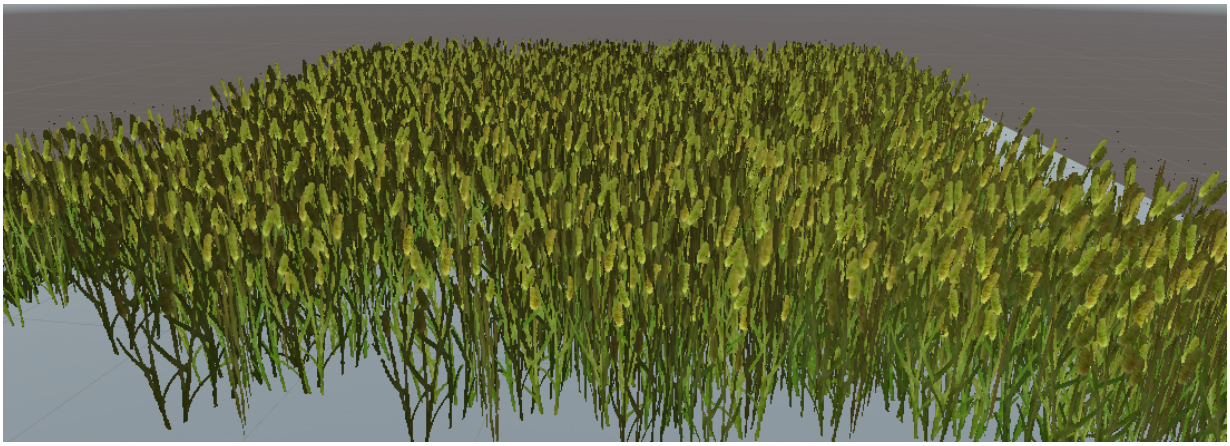


Figure 4.3: Screenshot of a field of plane based grass running in Unity

Our plane based approach is made up of 2 parts, a pixel shader that applies a grass texture to both sides of a plane and a script that can place these grass planes randomly onto a surface.

The Shader

This implementation makes only little actual use of shaders in its current from. It uses a pixel shader to filter out pixels of the grass texture based on alpha value. The shader also makes the texture appear on both sides of the quad by turning off back-face culling, which normally makes it so that only the front face of objects are rendered.

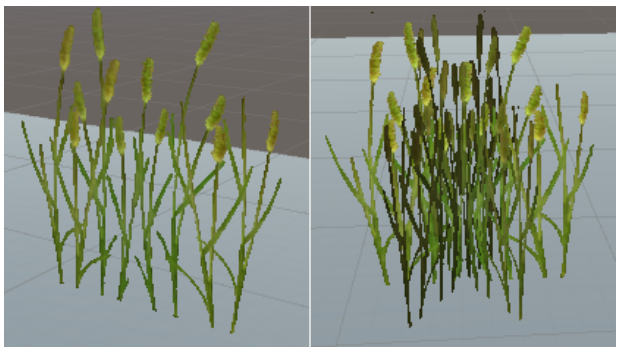


Figure 4.4: single and intersecting quads with the fragment shader applied.

The Script

The implementation uses a C# based script to project origins of grass instances down onto any underlying terrain: The script traces lines downward from random coordinates within a defined area. The points where any line hits a surface are then recorded into a list. Each recorded point is used to place three intersecting planes in the world, with the recorded point acting as their center. The aforementioned shader is then used to apply a grass texture that to each plane, so the grass is visible from all sides.

Advantages

- The script places down an object that can easily have different types of shaders applied to it.
- The script can be used to place any kind of predefined mesh so it could easily be modified to place other grass models.
- It is easier to make the grass look relatively realistic from a distance using this implementation.

Disadvantages

- The Shader turns of culling on the grass completely so it's rendered even when covered by other objects which takes up resources without adding to the visual fidelity.
- The script uses instantiating to place down the grass object but can only instantiate 1023 objects per generator. Since each instance of grass consists of three objects, only 341 grass instances can be placed per grass-generating script.

4.2.3 Point-based

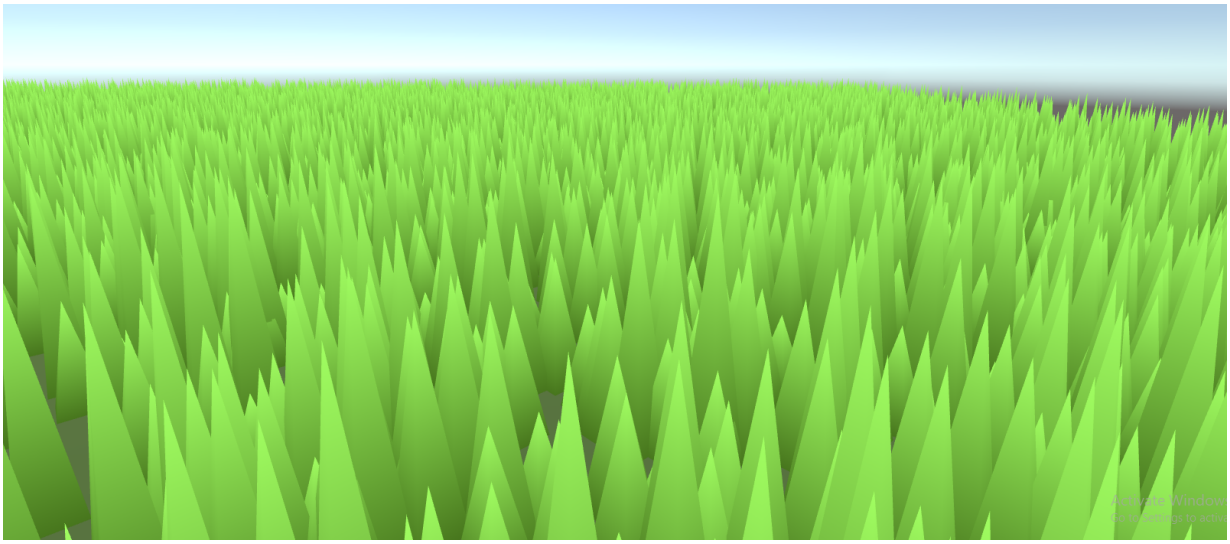


Figure 4.5: Screenshot of the point-based grass shader applied to a large mesh of points.

The point-based grass implementation is based on generating grass geometry almost entirely inside the rendering pipeline itself. The shader is intended to be used on meshes consisting only of point primitives in 3 dimensional space - also known as a *point cloud*. Each point in the point cloud acts as a base from which individual, tetrahedral blades of grass are defined in a geometry shader.

The Shader

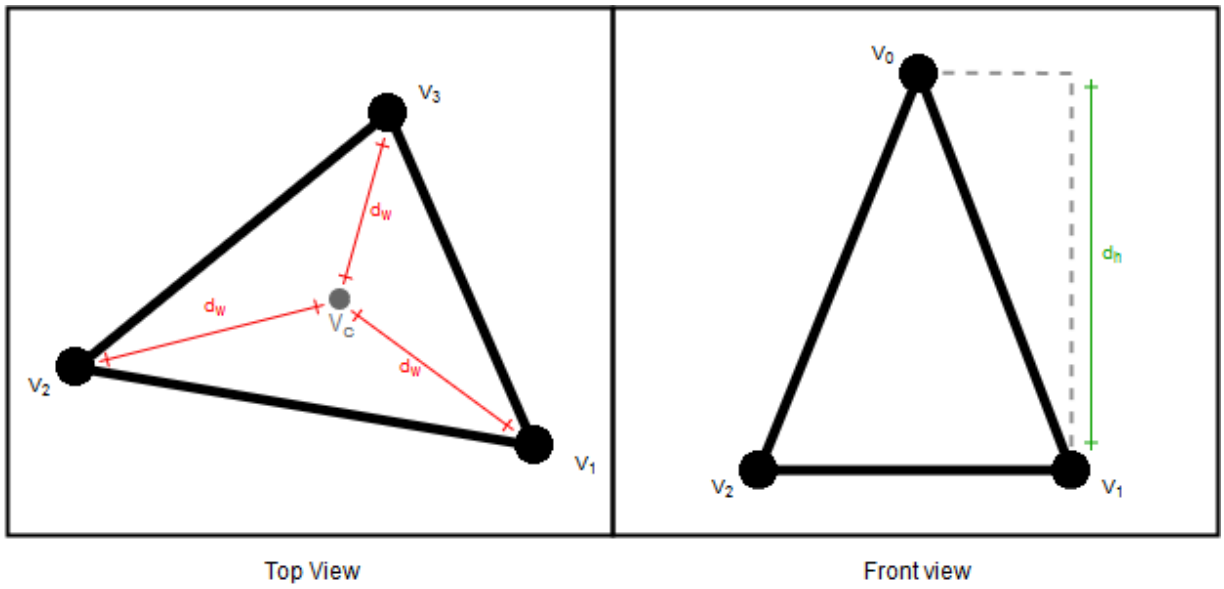


Figure 4.6: The basic structure of a single blade of grass in the point-based implementation, viewed from two angles: Top-view, looking down on the base of the grass, and front-view, looking at one side of the grass-tetrahedron.

Figure 4.6 shows how the geometry shader processes the individual point cloud vertices:

Each vertex in the mesh is defined as a center point V_c. The center point serves as the origin for the grass blade. From the center point, define a single 'tip' vertex V_0 and 3 side-vertices V_1, V_2, V_3. The base vectors for the coordinate system are used to define the side- and tip vertices at d_w and d_h distance from the center point as such:

```
//pos[0-4] are equivalent to V_0-4, heightCalculated is the randomized
    height distance d_h and halfWidth is d_w
pos[0] = float4(center + up * heightCalculated  , 1.0f);
pos[1] = float4(center + right * halfWidth      , 1.0f);
pos[2] = float4(center - right * halfWidth      , 1.0f);
pos[3] = float4(center + forward * halfWidth    , 1.0f);
```

d_w is a user-defined variable, set using ShaderLab parameters, and d_h is randomized between some user-defined range using a very simple Pseudo-Random Number Generator(PRNG), seeded with the xy world-space position of V_c:

```
 float randomRange(float2 seed, float min, float max)
{
    float num = frac(sin(dot(seed.xy,
        float2(12.9898, 78.233)))*
        43758.5453123);
    return (num + min) * (max-min);
}
```

V_0, V_1, V_2 and V_3 are then used to define the 3 faces of the blade of grass by linking the points together, following clockwise winding order:

Face 1:	V_0, V_1, V_2
Face 2:	V_2, V_3, V_0
Face 3:	V_0, V_3, V_1

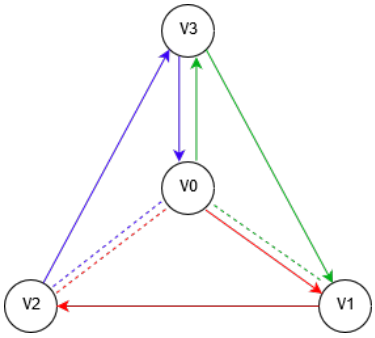


Figure 4.7: An illustration of how each blade of grass is defined.

This process is repeated for every vertex in the point cloud mesh. Tied together with this functionality is a very simple pixel shader that colors each blade of grass from top to bottom with a linear interpolation between two user-defined colors, so as to create the illusion of 'shade' in the grass blades without actually having to simulate it.

The Script

Integral to this implementation is how point cloud meshes are generated and placed in the 'world'. We use a simple Unity script for this functionality: Points in object-space are distributed randomly over a square area on the XZ-plane of the world.

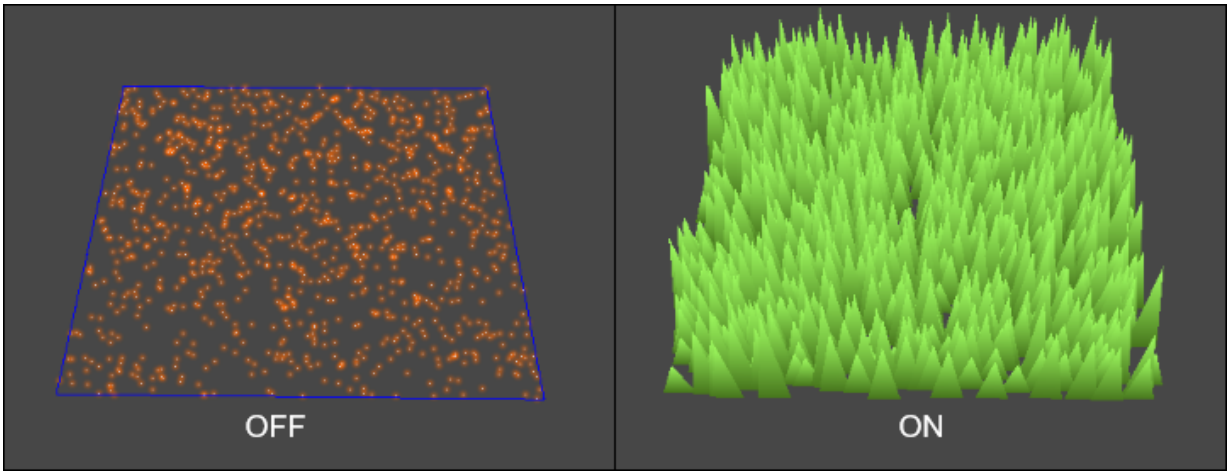


Figure 4.8: A mesh generated with no underlying terrain and the same mesh with the point cloud grass shader applied. The blue lines highlight the 'edge' of the mesh area where points are distributed randomly. Each point in the mesh is highlighted with an orange 'glow' around it.

These points are then projected, using Unity's raycasting mechanism, onto any terrain below, which allows point layout to conform to the contour of the terrain it is applied to. Finally, the projected points are then inserted into a mesh structure and the point cloud is defined.

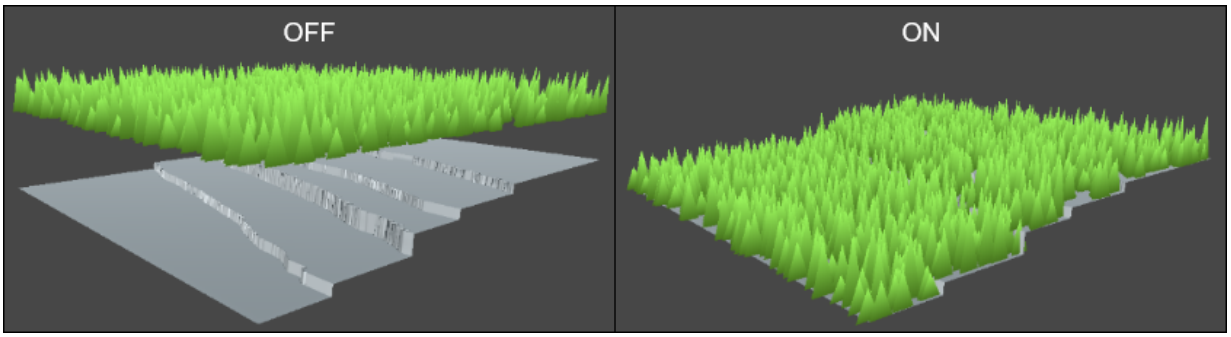


Figure 4.9: Two examples of a mesh generated with the Unity script with the point-based shader applied to them. On the left, the mesh is simply defined on a plane. On the right, the points of the mesh are projected onto the terrain below it.

Advantages

- Generating the model inside the pipeline itself gives us full control of the grass model at runtime, and makes it easy for us to potentially influence this at runtime.
- Generating the grass model entirely on the GPU allows for full utilization of GPU parallelism. Since the grass is numerous, this should free up a considerable amount of resources on the CPU for other tasks.

Disadvantages

- Generating the entire model inside the geometry shader means that the earlier pipeline stages, such as tessellation, can't directly affect the grass model. Solutions to this problem exist.
- This implementation should put a heavier load on the GPU than other implementations, since this requires more real-time processing.

4.2.4 Reflections & Choice of Type

After making a prototype implementation of each approach, the implementations were weighed against each other and compared, with the purpose of choosing which approach we would continue to develop. We ended up choosing the point-based approach, due to a variety of factors:

- Performance; the sheer load that the implementation puts on the computer hardware.
- Flexibility & Control; weighing the advantages and disadvantages of each approach.

Performance

We ran a couple of simple diagnostic tests on each prototype to compare the amount of computer resources that each approach would use. The intent behind our implementation was to be able to cover large areas of terrain with many grass instances, so performance was compared by how efficiently grass could be generated at various amounts of grass instances.

The tests results indicated that the point-based implementation could generate vastly more instances before becoming a limitation on resources. However, uncertainties in the tests and the individual differences between each implementation made it hard to conclude much from this indication.

Further details about the used test methodology and the individual test results can be found in chapter 5.

Flexibility

While the model-based approach would allow us complete control over each individual grass model, these models had to be statically defined before any shader effects could be applied to them - it would be considerably more complicated to change the model at runtime. This, together with the performance issues that comes with placing each model in RAM at all times, made the model-based approach rather unattractive.

The plane-based approach, on the other hand, allowed us less control over the shape of the grass model itself, but opened other possibilities in terms of the look of each grass cluster. The problem here would come in potentially implementing wind effects as well as having the same overall performance problems as the model-based approach. Another big limitation was the limit on how many grass clusters could be instantiated within unity scripts.

The point-based approach was, first of all, simpler to work with, but the fact that the entire model was generated inside the pipeline also allowed us easier control over the model during runtime as well as the effects being applied to it. Additionally, we preferred the look of the point-based implementation.

4.3 Implementing wind

After choosing our basic approach to how the grass models were placed; we could start implementing the various effects that would be applied to those models. The first of these were to make the grass behave like it was 'waving' in the wind.

The obvious implementation was to move the tip vertices of each blade of grass, in accordance to some 'wind' direction. In the process of implementing the wind effect, we decided on three overall requirements for the movement of the wind effect itself:

- The distance between the center and the tip of the grass blade should remain constant and the wind should be able to blow in any direction - Generic rotation around all axes was required.
- The grass should blow smoothly over time, between two rotation extremes - A rotation-cycle should be implemented based on time.
- The blades of grass should not move synchronously - Adjacent blades of grass should start their rotation-cycle at different points.

4.3.1 Which Type of Rotation?

In researching how we wanted to implement our rotation, we stumbled upon two different methods: Rotation matrices and Quaternion multiplication.

Rotation Matrices

Rotation matrices is a specific concept from linear algebra, that utilizes matrix multiplication of *n*x*n*-dimensional matrices to rotate points in R^n space. The idea is to construct matrices that, once multiplied with a given point, results in the point being rotated about a specific axis, with a specific angle. A rotation matrix can be constructed using the following formula(s):

[Ser15]

$$R_{\alpha,n} = \begin{bmatrix} a^2K + cos(\alpha) & abK - csin(\alpha) & acK + bsin(\alpha) \\ abK + csin(\alpha) & b^2K + cos(\alpha) & bcK - asin(\alpha) \\ acK - bsin(\alpha) & bcK + asin(\alpha) & c^2K + cos(\alpha) \end{bmatrix}$$

$$K = 1 - cos(\alpha)$$

$$n = [a, b, c] = ax + by + cz$$

Where α is the angle of rotation, and n is the axis of rotation (a vector)

Our implementation utilizes the above equation and uses a normalized vector as the axis of rotation:

```
//Rotate a point around a given axis with a given angle (in RAD)
float3 rotateAroundAxis(float3 origin, float a, float3 axis)
{
    float k = 1 - cos(a);
    float x = axis.x;
    float y = axis.y;
    float z = axis.z;
    float3x3 rotAxisMat = {
    pow(x, 2)*k + cos(a), x*y*k - z*sin(a), x*z*k + y*sin(a),
    x*y*k + z*sin(a), pow(y, 2)*k + cos(angle), y*z*k - x*sin(a),
    x*z*k - y*sin(a), y*z*k + x*sin(a), pow(z, 2)*k + cos(a) };
    return mul(rotAxisMat, origin); //mul is matrix multiplication
}
```

Quaternion Multiplication

Quaternion multiplication is a very common method of rotation in modern games and graphics computing. As the name implies, quaternion multiplication is based on a special number system called quaternions.

The code we use to implement this functionality was taken from user - mattatz on GitHub(source: [Nak18]). A copy can be found in appendix C.1

Below is a very summarized outline of the math behind rotation using quaternion multiplication.

In short, quaternions are 4-dimensional vectors whose components are partially comprised of 3 specific complex numbers called the fundamental *quaternion units*. Just as a regular 3-dimensional vector is comprised of the components *x, y* and *z*:

$\vec{v} = [a, b, c] = ax + by + cz$, where

a, b, c are numbers and v is a vector

So does the quaternion units *i, j* and *k* comprise a quaternion:

$q = [a, b, c, d] = a + bi + cj + dk$, where

a, b, c, d are numbers and q is a quaternion

In quaternions, the components that are multiplied by the quaternion units are collectively referred to as the vector components, and the remaining component is simply called the scalar. Following this principle, the above quaternion can also be represented as:

$$q_n = [a, \mathbf{v}]$$
$$\mathbf{v} = [b, c, d] = bi + cj + dk, \text{ where}$$

a is the scalar component, and **v** is the vector component

A quaternion with a scalar of 0 is called a *pure* quaternion.

The quaternion units are complex numbers and are governed by a special set of algebraic rules that specify the relation between them:

$$i^2 = j^2 = k^2 = ijk = -1$$

Without diving too deep, these relations are what allows us to use quaternions for rotation, and what makes quaternion multiplication work.

In parallel to rotation matrices, the idea of quaternion rotation is based on constructing quaternions that represent specific rotations about some axis with some angle. All quaternions used for rotation must be *unit* quaternions (parallel to unit vectors). Rotation quaternions are constructed using the following formula:

$$q = \left[cos\left(\frac{1}{2}\theta\right), sin\left(\frac{1}{2}\theta\right)\mathbf{v} \right], \text{ where}$$

θ is the angle of rotation, and

v is the 3-dimensional, normalized axis of rotation (replacing x, y, z with i, j, k)

By multiplying this quaternion q with its inverse q^{-1} and the vector you want rotated **p** - in pure quaternion form p - you get the rotated vector p': [Jim13]

$$p' = qpq^{-1} = [0, 2a(\mathbf{v} \times \mathbf{p}) + 2(\mathbf{v} \times (\mathbf{v} \times \mathbf{p})]$$
$$p = [0, \mathbf{p}], \text{ where}$$

p' is the rotated point/vector in pure quaternion form,

p is the 3D vector you want rotated,

q is the rotation quartenion, and

q^{-1} is the inverse of the rotation quartenion

The inverse of a quaternion is calculated by the following formula: [Mat19]

$$q = [a, b, c, d] = a + bi + cj + dk$$
$$q^{-1} = \frac{a - bi - cj - dk}{a^2 + b^2 + c^2 + d^2}$$

Theoretical Comparison and Choice

Because each instance of the two methods only corresponds to one specific rotation, and the fact that each strand of grass has its own rotation cycle, the rotation must be calculated for every individual strand of grass in the entire mesh. For this reason, the performance of each rotation is a very important factor.

We weigh the difference between the two previously mentioned methods based on the amount of operations they must use to instance themselves and rotate a given point/vector, as well as the amount of floats(numbers) they must write to memory:

Rotation Type	# Operations	# Floats
Matrix	15	9
Quaternion	41	4
Quaternion + Matrix	39	4

Table 4.1: Comparison of operation counts and floats stored for transforming 1 vector. Values from [Geo08]

Table 4.1 shows a stark difference between the two base methods; while quaternions store less than half the floats than matrices, quaternion rotation takes a little over 3 times the base-operations that matrix rotation does.

Instead of being directly used for rotation themselves, quaternions can be converted into rotation matrices. Skipping the instantiation of the inverse quaternion leads to a reduction in operations, also seen in table 4.1.

The strength of this method becomes evident when several vectors need to be rotated identically, since the quaternion only needs to be defined once:

Rotation Type	# Operations
Matrix	15n
Quaternion	41n
Quaternion + Matrix	24 + 15n

Table 4.2: Comparison of operation counts and floats stored for transforming n vectors. Values from [Geo08]

Since our current shader must construct a new rotation matrix for every single blade of grass, rotation matrices seem very performant in comparison. Our shaders intended application is for use in real-time graphics and therefore we value performance higher than memory usage. As such, we chose to use rotation matrices for our current implementation.

4.3.2 Smooth Wind

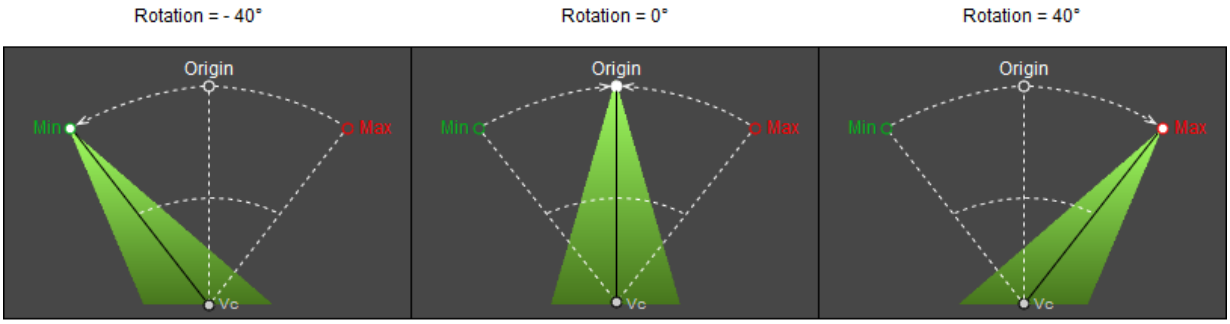


Figure 4.10: A diagram of the rotation functionality for a single blade of grass, moving between the minimum and maximum degrees of rotation. The wind effect is created by linearly interpolating the angle of rotation over time.

The rotation cycle of our shader is implemented using a simple linear interpolation between a rotation at a user-defined angle, and its negative. The interpolation factor is simply calculated from a sine-function, based on current system time.

4.3.3 Asynchronous Wind

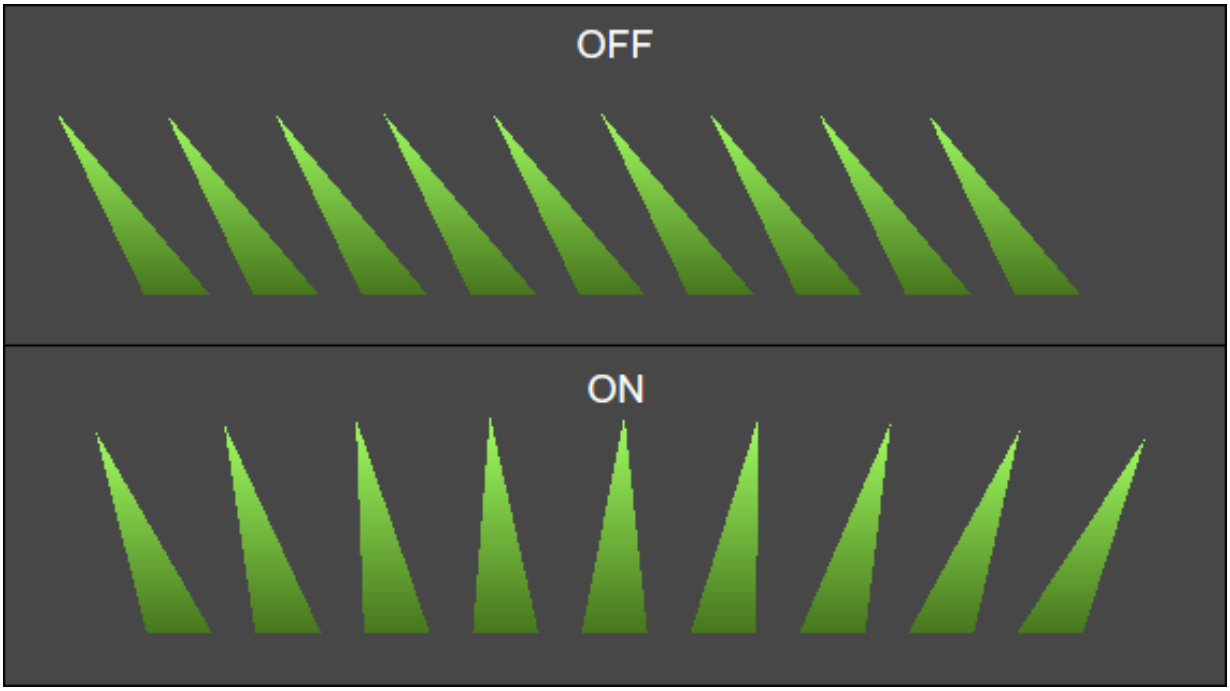


Figure 4.11: An example to illustrate the effect of asynchronous wind on our grass shader: The coordinate position of each blade of grass are used to offset the rotation cycle of each individual blade of grass. This results in a 'wavy' effect.

We achieved the requirement of adjacent blades not moving in unison, by offsetting their rotation cycle, i.e. the number we feed to the aforementioned sine-function, by the X and Z coordinates of the grass centerpoint.

4.3.4 Wind Implementation

Below is a simplified snippet of our final shader, encompassing the combined implementation of our wind implementation:

```
//Define rotation axis and angle based on given wind-direction
float3 directionVector = normalize(Wind_Direction_Vector);
float rotationAngle = angleToRad(Rotation_Angle) * sin(_Time[3] +
    center.x + center.z);
float3 rotationAxis = normalize(cross(up, directionVector));

//rotate tip vertex around rotationAxis by roationAngle, using
    rotation matrix method
Tip_Point = float4(rotateAroundAxis(pos[0].xyz - center, rotationAngle
    , rotationAxis) + center, 1);
```

The function *rotateAroundAxis()* is the same method described in section 4.3.1.

5 | Testing

In conjunction with the development of the various implementations of our grass, we have run simple performance tests on each iteration. The purpose of these tests was to identify the influence that each addition to our basic grass-rendering approach had on GPU resource requirements. This, in turn, would allow us to identify areas where optimization was necessary.

The following chapter is dedicated to explaining these tests, their results and the reflections that each test lead to.

5.1 General Test Methodology

All tests in this chapter were monitored by Unity's built-in diagnostic tools, but due to our experience and time limitations, features of these tools were ignored.

Tests were carried out in identical environments, under identical constraints. This standardization was intended to minimize external influences that could influence the outcome of each test - or at least keep those external influences identical between test runs.

The standardized test environment was as follows: A single Unity scene with no elements other than the subject to be tested, and a camera to facilitate the testing (Unity requires a camera to be placed in the scene before any rendering is done). Each subject was a square in the X/Z plane with side lengths of 4 world units. Grass elements were distributed evenly across the subject-squares in a grid-like pattern, by instancing scripts, tailored to each implementation. These scripts can be found in appendix D

All performance was measured in Frames Per Second (FPS), which is an indicator of how many images the program can render pr. second. This value is influenced by both the CPU and the GPU.

We expect the performance of each approach to differ, based on specifications of the computer they run on. For this reason, the tests were all be carried out on the same computer, with the same hardware. These specifications can be found in appendix D.4.

5.2 Testing the Different Approaches to Grass Rendering

Tests were carried out on each different implementation, mentioned in see Section 4.2. The purpose of these tests were to compare the performance of each implementation.

5.2.1 Test Description

Each approach was tested separately with increasing amounts of their respective grass instances. FPS measurements were taken directly from Unity's built in diagnostics tools, but these lacked any averaging functionality. For this reason, many of the resulting FPS values were approximated from observed maximum and minimum extremes during testing.

Due to the way that we currently place the plane-based grass clusters, the instancing script for the plane-based approach is limited, by Unity, to produce a maximum of 341 individual clusters. It is for this reason that the scale of the plane-based test is so limited.

5.2.2 Results

Model Based	
# Models	*Avg. FPS**
100	2243
400	1467
900	1134
1600	812
2500	593
3600	452
4900	340
6400	251
8100	185
10,000	149
40,000	30

Plane Based	
*# Plane Clusters***	*Avg. FPS**
64	2122
81	2045
100	1870
121	1740
144	1583
169	1540
196	1357
225	1286
256	1224
289	1128
324	1040

Point Based	
# Points	*Avg. FPS**
100	2940
10,000	2774
40,000	2836
90,000	210
160,000	101
250,000	64
360,000	43
490,000	31
650,000	22
810,000	18
1,000,000	14

** Average FPS is approximated from max/min readings, due to limitations in diagnostic tools.*
*** Plane clusters comprised of 3 individual planes. Tests are on different scales due to the limitations in our instancing-methods in relation to each approach.*

Table 5.1: Test results from each test of the different approaches to grass rendering.

5.2.3 Conclusion & Reflections

The tests show a clear difference between the three approaches:

Accounting purely for performance pr. instance, the point-based approach trumps the other two implementations by far - it can simply render vastly more instances of grass. However, as discussed previously, the differences between each implementation makes a direct comparison difficult.

Additionally, the test results from the point-based approach shows a large, strange decrease in FPS somewhere between 40.000 and 90.000 instances. This decrease is likely due to the size of the encompassing mesh growing too large for the CPU to send it to the GPU as a single package, although this is just speculation.

5.3 Testing our Wind Implementation

Tests were carried out to determine what impact our implemented wind effects would have on the grass rendering. See section 4.3 for a description of the implementation.

5.3.1 Test description

The main goal of these tests were to compare the performance of quaternion-based rotation to matrix-based rotation, with no wind effects as a baseline.

The test was carried out in much the same way as the test of the different grass rendering approaches - FPS was approximated from minimum and maximum values.

5.3.2 Results

No Wind	
# Points	*Avg. FPS**
100	2940
10,000	2774
40,000	2836
90,000	210
160,000	101
250,000	64
360,000	43
490,000	31
650,000	22
810,000	18
1,000,000	14

Matrix Rotation	
# Points	*Avg. FPS**
100	2497
10,000	2523
40,000	2510
90,000	195
160,000	101
250,000	60
360,000	39
490,000	29
650,000	22
810,000	17
1,000,000	13

Quaternion Rotation	
# Points	*Avg. FPS**
100	3006
10,000	2990
40,000	2998
90,000	192
160,000	101
250,000	60
360,000	40
490,000	29
650,000	22
810,000	17
1,000,000	13

** Average FPS is approximated from max/min readings, due to limitations in diagnostic tools.*

Table 5.2: Results from wind effect test. From left to right, each table corresponds to the following; Grass implementation with no wind effects, Grass implementation with wind using rotation matrices and Grass implementation with wind using quaternion rotation.

5.3.3 Conclusion

Although the results from table 5.2 shows differences between matrix and quaternion rotation at lower amounts of points. This large, strange reduction in performance at 90.000 points in each case, determine the above test to be inconclusive.

We suspect that our point-grass implementation becomes limited by the CPU somewhere between mesh sizes of 40.000 to 90.000 points. This is a serious flaw with out testing methodology, and one that might be fixed by splitting the grass-field into several meshes of equal density.

Matrix rotation seems to perform worse than quaternion rotation before the suspected break point, but this could, viably, be reduced to a combination of error in our approximation and noise on the GPU/CPU from other tasks.

6 Discussion

The following chapters will discuss various choices from our implementation process as well as our tests. We will highlight potential weaknesses of our implementation and propose viable alternatives to these.

6.1 Choices

6.1.1 Our choice of work environment

As a premise to our implementation, we decided to go for Unity as our programming environment - because of the fact that Unity would remove a lot of the complexity that comes with writing code for graphics applications. This choice served us well throughout the project, but the various rendering optimizations that Unity implements, ended up making it hard for us to gauge the performance of our shaders in isolation.

An alternative solution to our project would be to do our different implementations in other development environments, but this was not possible due to time-constraints.

6.1.2 Why did we choose to compare three grass generation methods

Our work included the prototyping and subsequent comparison of three specific approaches to grass generation: Point-based, plane-based, and model-based. We compared these to learn more about grass generation methods in general, but it might have served us better to focus our efforts on a single approach from the start of the project: This would have allowed to dive deeper into the more complex features of grass generation, like interaction with physics-objects, or various optimization methods like Occlusion Culling.

Additionally, we could have chosen three methods that were closer to each other in concept, but we would argue that a comparison of such methods would have little to no merit; the advantages and disadvantages of each implementation would likely be nearly identical.

6.2 Implementations

6.2.1 Generalizing rotations could make quaternion rotation the preferred choice

Our wind implementation calculates rotation in every geometry-shader pass, i.e. once for every vertex in the original mesh. We chose matrix rotation because the collective rotation function - including both generating the rotation matrix and applying it to a point - is much more efficient in this specific case. Theoretically, quaternion rotations should be more performant if we could somehow generate a shared rotation quaternion for all vertices and pass it into the rendering pipeline along with the mesh.

6.2.2 Distributing computations between other shader stages might improve performance

Our current implementation does most of its computation in the geometry shader. Certain parts of that computation could be placed in the vertex shader stage instead to improve performance. The computations for rotation, height-calculation and width-calculation are all independent of the points generated in the geometry shader itself.

6.3 Tests

6.3.1 FPS as a measure of performance

The FPS measurements from Unity's diagnostic tools are based on the collective performance of the program in the CPU and the pipeline in the GPU; The CPU determines which meshes should be sent to the GPU for rendering, as well as being responsible for handing that mesh data over to the GPU pipeline and initiating the rendering of the frame.

Since our implementations are comprised of both CPU computation(placing the grass-instances themselves) and GPU computation (applying effects to those instances). Because of this, it is reasonable to measure both CPU and GPU performance in conjunction with each other, so we hold that FPS is a satisfactory measure of performance. On the other hand, if the points of our tests were to measure GPU performance in isolation, we would need to use other diagnostic measures.

6.3.2 Testing environment is unreliable

A couple of potential problems present themselves with our test environment:

First of all, we only performed our tests on a single computer: We cannot outrule the possibility of our implementations performing slightly different on other computers, since almost all parts of the implementation could be affected by different hardware.

Second, our tests have the possibility of being influenced by other processes running on the same computer or just in Unity in general. This might explain the fluctuations in our FPS readings.

Third, we test our implementations in isolation from other elements in Unity, but the point of our implementations is for them to be used as a part of a game - which would entail the grass terrain running in conjunction with other elements of that game.

6.3.3 Unity optimization makes getting accurate readings harder

Unity has various built-in optimizations for its rendering processes. This made it harder to get accurate performance readings on our grass implementations. E.g. we placed the camera of the various tests 'inside' the subject grass patches, since FPS readings would remain completely stable between tests otherwise.

Without any knowledge of which optimizations are at work at any given time, our readings could possibly be far from real operation. This would be a good idea to remedy in further tests.

One could argue that these optimizations are valid to include in performance tests since our implementations were meant to work inside Unity anyway, but it certainly limits the usability of our readings for use in other contexts.

6.3.4 Approximation of test results are likely inaccurate

Due to our inexperience with Unity's diagnostic tools, we were unable to get frame-by-frame readings of FPS measurements and had to simply observe the diagnostic tool and the fluctuations of the numbers. As such, we had to approximate the actual medians from observed maximum and minimum readings during run-time. This is obviously an inaccurate method of measuring averages, especially since the FPS readings had a tendency to fluctuate a lot at lower instance amounts.

A suggested solution to this problem would be to run other, third-party diagnostic tools on compiled versions of the Unity program, for each test case. We would most likely be able to find a diagnostic tool that allowed us to get exact FPS averages.

It's worth noting that we did observe sufficient differences between each test case to give a general outline and to warrant some consideration of performance, even despite the inaccuracies.

6.3.5 More test parameters are probably necessary

Currently, the single measured value was FPS readings, which works decently well as an indicator of run-time performance. But in many cases, grass implementations are used in conjunction with various other elements that take up various other resources, than simply computation time in the CPU and GPU. Memory usage, among other things, is also an important factor to consider when choosing and optimizing a grass implementation.

Therefore we would like to have measured a variety of other different performance parameters in our tests as well.

6.3.6 Problems with the model-based implementation test

The model-based approach to grass rendering is applicable to many different types of models. Different models have different vertex counts, and as a result, require different resources. Our test is based on one specific grass model, so any conclusions of the test of the model-based approach can only relate specifically to the approach using that model.

Additionally, the model based instancing script creates individual Unity game-objects that represents each grass instance. Game-objects in unity are tied into various other processes other than rendering, which will be compounded in the model-based implementation as instance-count grows higher. Also, every game-object comes with some memory-overhead.

To achieve a fair comparison with the model-based approach, we would probably have to re-think our implementation of it.

6.3.7 Different implementations cannot be compared on the same scale

A central problem with the tests of the different approaches, is the fact that each implementation has different limitations:

- The plane-based approach is directly limited by our implementation in unity.
- The Model-based approach is limited by the memory overhead of each grass-instance.
- The Point-based approach is only limited by the size of its mesh.

While these limitations are important to consider when picking which approach to use, one might argue that because of these limitations, comparing the approaches through simple performance measurements is meaningless.

7 | Conclusion

How can we use shaders to generate terrains of grass in the Unity 5 game engine, that are rendered in real-time?
To solve this problem, we have implemented a two-part system comprised of a mesh-generation script and a shader. The shader takes meshes consisting of individual vertices - generated by the mesh-generation script - and uses these as a basis for generating simple tetrahedron-shaped grass strands.

Which shader types can be used for this and how?
As we have found out through our research into the field, the possibilities for different types of shader based grass terrain solutions are almost enumerable and different combinations of these are ideal for different circumstances.

We implemented three different approaches to our problem, each using the different stages in different ways.

The first was based on applying shader effects directly to 3D models of grass. This approach used a simple pixel shader to apply basic shading to the 3D object. The second approach used the pixel shader stage to apply 2D textures of grass to each plane in a cluster of planes. The third and final approach generated grass almost entirely inside the rendering pipeline, using the geometry shader.

Our final implementation was based on the third approach. It used the geometry shader to extrude individual blades of grass, and the pixel shader stage to apply a simple color-gradient from the top to the bottom of each strand.

How can we improve the practicality of the terrain generation?
Our final implementation is based on meshes of individual vertices. These meshes were generated by our own mesh-generation scripts, which would randomly place these vertices in a predefined 3D-plane. To improve the practicality of this implementation, we used Unity's Raycast functionality to project the vertices onto any terrain positioned below their initial positions, before they are included in the mesh. This allowed our meshes to follow the shape of the terrain below it, vastly improving usability.

Which features can we implement to make the grass terrain more closely imitate grass?
We implemented a variety of features to our grass generation solution, to better imitate the look and behavior of real-life grass:

- Our grass terrain solution randomizes the dimensions of each strand of grass. The purpose of this was to prevent the uniform look of a field of grass with no variation.
- Our solution attempts to emulate wind effects by rotating the tips of each strand of grass around its own center point using matrix rotation. The wind effect is offset by the positions of each strand, as to create 'waves' of wind in large fields of grass.

8 | Further Research

8.1 Next iteration - Dynamic Level-Of-Detail

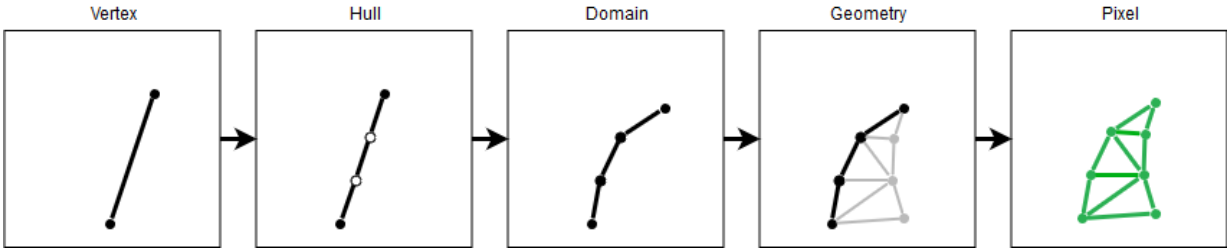


Figure 8.1: An illustration of how the proposed next iteration would use each primitive in the base mesh to create curved blades of grass.

Building on the previous iteration of our grass shader, the plan is to implement distance-based, dynamic Level-Of-Detail(LOD) in the shader itself, as well as some restructuring of the basic approach.

Dynamic LOD is a feature of some shader implementations that controls the detail of a model, based on its distance from the camera viewport. In our case, we would like to move beyond the simple, tetrahedronal shaped grass strands, into higher-detail, 'curved' strands - comprised of more vertices. The intention behind this iteration is to improve the visual fidelity of the grass while limiting the impact this increase in detail would have on performance, by gradually reducing the detail of each model, as it is positioned further away from the camera.

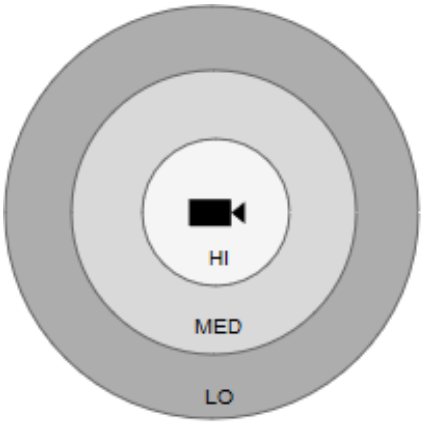


Figure 8.2: The LOD of each blade of grass is dependent on its distance from the camera subject: High-detail models close to the camera, medium-detailed models at intermediate distance, and low-details models beyond that.

To facilitate this, we would like to utilize the tessellation shader stages to split each blade of grass into several parts and move these so the grass blade becomes curved. Since points can't be subdivided, we thought of basing our next iteration on meshes of line-primitives, instead of points.

The basic overview of each shader stage in the new iteration is illustrated in figure 8.1 and is planned as follows:

1. The vertex shader rotates the tip vertex to follow the wind direction.
2. The hull shader divides the line into several vertices.
3. The domain shader moves those vertices to a curved position
4. The geometry shader extrudes a 3D shape from the curved line.
5. The pixel shader colors the newly generated 3D model.

This implementation would allows us to scale the detail of each blade of grass, simply by tweaking the amount of subdivisions that the tessellation shader stages perform.

8.2 Shader Intrinsic Functions

Shader intrinsic functions allow the developer to directly access graphics hardware instructions, in situations where those instructions would normally be abstracted by an API. It's like directly embedding highly optimized machine language code into high level code[Cha16], meaning that the code does not need to be compiled, as depicted in Figure 8.3. As such, shader intrinsic functions have the capabilities to massively improve the performance of certain shader implementations.

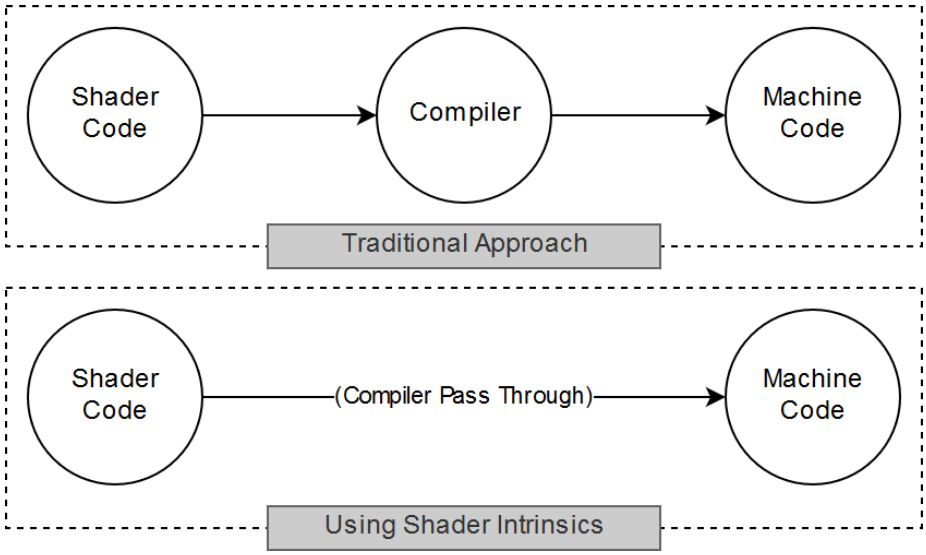


Figure 8.3: Overview of working with and without intrinsic shader functions.

Our proposal for further research would be to gain an understanding of exactly what these intrinsic functions are capable of and how we could possibly utilize them to improve upon our current shaders.

8.3 Collision Feature

A proposed conceptual addition to our grass shader was to imitate the way that grass moves away from a foreign object touching it. The naive way of doing this would be to implement collision detection where each object constantly asks if it is touching some other object. The problem

with this approach is that it requires a lot of computational power. Instead, we have considered an approximation of this method, where collisions are assumed to be happening within a given range.

We propose two different implementations for this:

- Dynamically deforming the mesh, in which points in the mesh are permanently displaced at the points where objects have collided with grass patches. This creates a sort of 'trampling' effect where trails are created in the grass itself.
- Bending individual grass strands away from the grass. Here, the tips of each grass strand would be rotated away from a detected point of collision.

8.4 Optimization

One of the goals of our grass shader is to generate as many strands of grass as possible. Luckily we still have plenty of ways to optimize this. However, our testing shows us a bottleneck-like problem that limits us to between 40.000 and 90.000 strands of grass before performance drops.

We expect this behaviour to be because machine has run out of allocated memory and, as a result, needs to constantly shift the data of the mesh in order to render the last strands of grass. A simple solution to this would be to allocate more memory for our program. Alternatively we could explore ways to lower the memory cost of our implementation. Once this is done, our other optimization methods would have a much greater impact.

8.4.1 Grass on Camera Position

A way to improve the overall performance of our grass shader would be to only render grass within a given radius of the camera position. This would limit the total amount of grass to be rendered, and thereby, the resources spent.

8.4.2 Pixel Shader as Wind

Our grass has a gradient of colour from top to bottom. When rotated, this fact makes the wind effect looks like waves of colour are traveling across the field of grass.

Instead of calculating rotation on distant blades of grass, it might be possible to get away with simply recolouring the blades of grass with a wave-like pattern instead. Although close up, it would likely not look realistic, it may be good enough at a distance. This would save computation power as the rotation is way more costly than a simple recolouring of the pixels.

8.4.3 Unified Rotation

In its current iteration, our grass shader imitates wind in waves by doing rotation-calculations for each individual blade of grass. Each rotation angle is offset by the world-space position of its center point, using a sine-function. As a result, many of the grass strands rotate in the exact same way. A way to optimize our wind effect would be to, somehow, apply the shared rotations to multiple strands of grass, rather than calculate the rotation for each.

Bibliography

[VBT95] Allan Vermeulen, Gabe Beged-Dov and Patrick Thompson. 'The pipeline design pattern'. In: *Proceedings of OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems*. Citeseer. 1995.

[Mar02] Francis T. Marchese. *Computer Graphics - Two dimensional viewing*. 2002. URL: http://csis.pace.edu/~marchese/CG/Lect7/cg_l7.htm. *Accessed 26th May 2019*.

[Lov05] Anthony Lovesey. 'A comparison of real time graphical shading languages'. In: *University of New Brunswick Canada, CS4983 Senior Technical Report* 70 (2005).

[Geo08] Geometrictools.com. *Rotation Representation and Performance Issues*. 2008. URL: https://www.geometrictools.com/Documentation/RotationIssues.pdf. *Accessed 26th May 2019*.

[Zep08] Zephyris. *A representation of the UV mapping of a cube*. 2008. URL: https://en.wikipedia.org/wiki/UV_mapping#/media/File:Cube_Representative_UV_Unwrapping.png. *Accessed 21-May-2019*.

[Jim13] Essentialmath.com Jim Van Verth. *Understanding Quaternions*. 2013. URL: https://www.haroldserrano.com/blog/rotations-in-computer-graphics. *Accessed 26th May 2019*.

[Gał14] Thomasz Gałaj. *What is a programmable rendering pipeline?* 2014. URL: https://shot511.github.io/2014-06-08-tutorial-04-what-is-programmable-rendering-pipeline/. *Accessed 05-May-2019*.

[Joe14a] Learn OpenGL.com Joey de Vries. *Coordinate Systems*. 2014. URL: https://learnopengl.com/img/getting-started/coordinate_systems.png. *Accessed 7-April-2019*.

[Joe14b] Learn OpenGL.com Joey de Vries. *Coordinate Systems*. 2014. URL: https://learnopengl.com/Getting-started/Coordinate-Systems. *Accessed 7-April-2019*.

[Sto14] Jon Story. *Don't be conservative with Conservative Rasterization*. 2014. URL: https://developer.nvidia.com/content/dont-be-conservative-conservative-rasterization. *Accessed 7-April-2019*.

[Scr15a] Scratchapixel.com. *Introduction to Polygon Meshes*. 2015. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh. *Accessed 26th May 2019*.

[Scr15b] Scratchapixel.com. *Rasterization: a Practical Implementation*. 2015. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage. *Accessed 26th May 2019*.

[Ser15] Harold Serrano. *Rotations in computer graphics*. 2015. URL: https://www.haroldserrano.com/blog/rotations-in-computer-graphics. *Accessed 26th May 2019*.

[Cha16] AMD / GPUOpen / Matthaeus Chajdas. *GCN Shader Extensions for Direct3D and Vulkan*. 2016. URL: https://gpuopen.com/gcn-shader-extensions-for-direct3d-and-vulkan/. *Accessed 26th May 2019*.

[Gro16] Khronos Group. *OpenGL, Face Culling*. 2016. URL: https://www.khronos.org/opengl/wiki/Face_Culling. *Accessed 26th May 2019*.

[Mic16] Christoph Michel. *Understanding front-faces - winding order and normals*. 2016. URL: https://cmichel.io/assets/2016/02/winding-order-triangle-unity.png. *Accessed 7-April-2019*.

[Mic17] Microsoft. *Writing Surface Shaders*. 2017. URL: https://docs.unity3d.com/Manual/SL-SurfaceShaders.html. *Accessed 26th May 2019*.

[Uni17] Unity. *Textures*. 2017. URL: https://docs.unity3d.com/Manual/Textures.html. *Accessed 26th May 2019*.

[Dev18] Samsung Developers. *What Is a Graphics API?* 2018. URL: https://developer.samsung.com/tech-insights/vulkan/what-is-a-graphics-api. *Accessed 26th May 2019*.

[Mic18a] Microsoft. *Direct3D 11 Advanced Stages Tessellation*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/direct3d-11-advanced-stages-tessellation. *Accessed 26th May 2019*.

[Mic18b] Microsoft. *Geometry Shader Stage*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/geometry-shader-stage. *Accessed 26th May 2019*.

[Mic18c] Microsoft. *Pixel Shader Stage*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/pixel-shader-stage. *Accessed 26th May 2019*.

[Mic18d] Microsoft. *RasterizerStage*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/d3d10-graphics-programming-guide-rasterizer-stage. *Accessed 26th May 2019*.

[Mic18e] Microsoft. *Shader Semantics*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3dhlsl/dx-graphics-hlsl-semantics. *Accessed 26th May 2019*.

[Mic18f] Microsoft. *Tessellation Progress*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/images/tess-prog.png. *Accessed 21-May-2019*.

[Mic18g] Microsoft. *The Graphics Pipeline*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/overviews-direct3d-11-graphics-pipeline. *Accessed 26th May 2019*.

[Mic18h] Microsoft. *Vertex Shader Stage*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/vertex-shader-stage. *Accessed 26th May 2019*.

[Mic18i] Microsoft. *Work with shaders and shader resources*. 2018. URL: https://docs.microsoft.com/en-us/windows/desktop/direct3dgetstarted/work-with-shaders-and-shader-resources. *Accessed 26th May 2019*.

[Nak18] Masatatsu Nakamura. *Quaternion structure for HLSL*. 2018. URL: https://gist.github.com/mattatz/40a91588d5fb38240403f198a938a593. *Accessed 26th May 2019*.

[Uni18a] Unity. *Materials, Shaders, Textures, and UVs*. 2018. URL: https://docs.unity3d.com/Packages/com.unity.probuilder@4.0/manual/workflow-texture-mapping.html. *Accessed 26th May 2019*.

[Uni18b] Unity. *Surface Shaders with DX11 / OpenGL Core Tessellation*. 2018. URL: https://docs.unity3d.com/Manual/SL-SurfaceShaderTessellation.html. *Accessed 26th May 2019*.

[Wik18] Wikibooks. *Cg Programming/Vertex Transformations*. 2018. URL: https://en.wikibooks.org/w/index.php?title=Cg_Programming/Vertex_Transformations&oldid=3365531. *Accessed 7-April-2019*.

[Mat19] MathWorks.com. *Quaternion inverse*. 2019. URL: https://se.mathworks.com/help/aeroblks/quaternioninverse.html. *Accessed 26th May 2019*.

[Sli19] Justin Slick. *What is 3D Rendering in the CG Pipepline?* 2019. URL: https://www.lifewire.com/what-is-rendering-1954. *Accessed 26th May 2019*.

A Appendix - Shader Examples

A.1 Vertex Shader Example Code - Offsetting vertices

```
v2f vert (inputStruct v)
{
    //declare output struct
    v2f o;

    //Get either 0 or 1 based on the index of the vertex being processed.
    //The result is 0 if index is divisible by 2, 1 otherwise.
    int transformFactor = !(v.index & 1);

    //use transformFactor to 'extend' the position of every other vertex, 0.1 units
    along its normal vector.
    float4 vertPos = v.vertex + (0.1 * transformFactor * v.normal);

    //convert the 'transformed' vertex position directly to clip space, using a built
    in function.
    vertPos = UnityObjectToClipPos(vertPos);

    //place the final vertex position into the output struct and return it.
    o.vertex = vertPos;
    return o;
}
```

A.2 Pixel Shader Example Code - Color gradient

```
fixed4 frag(v2f i) : COLOR
{
    //define two colors in RGBA
    fixed4 topColor = fixed4(1, 1, 1, 1);    //white
    fixed4 bottomColor = fixed4(1, 0, 0, 1); //red

    //linearly interpolate between the two colors, based on the fragment's position in
    respect to the mesh (uv position).
    fixed4 color = lerp(bottomColor, topColor, i.uv.y);

    //return the resulting color
    return color;
}
```

A.3 Geometry Shader Example Code - Exploding primitives

```
//declare the maximum vertex output of this geometry shader. This is for optimization.
[maxvertexcount(3)]
void geom(triangle v2g IN[3], inout TriangleStream<g2f> triStream)
{
    //read the three vertices of the input primitive.
    float3 v0 = IN[0].pos.xyz;
    float3 v1 = IN[1].pos.xyz;
    float3 v2 = IN[2].pos.xyz;

    //calculate the normal vector of the primitive.
    float3 normal = normalize(cross(v1 - v0, v2 - v1));

    //Create the new vertex positions by extending the position of every vertex in the
    original primitive along the normal vector by 0.1 units.
    v0 += normal * 0.1f;
    v1 += normal * 0.1f;
    v2 += normal * 0.1f;

    //convert every new vertex position directly to clip space, using a built in
    function.
    v0 = UnityObjectToClipPos(v0);
    v1 = UnityObjectToClipPos(v1);
    v2 = UnityObjectToClipPos(v2);

    //declare output struct
    g2f OUT;

    //use the output struct to append the new vertices to the geometry shader output.
    OUT.pos = v0;
    triStream.Append(OUT);
    OUT.pos = v1;
    triStream.Append(OUT);
    OUT.pos = v2;
    triStream.Append(OUT);
}
```

A.4 Tessellation Shader(s) Example Code - Uniform Tessellation

```
Shader "Examples/UniformTessellation"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _Uniform("Uniform Tessellation", Range(1, 64)) = 1
        _TessMap("Tessellation Map", 2D) = "black" {}
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100
        Pass
        {
            CGPROGRAM
            #pragma vertex tes
            #pragma fragment frag
            #pragma hull hulls
            #pragma domain domains
            #pragma target 4.6
            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };
            struct cp
            {
                float4 vertex : INTERNALTESSPOSITION;
                float2 uv : TEXCOORD0;
            };
            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 vertex : SV_POSITION;
            };
            struct tf
            {
                float edge[3] : SV_TessFactor;
                float inside : SV_InsideTessFactor;
            };
            sampler2D _MainTex;
            sampler2D _TessMap;
            float4 _MainTex_ST;
            float _Uniform;

            v2f vert(appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
```

```
52                  o.uv = TRANSFORM_TEX(v.uv, _MainTex);
53                  return o;
54              }
55
56              cp tes (appdata v)
57              {
58                  cp p;
59                  p.vertex = v.vertex;
60                  p.uv = v.uv;
61                  return p;
62              }
63
64              [UNITY_domain("tri")]
65              [UNITY_outputcontrolpoints(3)]
66              [UNITY_outputtopology("triangle_cw")]
67              [UNITY_partitioning("integer")]
68              [UNITY_patchconstantfunc("pcf")]
69              cp hulls(InputPatch<cp, 3> patch, uint id : SV_OutputControlPointID)
70              {
71                  return patch[id];
72              }
73
74              tf pcf(InputPatch<cp, 3> patch)
75              {
76                  float p0factor = tex2Dlod(_TessMap, float4(patch[0].uv.x, patch[0].uv.y
    , 0, 0)).r;
77                  float p1factor = tex2Dlod(_TessMap, float4(patch[1].uv.x, patch[1].uv.y
    , 0, 0)).r;
78                  float p2factor = tex2Dlod(_TessMap, float4(patch[2].uv.x, patch[2].uv.y
    , 0, 0)).r;
79                  float factor = (p0factor + p1factor + p2factor);
80                  tf f;
81                  f.edge[0] = factor > 0.0 ? _Uniform : 1.0;
82                  f.edge[1] = factor > 0.0 ? _Uniform : 1.0;
83                  f.edge[2] = factor > 0.0 ? _Uniform : 1.0;
84                  f.inside = factor > 0.0 ? _Uniform : 1.0;
85                  return f;
86              }
87
88              [UNITY_domain("tri")]
89              v2f domains(tf factors,
90                  OutputPatch<cp, 3> patch,
91                  float3 barycentricCoordinates : SV_DomainLocation)
92              {
93                  appdata data;
94                  data.vertex =
95                      patch[0].vertex * barycentricCoordinates.x +
96                      patch[1].vertex * barycentricCoordinates.y +
97                      patch[2].vertex * barycentricCoordinates.z;
98                  data.uv =
99                      patch[0].uv * barycentricCoordinates.x +
100                     patch[1].uv * barycentricCoordinates.y +
101                     patch[2].uv * barycentricCoordinates.z;
102                 return vert(data);
103             }
```

```
104
105             fixed4 frag(v2f i) : SV_Target
106             {
107                 fixed4 col = tex2D(_MainTex, i.uv);
108                 return col;
109             }
110             ENDCG
111         }
112     }
113 }
```

A.5 Surface Shader - Tessellation, UV and Displacement Mapping

This dynamic LOD tessellation implementation, using a surface shader, was implemented using this Unity's tessellation shader tutorials[Uni18b]. It implements many advanced effects easily using the surface shader.

```
Shader "Custom/TessellationGrass" {
  Properties{ //Properties from Unity
    _Tess("Tessellation", Range(1,32)) = 4
    _MainTex("Base (RGB)", 2D) = "white" {}
    _DispTex("Disp Texture", 2D) = "gray" {}
    _NormalMap("Normalmap", 2D) = "bump" {}
    _Displacement("Displacement", Range(0, 1.0)) = 0.3
    _Color("Color", color) = (1,1,1,0)
    _SpecColor("Spec color", color) = (0.5,0.5,0.5,0.5)
  }
    SubShader{
      Tags { "RenderType" = "Opaque" }
      LOD 300

      CGPROGRAM
      //Multiple compiler specifics are defined like to use the surface shader pragma, the tessellation type; distance based, phong shading etc.
      #pragma surface surf BlinnPhong addshadow fullforwardshadows vertex:disp tessellate:tessDistance nolightmap
      #pragma target 4.6
      #include "Tessellation.cginc"

      struct appdata { //Primitives from Unity (tris)
        float4 vertex : POSITION;
        float4 tangent : TANGENT;
        float3 normal : NORMAL;
        float2 texcoord : TEXCOORD0;
      };

      float _Tess; //Anchor

      float4 tessDistance(appdata v0, appdata v1, appdata v2) { //The distance based tessellation, based off of the Unity camera
        float minDist = 2.0;
        float maxDist = 10.0;
        return UnityDistanceBasedTess(v0.vertex, v1.vertex, v2.vertex, minDist, maxDist, _Tess);
      }

      sampler2D _DispTex;
      float _Displacement;

      void disp(inout appdata v) //Hands displacement according to the _DispTex anchor (2d texture)
      {
        float d = tex2Dlod(_DispTex, float4(v.texcoord.xy,0,0)).r * _Displacement;
        v.vertex.xyz += v.normal * d;
```

```
43      }
44
45      struct Input {
46        float2 uv_MainTex;
47      };
48
49      sampler2D _MainTex;
50      sampler2D _NormalMap;
51      fixed4 _Color;
52
53      void surf(Input IN, inout SurfaceOutput o) { //Surface shader specific function
    which applies the main texture and colour, albedo, specular, gloss and normal
    effects.
54        half4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
55        o.Albedo = c.rgb;
56        o.Specular = 0.2;
57        o.Gloss = 1.0;
58        o.Normal = UnpackNormal(tex2D(_NormalMap, IN.uv_MainTex));
59      }
60      ENDCG
61    }
62      FallBack "Diffuse"
63  }
```

B Appendix - Shader Implementations

B.1 Model-Based Implementation Code

This is the complete ShaderLab code for the model-based grass implementation without wind effects applied. This ShaderLab code defines 2 separate shaders that work on the mesh:

1. A simple vertex shader *vert()* that projects the object-space positions of each vertex to clip-space.
2. A pixel shader *frag()* that paints different brightness values of a color on the sides of the model.

```
Shader "Custom/ModelBasedShaderNoWind"
{
    Properties
    {
        _RampTex("Ramp", 2D) = "white" {}
        _grassColor("Grass Color", Color) = (1,1,1,1)
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            sampler2D _RampTex;
            float4 _grassColor;
            float4 _LightColor0; //unity delivers

            struct vertexOutput
            {
                float4 pos : SV_POSITION;
                float3 normal : NORMAL;
            };

            struct vertexInput
            {
                float4 pos : POSITION;
                float3 normal : NORMAL;
            };

            vertexOutput vert(vertexInput vIn)
            {
                vertexOutput vOut;
                vOut.pos = UnityObjectToClipPos(vIn.pos);
                float4 normal4 = float4(vIn.normal, 0.0);
                vOut.normal = normalize(mul(normal4, unity_WorldToObject).xyz);

                return vOut;
            }
```

```
41
42              float4 frag(vertexOutput fIn) : COLOR
43              {
44                  float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
45                  float ramp = clamp(dot(fIn.normal, lightDir), 0.001, 1.0);
46                  float3 lighting = tex2D(_RampTex, float2(ramp, 0.5)).rgb;
47                  float3 rgb = lighting * _grassColor.rgb;
48
49                  return float4(rgb, 1.0);
50              }
51              ENDCG
52          }
53      }
54      FallBack "Diffuse"
55  }
```

B.2 Plane-Based Implementation Code

```
1  // Upgrade NOTE: upgraded instancing buffer 'Props' to new syntax.
2
3  Shader "Custom/GrassTerrainShader" {
4    Properties{
5      _Color("Color", Color) = (1,1,1,1)
6      _MainTex("Albedo (RGB)", 2D) = "white" {}
7      _Glossiness("Smoothness", Range(0,1)) = 0.5
8      _Metallic("Metallic", Range(0,1)) = 0.0
9      _Cutoff("Alpha cutoff", Range(0,1)) = 0.5
10   }
11     SubShader{
12       Tags { "Queue" = "AlphaTest" "IgnoreProjector" = "True" "RenderType" = "
     TransparentCutout" }
13       LOD 200
14       //Disabel culling to force drawing on both sides
15       Cull Off
16
17       CGPROGRAM
18       // Physically based Standard lighting model, and enable shadows on all light
     types
19       #pragma surface surf Standard fullforwardshadows alphatest:_Cutoff
20
21       // Use shader model 3.0 target, to get nicer looking lighting
22       #pragma target 3.0
23
24       sampler2D _MainTex;
25
26       struct Input {
27         float2 uv_MainTex;
28       };
29
30       half _Glossiness;
31       half _Metallic;
32       fixed4 _Color;
33
```

```
34      // Add instancing support for this shader. You need to check 'Enable Instancing'
    on materials that use the shader.
35      // See https://docs.unity3d.com/Manual/GPUInstancing.html for more information
    about instancing.
36      // #pragma instancing_options assumeuniformscaling
37      UNITY_INSTANCING_BUFFER_START(Props)
38        // put more per-instance properties here
39      UNITY_INSTANCING_BUFFER_END(Props)
40
41      void surf(Input IN, inout SurfaceOutputStandard o) {
42        // Albedo comes from a texture tinted by color
43        fixed4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
44        o.Albedo = c.rgb;
45        o.Metallic = _Metallic;
46        o.Smoothness = _Glossiness;
47        o.Alpha = c.a;
48      }
49      ENDCG
50    }
51      FallBack "Diffuse"
52  }
```

B.3 Point-Based Implementation Code

This is the complete ShaderLab code for the point-based grass implementation without wind effects applied. This ShaderLab code defines 3 separate shaders that work on the mesh:

1. A simple vertex shader *vert()* that converts vertices to world-space.
2. A geometry shader *geom()* that handles the construction of each blade of grass.
3. A pixel shader *frag()* that paints each blade of grass with a linear interpolation between two defined colors.

```
1  Shader "Grass/PointGrass/PointGrassThicc"
2  {
3      //Define properties that should be accessible in the unity game engine and their
       default values.
4      Properties
5      {
6          _GHeight("Grass Height", range(0, 3)) = 0.5
7          _GWidth("Grass Width", range(0, 0.5)) = 0.2
8          _GColBot("Bottom Color", Color) = (0,0,0,0)
9          _GColTop("Top Color", Color) = (1,1,1,1)
10     }
11     SubShader
12     {
13         Pass
14         {
15             CGPROGRAM
16             #include "UnityCG.cginc"
17             #include "Assets/_Shaders/ShaderUtils/ShaderUtilities.cginc"
18             #pragma vertex vert
19             #pragma geometry geom
20             #pragma fragment frag
```

```
21
22              float _GHeight;
23              float _GWidth;
24              float4 _GColBot;
25              float4 _GColTop;
26
27              struct v2g
28              {
29                  float4 vertex : POSITION;
30              };
31
32              struct g2f
33              {
34                  float2 uv : UV;
35                  float4 vertex : POSITION;
36              };
37
38              v2g vert(appdata_base v)
39              {
40                  v2g o;
41                  o.vertex = mul(unity_ObjectToWorld, v.vertex);
42
43                  return o;
44              }
45
46              [maxvertexcount(8)]
47              void geom(point v2g IN[1], inout TriangleStream<g2f> triStream)
48              {
49                  float3 center = IN[0].vertex.xyz;
50
51                  //define base vectors to use in movement.
52                  float3 up = float3(0, 1, 0);
53                  float3 right = float3(1, 0, 0);
54                  float3 forward = float3(0, 0, 1);
55
56                  //side vertexes are offset half of _GWidth from center of grass.
57                  float halfWidth = (_GWidth) / 2;
58
59                  //grass height is randomized based on XZ world coordinates of grass.
60                  float heightCalculated = _GHeight * randomRange(center.xz, 0.5, 1);
61
62                  float4 pos[4];
63                  //WINDING ORDER CLOCKWISE, REMEMBER
64                  pos[0] = float4(center + up * heightCalculated, 1.0f);
65                  pos[1] = float4(center + right * halfWidth, 1.0f);
66                  pos[2] = float4(center - right * halfWidth, 1.0f);
67                  pos[3] = float4(center + forward * halfWidth, 1.0f);
68
69                  g2f OUT;
70
71                  //front face
72                  //top
73                  OUT.vertex = mul(UNITY_MATRIX_VP, pos[0]);
74                  OUT.uv = float2(0, 1);
75                  triStream.Append(OUT);
```

```
76                  //right
77                  OUT.vertex = mul(UNITY_MATRIX_VP, pos[1]);
78                  OUT.uv = float2(1, 0);
79                  triStream.Append(OUT);
80                  //left
81                  OUT.vertex = mul(UNITY_MATRIX_VP, pos[2]);
82                  OUT.uv = float2(0, 0);
83                  triStream.Append(OUT);
84
85                  //back face 1
86                  OUT.vertex = mul(UNITY_MATRIX_VP, pos[3]);
87                  OUT.uv = float2(0, 0);
88                  triStream.Append(OUT);
89
90                  OUT.vertex = mul(UNITY_MATRIX_VP, pos[0]);
91                  OUT.uv = float2(0, 1);
92                  triStream.Append(OUT);
93
94                  //back face 2
95                  OUT.vertex = mul(UNITY_MATRIX_VP, pos[3]);
96                  OUT.uv = float2(0, 0);
97                  triStream.Append(OUT);
98
99                  OUT.vertex = mul(UNITY_MATRIX_VP, pos[1]);
100                 OUT.uv = float2(0, 0);
101                 triStream.Append(OUT);
102             }
103
104             half4 frag(g2f IN) : COLOR
105             {
106                 //linearly interpolate between two colors _GColBot and _GColTop based
    on fragment y-coordinate
107                 float4 col = lerp(_GColBot, _GColTop, IN.uv.y);
108                 return col;
109             }
110
111             ENDCG
112         }
113     }
114 }
```

C | Appendix - Sourced Code

C.1 Library for Quaternion Rotation

The below code was sourced from the github gist: `https://gist.github.com/mattatz/40a91588d5fb38240403f198a938a593`

Created by Masatatsu Nakamura (Mattatz on GitHub).

```
#ifndef __QUATERNION_INCLUDED__
#define __QUATERNION_INCLUDED__

#define QUATERNION_IDENTITY float4(0, 0, 0, 1)

#ifndef PI
#define PI 3.14159265359f
#endif

// Quaternion multiplication
// http://mathworld.wolfram.com/Quaternion.html
float4 qmul(float4 q1, float4 q2)
{
    return float4(
        q2.xyz * q1.w + q1.xyz * q2.w + cross(q1.xyz, q2.xyz),
        q1.w * q2.w - dot(q1.xyz, q2.xyz)
        );
}

// Vector rotation with a quaternion
// http://mathworld.wolfram.com/Quaternion.html
float3 rotate_vector(float3 v, float4 r)
{
    float4 r_c = r * float4(-1, -1, -1, 1);
    return qmul(r, qmul(float4(v, 0), r_c)).xyz;
}

// A given angle of rotation about a given axis
float4 rotate_angle_axis(float angle, float3 axis)
{
    float sn = sin(angle * 0.5);
    float cs = cos(angle * 0.5);
    return float4(axis * sn, cs);
}

// https://stackoverflow.com/questions/1171849/finding-quaternion-representing-the-
    rotation-from-one-vector-to-another
float4 from_to_rotation(float3 v1, float3 v2)
{
    float4 q;
    float d = dot(v1, v2);
    if (d < -0.999999)
    {
        float3 right = float3(1, 0, 0);
```

```
        float3 up = float3(0, 1, 0);
        float3 tmp = cross(right, v1);
        if (length(tmp) < 0.000001)
        {
            tmp = cross(up, v1);
        }
        tmp = normalize(tmp);
        q = rotate_angle_axis(PI, tmp);
    }
    else if (d > 0.999999) {
        q = QUATERNION_IDENTITY;
    }
    else {
        q.xyz = cross(v1, v2);
        q.w = 1 + d;
        q = normalize(q);
    }
    return q;
}

float4 q_conj(float4 q)
{
    return float4(-q.x, -q.y, -q.z, q.w);
}

// https://jp.mathworks.com/help/aeroblks/quaternioninverse.html
float4 q_inverse(float4 q)
{
    float4 conj = q_conj(q);
    return conj / (q.x * q.x + q.y * q.y + q.z * q.z + q.w * q.w);
}

float4 q_diff(float4 q1, float4 q2)
{
    return q2 * q_inverse(q1);
}

float4 q_look_at(float3 forward, float3 up)
{
    float3 right = normalize(cross(forward, up));
    up = normalize(cross(forward, right));

    float m00 = right.x;
    float m01 = right.y;
    float m02 = right.z;
    float m10 = up.x;
    float m11 = up.y;
    float m12 = up.z;
    float m20 = forward.x;
    float m21 = forward.y;
    float m22 = forward.z;

    float num8 = (m00 + m11) + m22;
    float4 q = QUATERNION_IDENTITY;
    if (num8 > 0.0)
```

```
99      {
100         float num = sqrt(num8 + 1.0);
101         q.w = num * 0.5;
102         num = 0.5 / num;
103         q.x = (m12 - m21) * num;
104         q.y = (m20 - m02) * num;
105         q.z = (m01 - m10) * num;
106         return q;
107     }
108
109     if ((m00 >= m11) && (m00 >= m22))
110     {
111         float num7 = sqrt(((1.0 + m00) - m11) - m22);
112         float num4 = 0.5 / num7;
113         q.x = 0.5 * num7;
114         q.y = (m01 + m10) * num4;
115         q.z = (m02 + m20) * num4;
116         q.w = (m12 - m21) * num4;
117         return q;
118     }
119
120     if (m11 > m22)
121     {
122         float num6 = sqrt(((1.0 + m11) - m00) - m22);
123         float num3 = 0.5 / num6;
124         q.x = (m10 + m01) * num3;
125         q.y = 0.5 * num6;
126         q.z = (m21 + m12) * num3;
127         q.w = (m20 - m02) * num3;
128         return q;
129     }
130
131     float num5 = sqrt(((1.0 + m22) - m00) - m11);
132     float num2 = 0.5 / num5;
133     q.x = (m20 + m02) * num2;
134     q.y = (m21 + m12) * num2;
135     q.z = 0.5 * num5;
136     q.w = (m01 - m10) * num2;
137     return q;
138 }
139
140 float4 q_slerp(in float4 a, in float4 b, float t)
141 {
142     // if either input is zero, return the other.
143     if (length(a) == 0.0)
144     {
145         if (length(b) == 0.0)
146         {
147             return QUATERNION_IDENTITY;
148         }
149         return b;
150     }
151     else if (length(b) == 0.0)
152     {
153         return a;
```

```
154     }
155
156     float cosHalfAngle = a.w * b.w + dot(a.xyz, b.xyz);
157
158     if (cosHalfAngle >= 1.0 || cosHalfAngle <= -1.0)
159     {
160         return a;
161     }
162     else if (cosHalfAngle < 0.0)
163     {
164         b.xyz = -b.xyz;
165         b.w = -b.w;
166         cosHalfAngle = -cosHalfAngle;
167     }
168
169     float blendA;
170     float blendB;
171     if (cosHalfAngle < 0.99)
172     {
173         // do proper slerp for big angles
174         float halfAngle = acos(cosHalfAngle);
175         float sinHalfAngle = sin(halfAngle);
176         float oneOverSinHalfAngle = 1.0 / sinHalfAngle;
177         blendA = sin(halfAngle * (1.0 - t)) * oneOverSinHalfAngle;
178         blendB = sin(halfAngle * t) * oneOverSinHalfAngle;
179     }
180     else
181     {
182         // do lerp if angle is really small.
183         blendA = 1.0 - t;
184         blendB = t;
185     }
186
187     float4 result = float4(blendA * a.xyz + blendB * b.xyz, blendA * a.w + blendB * b.w
    );
188     if (length(result) > 0.0)
189     {
190         return normalize(result);
191     }
192     return QUATERNION_IDENTITY;
193 }
194
195 float4x4 quaternion_to_matrix(float4 quat)
196 {
197     float4x4 m = float4x4(float4(0, 0, 0, 0), float4(0, 0, 0, 0), float4(0, 0, 0, 0),
    float4(0, 0, 0, 0));
198
199     float x = quat.x, y = quat.y, z = quat.z, w = quat.w;
200     float x2 = x + x, y2 = y + y, z2 = z + z;
201     float xx = x * x2, xy = x * y2, xz = x * z2;
202     float yy = y * y2, yz = y * z2, zz = z * z2;
203     float wx = w * x2, wy = w * y2, wz = w * z2;
204
205     m[0][0] = 1.0 - (yy + zz);
206     m[0][1] = xy - wz;
```

```
207     m[0][2] = xz + wy;
208
209     m[1][0] = xy + wz;
210     m[1][1] = 1.0 - (xx + zz);
211     m[1][2] = yz - wx;
212
213     m[2][0] = xz - wy;
214     m[2][1] = yz + wx;
215     m[2][2] = 1.0 - (xx + yy);
216
217     m[3][3] = 1.0;
218
219     return m;
220 }
221
222 #endif
```

D Appendix - Test instancing scripts & computer specification

Below are the scripts used to place instances of each grass implementation, for the purpose of testing performance.

D.1 Instancing script for model-based implementation

```
//Evenly distributes copies of the defined gameobject in an area defined by areaSize,
    in a grid pattern
void InstantiateGrass()
    {
        float distanceX = areaSize.x / grassNumber;
        float distanceZ = areaSize.y / grassNumber;

        for (int i = 0; i < grassNumber; ++i)
        {
            Vector3 origin = transform.position;
            origin.x = -(areaSize.x / 2) + i * distanceX;

            for(int j = 0; j < grassNumber; j++)
            {
                origin.z = -(areaSize.y / 2) + j * distanceZ;

                //make copies of the defined gameobject (prefab) at the given position
    (origin)
                Instantiate(prefab, origin, Quaternion.identity, this.transform);
            }
        }
    }
```

D.2 Instancing script for plane-based implementation

```
    //Update is a Unity built-in function that is run once for each in-game frame.
    void Update()
    {
        List<Matrix4x4> grassMaterices = new List<Matrix4x4>(grassNumber*3);

        float distanceX = size.x / grassNumber;
        float distanceZ = size.y / grassNumber;

        for (int i = 0; i < grassNumber; ++i)
        {
            Vector3 origin = transform.position;
            origin.x = -(size.x / 2) + i * distanceX;

            for(int j = 0; j < grassNumber; j++)
            {
```

```
                origin.z = -(size.y / 2) + j * distanceZ;

                //define the three planes that make up the plane cluster
                grassMaterices.Add(Matrix4x4.TRS(origin, Quaternion.identity, Vector3.
    one));
                grassMaterices.Add(Matrix4x4.TRS(origin, Quaternion.AngleAxis(60,
    Vector3.up), Vector3.one));
                grassMaterices.Add(Matrix4x4.TRS(origin, Quaternion.AngleAxis(120,
    Vector3.up), Vector3.one));
            }

        }
        //Inputs the defined mesh (grassMesh) directly into the GPU rendering pipeline,
     at the given position.
        Graphics.DrawMeshInstanced(grassMesh, 0, material, grassMaterices);
    }
```

D.3 Instancing script for point-based implementation

```
//Creates a mesh of points, evenly distributed over the area defined by sizeX and sizeZ
void CreateMeshAll(Mesh mesh)
    {
        numPoints = numPointsX * numPointsZ;
        Vector3[] points = new Vector3[numPoints];
        int[] indecies = new int[numPoints];
        Color[] colors = new Color[numPoints];

        float distanceX = sizeX / numPointsX;
        float distanceZ = sizeZ / numPointsZ;

        int index = 0;
        for (int i = 0; i < numPointsX; ++i)
        {
            float posX = (-sizeX/ 2) + i * distanceX;

            for(int j = 0; j < numPointsZ; j++)
            {
                float posZ = -(sizeZ /2 ) + j * distanceZ;

                points[index] = new Vector3(posX, 0, posZ);
                indecies[index] = index;
                colors[index] = Color.white;
                index++;
            }
        }
        //allows meshes of more than ~65,000 vertices
        mesh.indexFormat = UnityEngine.Rendering.IndexFormat.UInt32;
        mesh.vertices = points;
        mesh.colors = colors;
        mesh.SetIndices(indecies, MeshTopology.Points, 0);
    }
```

D.4 Test computer specification

Motherboard:	Asus ROG STRIX B360-I Gaming
CPU:	Intel Core i5 8600K
GPU:	Nvidia GeForce GTX 970
RAM:	16GB DDR4 Dual-Channel 2133MHz

* All parts running at stock clock-speeds.

E | Appendix - Program Manual

This section is a short guide on how to get a minimal instance of our final grass terrain implementation up and running in Unity.

E.1 Requirements

- A functional computer running Windows 10 OS.
- A Direct3D 11 installation.
- Unity 5, release 2018.3.6f1.
- A basic 3D project open in Unity with a standard Camera(created automatically).

E.2 Pre-requisite: Importing project files into the Unity editor

The attached project folder is a Unity project with the following steps already done, so this can be skipped.

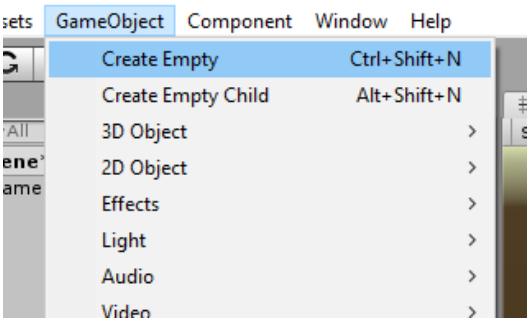
Our implementation works based on 4 files, that must be included in the Unity project files before we can proceed:

1. The mesh-generator script - `ProjectPointCloudMeshGen.cs`
2. The Material that we apply the grass shader to - `PointGrassMat.mat`
3. The utility file that the shader requires to compile properly - `ShaderUtilities.cginc`
4. The grass shader itself - `PointGrass7_WaveWind.shader`

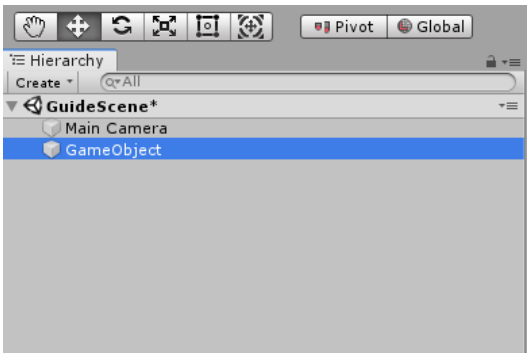
Each of these files represent a component of our implementation and must all be included into the Unity `Assets` folder, found under the `Project` sub-window. One thing to note is that the utility file must have the location path: `Assets/_Shaders/ShaderUtils/ShaderUtilities.cginc`

All other files can be placed freely.

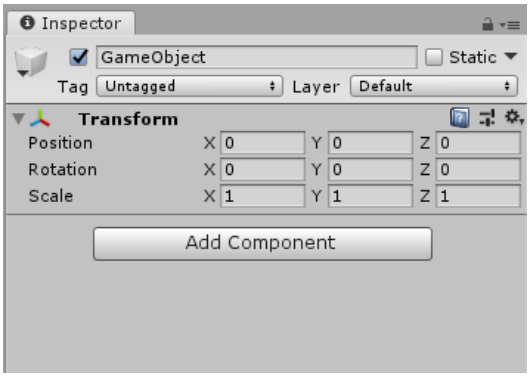
E.3 Setting up the GameObject



Create empty game objects using the GameObject tab.



Game objects can be found in the Hierarchy window.



The Inspector shows lists all game object components and their attributes. All game objects are created with a Transform component that describes 3D position.

The basic building blocks of Unity are the so-called *Game Objects*. Game objects are essentially containers for various other standard and user-defined components.

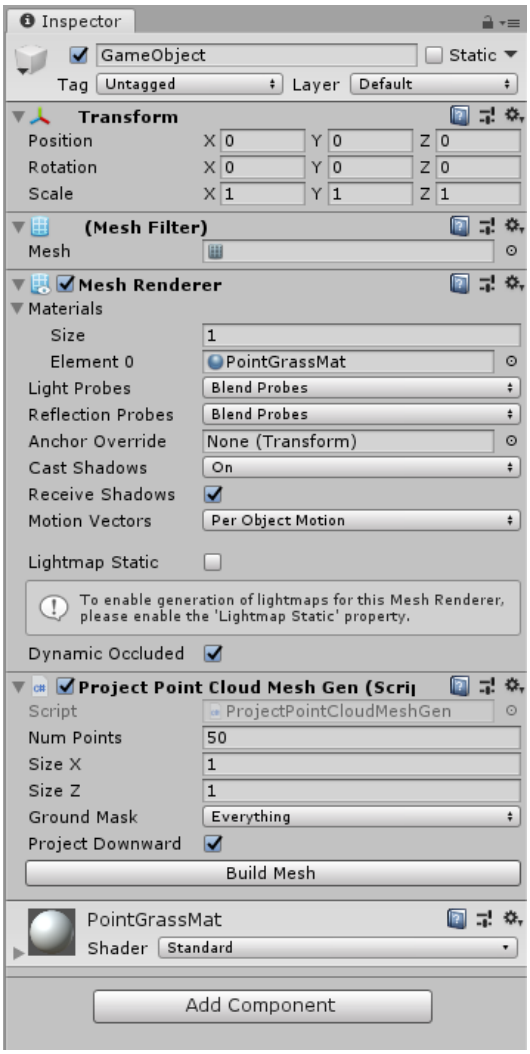
We will be using an empty game object to hold each of our grass generation components, so we must first create it:

- Click the `GameObject` tab at the top of the Unity window. This will bring up a menu used for creating the various types of standard game objects.
- Click the `Create Empty` option under this menu.

This will create a game object in the scene Hierarchy, which is found under the `Hierarchy` sub-window.

Click the game object instance and notice how the `Inspector` sub-window changes. The inspector gives a list of all the components that a game object includes. Currently, the only component attached to our game object is the `Transform` component. Let's add some more.

E.4 Adding the mesh generator and generating a mesh



The mesh generator script automatically adds two other components, used for rendering the mesh. The public variables are editable from the editor itself.

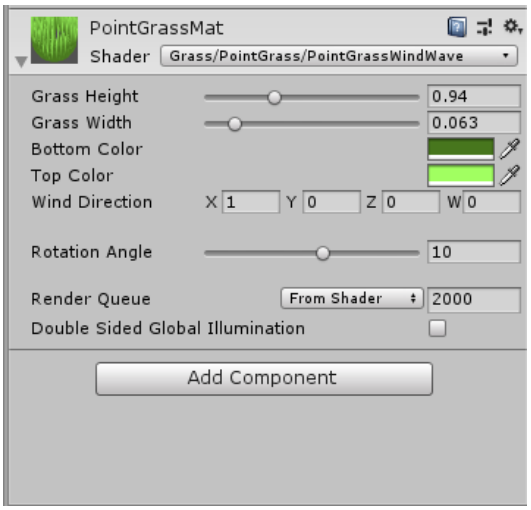
The script in `ProjectPointCloudMeshGen.cs` is a component that allows us to generate point-cloud meshes for our grass-shader to be applied to. We'll add this by dragging the script from the project window, directly into the inspector window itself. Now look at the inspector window again.

The script is set up to automatically add two other components, `(Mesh Filter)` and `Mesh Renderer` to the game object. Unity needs these to handle our mesh-rendering, but we wont be fiddling with them, so just ignore them.

Under the script component we see the various attributes / public variables that the script contains. These attributes start out with some default values that can be changed to adjust the mesh-generation. Most of them are self-explanatory, but `Ground Mask` is a little more complex. `Ground Mask` is tied into how Unity handles Raycasting and we wont be diving further into this - just leave the value at `Everything`.

To generate the mesh we will be using, click the `Build Mesh` button at the bottom of the script attributes. This will place a number of purple points, representing the mesh, in the Unity `Scene` window.

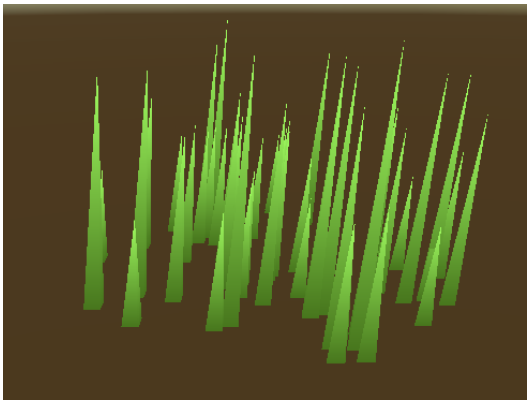
E.5 Applying the material and shader



The material component shows the properties of the shader that its bound to.

Unity uses so-called *Material* components to keep track of which shaders should be applied to an object. `PointGrassMat.mat` is such a material. Drag `PointGrassMat.mat` into an empty space on the inspector for the game object. If the mesh doesn't look like grass yet, drag the `PointGrass7_WaveWind.shader` file onto the material component to apply the shader. This will also open the properties for that shader, under the given material component. Click the little triangle on the shader component in the inspector to open or close these properties.

E.6 Adjusting the shader



Use the shader properties to modify the look and movement of each blade of grass.

The properties shown under the shader properties have the following effect on the look of the grass:

- `Grass Height`: The height of the grass.
- `Grass Width`: The total width of the grass.
- `Bottom Color` and `Top Color`: The colors that the grass interpolates over to create a smooth color-transition.
- `Wind Direction`: A vector describing the direction that the wind is blowing in.
- `Rotation Angle`: The extremes of the wind-rotation mechanism. Larger values make the grass rotate faster, as well as rotate more.

The remainder of the properties are automatically added to the shader, and we simply ignore these.