# CS354 Assignment 4 Design Document

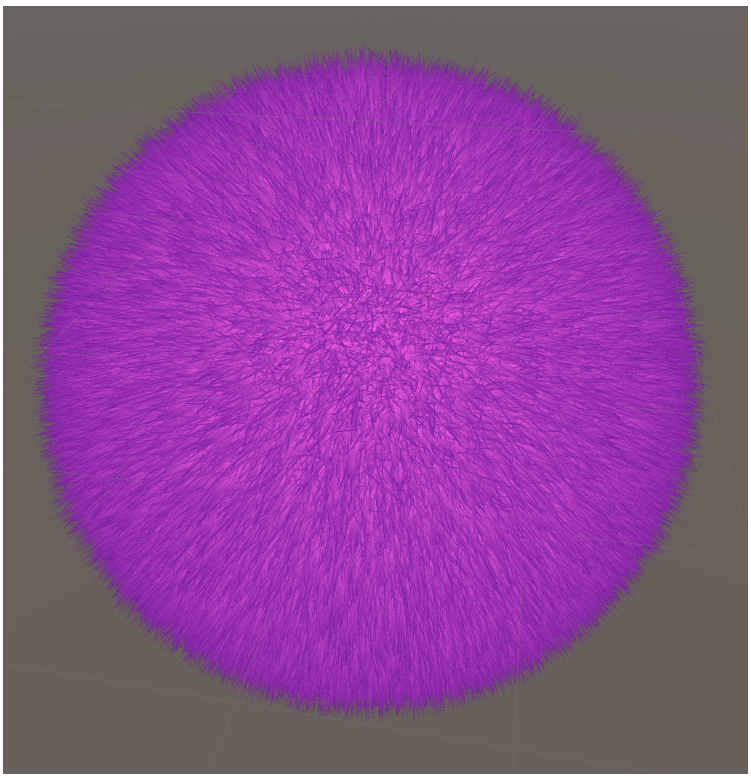
Joaquin Niles, Chinmay Walavalkar, James Stewart

## Overview

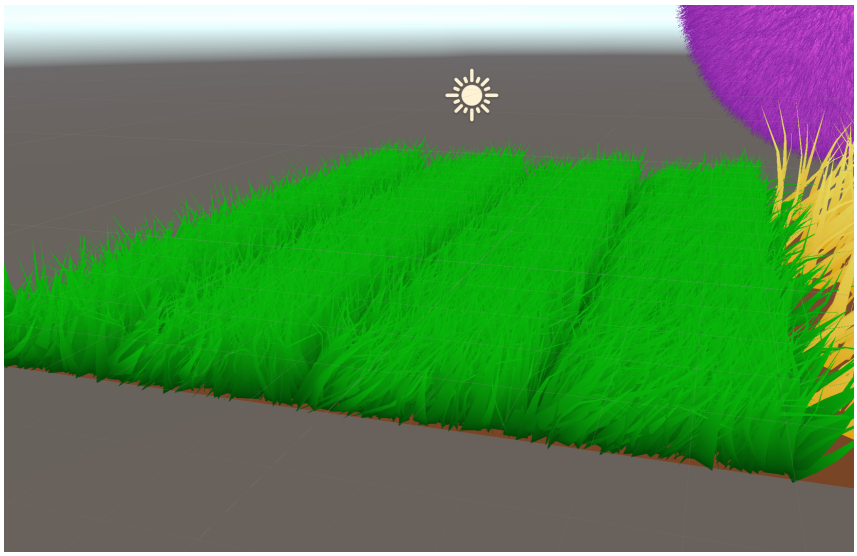
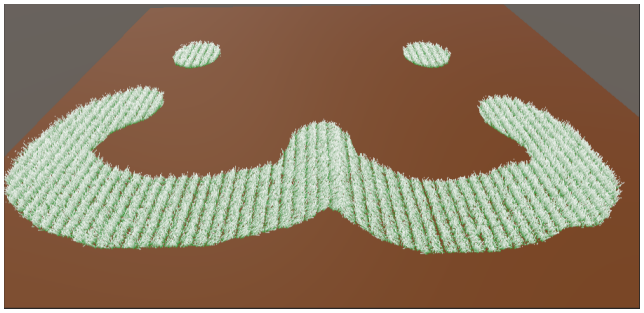
For this project we set out to create a non-photorealistic grass model. This involved working with shaders in unity to create grass groundcover and animate the grass. We had to learn how to use geometry and tesselation shaders to get a denser field of grass and smoother individual blades, as well as the geometry that goes into procedurally modeling grass in a way that looks good without using too many resources.

## Feature Implementation Strategies

| Mechanic | Results |
|---|---|
| Grass Colors | **Implementation:** This was relatively simple to implement. We define a base color and a tip color. Then, based on the vertical component of our current uv coordinate of the grass blade, we interpolate between the colors using lerp, which is a built-in unity function that lets us interpolate between 2 vectors (which in this case are colors).<br><br>`return lerp(_Base_Color, _Tip_Color, g.uv.y);` |
| Grass Geometry Generation | **Implementation:** For generating our grass, we generate a blade on each vertex of the mesh we put the grass material on. All grass blades are defined in tangent space, meaning that if the mesh the grass is on is slanted the grass will also be slanted (its basically defined along the normal of the mesh surface). |

| | |
|---|---|
| |  <br><br>The actual geometry for the grass is semi-complex. We have a predetermined amount of segments, consisting of 2 vertices each, and a tip vertex at the top. We generate these segments one at a time through a for-loop, where we then determine how far we want to offset for grass curvature.<br><br>`float3 vOffset = float3(width * (1 - t), pow(t, _Grass_Curvature) * forward, height * t);`<br><br>To transform the tip of the grass blade correctly, we apply 5 different matrices, those being the tangent matrix, the wind matrix, the rotation matrix, the bend matrix, and the sway matrix, which are applied in that order. The tangent matrix acts as a pseudo model matrix, just defined tangentially to the mesh.<br><br>`float3x3 tipTfMatrix = mul(mul(mul(mul(tangentToLocal, windMatrix), rotMatrix), bendMatrix), swayMatrix);` |
| Tessellation | **Implementation:** Tessellation in Unity URP is implemented through hull and domain shaders, which occur before the geometry shader, where the actual grass is generated. In tessellation, we subdivide the mesh so that we have more vertices to put grass blades on. To decide where these new vertices are located on the mesh, we use a heuristic derived from catlikecoding's guide on implementing tessellation in Unity URP. There are 4 factors that affect how we subdivide a mesh through tessellation, 3 edges for the new triangle we make, and a |

| | |
|---|---|
| | representation for the inside of the triangle. These factors tell us how many sub-edges we want to make per edge. For our heuristic, we take the distance between the 2 vertices that make up the edge, and then divide by a variable that represents the distance between grass blades.<br><br>`return edgeLength / _GrassTessellationDistance;`<br><br>For the inside triangle factor, we take the average of the factors for each edge, this will give us different factors for each edge, which gives us a non-uniform distribution of the new vertices that gives us a result that makes the grass look more natural.<br><br>`f.inside = (f.edge[0] + f.edge[1] + f.edge[2]) / 3.0f;`<br><br>For the new vertices generated using tessellation, we need to assign attributes to them, such as their positions and normals. To get those, we are given barycentric coordinates to interpolate between, so that we can get accurate values for the newly generated vertices. |
| Grass Motion | **Implementation:** The grass motion consists of two types, directional wind and grass sway. The direction of the two is the same (wind direction) to keep the dynamics cohesive and are implemented somewhat similarly.<br><br>The First, directional wind, uses the position of a blade of grass, the built in hlsl _Time value, and various user inputs to give the impression of wind blowing across a scene. Originally the wind was intended to follow a Gerstner wave, but currently can be represented by the equation<br><br>`wind = clamp(((TWO_PI / _Wavelength)  * (vPos.x * windAxis.y + vPos.z * windAxis.x - _Wind_Speed * _Time.x)) % 1, -_Wind_Strength, _Wind_Strength)`<br><br>Where `_Wavelength`, `_Wind_Speed`, `_Wind_Strength`, are set by the user and `windAxis.y` and `windAxis.x` are the normalized versions of `_Wind_Direction_Y` and `_Wind_Direction_X`, which are also set by the user. The modulo does result in some unwanted behavior where blades of grass will jump suddenly at the edge of a gust of wind but overall, the results look good and there is a clear sense of wind moving across the screen. The direction of the wind is set by the user using the aforementioned `_Wind_Direction_Y` and `_Wind_Direction_`. A rotation matrix is then generated with `wind` as the angle and `normalize(float3(_Wind_Direction_Y, _Wind_Direction_X, 0.0f))` as the axis of rotation. This matrix is then used in calculating the tip position of each blade of grass<br><br>The Second, grass sway, is implemented in almost the same manner but is more randomized. A rotation matrix is generated using the same axis of rotation. The angle of rotation is calculated using the equation<br><br>`sway = (_Sway_Intensity / 20) * normalize(float3(rand(vPos), rand(vPos), 0.0f)) * ((rand(vPos) - _Sway_Speed * _SinTime.x));`<br><br>Where the `_Sway_Intensity` and `_Sway_Speed` are set by the user. The |

| | |
|---|---|
| | range of these values is quite large but results tend to look best with small values for each. The resulting rotation matrix is again used in calculating the position of the tip of a blade of grass.<br><br>  |
| Grass Visibility Mapping | **Implementation:**<br>By utilizing a grayscale image, we can create maps that determine where grass is and isn't rendered. To sample the map, we use tex2Dlod, a HLSL built in function for texturing 2D images.<br><br>`tex2Dlod(_VisibilityMap, float4(input[0].uv, 0, 0)).r;`<br><br>By determining the color of the sample, and comparing the 'gray' value to a threshold, we can create more dynamic looking terrain or abominations like the image below.<br><br>  |
| FPS benchmarking | **Findings:** Before talking about results, it's important to know that Unity has frustum culling automatically implemented as part of its engine, so any results are based on the optimizations given by Unity.<br><br>As for the results. The FPS never dropped below 200, and reached an upwards of 700-800 FPS, even when having the whole scene, which consisted of a large sphere, a large field, and 3 smaller fields within the frustum at once. The average when everything was within the camera frustum was around 300-400 FPS, usually around 320 FPS. |

# References:


Daniel Ilett, *Making Zelda: Breath of the Wild Stylised Grass in Unity URP*, https://www.youtube.com/watch?v=MeyW_aYE82s

Daniel Ilett, *Six Grass Rendering Techniques in Unity*, https://www.youtube.com/watch?v=uHDmqfdVkak&t=274s

Jasper Flick, *Tessellation*, https://catlikecoding.com/unity/tutorials/advanced-rendering/tessellation/

Jasper Flick, *Waves*, https://catlikecoding.com/unity/tutorials/flow/waves/

Roystan, *Grass Shader,* https://roystan.net/articles/grass-shader/

Stylized Station, *How Grass Works in Ghost of Tsushima*, https://www.youtube.com/watch?v=G8HH_pMKOhk

Random number generation for HLSL, https://answers.unity.com/questions/399751/randomity-in-cg-shaders-beginner.html?childToView=624136#answer-624136

3x3 Rotation Matrix Construction, https://gist.github.com/keijiro/ee439d5e7388f3aafc5296005c8c3f33

Camera Control script is sourced from: https://gist.github.com/gunderson/d7f096bd07874f31671306318019d996