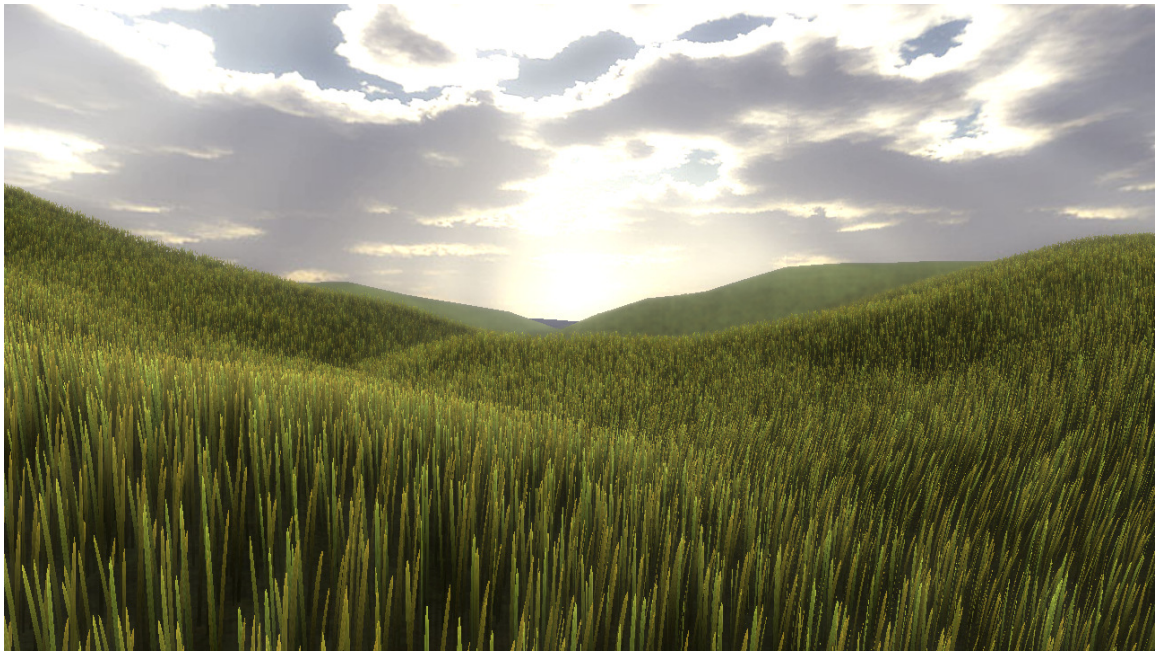
REALISTIC REAL-TIME GRASS RENDERING

MASTERS OF INTERACTIVE TECHNOLOGY

THESIS



To the Graduate Faculty:

I am submitting herewith a project written by Edward Lee entitled “Realistic Real-Time Grass Rendering”. I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Interactive Technology in Digital Game Development, with Specialization in Software Development.

_____

Wouter van Oortmerssen, PhD.
Supervisor

I have read this Project
and recommend its acceptance:

_____

Jani Kajala
Advisor or Second Reader

Accepted for the Faculty:

_____

Gary Brubaker
Deputy Director, Academics
SMU Guildhall

REALISTIC REAL-TIME GRASS RENDERING

A Thesis Presented to the Graduate Faculty of

The Guildhall

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Masters of Interactive Technology

with a

Major in Software Development

by

Edward Lee, B.S.
Bachelor of Science Mathematics of Computation, University of California, Los Angeles
December 17, 2010

ACKNOWLEDGEMENTS

I would like to acknowledge the following people who have helped in one way or another during the development of this thesis. First want to give thanks to my supervisor, Wouter van Oortmerssen, PhD., who has enlightened me with insightful advice that made it possible to render so many blades of grass in real-time. The faculty professors, Jeff Wofford and Jani Kajala, have provided valuable technical advice that has affected the performance of the simulation. Also, I would like to thanks Kevin Boulanger, PhD., for providing invaluable advice about varying rendering techniques for vegetation. The art direction of this project was heavily influenced by the faculty art professors Eric Walker and David Cherry.

Lee, Edward B.S., University of California, Los Angeles, 2008

REALISTIC REALTIME GRASS RENDERING

Supervisor: Wouter van Oortmerssen, PhD.

Master of Interactive Technology conferred January 4, 2011

Thesis completed December 9, 2010

This master's thesis outlines an algorithm that renders an infinite field of grass in real-time. My technique relies heavily on the geometry shader in order to procedurally generate each blade in real-time. Wind conditions are affected by the player movement, in which vector fields is generated and diffused mathematically throughout the scene.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

DEDICATIONS

I would love to dedicate this work to my lovely parents and grandparents as well as my girlfriend, Robyn Hirokawa, all of whom have showed nothing but support during this entire process.

CHAPTER 1: INTRODUCTION

Grass is the most prevalent form of vegetation on the Earth's surface. Thus, it is an essential element in rendering realistic outdoor scenes. But due to the geometric complexity of grass, rendering it realistically in real-time can be difficult and computationally expensive. Furthermore, to produce life-like representations of grass requires faithful physical simulations of movement provided by wind currents.

BACKGROUND OF THE STUDY

Although the shape of a single grass blade of grass can easily be represented by a handful of triangles, blades never exist naturally in sparse formation. In nature, grass can span an entire field, with millions of blades covering up a large meadow. Traditionally, little effort has been placed on rendering realistic grass because of its geometric complexity. In practice, graphics engines have approximated the portrayal of grass by scattering sparse patches of geometry across the entire scene. But as consumer hardware and gaming consoles become more computationally effective, the possibility for geometrically complex simulations of grass becomes more promising. New advances in graphics hardware technology as well as innovations in real-time rendering techniques allow for more efficient rendering of complex objects. Games such as the PSN title *Flower* have proven that grass rendering is possible on current console technology.

There are several new developments in the field of graphics that had made the rendering of complex grass possible. One significant development is *geometry instancing* (Kalra, 2009), which allowed rendering of multiple copies of the same mesh in a scene with one render call. Using geometric instancing allows grass patches to be rendered throughout the scene with one render call, thus reducing the communication overhead between the graphics engine and GPU.

Another significant development in grass rendering is the use of patterned displacement maps (Bakay, Lalonde, & Heidrich, 2002). This technique is extremely memory efficient and can be used to generate shells in arbitrary slicing directions for high frequency data used to represent fields of grass. This technique also allows for physical simulations of wind on individual blades.

Furthermore, focusing on the geometry alone is not sufficient; emphasis must also be placed on the lighting of grass in order to produce believable results. A recently development of *bidirectional texture function* (Shah, Kontinnen, & Pattanaik, 2005) allows for efficient representations of highly detailed surfaces under varying illumination and viewing conditions. BTF is employed to synthesize grass textures for given viewing and illumination conditions.

PROBLEM AND PURPOSE STATEMENT

The purpose of my study is to research and implement an efficient way to render grass which can run on current consumer graphics hardware. The problem with current methods of rendering grass is that they traditionally use patched geometry to render grass. This causes a very patterned and artificial appearance that is not very believable.

The ultimate goal is to develop a method that allows for real-time rendering of an infinite surface of dense grass, contoured on to the terrain. This can be achieved by using the levels of detail approach combined with frustum culling and alpha-to-coverage rendering methods.

We achieve seamless level of detail by segmenting the rendering of grass into various levels of details. Different levels of detail are rendered depending on how close the user is to the current patch of grass. When the user is close to the patch, each individual grass blade is represented by high-detailed geometry. Grass blades that are medium distance away are represented by less-detailed geometry. And finally, distant patches are rendered using simple flat textures that contour the terrain.

Wind conditions of the scene are simulated by using a customized vector field that draws heavily from fluid dynamics. The viewer of the scene is able to apply wind forces to the scene by simply moving around in the environment.

Research or Project Question

Is there a way to implement realistic grass that is rendered in real-time, has no popping due to LOD changes, with grass blades that curve and bend according to wind conditions?

SIGNIFICANCE OF THE STUDY

As computer graphics move further towards hyper-realism, rendering life-like simulations of outdoor scenes involving grass becomes progressively more significant for player immersion. Outdoor scenes have conventionally been described by sparse or patched geometry, often negatively impacting its believability and thus negating player

immersion. Being able to efficiently render the complex geometry of grass in real-time can significantly improve the visual impact of the scene and enhance its believability.

OVERVIEW OF METHODOLOGY

Given that the most important aspect of developing an effective grass-rendering algorithm is computational efficiency, this study was conducted through the process of implementing a single method and iterating upon it while comparing the speed and efficiency of each iteration while attempting to maintain the highest visual fidelity of the scene. The analysis will be purely quantitative and the development process will have its greatest emphasis on attaining maximal frame rates. However, due to the fact that all cards have varying bottlenecks, the study will be implemented on varying resolutions to determine where pixel shader bottlenecks become geometry bottlenecks.

A secondary focus of the study involves the believability of the grass rendering. Since the purpose of this study is to develop an algorithm to render immersive, realistic grass, having a faithful simulation of real-life grass is essential. The believability of the geometry is a qualitative measure which is judged on how realistic the grass appears in comparison to real-life. Believability is a result of how dense we are able to populate the scene as well as how closely we are able to simulate realistic grass animation and light conditions.

Furthermore, a balance has to be made between computational efficiency and grass rendering fidelity. For example, the most realistic method would be to render the entire scene with millions of grass blades. However, doing so would slow the frame rate to a crawl. Also, having too complex of a lighting equation might put too much load on

the GPU. Therefore, the best lighting technique will be determined during the implementation process. To achieve interactive frame rates, we use a level-of-detail scheme combined with frustum-culling to maximize rendering efficiency.

The study does not use sampling methods and therefore does not necessitate the involvement of participants. And since it does not require participants, ethical considerations involving the methodology of this study are insignificant. The entire study will be conducted solely based on results attained from the varying implementations.

LIMITATIONS

Given that this algorithm relies heavily on the geometry shader, hardware that does not support geometry shaders, such as older graphics cards or consoles, will be unable to run the simulation.

DEFINITIONS

Displacement map is the computer graphics technique of using a texture to cause an effect where the actual geometric position of points over the textured surface is displaced.

Level-of-Detail (LOD) is the computer graphics technique for decreasing the complexity of 3D objects according to the position of the viewer or from the object's importance in the scene.

Vertex shader (VS) is a shader program executed on the graphics processing unit. Vertex Shaders are run for each vertex in a geometry.

Geometry Shader (GS) is a shader program that is executed after the vertex shader in the GPU pipeline. The geometry shader takes in a whole primitive and can alter or generate vertices on the fly.

Pixel Shader (PS) is a shader program that is executed after the vertex shader. The pixel shader essentially interpolates the vertex values that are passed through the vertex shader.

Alpha-to-coverage is a multisampling technique which uses the alpha channel of textures as a coverage mask for anti-aliasing.

Frame-rate is a quantitative measurement of how many frames are rendered per second.

CHAPTER 2: LITERATURE AND FIELD REVIEW

The literature published on techniques for rendering grass has been quite broad, ranging from volumetric rendering to displacement maps. The technique I used to implement grass is a selective mixture of techniques which are presented in these papers in addition with more advanced wind-simulation techniques modeled with the custom vector fields.

LITERATURE REVIEW

The process of searching for valuable literature on the topic of grass rendering has included probing through papers published in conferences such as SIGGRAPH and GDC as well as academic white papers published by graduate and doctorate students of computer science. A lot of valuable information on a variety of techniques was presented in these papers.

A white paper by Boulanger (Boulanger, 2008) presents a method that allows for real-time rendering of vast surfaces of grass with dynamic lighting, dynamic shadows and anti-aliasing. His approach uses levels-of-detail to provide an effective compromise between rendering complexity and rendering speed. By using GPU-based grass rendering system, he integrates dynamic illumination and the parallax effect to display large fields of grass. To generate grass patches, Boulanger launches a particle at the root of each blade and marks the position the particle at key steps. Applying gravity to the particle

allows for a curved trajectory. These key steps become segments of the grass blade in which the vertices are built around to produce the geometric representation of the blade. Rendering these patches using geometric instancing allows for rendering of several patches in one render call. To simulation the motion of grass due to wind, he shears the base volume in the direction of the wind vector. Wind calculations and vertex translations are all done in the vertex shader. In the fragment shader, Boulanger calculates a shadow mask that is mapped onto a cylinder which is then used as a visibility function on casting shadows. To effectively render vast amounts of grass, Boulanger uses three levels of details of the grass in which he blends between them for smooth detail transitions.

Bakay (2002) presents another approach to render fields of grass while tackling the problem of grass animating in the wind in real-time. His approach employs the GPU's vertex shader to displace transparent shells that represent the body of the grass's blade. These shells are translated along the surface normals of the terrain. This allows him to achieve convincing results of the grass's movement due to wind vectors. A single texture

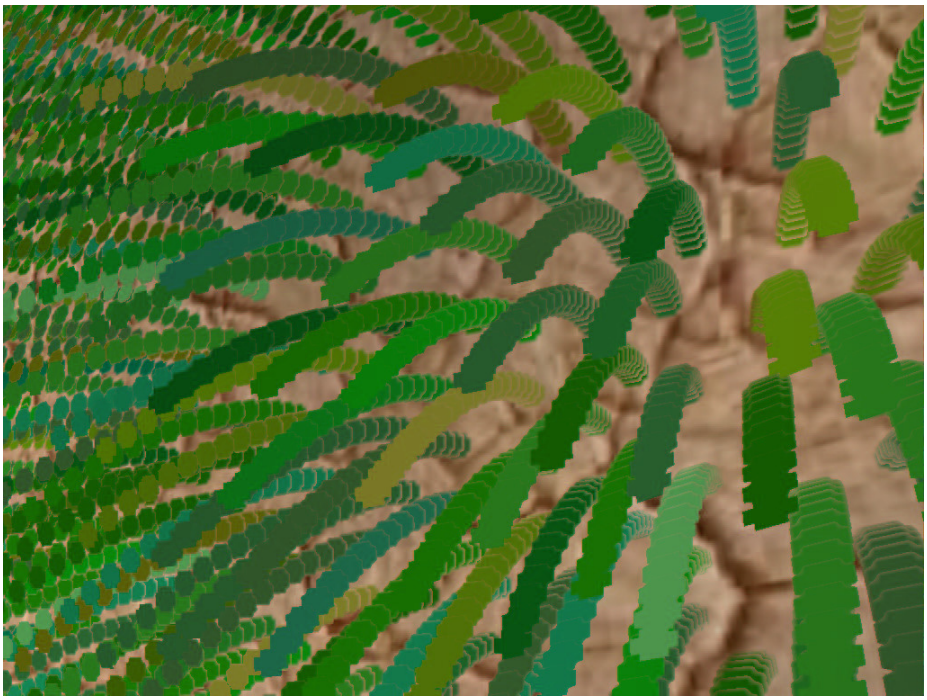


Figure 1 - Grass Rendering Using Shells

is used to generate all the shell textures, in which he encodes the height of the grass in the alpha channel. His approach relies heavily on the GPU and can be applied to similar surfaces such as hair and fur. The issue with this method is that if viewed up close, the grass would appear very chunky and unrealistic. Furthermore, lighting models on the shell method will appear unrealistic because the shells used to model grass are round while grass blades are flat.

But grass cannot just be visually appealing; to provide the ultimate realism in fields of grass, each of the grass blades must properly react and collide with objects. *GPU-Based Responsive Grass* (Jens, Kolb, & Salama, 2009) provided a method of using GPU-based distance maps to test for collision and for collision resolution. In his method, vertices are dynamically lit with ambient occlusion based on irradiance volume techniques. Alpha-to-coverage completes the illusion by simulating the semitransparent nature of grass. However, his method is highly dependent on the position and the density of the grass. More specifically, it requires the grass regions to be on flat surfaces, which may be unusable due to the fact that naturally-occurring grass may not necessarily exist on perfectly flat surfaces.

Anu Kalra (2007) provides a method of using DirectX10's geometric instancing technique to efficiently render thousands of blades of grass. Geometry instancing allows the reuse of geometry when drawing multiple similar objects in a scene. This can be extremely helpful in rendering grass since it requires rendering the same object multiple times. His method also uses alpha-to-coverage to help render the meadow without alpha-sorting thousands of blades of grass. The shortcoming of this method is it renders each

blade as a stiff quad and therefore does not allow for blades of grass to curve according to wind conditions.

FIELD REVIEW

Due to the geometric complexity of rendering realistic grass, games have traditionally rendered grass using sparse geometry or patches. But as rendering technology becomes more efficient and graphics cards become more effective, the capability of rendering complex geometry becomes more of a possibility. It wasn't until recently that consumer graphics hardware and gaming consoles were powerful enough to approach the realistic rendering of large fields of grass.

The most notable game that have used grass rendering is the PSN title *Flower*. *Flower* is a PlayStation Network game developed by thatgamecompany where the player must control the wind to blow a flower petal thru vast fields of grass. Their grass rendering implementation used the full power of Playstation 3's multi-cell processors and became their flagship technology for the product. Having such a high level of realism in the grass allowed the user to be fully immersed in its environment. Without proper grass rendering, the user would never be able to feel as if he/she was fully immersed in the lush grassland. Their graphics engine was able to realistically change between levels of detail, allowing players to travel seamlessly throughout a large open terrain. All these elements are combined together to make the players feel a sense of open freedom.

Figure 2 - *Flower* in-game screenshot

Another game that has a believable grass implementation is *Far Cry 2*. However, their implementation was not as convincing as *Flower* since their technique of rendering grass involved populating the scene with patched geometry. Their wind simulation seems to be generated using simple sine waves and does not seem appear to be completely convincing since there are no global or local wind conditions applied the grass.

The most recent game to have notable grass rendering technology is *Red Dead Redemption*. However, their implementation is similar to that of *Far Cry 2* such that grass is only represented by sparse patches of grass. Their grass is implemented via static meshes with no influence in wind conditions, thus making the grass appear very stiff and non-realistic.

SUMMARY

As shown in the field review, game development is still not at a point where effective real-time rendering of complex hyper-realistic grass scenes is a viable option.

Even the most advanced engines in game development such as CryTek and RAGE (Rockstar's in-house engine) resort to using simple sparse geometry to represent grass. *Flower* is an exception since it had very limited gameplay and thus was able to devote all of PlayStation 3's multi-cell processors in rendering grass. Most games out there will not have the luxury of being able to devote all of its computational resources in to rendering grass. More specifically, most games must also share computational resources with the game's subsystems such as AI and other game logic.

However, the literature review on the topic seems to be more promising. A lot of new advancements in the academic realm have been made in developing effective and efficient techniques for rendering grass. Techniques such as volume rendering and geometric instancing (Boulanger, 2008) allows for many patches at different locations on the terrain to be rendered with one rendering call. Combined with the level-of-detail technique, it allows large sets of grass to be rendered at real-time speeds. Custom vector field mathematics will be utilized to model wind conditions. Combining all these techniques together, I believe that it is possible to render realistic grass in real time.

CHAPTER 3: METHODOLOGY

Grass is the most dominate form of vegetation on the earth's surface, thus making it an essential element in rendering outdoor scenes. Traditionally, video games have not focused much on rendering realistic grass due to the geometric complexity and expense of rendering realistic grass. Traditional representations of grass usually involve rendering patterned patches of grass which negatively affects player immersion. However, as the computational power of modern graphics processors become progressively more effective, the possibility of rendering realistic outdoor scenes with lush fields of grass becomes more promising.

Grass fields have traditionally been animated using simple sinusoidal waves, often causing the scene to appear artificial and lifeless. To simulate realistic wind conditions, a custom vector field is applied to calculate the wind vectors that influence the grass patches. Ultimately, the goal of this study is to discover and implement a method that allows for rendering dense grass scenes, with no level-of-detail popping and with realistic wind conditions.

PRODUCT AND DEVELOPMENT PROCESS

The research is designed around developing a computationally efficient rendering method that can be run on current consumer hardware. During the research process, a single method would be implemented, iterated upon, and evaluated on several factors,

with the highest emphasis on frame rate and visual fidelity. However, just having a high frame rate is not sufficient; the scene must also appear realistic and immersive to the user. The grass should appear dense and each blade should interact with the wind forces in a believable manner. The end product of this research would be a realization of a method that balances realism and computational efficiency.

Terrain

The goal of this study is to simulate an infinite field of grass. The player should be able to walk in any direction and observe waving fields of grass on rolling hills.

To create the terrain layout, a 2D height field is generated using Perlin noise. The Perlin noise algorithm produces a pseudo-random appearance which gives a natural looking height-map. Thus, Perlin noise is widely used in computer graphics to generate natural effects like cloud, smoke and fire.

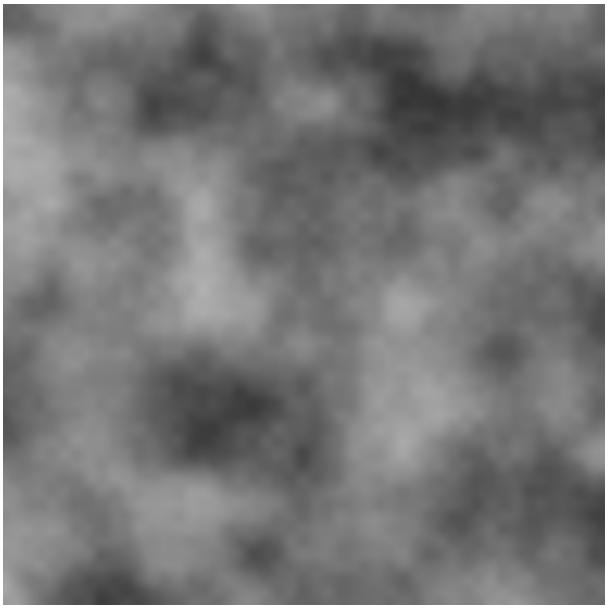


Figure 3 - Perlin Noise 2D Texture

The entire terrain layout is generated mathematically using the heightmap produced from the Perlin noise algorithm. Each position in the 2D heightmap defines the normalized height value of the terrain at that specific position. The figure above shows a heightmap that is generated using the Perlin noise algorithm. The values are normalized such that the black pixels have the value 0.0 and the white pixels have the value 1.0. To achieve the desired result, simply scale each value in the height by the maximum height of the terrain.

The Perlin noise heightmap represents a bounded layout of the terrain. However, in this study, the player must be able to travel freely in any direction. Thus, precaution must take taken to ensure that the terrain can be tileable. The player should be able to continuously travel along the seams of where the terrain is being tiled together without noticing any gaps. To accomplish this, the values at the edges of the heightmap need to be averaged out so that the heights are equal. This allows the height values at the edges of the terrain to be equal and therefore tileable.

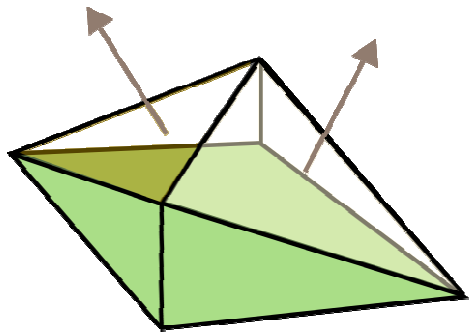


Figure 4 - Each Quad in Terrain has Two Planes

Four neighboring positions in the heightmap define a quad in the terrain. Each position in the heightmap can be spaced out to reflect the width of the terrain. For each quad in the terrain, there exist two triangles which each lie on a separate plane in

Euclidean space. During load-time, the plane data for all quads are calculated and saved to a buffer. To calculate the plane equation of a triangle, take the three vertices and calculate the cross product. The result of the cross product is the vector that defines the normal of the plane. So solve for the plane's height value (or D), take any vertex of the triangle, plug into the plane equation and solve for D.

$$Ax + By + Cz + D = 0$$

$$D = -Ax - By - Cz$$

Rendering the terrain floor is trivial once all the vertices in the terrain have been determined. The simplest way is to render the terrain layout wherever it is needed, tiling it wherever it is visible. However, frustum culling and other visible occlusion techniques can be used to minimize the number of vertices to be rendered by the GPU.

Wind Simulation

In order for the user to feel a sense of immersion in the grassy scene, the player must be able to interact with environment. In this study, the user acts as the only wind component of the simulation, in which the viewer navigates the scene and exerts wind onto the grassy field environment. The player will be in total control of all the wind conditions in the scene to maximize the interactivity.

Firstly, it is necessary to generate a grid of vectors that will be used to represent the current state of all wind vectors in the simulation. The wind vectors are represented by a vector field, with each vector corresponding to one patch or grid in the simulation space. Each grid consists of two vectors, one that pertains to the wind's velocity and another that pertains to the wind's acceleration. The wind's velocity vector determines

how much force to exert onto the grass blades in that patch. The figure below shows the vector field $f(x, y) = \{x, y\}$, in which force is applied in a radial pattern away from the player (origin).

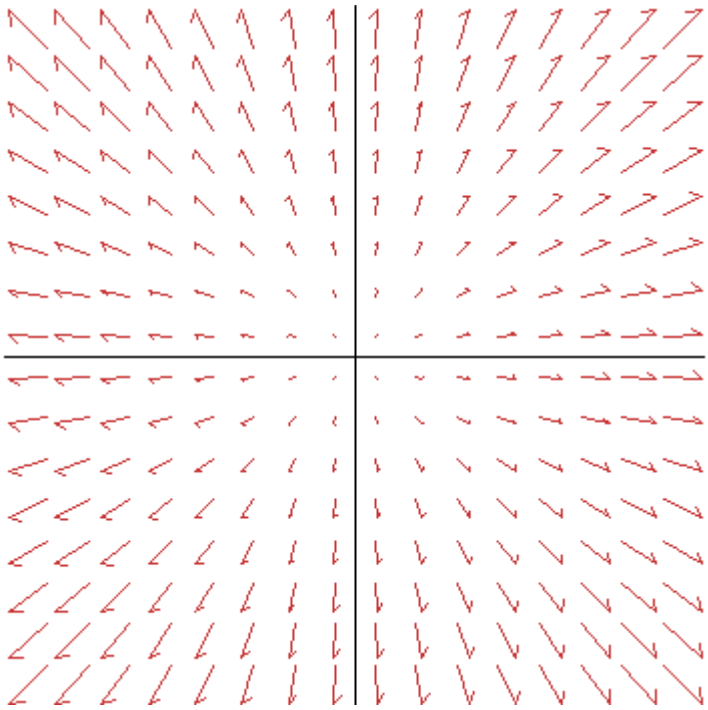


Figure 5 – Example Vector Field: f(x,y)={x,y}

The wind force is applied in the direction that the player is moving. The magnitude of the wind force is proportional to the magnitude of the velocity of the player. More specifically, the magnitude of the wind force applied to the scene should be strong near the player and decrease gradually the farther away the grass patch is from the player.

Figure 5's vector field is not a good candidate to represent the wind vector field because the wind force should be strong near the origin and decrease gradually the farther the vector is from the player. However, the vector field shows that the magnitude of the vector increases as the vector is farther away from the origin.

One might be tempted to just divide the vectors by the magnitude-squared of the vector. However, this causes the curve to become asymptotic when the vectors are extremely close to the origin. That is, the magnitude of the vector approaches infinity as the vector approaches the origin.

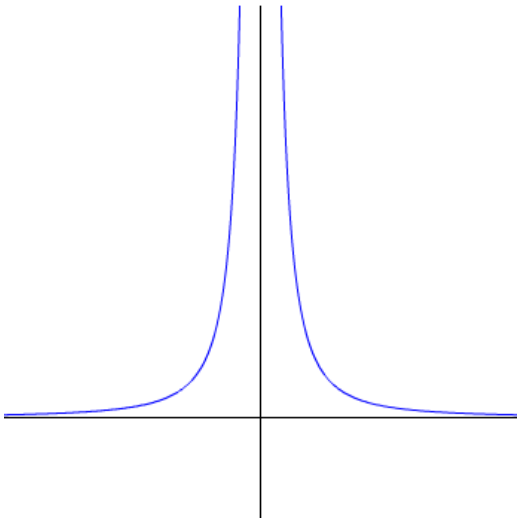


Figure 6- Asymptotic Issues of Diving by Length-Squared: $f(x) = \frac{1}{x^2}$

So what is really desired is a coefficient that can be multiplied to each vector such that there are no asymptotic fallacies. To simulate this gradual change in magnitude of the wind force, the vector fields generated from the equation $f(x, y) = \{x, y\}$ must be multiplied by a coefficient which is derived from the classical statistical bell curve.

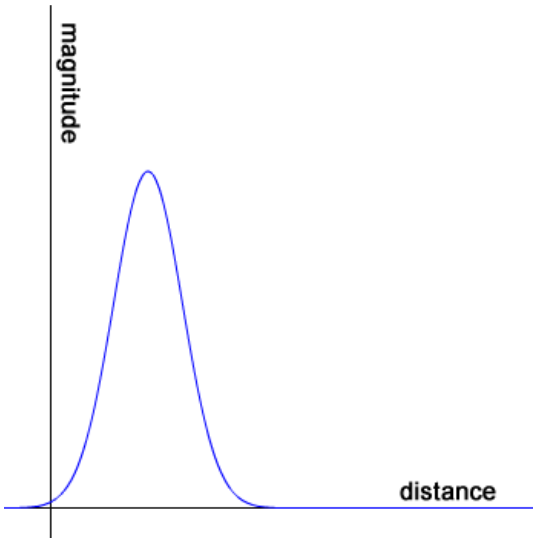


Figure 7 - Bell Curve Function: $f(x) = e^{-(10x-2)^2}$

Notice that, in the figure above, the magnitude of the vector is strong near the origin and decreases as the vectors span out into the distance. Having the magnitude start at zero will cause the vector field to not have asymptotic issues.

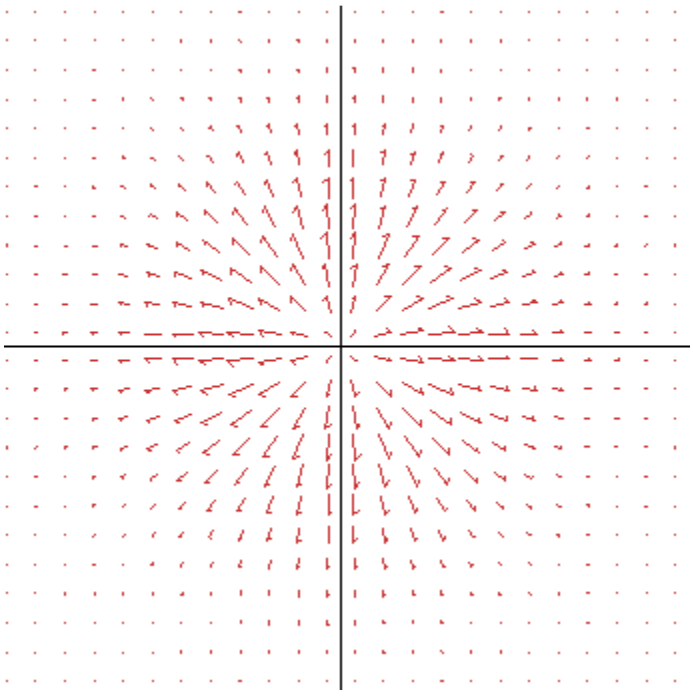


Figure 8 - Vector Field with Bell Curve Coefficients: $f(x,y) = \{xe^{-(x^2+y^2)},\ ye^{-(x^2+y^2)}\}$

The figure above depicts a vector field with strong wind vectors near the player and gradually weak vectors as the vectors move outwards.

However, wind should only blow in the general forward direction that the player is moving. A player who is moving forward does not blow wind in the opposite direction. The current vector field blows wind uniformly around the player (origin). To resolve this issue, first take the dot product between the wind vector and the movement direction to determine if it the current vector is facing the same direction. Clamp the dot product result to the range [0,1] so that if the value of the dot product less than zero (facing opposite directions), the vector will become zero and therefore not affect the vector field. Multiply the current vector by this value will attain the desired vector field.

The vector field that is described above becomes the acceleration vector field that is applied to the wind vector field simulation. During each update of the simulation, apply a simple Euler Integration for each wind velocity vector in the grid to determine the current velocity of the wind at the specific grid position. Dampen the acceleration and diffuse the velocity vectors with its surrounding vectors so that the velocity vector fields do not grow too large. To calculate the current velocity at a grid, apply Euler integration:

$$CurrVel = CurrVel + Accel * deltaTime$$

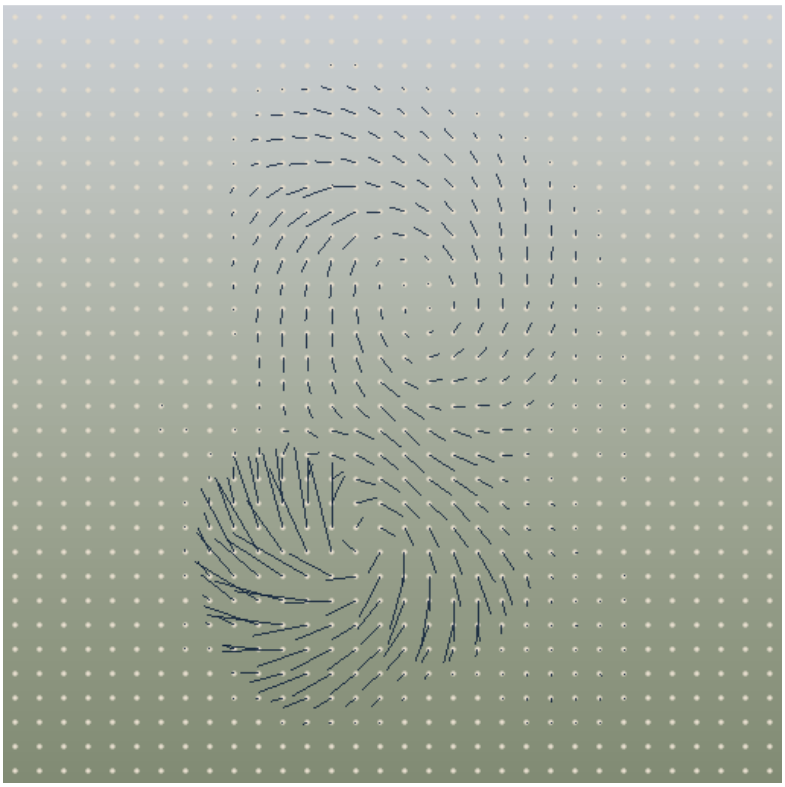


Figure 9 - Velocity Vector Field

The user is expected to be able to travel in any direction for a limitless amount of time. However, the wind vector field is bounded. It is inaccurate to simply tile the vector field infinitely since it does not make any sense to tile wind vectors across infinite space. Since the edges of the vector field generally have a near-zero magnitude, the vector field can be made to "follow" the player.

In order to accomplish this, track the current grid position of the player so that whenever the player moves to another grid, all the vectors are shifted the direction that the player traveled. For example, say the player moves up one grid; shift all the grids up one so that the player is always maintained near the center. This causes the player to be always near the center of the vector grid while giving the impression of an infinite vector field.

Vertex Buffer – Root Positions

Given that the communication overhead between the CPU and the GPU is relatively expensive, one overarching goal of this project is to put as much computation on to the GPU as possible to decrease the risk of bottlenecking the computer bus. This is accomplished by passing the minimal amount of data needed for the GPU to procedurally generate the grass blades.

The first step of the process is to generate a grass patch vertex buffer. Each vertex in the vertex buffer defines the root positions of the grass blades in a single patch. This vertex buffer is created during load-time and then passed to the GPU. The vertex buffer only contains the positional data of each of the grass blade's root position in objects space. This is done by randomly populating a square patch with random positions in the x-z plane.

Note that only one vertex buffer is needed for all levels-of-detail. Since the root positions are randomly populated in the patch, the researcher only needs to decrease the number of root positions passed through the graphics pipeline for each level-of-detail during the render call.

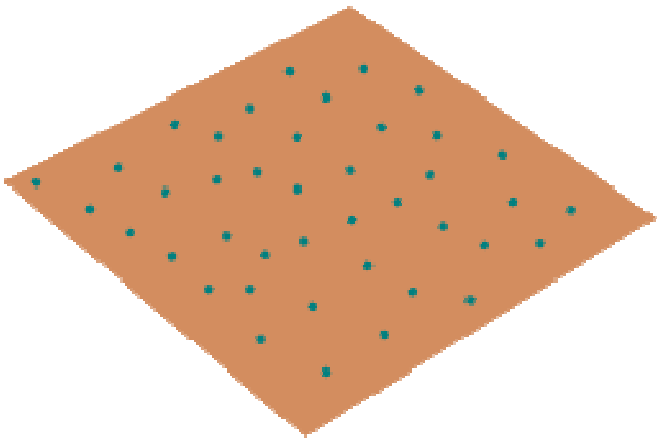


Figure 10 - Grass Patch Vertex Buffer of random Root Positions

The patch only makes a small chunk of the overall grass field. To cover an entire field, the patch is reused and rendered wherever it is needed. Using this patch, the algorithm can populate the entire scene to give the appearance of a dense field.

In this algorithm, all the positions that the grass patch needs to be rendered are first gathered. Using the camera's frustum, a container (STL vector) is populated with grid positions that exist within the frustum. Since the patches can be sloped to contour the terrain, the patches are represented by axis-aligned bounding boxes (AABB) that bound the entire patch. A simple method would be to gather all the positions within the bounding box of the frustum, and then apply a simple frustum-collision check (AABB versus frustum) on each patch to determine if the patches can be visible within the camera's frustum.

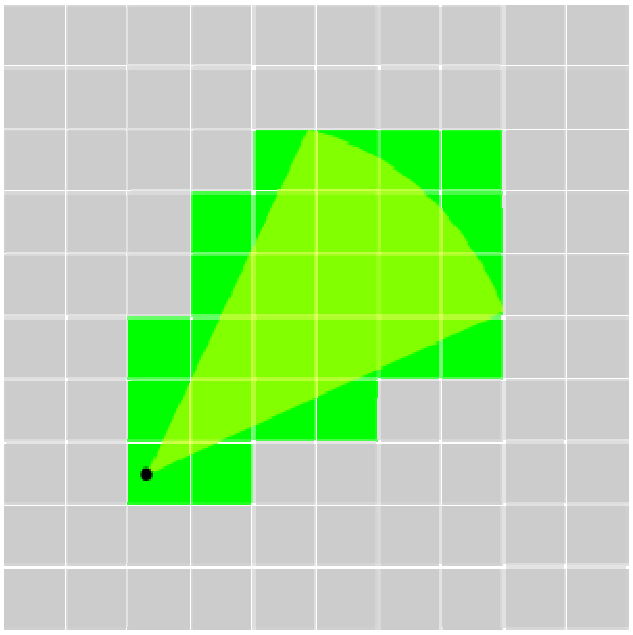


Figure 11 - Patches Gathered Within Frustum

The researcher must note that grass blades can sway beyond the patch's bounding box. For example, if there is a heavy wind, the grass blade may lean outside of its bounds. This maybe cause issues where patches are culled out of the scene even though the grass blades may be seen. In order to fix this issue, simply extend the bounding box to account for the movement of the grass blades.

Level-of-Detail

Level-of-detail schemes increase the efficiency of rendering by only rendering detail only when it is needed. So whenever a grass patch position is added to the container (of all positions to be rendered), its distance from the position of the camera is used to calculate its level-of-detail (LOD). Since the first level-of-detail contains the most geometrically-complex details of the grass, the method must provide as much detail as possible for closer blades. For patches that are positioned at a great distance away from

the camera, less complex geometry can be used to represent far-away blades. This method allows the GPU to process the minimal number of vertices needed to fully represent the scene.

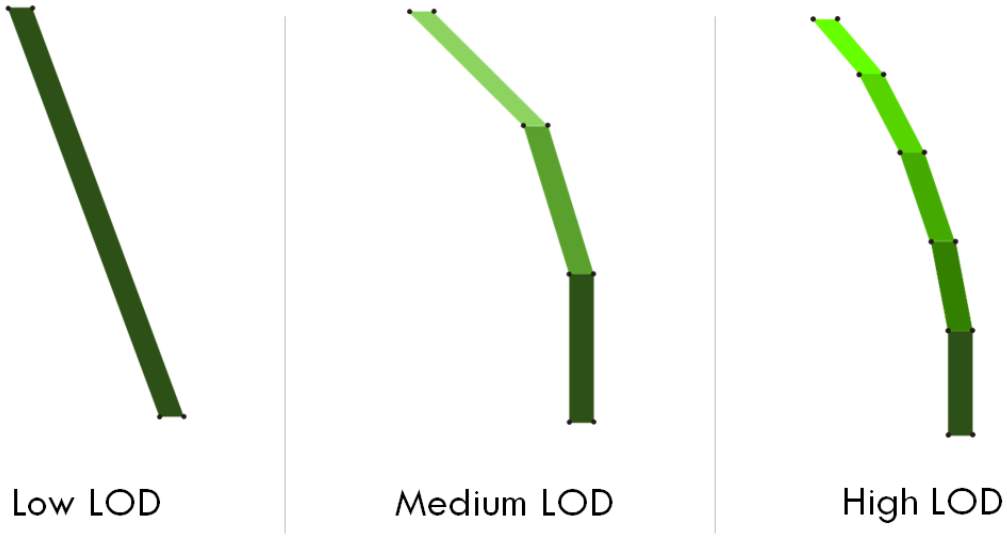


Figure 12 - Grass's Level-of-Detail

The function to map distance-to-camera to level-of-detail should have an inverse-squared relation. The level-of-detail should be extremely detailed near the camera and decrease drastically in complexity as the patches move out into the distance.

Figure 13 - Inverse Squared Function: $f(x) = \frac{1}{(x-0.6)^2}$

The function must be tuned slightly so that the there is no visible popping when the player is moving and transitioning between two levels-of-detail. This tuning depends on the camera's field of field and patch size.

It is essential that the closest patch to the camera not only have the highest detail but also have the highest density of blades. As the patch become farther away from the camera, the density of the patch becomes less significant. Therefore, the number of grass blades needed to represent the grass patch is proportional to the distance that the patch is to the camera. Decreasing the number of grass blades in distant patches reduces their geometric complexity, which allows for a greater number of vertices to be focused on closer patches where it is most essential.

The researcher must make sure to gradually decrease the number of blades per level-of-detail or else the viewer may notice some “popping” when patches transition between levels-of-detail. Like the function that maps distance-to-camera to level-of-detail, the number of grass blades rendered for a patch should have an inverse-squared relation to its distance.

Rendering Setup

Before rendering the grass patches, the wind vector field must be sent in to a buffer in the GPU. Then, the vertex buffer is bound as well as all the textures need to render the grass blades.

For each position in the grid, the position, level-of-detail and the current terrain data of that specific grid is sent to the GPU and the render call is summoned. The terrain data exists as two float4 variables, each of which represents a plane for the two planes that exist within a patch. The level-of-detail exists as an integer and the grid position exists as a float2. All data pertaining to rendering the grids can be sent per render call or can be sent all in one batch to decrease the communication overhead between the CPU and the GPU.

Since the number of root positions decrease as the level-of-detail decreases, the user only needs to limit the number of vertices sent from the vertex buffer in through the graphics pipeline when invoking the render call for a specific patch. Note that the user only needs one vertex buffer for all levels-of-detail, thus allowing only one vertex buffer to be bound for all levels-of-detail.

Geometry Shader

When the GPU renders the patch's vertex buffer, the vertex buffer's vertices are first passed through the geometry shader. Using the level-of-detail integer passed in during the render call, the geometry shader dynamically chooses how detailed the grass blades will be and how much vertices must be used to represent the grass blade.

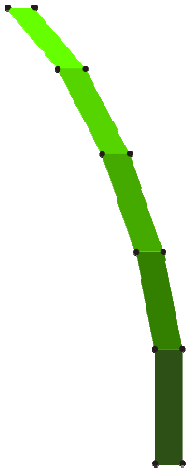


Figure 14 - High Level-of-Detail Grass Blade Segments

The detail of the grass is determined by how many triangles the grass blade will be represented by. Essentially, the level-of-detail determines the how many times to segment the grass blade. The more segments a grass blade has, the more vertices must be pushed through the GPU pipeline. The high level-of-detail grass blades will have more segments in order have a more defined curve. However, low level-of-detail grass blades only need to be represented by a few triangles.

The vertices passed in from the vertex buffer represent the root positions of each blade. The geometry shader uses the root position in the vertex buffer to determine where to procedurally generate the blade.

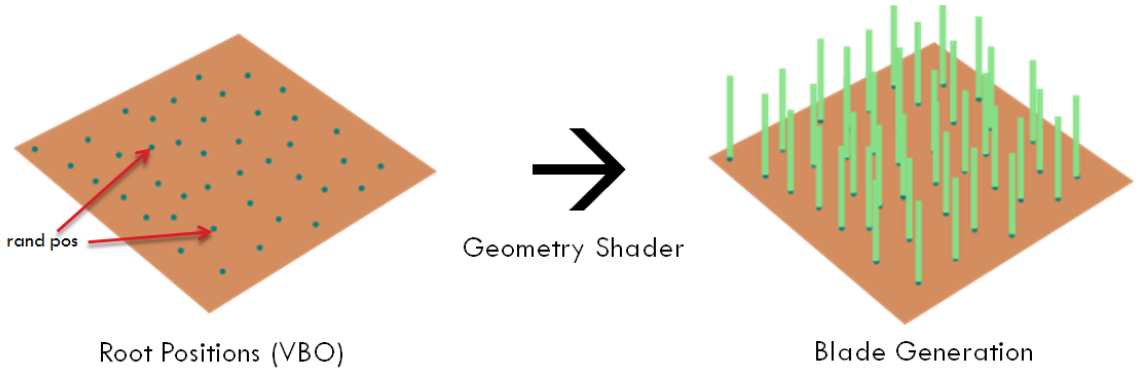


Figure 15 - Vertex Buffer Fed through Geometry Shader

The curvature and positioning of the vertices of the grass blade are described in the following section. Nevertheless, given the curvature of the blade, its vertices can be interpolated along the curve. The geometry shader dynamically generates the blade's level-of-detail based on the blade's distance from the camera. However, this requires the geometry shader to generate a variable number of vertices. The geometry shader works best when it knows exactly how many vertices are to be generated from the shader program. To make the rendering efficient on the GPU, the geometry shader must be set up in such a way that each level-of-detail corresponds to a constant number of vertices.

The geometry shader essentially determines the number of vertices that are generated per blade and builds the blade. The geometry shader first collects the vertices of the blade and displaces them depending on the terrain and wind forces. It is responsible for translating the vertices to align to the curvature of the blades.

Vertex Displacement

The geometry shader is essentially the meat of the entire algorithm. It handles not only the displacement of vertices due to wind forces but also translates the vertices to contour the terrain.

Terrain Contour

The first step of the vertex displacement is to make the grass blade's vertices contour to the terrain's form. In order to do this, the grass blade's root position must be used to determine where it exists on the terrain. A terrain quad consists of two triangles, each which exists on a separate plane (normal and distance). With the planar data of the current grid already provided, what's left is to figure out which of the two planes the vertex exists in so that the vertex can be positioned onto the plane.

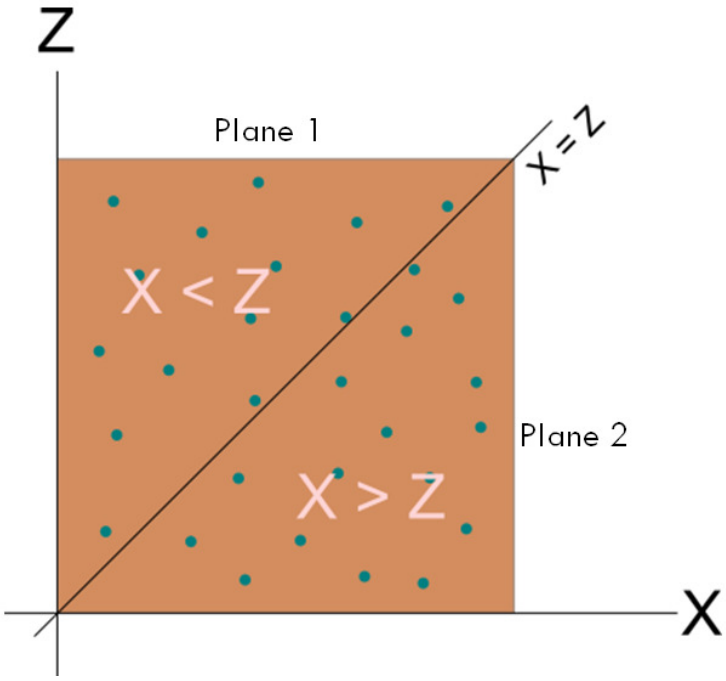


Figure 16 - Patch Contoured to Terrain

The figure above shows a vertex buffer patch in object-space. If the vertex's x-position is greater than its z-position, we contour the root position to the top-left plane; else, the root position is contoured to the bottom-right plane.

In order to contour the plane, simply translate the y-component of the vertex position using the formula:

$$RootY = \frac{-(D + A * RootX + C * RootZ)}{B}$$

Where A, B, C, D are from the plane equation:

$$Ax + By + Cz + D = 0$$

Randomization

In nature, grass blades do not exist in uniform patterns. Not only does the distribution of the blades appear random, but each and every grass blade that exists in a field will appear to have unique features. Since grass is a form of vegetation that grows serially (each seed yields a single blade), each grass blade will exist in different stages of its life. A grass blade might be young and new, with rich saturation and stiffer constitution, while another grass blade may be approaching its final stage of life, in which it will have a slightly more orange hue and a decayed structure. All these variations in a single patch must be accounted for when rendering grass fields in order to make the grass field appear natural.

Figure 17 - No randomness makes the scene seem artificial

Unfortunately, the GPU does not provide any functionality in order to generate a random number. To simulate randomness, graphics developers have traditionally sampled from a random noise texture.

However, a better method would be to use the grass blade's root position to generate a random number. The previous section described how the root positions of a patch are randomly generated during load time. Utilizing the fact that the root positions are random, the root position can be used generate another random number by using the formula:

$$f(Pos) = \sin\left(kHalfPi * frac(Pos.x) + kHalfPi * frac(Pos.y)\right.$$

This formula above returns a normalized random number given a root position (Pos). The $frac()$ function returns the fraction of a number (0.0 to 1.0). The fraction is taken so that the value returned from the $sin()$ function is not periodic. Then, multiply

each of the fraction's number by half-PI to provide a result between the values of 0 to PI. Taking the sin of that value will generate a value from 0.0 to 1.0. This random number can be used to randomize certain features of the grass blade by using the formula:

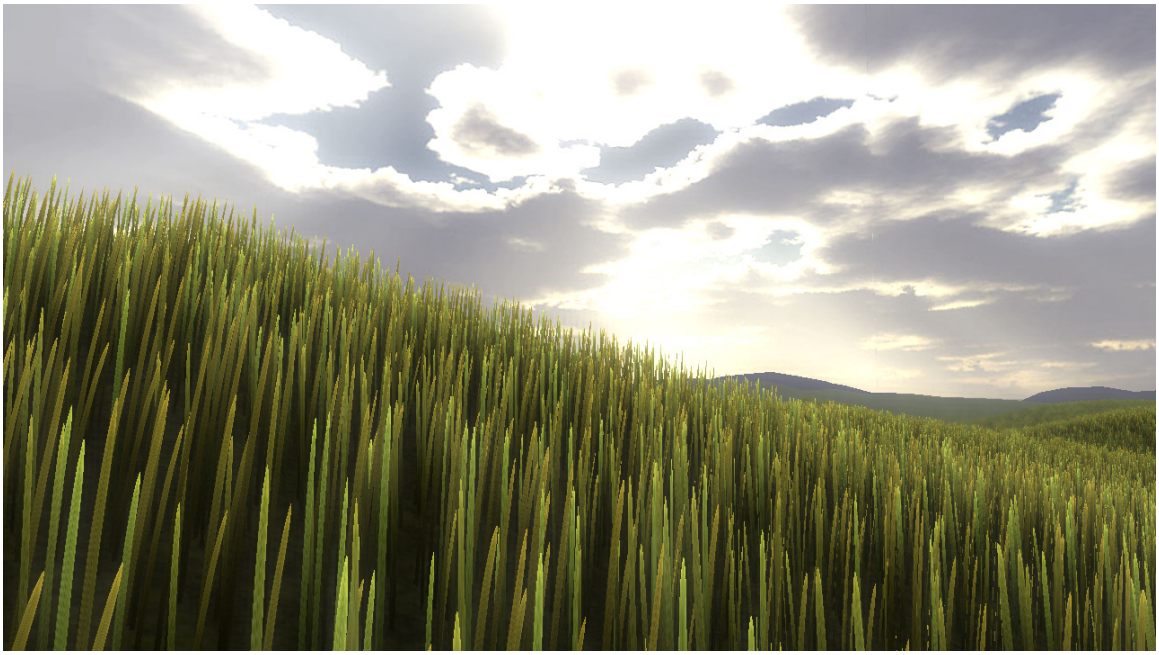


Figure 18 - Randomness makes the scene appear more natural

This normalized random number is used in several applications in the shader when creating the blade. It is used to orient the grass blades such that each individual blade is facing a different direction. If each blade is not randomly oriented, all blades will result in a single orientation, with all blades facing in one direction, causing the scene to appear strange when viewed from different angles.

The random number is also used to randomize the heights and color of each blade in order to simulate the various stages of life that the blades are in. Also, grass blades maybe naturally lean in random directions, which can be easily simulated with the random number.

Blending

Due to the computational inefficiency of rendering so many blades of grass, any opportunity to decrease the geometry complexity of the scene benefits the simulation by decreasing the load on the GPU.

The farther out the grass patch is to the camera, the less need there is in rendering the highly-detailed blades of grass. Blades that are extremely far away are too clumped together for the viewer to be able to notice the individual geometry of every blade. So instead of rendering each individual blade, far away grass blades can be represented by flat triangles that contour the terrain.

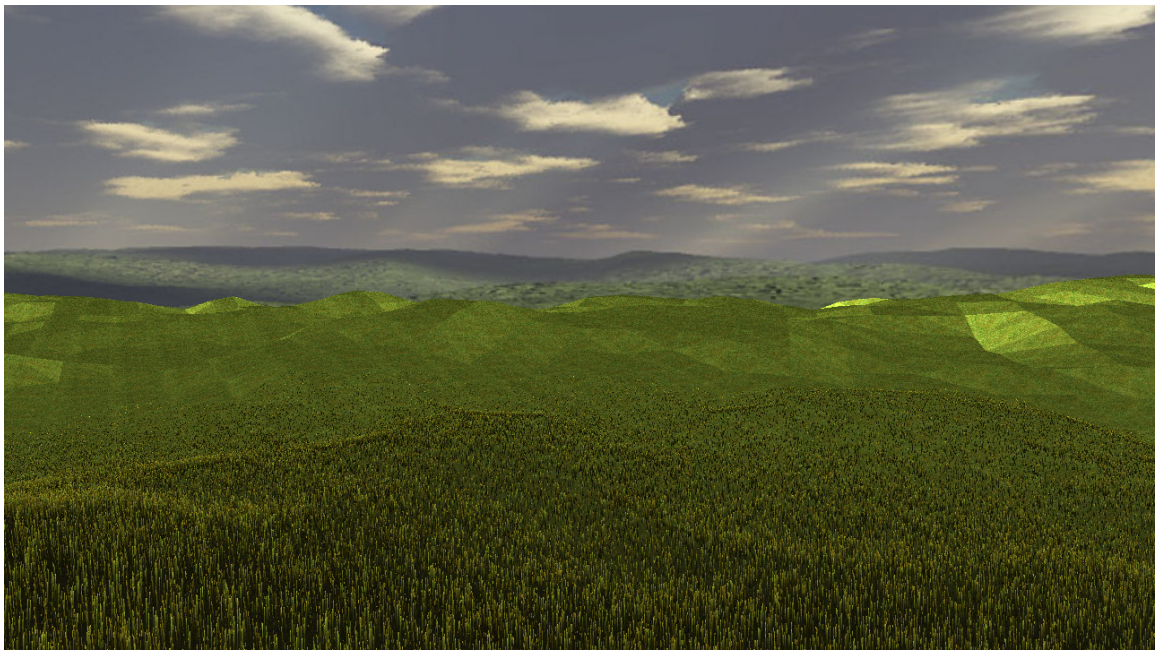


Figure 19 - Flat Patches in the distance

The figure above shows what it looks like when far away patches are being rendered by flat geometry. In order to blend smoothly between the patches that are

rendered by individual grass blades and the flat patches, simply decrease the height of the blade the farther the grass blade until it transitions into the flat patches.

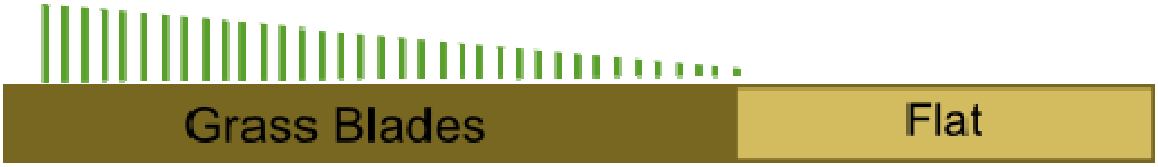


Figure 20 - Grass blades slowly decrease in height

In order to make the transition between the detailed grass patches and the flat patches appear more seamless, the coefficient of the height can be determined by using a simple bell curve.

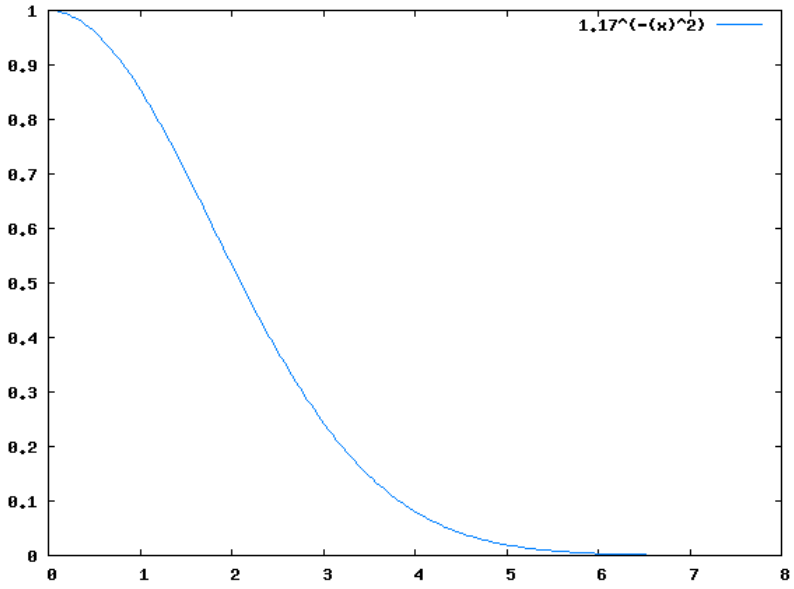


Figure 21- Bell Curve e^{-x^2}

The shape of the bell curve provides a good coefficient so that the height of the blade smoothly transitions into the floor. Multiplying the height by the coefficient provided by the bell curve will provide accurate results. Of course, the coefficients of the

bell curve function needs to be tweaked to account for the distance to the far plane as well as the size of the patches.

Finally, in order to make the transitions completely smooth, post-processing effects are applied to the final rendering of the scene. The post-processing techniques are discussed in the follow sections.

Wind Force

Wind is a huge component in adding interactivity in the scene. In this simulation, the player acts as the wind component, blowing wind in the direction of player's movement. The previous sections described how the wind vector fields were calculated. This section discusses how the blade interacts with wind forces.

First, to fully understand how grass blades interact with the wind, analysis must be made on the interactions between the surface area of the grass blade and the wind vector. Depending on the wind forces and the current state of the blade, wind affects the blade in various ways.

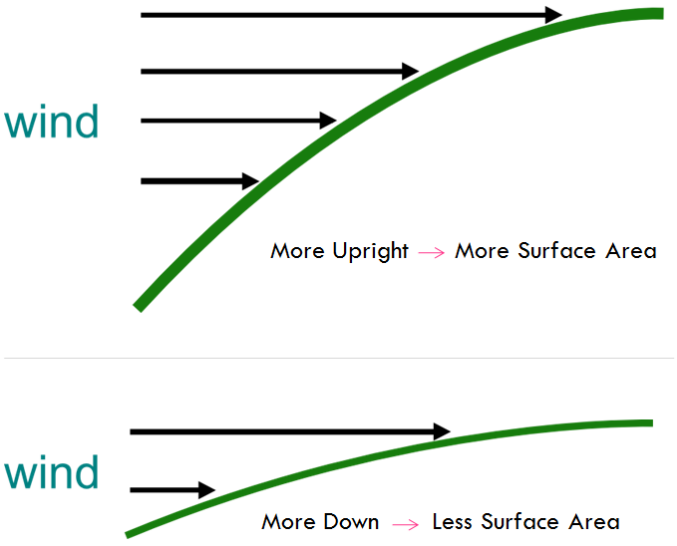


Figure 22 - Interactions between wind vector and blade

The figure above shows the relationship between the wind vectors and the grass blade. In this algorithm, the wind vectors blow parallel to the x-z plane. The more upright the blade is, the more surface area it has to be influenced by the wind forces. On the other hand, the more downwards the grass blade is leaning, the less surface area it has to be influenced by the wind. Therefore, the amount of influence the wind has on the blade is directly proportional to the grass blade's current state.

Figure 23 - Grass blade's oscillation pattern

Grass blades that are more upright tend to have greater oscillation when reacting with the wind. Because it has more surface area, the influence of the wind on the grass blade is much greater and thus affects the blade more strongly. The figure above shows two grass blades at different states. The upright grass has a greater zone of oscillation while the grass blade that is more downwards-leaning has a smaller zone of oscillation. The oscillation of the blade can be simulated within the shader by taking the oscillation zone and oscillating the curve's vertices within the zone using the sinusoidal wave.

To make the movement of the grass appear more natural, oscillations of every blade in the grass field cannot be in uniform. That is, if all blades of grass were oscillating at the same rate, the scene would look unnatural. In nature, different blades have different weights and thus different momentums when interacting with wind conditions. To simulate these conditions, simply skew the oscillation zone per blade as

well as doing a phase shift on the oscillation. The offset and phase shift amount can be determined by using the random number as discussed in previous sections.

Figure 24 - Grass reacting to wind conditions

The figure above shows the grass field interacting with the wind. Notice that each blade to react with the wind differently because of the offsetting of the oscillation. This approach of slightly randomizing the interaction between the blades and the wind makes the scene appear much more natural and believable.

Figure 25 - Discrete Wind Vectors

One issue with using a discrete wind vector field is that each grid in the vector field corresponds to one patch of grass. That is, having an entire grass patch be influenced by one discrete wind vector will oftentimes make the scene appear very artificial since every blade in a patch will lean in a single direction of the wind vector. The figure above shows an instance of when two wind vectors are pointing in opposite directions. The edges between two separate grass patches are very visible.

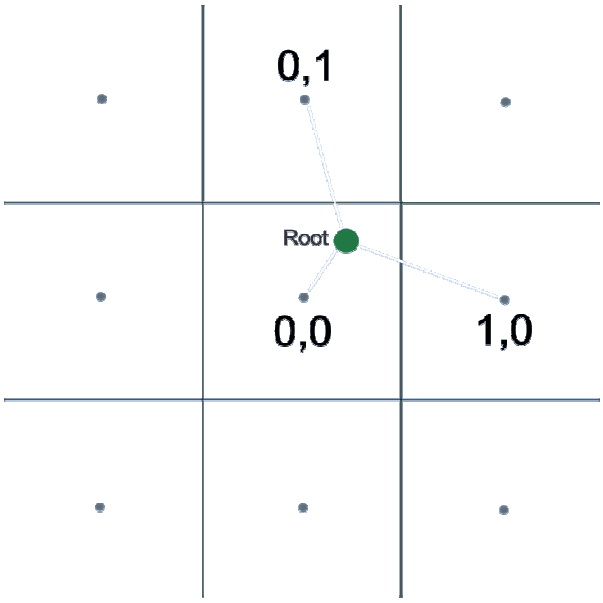


Figure 26 - Linearly interpolating between neighboring wind vectors

In order to have a continuous wind vector field, each blade must sample surrounding wind vectors and linearly interpolate between them. Depending on the position of the blade's root relative to the centers of other patches, the wind vector that influenced that specific blade can be calculated by linearly interpolating between three wind vectors

1. The wind vector of the current patch that the blade is in
2. Left or right wind vector, depending on if the blade is on the left of the its patch
3. Top or bottom wind vector, depending on if the blade is on the top or bottom of the center of its patch

When determining the wind vector applied on to each individual blade, linearly interpolating between the neighboring wind vectors results in a continuous vector field. This method provides convincing results with no discrete gaps in between patches.

Vertex Displacement

Now that the wind forces and its interactions with the blade have been determined, the shader can displace the vertices of the grass blades according to the desired shape of the blade.

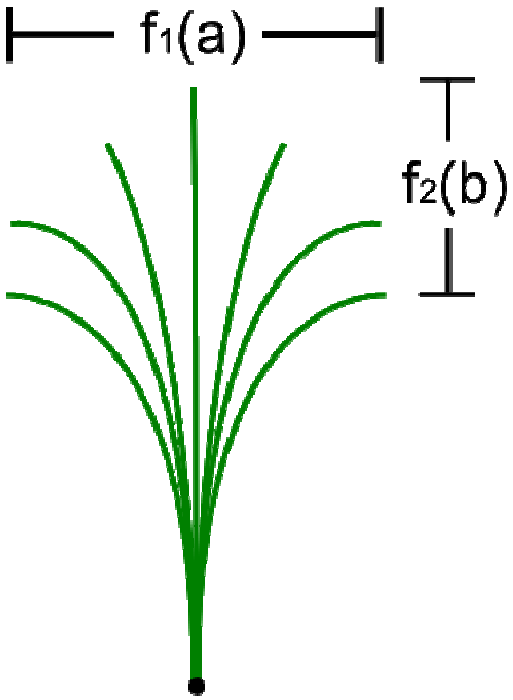


Figure 27 - Curvature of a blade due to wind forces

The displacement of the vertices occurs along the curvature of the blade. The greater the wind force, the more the vertex will be translated horizontally. The figure above shows how the blade curves in relation to the wind. The amount that the vertex is displaced horizontally is directly proportional the magnitude of the wind. However, the vertex cannot only be displaced horizontally or else the blade will appear to stretch as it sways side to side. The vertex must also be displaced vertically in order to account for the curvature of the blade. Thus, the greater the wind force is on the blade, the more the

vertex is displaced downwards in the y-direction. The function that relates wind force and displacement is as follows:

$$Pos.xz = Pos.xz + WindVec * WindCoeff$$

$$Pos.y = Pos.y - length(WindVec) * WindCoeff * 0.5$$

The WindCoeff coefficient is determined by how far the current vertex is from the root of the blade. That is, the higher up the vertex is on the blade, the more the vertex is affected by wind. The vertex of the root of the blade should not move at all since it is attached to the floor and thus should have a WindCoeff of zero. The vertex at the top of the blade should be the most affected by wind, and thus should have a WindCoeff of one.

Displacing the vertex horizontally and vertically allows the blade to have a constant length and also provides a nice curvature to the blade.

Vertex Shader

The vertex shader is extremely minimalistic. Since the geometry shader does all the hard work of creating and displacing vertices, the vertex shader only needs to pass the vertex data into the pixel shader, in which the vertex's values are interpolated.

Pixel Shader

After the vertices are passed through the vertex shader, its values are interpolated and then fed into the pixel shader. The pixel shader handles a lot of the coloration and texturing of the grass blades. The first step in the pixel shader is to render each blade's shape by using a technique called *alpha-to-coverage*. Alpha-to-coverage is a multisampling computer graphics technique which is often used to render dense vegetation or grass. This technique uses one texture to as the color map and another

texture for the alpha map. Depending on the alpha values of the alpha map, certain pixels are culled so that only the shape of the blade remains. In DirectionX10's HLSL, this can be done by using the `clip()` function.

Figure 28 - Grass Blade's Diffuse and Alpha Map

The second step of the pixel shader is to handle the color tinting of each blade. In nature, some grass blades may be healthy and green while others may be old and withered. The pixel shader program simulates this variation in color by using the normalized random number (mentioned previously) and linearly interpolating between the diffuse color and an orange color. This process provides a field of grass that has a natural coloration to each blade.

Ambient occlusion occurs when grass blades affect the attenuation of light due to the occlusion of nearby blades. In essence, it is what causes the grass blades to appear darker towards its root while appearing lighter towards its top. To simulate the effect of ambient occlusion on the blades, the pixel shader must keep track of the pixel's relative height in the blade. That is, a pixel that is located near the root of the blade will have a low value while pixels at the top of the blade will have a value of one (bright). The process produces a quick and believable way to simulate the natural effect of ambient occlusion on grass blades.

Finally, other lighting techniques in the pixel shader such as per-pixel lighting and specular lighting can be applied to each blade. The normal of the current pixel can be

used to calculate certain lighting techniques. The normal is calculated in the geometry shader by taking the three vertices and calculating their cross product. The normal is fed into the vertex shader and linearly interpolated by the pixels shader. Taking the dot product of the normal by the light position achieves a per-pixel lighting effect.

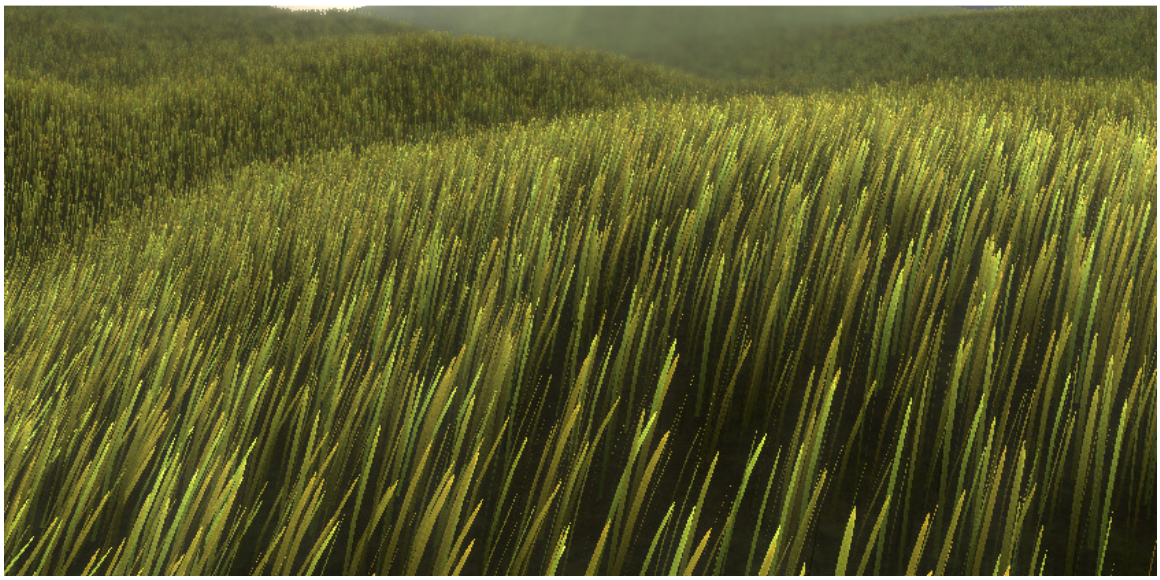


Figure 29 - Pixel shading on the grass blades

Flicker Prevention

The average grassy scene consists of tens-of-thousands of grass blades that are all rendered onto the screen. However, since the screen space is of limited supply, there exists an issue of not having enough screen resolution to accurately represent the all the pixels that are being rasterized in the scene. A typical problem with rendering complex scenes occurs when several pixels are competing to be rendered onto the same screen pixel. Although this may not be noticeable in screenshots, the problem arises during the animation of the scene.

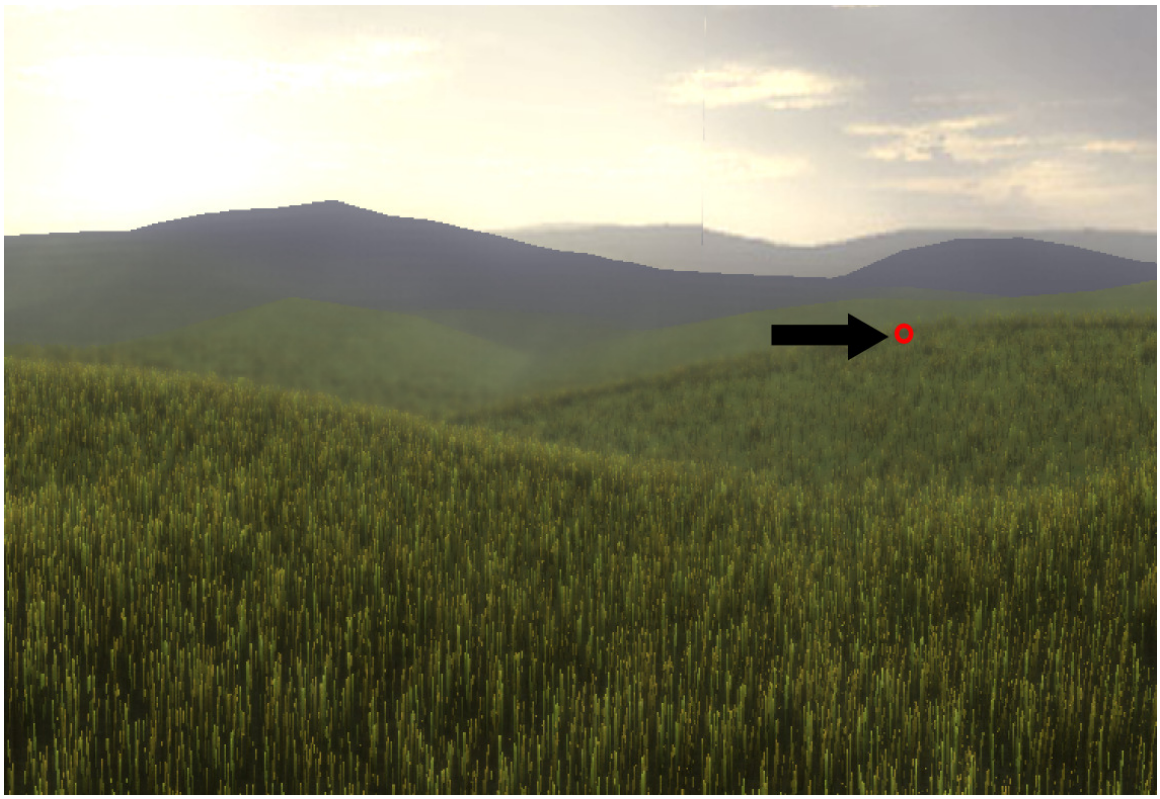


Figure 30 - Several blades of grass competing for same pixel

In this study, several grass blades far in the distance may be competing for the same pixel on screen. This occurs when a patch of grass is so clumped in the distance that several of its grass blades may exist as one pixel in screen-space. The figure above shows a dense scene which has the black arrow pointing to a problematic area where several blades of grass are contending to be rendered at that location. When several pixels are competing for the same screen-pixel, the screen-pixel will flicker as the GPU "randomly" chooses which pixel to render into that screen-pixel. Of course, this is not random since the render depends on the position and the orientation of the camera. Nonetheless, the camera may be moving or the grass blades may be animated, which causes the pixels to

flicker in real-time. This makes the animated scene to appear unnatural, which can break player immersion if not handled correctly.

A big part of the reason why this visual artifact exists is due to the fact that so many blades of grass are competing for the same pixel on screen. To handle this, simply render fewer grass blades for patches that are far in the distance, which allows for less contention of the screen space pixel of distant grass geometry.

Since the grass patch's vertex buffer was generated by randomly populating a square, the user only needs to decrease the number of root positions that is to be rendered during the render call. This allows a single vertex buffer to be used during the entire rendering of the scene. Decreasing the blade count gradually (depending on distance) allows for a smooth transition when the player traverses the scene, else the player may experience popping as the level-of-detail changes for each patch in the scene.

Apart from reducing the contention of rendering on to the pixel, decreasing the blade count gradually also reduces the amount of vertices that are passed through the graphics pipeline, thereby decreasing the workload and allowing for more complex calculations elsewhere. Since distant patches can barely be seen by the user, the density of a patch becomes less significant as the patch fades in to the distance.

Furthermore, visual artifact can occur in distant patches where the blades are so thin that the grass blade flickers between the terrain and the blade's pixels. In order to solve this issue, simply increase the thickness of the blade as the grass is farther into the distance. The thickness is proportional to the distance that the grass blade is to the camera. That is, the width of the blade can be found by linearly interpolating between the minimum and maximum blade width based on distance of the blade to the camera:

$$width = lerp(\min width, \max width, distance\ normalized)$$

This allows distant grass blades to fill the screen, which reduces the problem of flickering in distance grass blades.

Although this solution solves some of the issues regarding pixel flickering, it does not completely remedy all flickering artifacts. For example, grass blades that are directly perpendicular to the orientation of the camera will be so thin that it also is causes similar issues. Nonetheless, applying such techniques reduces the visual artifacts of flickering, thereby improving the visual quality of the animated scene.

Post-Processing

The entire scene is first rendered to a render target. Using this render target, several post-processing effects can be applied in order enhance the visual quality of the scene.

Fog

The first post-processing effect applied to the scene is a technique similar to fog. This technique is used to simulate the effect that occurs naturally when looking at far away objects. In particular, far away objects tend to have a more desaturated color compared to objects that are near the camera.

Figure 31 - Objects appear more desaturated in the distance

Notice in the figure above that the colors of nearby objects tend to have a more vibrant color in comparison to the distant mountainous ridges which have a washed-out, desaturated look. This desaturation effect is due to the fact that the distant objects' light rays must travel at a greater distance than the light rays of nearby object. These light rays are intercepted by molecules (dust, pollen, air) in the air which absorb the light rays. The longer path the light ray must travel to reach the viewer, the more the light ray collides with air molecules and thus the more desaturated the colors become.

The process to add fog into the final scene is quite simple. The render target stores the depth buffer of the scene, which provides non-linear depth information of each pixel in the render target. This value can be easily converted to the actually depth using a simple calculation. Then, using the depth information linearly interpolate between the raw render target's diffuse color and the environmental fog color.

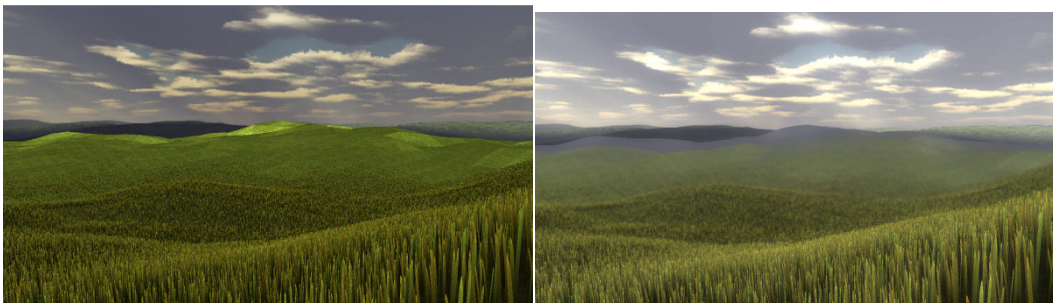


Figure 32 - Raw Render Target and Fog Post-Processing

The figure above shows the same scene being rendered with and without fog post-processing enabled. The fog makes the scene more visually appealing since it makes the setting appear more natural. Fog is important visual element in a scene because it is not only observed in nature; it also provides visual cues for the thickness of the atmosphere.

Bloom

Bloom is a post-processing effect that aims to reproduce the imaging artifacts of real-world cameras and the human eye lens. It is based on the fact that in the real world, the lens can never focus perfectly and will convolve the incoming light rays to bounce and blur within the retina, causing the light to blur onto nearby pixels.

To achieve this bloom effect, the current raw render target of the scene must be provided. For each pixel in the render target, calculate its luminosity by taking the RGB vector and applying the dot product with the luminosity vector (0.30, 0.59, 0.11). This value describes how intensely bright the color value is. The luminosity of the pixel is passed through a threshold function such that given a luminosity value x and for a certain threshold value c:

$$f(x) = \begin{cases} 0 \; if \; x \leq c \\ x \; if \; x > c \end{cases}$$

Save this threshold data onto another render target. Then, apply a Gaussian blur to the threshold render target to give a nice bloom effect and save that data to another render target. Finally, add the values from the blurred threshold render target onto the original render target, giving a nice bloom effect.

Figure 33 - Threshold Render Target

Figure 34 - Blurred Threshold Render Target (Gaussian Blur)

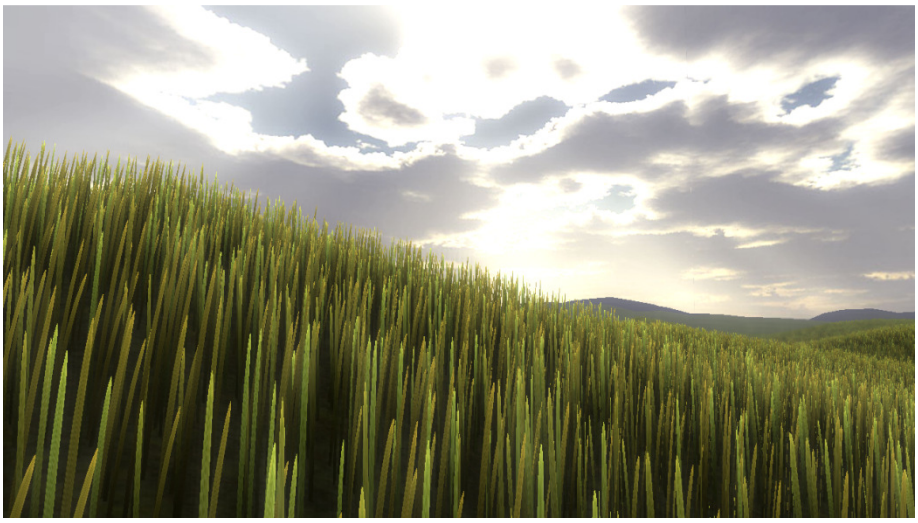


Figure 35 - Combing Blurred Render Target with Original Image

Depth-of-Field

In optics, when the camera lens focuses is on a particular object, the object will appear sharp while out-of-focus objects will appear blurred. This effect is due to the lens' distance to its subject, its focal length as well as its circle-of-confusion. It is an important aspect in cinematic visual vocabulary and photo-realistic rendering in which the technique can be used to focus the viewer on certain parts of the scene.

The traditional approach to depth-of-field is to render the scene several times around the focal point, then blur the render targets together to produce the depth-of-field effect. But due to the extreme complexity of the grass scene, rendering it multiple times is simply infeasible.

To achieve an effect similar to depth-of-field, the current render target is first blurred and then outputted to another render target. Then, render the scene by sampling the depth buffer (similar to the fog). Given the target depth (where you want the player to focus on), calculate, for each pixel, the difference of depth between the current depth and

the target depth. Normalize the value and use the result as the linear interpolation factor between using the original render target and the blurred render target. This technique results in the final image, in which the closer the pixels are to the target depth, the more defined the pixels are; while pixels that have are really far away from the target depth will have a more blurred appearance.

Figure 36 - Raw Render Target

Figure 37 - Raw Render Target with Gaussian blur

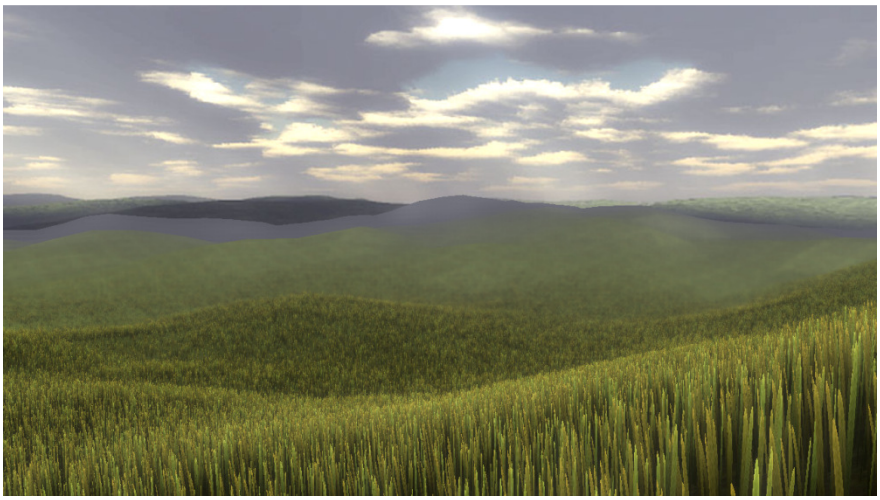


Figure 38 - Results of Depth-of-Field algorithm

The figure above shows the results from the depth of field. Notice that the closer grass blades appear more defined while the blades in the distance appear blurred and out of focus. Not only does it help focus the player on the more detailed nearby grass blades but it provides a good way to seamlessly blend between the areas where the grass fades into the flat terrain.

Shadows

In real life, the grass blades not only cast shadows on the ground but also on each other. In order to have a truly realistic grass simulation, some amount of shadowing must be implemented in order achieve photo-realism.

The process to render shadows in the scene is to employ the standard shadow mapping algorithm. The first step of the algorithm is to render the scene from the point of view of the light to capture the depth of the scene. The light's position, projection and view matrices are stored and sent into the GPU. Outdoor lighting is generally handled as an orthographic projection from the direction of the sun. Note that since this step only

acquires the depth of the scene. Any lighting and texturing of the blades are superfluous and should be removed from the pixel shader. However, to still preserve the shape of the blade (i.e. the tips of the blades are rounded), alpha-to-coverage must still be implemented in the pixel shader to clip out unneeded pixels.

Then, to render the shadows in the pixel shader, simply compare the depths of the current pixel to the depth from the shadow map. This is done by bringing the pixel to the view-space of the light. If the depth of the shadow map's pixel is less than the current pixel, then there is a shadow; else, it is fully lit. Note that there are advanced shadowing techniques that allows for greater realism, but with the cost of performance.

This process requires the grass blades to be rendered twice: once to capture the shadow depth and once more during the final rendering of the scene. Since rendering the geometric complexity of grass is extremely computationally expensive, rendering the entire scene twice greatly overwhelms the GPU and drops the frame-rate significantly. Thus, an efficient way to solve this problem would be to render only a few select patches that require shadows. The patches that need shadows are generally the ones that are closest to the position of the camera.

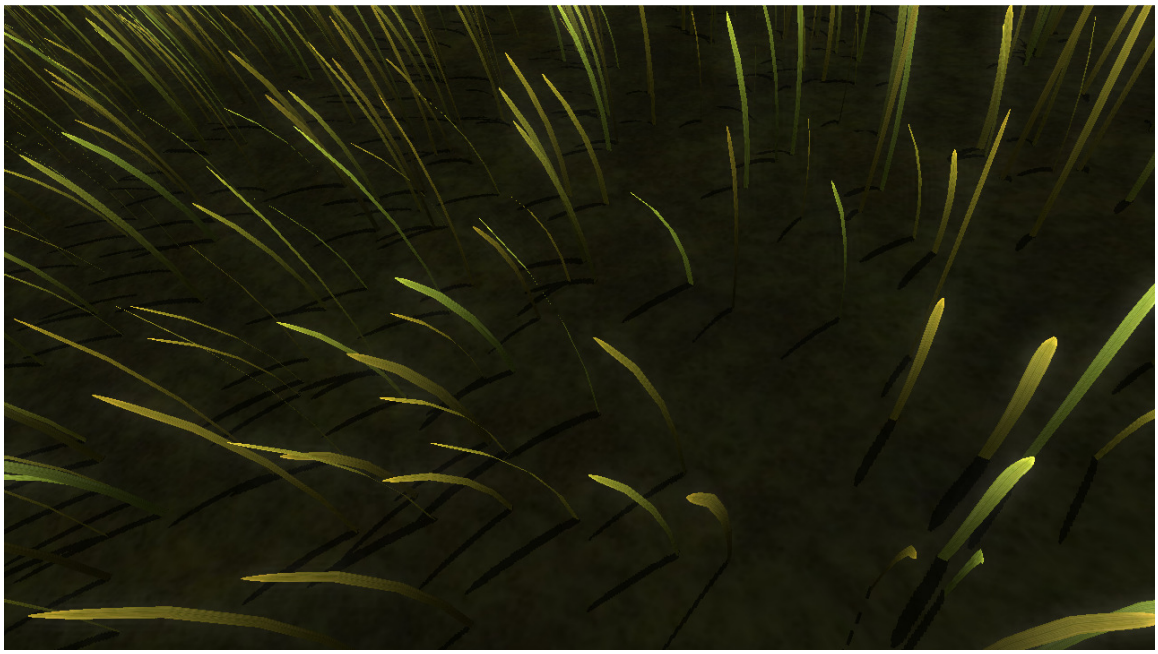


Figure 39 - Shadows casted on ground

There are several issues with rendering shadows in such a highly complex setting. Since the shadow map algorithm is only applied on nearby geometry, this means that the patches with the highest level-of-detail must be rendered twice. The patches with the highest level-of-detail are extremely expensive and rendering them twice can greatly decrease the frame-rate.

Not only does it require the GPU to handle rendering complex geometric data twice, but simply adding extra data to the shader structures that are passed between stages in the shader pipeline can greatly decrease frame rate. More specifically, in order to render shadows according to the shadow-mapping algorithm, the pixel shader requires two extra fields that must be passed from the vertex shader: *world position* and *projection texture coordinates*. Simply adding these two fields of the structure without any

implementation of shadow mapping slows down the frame rate. With the shadow mapping implemented, the frame rate drops significantly.

The issue of keeping frame rates high while rendering shadows is two-fold. First, shadows look the best when grass blades cast shadows on other blades. However, in order to preserve interactive frame-rates, the blade count of the patch must be decreased to lessen the workload on the GPU. Decreasing the number of grass blades causes the blades to be more spread out and therefore decreases the possibility of casting shadows on each other.

Figure 38 shows an image of the blades projecting shadows onto the floor after the number of blades has been reduced. However, the results run at a slow frame-rate of 10-15 frames-per-second (on 640x320 setting). Ultimately, due to the slowness of rendering shadows, shadows has been removed from the final project.

DATA COLLECTION AND PROCEDURES

The data collection process is quite simple: first implement the method proposed, document all the details of the method by extracting the frame time, blade count, and screen resolutions of the simulation.

Test Description

The data that was collected from the project are the frame rate and the polygon count from rendering the scene. These two variables are completely quantitative. The frame rate can be gathered using Nvidia's *perfmon* application. The blade count can be calculated during the application run-time. Each variation of the core method will run on

varying resolutions. This allows us to find what fill-rate, rasterization or pixel shader bottlenecks exist are when rendering.

Surveys and other statistical survey methods would not be needed since no participants are involved in this study. And since no participants are involved in this study, no ethical considerations are required to conduct this study.

Data analysis methods involve analyzing the changes in frame-rates for each variation to see which set-up can be used to maximize grass-rendering fidelity while still allowing interactive frame rates.

SUMMARY

Since the focus of this study is to discover and develop a reliable algorithm to render a dense field of grass, no statistical survey methods are required for the purposes of this study. Data is gathered by implementing various algorithms and measuring the difference in run-time performance. By comparing the differences between various schemes and iterating on the core implementation, the goal is to ultimately discover a scheme that is a perfect balance between believability and performance.

CHAPTER 4: RESULTS

This study is designed around discovering an algorithm that renders a dense and infinite field of grass on current consumer hardware. During the development process, a single method was implemented, iterated upon, and evaluated on several factors, with the greatest emphasis on frame rate.

This section outlines the results that were gathered from various configurations of the application. Each configuration varies in screen resolution and number of grass blades per patch. The purpose of comparing the screen resolution against the number of grass blades per patch is to determine where pixel shader bottlenecks become geometry shader bottlenecks.

The more grass blades that exists in a patch, the denser the field appears. Note that increasing the number of blades in a patch linearly increases the total number of grass blades that are rendered since the same grass patch are populated throughout the scene to cover the entire scene that exists within the frustum. Generally speaking, increasing the number of grass blades in a patch makes the scene appear more realistic since in nature, grass grows in very dense formations. Therefore, the goal is to push as many grass blades into a patch while still maintaining interactive frame-rates (30-60 fps).

The grass patch is set at a constant width of 340 units. However, the number of blades in a patch is dependent on the level-of detail. More specifically, the number of

grass blades gradually decreases as the patch is farther away from the camera. The test configuration uses four levels-of-detail. The coefficients of the number of grass blades in each level-of-detail are:

LOD 1	LOD 2	LOD 3	LOD 4
0.3	0.85	0.95	1.0

All other configurations, such as the wind vector simulation, terrain data, camera position and the view frustum, are held constant so that the two variables can be more accurately compared.

Development System

The development machine used during testing was the Alienware M17x gaming laptop. Its specifications are outlined below:

- Processor: 2.4+ Ghz Intel Core 2 Duo P8600 Processor
- Memory: 4 GB of System RAM
- Video Card: NVIDIA GTX 260M
- Operating System: Windows Vista or Windows 7

Results

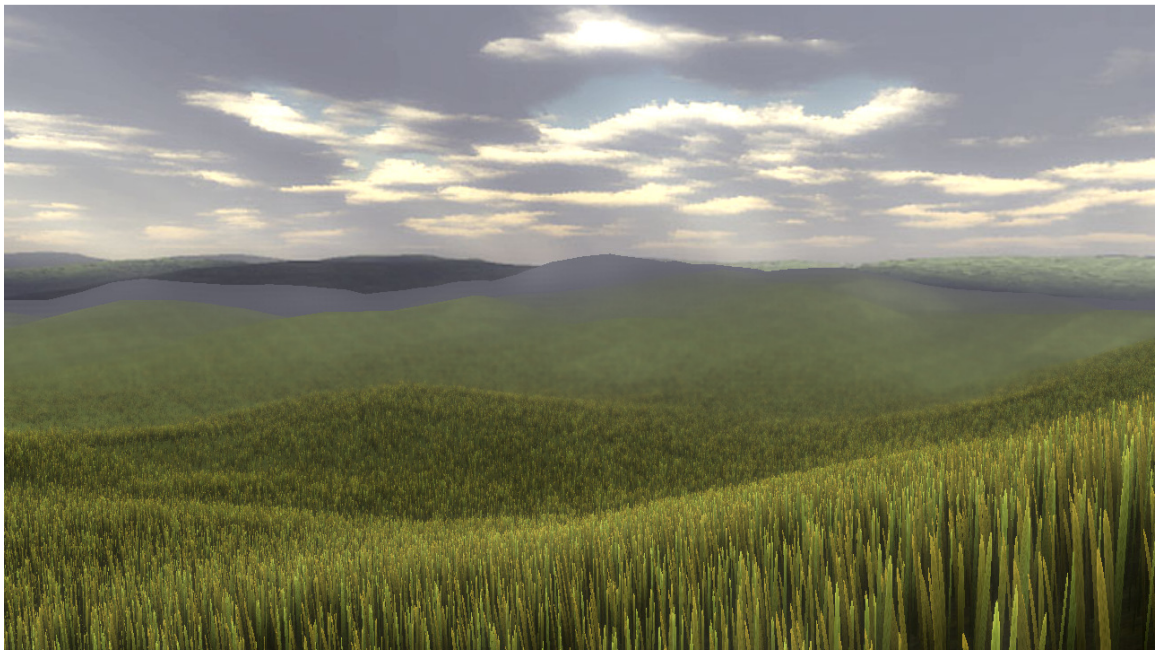


Figure 40 - View of the scene that is used for all test cases

The figure above shows the standardized view of the scene that is used for all test configurations. The camera is positioned in such a way so that the frustum spans across the entire field in order to maximum number of patches is rendered. There are about thirty grass patches rendered, with all their level-of-details (based on camera distance) consistent. All permutations of standard screen resolutions and blades counts are implemented and the frame-rates are recorded:

	1000	**1500**	**2000**	**2500**	**3000**
800x600	73	58	45	37	31
1280x720	57	42	33	28	24
1680x1050	49	39	31	26	23
1920x1200	44	35	28	24	21

Table 1 - FPS vs Resolution

The table above shows the average frame-rates gathered from the various configurations. All the results that have a frame-rate of 30 or above are highlighted in yellow. Note that the columns only represent the maximum number of grass blades for a patch at the highest level-of-detail. Lower level-of-detail patches contain a percentage of the number of blades in the highest level-of-detail patch.

The graph of the table is shown below:

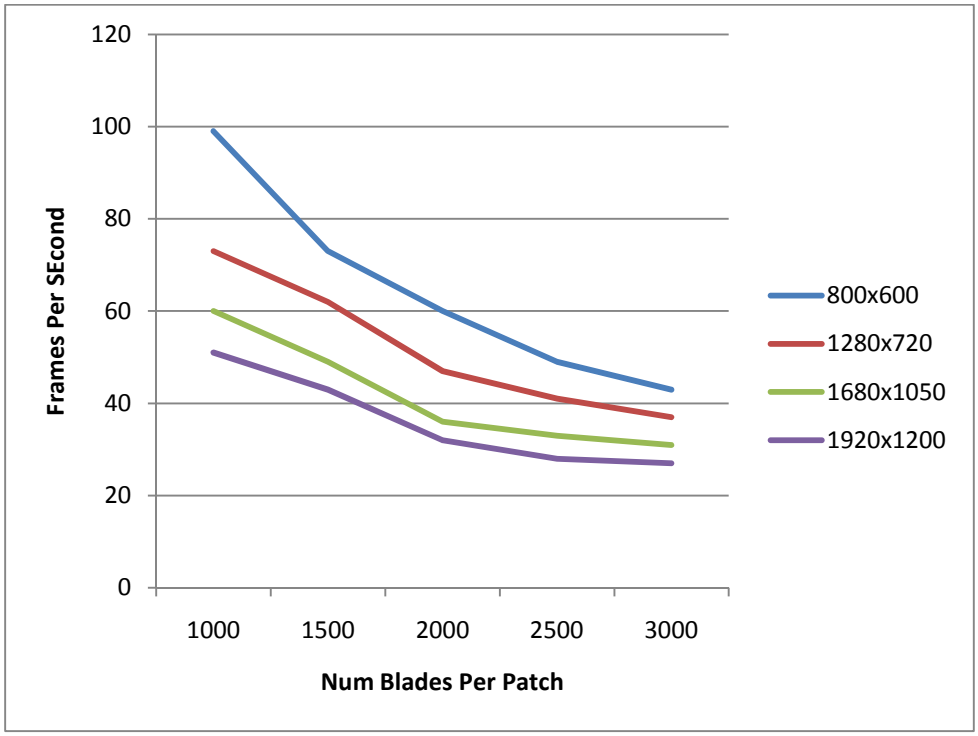


Figure 41 - Graph of Number of Blades vs FPS

Each line refers to a specific screen resolution. The regression model that fits the data is an inverse-squared ($1/x^2$) relationship between the frame-rate and the number of blades. That is, adding more grass blades decreases the frames-rate at inversed-squared rate.

Level-of-Detail Comparison

Without the implementation of decreasing number of grass blades based on level-of-detail, we achieve this result:

	1000	1500	2000	2500	3000
800x600	58	41	32	27	23
1280x720	43	33	26	21	17
1680x1050	39	29	23	19	15
1920x1200	34	26	20	17	14

Table 2 - FPS vs Grass Blades (Without LOD)

A graph of the data is shown below:

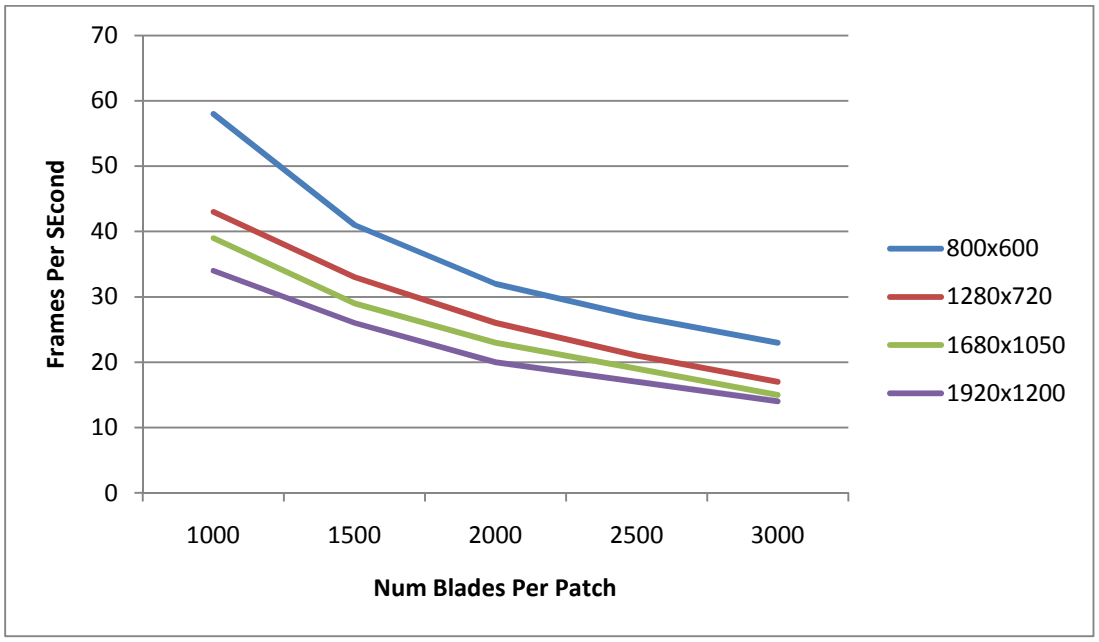


Figure 42 - Grass blades vs FPS (without LOD implementation)

Notice that the FPS is greatly reduced from the previous data (with LOD implemented). This is due to the fact that the number of grass blades does not decrease as the grass patches are farther away from the camera. Rendering the maximum number of

grass blades for all patches in scene causes the graphics processor to have to work harder in order to handle the extra tens-of-thousands more blades, which causes the significant drop in FPS.

Geometry Shader Comparison

A lot of the computational complexity related to vertex displacement occurs in the geometry shader. For example, each grass blade must not only linearly interpolate between surrounding wind vectors, but also must contour to the terrain as well as handling randomization between the blades. Furthermore, each vertex is subject to several computationally heavy mathematical operations such as sine, cosine and exponential functions.

To compare the true cost of the geometry shader, the scene was again rendered with the same parameters (resolutions, LOD and camera positions) but this time without all the vertex displacements implemented. Doing this measures the performance difference between having the computationally expensive vertex displacements implemented and having no vertex displacements implemented.

	1000	**1500**	**2000**	**2500**	**3000**
800x600	99	73	60	49	43
1280x720	73	62	47	41	37
1680x1050	60	49	36	33	31
1920x1200	51	43	32	28	27

Table 3 - FPS vs Resolution without Vertex Displacement

The table above shows the average frame-rates for various configurations with all the complex vertex displacements turned off. A graph of the results is shown below:

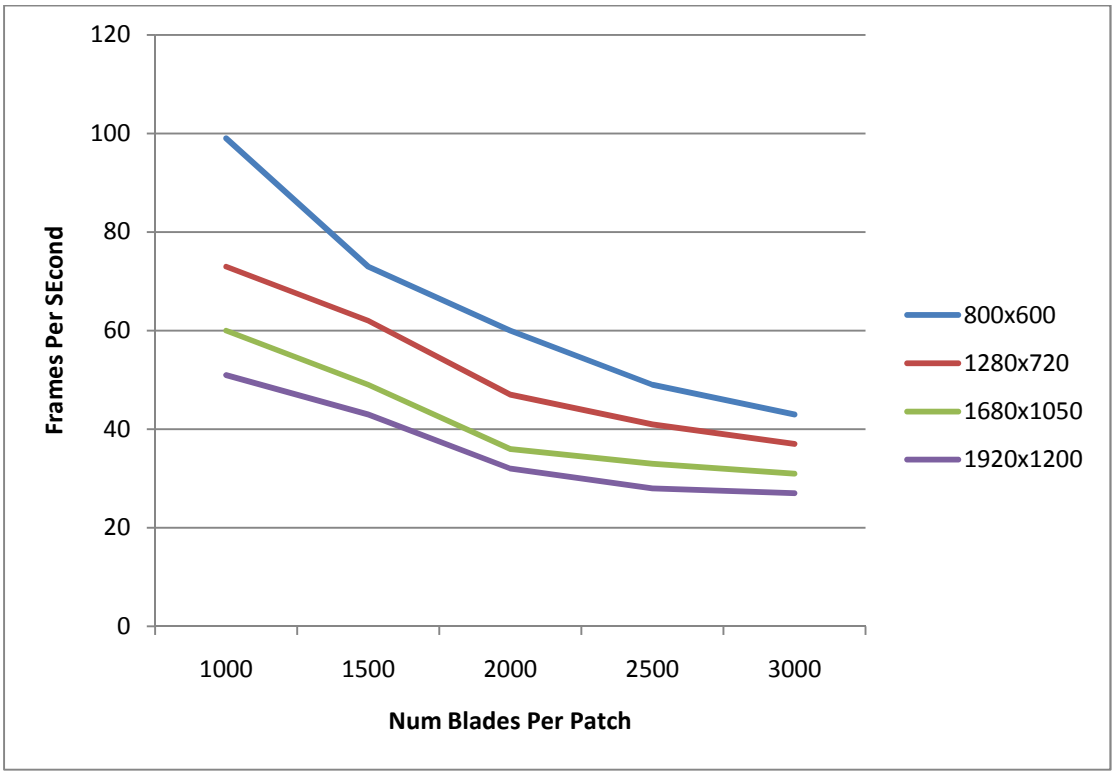


Figure 43 - Graph of frame-rates with vertex displacements removed

Notice that the graph appears essentially the same as the results from before but with all the values shifted upwards.

CHAPTER 5: DISCUSSION

Due to the computational complexity of rendering dense grass, one of the major goals of this study was to configure the algorithm such that the maximum number of grass blades could be rendered on the scene while maintaining interactive frame rates. A good balance must be made between all stages of the GPU pipeline such that no one step would completely bottleneck the graphic processor.

DISCUSSION

Examining the data gathered from the various configurations, it appears that the geometry shader bottlenecks the graphics pipeline much more than any other stage. In particular, the frame rate decreases at a quadratic rate as more root positions are added to the initial vertex buffer. However, keeping the root position count constant while increasing the screen resolution only decreases the frame rate at a linear rate.

This result is quite interesting since traditionally, it is the pixel shader's fill-rate that bottlenecks the rendering process and not the geometry or vertex shader. That is, usually the texture sampling and lighting calculations are what restrict the rendering process. However, the bottleneck of this algorithm lies in the geometry shader stage of the pipeline.

The geometry shader is relatively fast in generating vertices on-the-fly. Given that the number of vertices generated per blade is constant for each level-of-detail, the

geometry shader can quickly spawn enough vertices to describe the geometry of the blade. More specifically, if the geometry shader knows exactly how many vertices are going to be created for each root position in the vertex buffer, its process in the pipeline is greatly optimized. After the vertices are created for the blade, each vertex must be displaced with respect to wind conditions and terrain alignment. The vertex displacement is where most of the heavy-lifting is done.

As more root positions are being pushed through the pipeline and into the geometry shader, more calculations must be made to handle the translations of the vertices to align to the curvature of the blade. For each vertex on each blade of grass, the shader must sample surrounding wind vectors in the vector field in other to smoothly interpolate between discrete vectors. Afterwards, it calculates the translation of the vertex by examining the interactions between the blade and the wind vector. Randomization schemes must also be applied in order for the grass blades to appear "natural". All of these calculations are done on the geometry shader for each vertex, in which adding more vertices to a patch can quickly overwhelm the processor.

In this study, the amount of calculations done in the pixel shader is generally pretty light. The pixel shader is mainly used to calculate simple lighting effects, alpha-clipping, and color tinting. From the fact that increasing the screen resolution only decreases the frame-rate at a linear rate is testament to the fact that the calculations in the pixel shader is not enough to become the bottleneck of the rendering process.

LIMITATIONS

There are several limitations that constrain this study – most of which surround the hardware of the system. The development of this algorithm was geared toward graphics processors that have geometry shaders including in its graphics processor. Geometry shaders are available to operating systems that support DirectX10, such as Windows Vista and later. Current generation consoles, such as the Xbox360, PlayStation3 and Nintendo Wii, do not have geometry shaders available in its graphics pipeline. However, it is predicted that the next generation consoles would have such functionalities.

Since this study relies heavily on procedurally generating blades in the geometry shader, not having that capability would definitely distort the possibilities of being able to rendering so many blades of grass. However, that is not to say that it is impossible. One solution to this issue would be to create a vertex buffer for the highest level-of-detail patch and to generate separate index buffers per level-of-detail. When rendering the scene, simply invoke the render call manually for each level-of-detail to populate the scene. However, doing so incurs the overhead of having to bind each level-of-detail's index buffer as well as having to include more meta-data in the vertex buffer.

Furthermore, the original algorithm relies on procedurally generating vertices on the fly. That is, vertices are created in the geometry shader and destroyed once the rendering is complete. Without the geometry shader enabled, the GPU would have to store the vertex and index buffers pertaining to each level-of-detail, thereby occupying GPU memory.

With the progression of graphics hardware technology, newer functionalities are being invented and integrated into the graphics pipeline. One important operation that is being supported by newer graphics models, such as DirectX11, is the *tessellator*. The tessellator is capable of subdiving polygons to dynamically create vertices on-the-fly to provide higher resolutions for a given geometric model. One thing that held this study back from having truly seamless transitions in the levels-of-detail is the geometry shader. Since the geometry shader works best when it "knows" exactly how many vertices are being created, it requires the developer to code separate geometry shaders per level-of-detail. With the tessellator, the vertices of the grass blades can be created dynamically along the curvature of the blade, thereby providing better results in smoothly transitioning between levels-of-detail. Given that the tessellator is capable of handling all the dynamic level-of-detail transitions, this makes it such that the user does not need to invoke separate render calls for each level-of-detail, thus decreasing the number of render calls required.

CONCLUSION

One of the greatest findings during the development of this algorithm was discovering the sheer power of the graphics pipeline. During each frame, hundreds-of-thousands of vertices are being procedurally generated in the geometry shader and rendered on to the screen. The geometry shader does an incredibly fast job in generating vertices on the fly (as long as it knows exactly the number of vertices that are being created).

This study demonstrates that it is possible to achieve realistic and dense fields of grass on current consumer hardware. As graphics processors continue to evolve and progress, the possibilities of rendering more complex representations of grass becomes more promising. When rendering grass, graphics pipeline bottlenecks might be less of an issue as computational power in future graphics processors becomes more effective and efficient, thereby giving way to more computationally complex lighting techniques and hyper-realistic wind conditions.

As games continue to move towards hyper-realism, the importance of rendering realistic grass becomes more pertinent in allowing the player to be immersed in outdoor environments. There may be one day where rendering grass as proposed in the study only consumes a small portion of the computational power of the system. Fortunately, this study provides a good framework for graphics developers to render more complex real-time grass. Future graphics processors may be effective enough to allow for more complex lighting effects (such as dynamic shadows, real-time ambient occlusion, etc), which can be easily streamlined into the current algorithm. Also, developers may also consider computationally complex wind simulations based on fluid dynamics (such as Navier-Stokes). The possibilities of visual improvement may be endless. But as real-time graphics continue to evolve, so must its grass simulation.

REFERENCES

Bakay, B., Lalonde, P., & Heidrich, W. (2002). Real Time Animated Grass. *Eurographics.* Saarbruecken: University of British Columbia.

Boulanger, K. (2008). *Real-Time Realistic Rendering of Nature Scenes With Dynamic Lighting.* Orlando: University of Central Florida.

Jens, O., Kolb, A., & Salama, C. R. (2009). *GPU-based Responsive Grass.* 2008: University of Siegen.

Kalra, A. (2009). *Rendering Grass with Instancing in DirectX* 10.* Santa Clara: Intel.

Rockstar San Diego. (2010, May 18). Red Dead Redemption. *Red Dead Redemption* . San Diego, California, United States of America: Rockstar Games.

Shah, M., Kontinnen, J., & Pattanaik, S. (2005). Real-time Rendering of Realistic-looking Grass. *Eurographics Workshop on Natural Phenomena* (pp. 77 - 82). New York: ACM.

thatgamecompany. (2009, February 12). Flower. *Flower* . Santa Monica, California, United States of america: Sony Computer Entertainment.

Ubisoft Montreal. (2008, October 21). Far Cry 2. *Far Cry 2* . Montreal, Quebec, Canada: Ubisoft.

VITA

Edward Lee is a full-time software development graduate student at the Guildhall at SMU. The son of Annie Lee and Leo Lee, Edward Lee was born on October 29, 1986 in Los Angeles, California. He attended the University of California, Los Angeles and graduated with a Bachelor of Science degree in Mathematics of Computing, with a minor in Statistics. During his undergraduate career, he has published a paper, *Statistical Filtering of Global Illumination*, through the Institute of Pure and Applied Mathematics at UCLA. After graduation, Edward worked for a year as a web developer at PriceGrabber before deciding to switch to the game industry.

Permanent address: 1250 El Molino Ave, Pasadena, CA 91106

This thesis was typed by Edward Lee.