Rendering Interactive Dynamic Glass

MIKE KUSHNER

Master of Science Thesis
Stockholm, Sweden 2008
Rendering Interactive Dynamic Grass

M I K E  K U S H N E R

Master’s Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2008
Supervisor at CSC was Lars Kjelldahl
Examiner was Lars Kjelldahl

TRITA-CSC-E 2008:113
ISRN-KTH/CSC/E--08/113--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.csc.kth.se
Abstract

Of the modern day methods for rendering grass in real-time, few take physical interaction into account. This has largely been due to constraints in hardware, making it impractical to render enough meshes. The goal of this thesis is to examine previous methods for rendering grass, examine the advances in modern hardware and new methods that may be applicable and finally implement the best solution.

This thesis was written at Avalanche Studios and CSC at the Royal Institute of Technology in Stockholm in the spring of 2006.

Referat

Att rendera interaktivt dynamiskt gräs

Det finns i dagens läge många metoder för att rendera gräs i realtid, dock få som tillåter fysisk interaktion. Begränsningarna har framför allt varit praktiska, då hårdvaran inte kunnat rendera tillräckligt många meshar samtidigt. Målet med detta examensarbete är att undersöka tidigare metoder, framsteg inom hårdvara och undersöka vilka nya metoder som kan appliceras på en implementation.

Detta exjobb utfördes åt Avalanche Studios och CSC vid Kungliga Tekniska Högskolan våren 2006.
## Contents

1 Introduction ................................................................. 1
   1.1 Problem ................................................................ 1
   1.2 Objective ................................................................ 2
   1.3 Method ................................................................ 2
   1.4 Definitions ................................................................. 2
   1.5 About Avalanche Studios .................................................. 4

2 Previous work and Related Research .......................... 5
   2.1 Methods for Rendering Grass .................................................... 5
      2.1.1 The trivial solution .................................................. 5
      2.1.2 Volumetric textures ................................................. 5
      2.1.3 Particle systems ...................................................... 6
      2.1.4 Static textures, 2D .................................................... 6
      2.1.5 Static textures in structures, 2.5D .................................. 6
      2.1.6 Bidirectional Texture Functions ..................................... 8
      2.1.7 Variable LODs ....................................................... 9
   2.2 Plant Distribution ............................................................. 10
      2.2.1 Explicit models ....................................................... 10
      2.2.2 Procedural Models .................................................... 11
      2.2.3 Summary ............................................................... 11
   2.3 A study of NVIDIA’s Nalu tech demo ......................................... 11
      2.3.1 Overview ............................................................... 11
      2.3.2 Geometry ............................................................. 11
      2.3.3 Lighting ............................................................... 13
      2.3.4 Summary .............................................................. 14

3 Theory ................................................................ 15
   3.1 Timing ................................................................ 15
   3.2 Instancing ................................................................. 15
      3.2.1 Overview ............................................................. 15
      3.2.2 The Instance .......................................................... 16
   3.3 Physics ................................................................ 17
      3.3.1 The Forces that Bind Us .............................................. 17
CONTENTS

7 Conclusion 47

8 Acknowledgments 49
  8.1 Additional Reference 49

Bibliography 51
Chapter 1

Introduction

According to NVIDIA, development in GPU technology is following Moore’s law squared [Wlo05]. Whether or not NVIDIA will hold true to this claim, and if so for how long, we will for at least some foreseeable future be able to outsource more and more work to the graphics board and free up CPU-cycles. This development is partly reflected in the design of Microsoft’s Windows Vista, which utilizes the GPU for much of the operating system’s visual interface.

One of the holy grails in computer games is the word *immersion*, that is to say having the player totally immersed in the virtual environment, at best stimulating the player’s previous memories and defining the gaming session as an experience. A factor that detracts from a player’s immersion is lack of detail or lack of realism in detail, and detail is in itself limited by the capabilities of the hardware. The usual solution for this is to simply cheat, using tricks of light, shadow and color to give a surface the illusion of depth. These techniques are difficult to apply on more complex objects such as flora, where there it is the sheer amount of objects that limit performance and not their level of detail. Rendering a blade of grass is a trivial operation. Rendering a complete field of grass able to react to wind and force is what’s tricky.

1.1 Problem

If we had a machine with an unlimited amount of CPU, GPU and memory we simply would not have a problem. We could render an entire field in real-time without even flinching. Sadly we don’t, and this illustrates the main hurdle when it comes to a lot of computing and graphics in particular.

We will in the foreseeable future not be able to render a whole field of grass, but neither is it necessary to do so. It’s not about rendering whole field but how to give the user the illusion that we just have done so. The complexity of the problem can be reduced to a question of how many triangles we can process at one time. All physical objects on the screen consist of triangles, and the more triangles we can put on the screen the greater the detail. Additionally, given that we want to include
lighting calculations in our scene, computational complexity increases exponentially in proportion to the number of polygons rendered [SKP05].

A severe limitation of what we can accomplish is also set by the fact that flora rarely is more than a cosmetic detail in a game, and thus can’t justifiably use resources needed by the computer for more game-essential processes.

1.2 Objective

The point of this project is to investigate existing technologies used to render large volumes of objects and techniques used to render flora and to create an implementation able to display my chosen method. The scope of this project is not to create a photo realistic method for field simulation, but a convincing method using today’s technology and implementable on tomorrow’s hardware. Therefore my solution will prioritize approximation over exactness and feasibility over realism.

1.3 Method

Even though I had access to Avalanche’s game engine during my implementation phase, I instead decided to build my own sandbox partly as a learning experience, and partly because a finished game environment can be a cumbersome place to test a theoretical concept. In the end, I decided to perform my implementation in the following steps:

1. Examine existing solutions.
2. State simulation goals, i.e level of detail and realism.
3. Choose the most suitable solution and adapt it using modern GPU technology.
4. Create a sandbox environment, design the basic architecture for communication between CPU and GPU.
5. Design a wind model.
6. Design a spring model to represent a singular blade of grass.
7. Implement a model for representing a mass of springs.
8. Design a model for transferring the spring information to a field of grass.
9. Implement all of the above and then performance test the model.

1.4 Definitions

The reader of this work is presumed to have a background in computer science and a passing familiarity with computer graphics. Even so, certain concepts and terms may be unknown to the reader, and so I have chosen to list them in this section.
1.4. DEFINITIONS

**Angle/Axis-vector** A way of describing a direction in space, as defined by a three dimensional vector acting as an axis and a rotation about that axis.

**DIP** A single call to the graphics board to Draw an Indexed Primitive.

**DirectX** A set of standard APIs developed by Microsoft for graphics and game programming.

**Euler Angle** A direction as defined by a three dimensional vector. Is prone to gimbal locks.

**FPS** Frames Per Second - the number of updates of screen content that the computer can perform per second. Any values under fifty frames per second detract noticeably from the user’s immersion. A common measurement of an application’s efficiency/a graphics card’s capacity. Can also in gaming terminology denote a First Person Shooter.

**GPU** Graphics Processing Unit. A dedicated processor for performing graphics calculations. The GPU’s architecture is highly parallelized, resulting in a higher degree of efficiency at performing matrix-operations in comparison to the CPU.

**HLSL** Hi-Level Shading Language - a high level shader language developed by Microsoft.

**LOD** Level Of Detail

**ODE** Ordinary Differential Equation

**Quaternion** An extension of complex numbers, a quaternion is a four dimensional vector consisting of a real and three imaginary parts. Can conveniently be converted into Axis/Angle vectors and rotation matrices. Commonly used to describe rotation due to ease of use in regard to transforms, low memory cost and avoidance of gimbal-lock.

**RK4** Fourth Order Runge-Kutta, a method for solving ODEs.

**Shader** A small program written in Cg, HLSL, GLSL or shader-assembler which is executed in the GPU. The shader in itself consists of two parts, a Vertex Shader and a Pixel Shader, which perform effects at vertex and pixel-levels respectively. Commonly, texturing and transformations are performed in the vertex shader while light calculations are performed at pixel-level. The capabilities of a given shader depends on the graphics card on which it is run.

**SLERP** Spherical Linear intERPolation, a technique commonly used in quaternion calculus.
1.5 About Avalanche Studios

Avalanche Studios is a game development studio based in Stockholm founded 2003. The studio is noted for creating the Avalanche Engine, a graphics engine capable of rendering massive landscapes. The company released its first game, *Just Cause*, in 2006.
Chapter 2

Previous work and Related Research

Naturally, quite a lot of research has been performed in virtual botany and different methods for simulating natural environments. This chapter will give an overview of previous research, existing techniques and will further explain the complexity of the problem.

2.1 Methods for Rendering Grass

Several methods for displaying the actual grass have been previously discussed. As I previously stated, it's not the actual complexity of the objects drawn that constitutes the problem, but the sheer volume of them. Thus, techniques for polygon reduction don’t significantly alleviate the situation [PC01].

2.1.1 The trivial solution

The trivial solution is naturally to draw each blade of grass as a 3D-object, perform collision detection on each blade and then simply render them to the screen. This method is inefficient and even if hardware does advance to the point of where it is feasible, it is a clumsy solution. The singular advantage to it is realism, which in the optimal case is absolute.

2.1.2 Volumetric textures

Fur is often simulated using volumetric textures and has been suggested as a viable solution [PC01]. Volumetric textures are three dimensional textures, in essence a sandwich of 2D-textures layered onto each other. But as these don’t allow any animation whatsoever they are of little interest to us besides a mention. An implementation of volumetric textures can be seen in figure 2.1.
2.1.3 Particle systems

Particles systems were suggested for simulating grass as early as 1983. While easy to animate and suitable for simulating grass at long range, detail at close range is lacking [SKP05].

2.1.4 Static textures, 2D

Static textures detract from the geometric complexity of the scene by displaying details in the form of textures. The simplest variant is simply to map the texture onto the terrain. As can be imagined, this method is effective yet results in a total lack of detail and realism when implemented on its own.

2.1.5 Static textures in structures, 2.5D

The trivial method for displaying a static texture object is to have to polygon at a constant tangent to the viewer, resulting in what is known as a billboard and in context a 2D-object in a 3D environment. Since all the detail is static, shadow equations become troublesome and true light equations nearly impossible.
Billboards are normally only suitable to simulate detail in static renders or at long range, any movement at close range will reveal the lack of realism [SKP05]. An example of a grass implementation using this technique is NVIDIA’s grass demo, as can be seen in figure 2.2. A somewhat more advanced technique is to use non-

![Figure 2.2. A screenshot from NVIDIA’s Grass-demo. Note that each blade of grass (represented by a plane) is perpendicular to the viewer.](image)
billboarding clusters of static texture objects. One of the earliest implementations is by Codecult in their Codecreatures Benchmark from 2002, a later example can be found in the Nature Scene Tech Demo by NVIDIA. Both implementations utilize square texture objects arrayed in an approximately triangular pattern, viewed from above either as stars or as inset triangles. By arraying these structures in clusters and setting a slight transparency through either alpha transparency or the more effective method of dissolving, an illusion is given of dense brush seemingly created at random and different from different viewpoints [Wha05]. A good example of this technique can be seen in figure 2.3.

Since 2.5D rendering gives a sense of depth, the question of animation rises. The CodeCreates-demo calculates the animation in the CPU and performs the animation in clusters, allowing for relatively advanced wind motion but required at least one draw call per cluster. Performance versus realism is gaged by cluster size;
larger clusters result in fewer draw calls but a higher degree of synchronization in the animation. Another approach more commonly used today is performing the animation per grass object, letting the GPU perform the calculations necessary. Since this method requires only one draw call, it results in a lower CPU load and avoids synchronization, but does not allow for advanced wind calculations [Pel04].

While this technique gives quite effective results, it is incapable of performing true animation and is limited by the fact that the structures become apparent when viewed from above, as can be seen in figure 2.4. Proponents of the billboard technique point out that by using billboards, in spite of a loss of parallax, perspective is not lost [Wha05].

2.1.6 Bidirectional Texture Functions

Implemented by Shah et al. in 2005, the BTF-method builds on rendering the grass in question from every possible angle and light condition beforehand and storing these images in memory. Rendering of the grass in real time is performed in the pixel
2.1. METHODS FOR RENDERING GRASS

Figure 2.4. Another screenshot from the game *The Elder Scrolls IV: Oblivion* from Bethesda Softworks. The same scene as figure 2.3, but viewed from a higher angle. The planar shapes that make up each bushel become apparent, an effect aggravated by animation. I have filled in a few of the more obvious edges on the nearest bushels.

shader by superimposing the stored grass on the terrain. The interesting aspect of this method is that it builds a full field of grass without any draw calls, and thus there is no CPU drain.

Shah et al. built their texture library using 81 viewing directions under 21 light conditions, resulting in a total 1701 separate 128x128 textures. The main consequence, besides the memory needed to store that amount of textures in GPU memory, is of course that the grass is entirely undynamic and non interactive, thus not of interest of this thesis. Shah et al. mention that the main improvement to their method needed is compression. Depending on future advances in hardware and compression techniques their method may in conjunction with other methods be of serious interest in the future [SKP05].

2.1.7 Variable LODs

Of the above methods, most lend themselves best at rendering grass at either short or long range, if at all. The defining difference between these methods is normally
in the LODs, the levels of detail. By adapting the method used to the distance from the camera (and hence the amount of detail that the user can distinguish) and blending them seamlessly, a complete scene can be rendered using the best characteristics from each paradigm.

Perbet et al. implemented in 2001 a method using three LODs. In their method they render closest to the camera true 3D meshes representing blades of grass performing relatively complex wind operations [PC01]. At a set distance from the camera they instead render layers of flat billboards with textures based on the 3D-grass, somewhat like a volumetric texture but allowing for rudimentary wind animation. The last layer consists of a simple flat 2D-texture mapped onto the underlying terrain.

The problem that arises when using different techniques to render a single scene is the transition between different layers, when the difference can become apparent. Perbet et al. solved this by providing transition states. Transition between the 2D and 2.5D state is solved simply by having the texture slowly fade in over a certain distance depending on the camera’s speed as well as growing in size in order to allow for occurrences when the grass is in contrast to the background. On the other hand, fading between the 2.5D and 3D-layers would give poor results as blades of grass would appear to phase in and out, making the transition apparent to the user. The transition state between 2.5D and 3D layers consists of a a mixture of the two, as a patch transitions from 2.5D into 3D both the texture object and the true blade of grass are displayed. Since the texture is based on the 3D blades, the newly created blade is aligned after the texture and fades in as the texture fades out. Perbet et al. mention that this method creates following two artifacts: the root of the blade of grass seems to 'jump' as the texture is expanded into true 3D, and shading mismatches between the two types of objects. According the Perbet et al. the former is barely noticeable in a full field of grass (from a viewer standing on the ground), while the latter is solved by fading new objects in gradually.

2.2 Plant Distribution

2.2.1 Explicit models

The average person has excellent pattern recognition skills, and repetitive placement of flora is a dead give-away for a scene trying to look natural. At the same time, purely random placement can create certain issues when it comes to switching LODs. Whatley chose the random approach in his article in GPU Gems 2, mostly in order to prioritize ease of implementation and quick rendering in lieu of realism [Wha05]. In his implementation he projects a grid in front of the camera. For each cell in the grid he examines the ground by ray casting from the camera to detect from the terrain’s material properties if that patch is suitable for planting. He does so until the desired density is reached. Whatley states that his method’s main disadvantages are that it can’t handle land bridges, and that since it searches at random, an inordinate amount of CPU time may be used trying to reach the desired
2.2.2 **Procedural Models**

Nature, random as it may seem at times, actually follows a set of rules that we in our turn can use to produce a convincing method for placing flora in a seemingly random yet predictable pattern. Commonly one uses an ecological model such as cellular automata or a reaction-diffusion process to calculate population density while a point pattern generation model or an individual-based population model is used to obtain locations of individual plants. Deussen *et al.* implemented in 1998 a model based on what they refer to as biological neighborhoods. Plants are represented as circles, nodes in an open L-system that branches out over the terrain. The population is established by spreading each species of plant out at random and is then thinned by examining which circles intersect and then culling the smaller of the two. As the simulation is run, plants will grow and when a plant reaches a certain size it is considered aged and dies. This model can be enlarged by including information such as terrain preference, different growth rates and randomizing factors [DHL+98].

2.2.3 **Summary**

Although it would be interesting to procedurally place our grass, it’s arguable whether there would be any noticeable difference except in spots where the grass interacts with other more dominant species such as trees. This can easily be compensated for by other means, and is really not worth the computation time. What on the other hand is desirable is a procedural (yet seemingly natural) system of placing grass which is predictable in order to seamlessly be able to switch between different LODs. Perhaps a variant of a hash algorithm to calculate position using the grid as seed can be applied in our case?

2.3 **A study of NVIDIA’s Nalu tech demo**

2.3.1 **Overview**

In 2005 Nguyen *et al.* at NVIDIA produced the Nalu-demo to showcase their GeForce 6800 series graphics boards. The demo’s main selling point is its protagonist, Nalu the mermaid, and her flowing mass of hair. The hair is self shadowing, scatters light and moves realistically in Nalu’s underwater environment. The reason I chose to study this demo are the similarities between hair and grass physics, as well as the fact that the demo is well documented in *GPU Gems 2*.

2.3.2 **Geometry**

Nalu’s hair is constructed from simple line primitives, 4095 strands using 123,000 vertices to be exact. One of NVIDIA’s goals was to appear to use true dynamics
appear is the operative word), that is to say, handle collision detection against Nalu’s head and shoulders and have her hair move realistically around her in the water. Animating and controlling 4095 individual strands of hair is of course clumsy, hence Nguyen et al. implemented a control hair on each vertex of Nalu’s scalp upon which they calculate movement and collision detection. Each hair is in a sense grown, that is to say, created on Nalu’s mesh procedurally. Each individual control hair consists of seven vertices, whose positions are then used to create Bezier curves and finally tessellated into 36 vertices, giving smooth lines. The final step is interpolating the control hairs on the ends of each triangle to create a dense mass of hair from the surface of the mesh.

Collision detection is handled through a rig of spheres placed around Nalu’s head and shoulders, a method that Nguyen et al. motivated with ease of implementation and computational simplicity. A problem they ran into was that certain hair segments were wider than the collision spheres, resulting in hairs intersecting Nalu’s mesh. Nguyen et al. solved this by attaching collision beads to the vertices on the control hairs. According to Nguyen et al. this method proved faster and safer than running collision detection on the segments themselves. [ND05] A wire mesh rendering of Nalu can be seen in figure 2.6.

Figure 2.5. A screenshot from NVIDIA’s Nalu demo
2.3. A STUDY OF NVIDIA’S NALU TECH DEMO

Figure 2.6. A wire mesh rendering of Nalu

2.3.3 Lighting

Pulvis et umbra sumus (and pretty lighting effects -ed).
-Horace

Nguyen et al. chose the hair reflectance model presented by Marschner et al. at SIGGRAPH 2003 titled Light Scattering from Human Hair Fibres. In short, it considers each hair a translucent cylinder. It then estimates whether a given beam of light follows one of the following paths:

1. It may reflect directly off the surface, towards the viewer.

2. It may refract into the hair and bounce off the inside, towards the viewer.

3. I may refract into the hair, reflect on the inside surface and finally refract back towards the viewer.

Due to the shape of natural hair (which is more like a corrugated tube than a smooth cylinder), this representation gives excellent results while being computationally feasible.
Chapter 2. Previous Work and Related Research

Nguyen et al. state that neither of the two most common methods for simulating shadows today, stencil shadow volumes and shadow maps, are suitable for hair. Stencil shadow volumes become intractable due to the sheer amount of geometry while shadow maps lead to aliasing due to the level of detail. Instead they chose a method called Opacity shadow maps, which allow for fractional shadow values, unlike traditional shadow maps that only allow discreet values. Opacity shadow maps allow for the handling of volumetric objects and antialiasing [ND05]. Figure 2.7 showcases some of these methods.

Figure 2.7. A screenshot from NVIDIA’s Nalu demo showing some of the lighting effects applied by Nguyen et al.

2.3.4 Summary

Nalu’s hair does indeed look quite full and moves beautifully, but sadly the area of her scalp doesn’t represent a field. In a few years representing a field using line primitives may well be feasible (it may even be today), but we can cover a lot more pixels using textured polygons. What we should examine though is Nguyen’s application of control hairs.
Chapter 3

Theory

3.1 Timing

Timing is a classic problem within real time computer simulation. The problem hinges on the balance of accuracy against precision. The more precise an answer one seeks, the larger the overhead required to compile that answer and hence lesser the accuracy. Though it may seem obvious that accuracy is preferable, when it comes to simulations involving small time steps, poor precision can be catastrophic (where as poor accuracy may merely be considered an annoyance). Due to the fact that I chose not to use fixed time steps, poor resolution (where $\Delta t_i = \Delta t_{i-1}$) would cause frame loss, i.e. cycles would be wasted performing pointless calculations.

NVIDIA have thoughtfully provided a tool on their website to compare performance of different timers [NVI99]. The tool compared TimeGetTime(), GetTickCount(), the QueryPerformanceCounter and an assembly based counter. Results will vary from computer to computer, but of the four methods compared QueryPerformanceCounter was clearly the slowest, followed by TimeGetTime() and the assembly based timer, and GetTickCount() the fastest. This corresponds well with the findings of Wassenberg mentioned earlier. [Was04]

3.2 Instancing

3.2.1 Overview

As previously stated, one of the classical problems in graphics programming is how to produce as many triangles onto the screen as possible. Batching is a method to minimize draw calls, but it would be better yet if we could minimize the size of each batch submitted to the graphics card, freeing up the CPU as much as possible. [Car05]
3.2.2 The Instance

In a situation where one would be rendering a large amount of largely identical meshes with little texture variation, one could almost wish that the same vertex buffer could be used time and time again, preferably under a single draw call as well. This one mesh is in essence an instance and using instancing we can reuse it as many times as we would like under that one draw call. The CPU burden would be no greater for drawing a thousand meshes than for drawing one [Car05]. In a mesh dense tech demo by NVIDIA, the framerate jumped from 8-11 fps to 35-45 fps using instancing, a remarkable difference [Dud04]. In a sentence, instancing is a method to avoid DIP calls and minimize batching [Ceb05]. The idea of instancing has been around for a long time [DHL+98], but it has only recently been introduced properly in Microsoft’s DirectX 9 API. Carucci states that there are four methods of instancing, which are as follows: [Car05]

Static batching

Each instance is transformed once in world space, receives its attributes and is then sent directly to the GPU with every frame. The advantage is, since we will only be touching the batch once, we can create the buffer with the D3DUSAGE_WRITEONLY flag, thereby hinting to the driver to put the batch in the fastest possible memory. Static batching is the fastest but least dynamic method. Once the batch is transformed and submitted it’s there, never to be touched again. We can’t perform any skinning or transforms in the GPU.

Dynamic batching

Each instance is transformed in world space as above but leaves the vertex buffer open to editing, allowing us to resubmit transformed vertices and also perform skinning. This method gives us a great deal of control but also drains memory as well as offsets performance due to the amount of time used emptying and resetting the vertex buffer.

Vertex constants instancing

This technique uses vertex constants to store instance attributes. Since we only have a by the graphics card limited amount of vertex constant we are thus limited to the amount of instances that we can send per batch. Carucci mentions that we can on a DirectX 9-class GPU use max 40 instances per batch or 70 with some optimization. Due to the fact that the vertex constants are occupied for the instancing operation there is no support for skinning using this method.

Batching with Geometry Instancing API

Introduced in the DirectX 9.0c API, this is the first API-method to give designed support for instancing. It is superior to the above methods being dynamic, de-
manding minimal effort from the CPU and having a minimal memory footprint. The principal is as simple as it is ingenious, one simply sends to the GPU along with the vertex buffer a value representing the number of vertices in the buffer to be instanced and a value representing the number of sought instances respectively. This method does not support skinning and only allows us to instance a single geometry per draw call. Using this method requires DirectX 9.0c or newer and PS/VS 3.0 compatible hardware [Ceb05].

### 3.3 Physics

Our primary object of study in this thesis is a blade of grass and its respective segments. This section first examines the forces involved in my simulation, and then describes the corresponding physics equations.

#### 3.3.1 The Forces that Bind Us

**External Forces and Collisions**

In this model, each segment is subjected to three external source of forces: gravity, wind and collisions. Wind and gravity are easy to apply as linear forces on each point, these forces being converted subsequently into torque. The dynamics of collisions are described later in this section.

#### 3.3.2 Linear Force, Velocity and Momentum

Given a force $F$ upon a point on an object for a period $t$, the linear movement for that point (note: not the object itself, as there may be dissipation due to angular forces) can be described using the following method. The position $p_t$ can be calculated through

$$x_{t+\Delta t} = x_t + v_{linear} \Delta t + 0.5 \times F/m \times \Delta t^2$$

where $m$ is the object’s mass and $v_{linear}$ is the object’s linear velocity at the time $t$. An important factor when calculating collisions is an object’s linear momentum, which can be found through

$$P_{t+\Delta t} = P_t + F \Delta t$$

#### 3.3.3 Angular Force (i.e. torque), Velocity and Momentum

Given a force $F$ upon a spring (each axis is solved on its own and is independent of the other) from wind or collision, the *torque* can be defined as

$$\tau = r F \sin \theta$$

where $r$ is the perpendicular distance from the axis of rotation and $\theta$ is angle of the displacement arm. Note that $\theta$ has a different value depending on whether the
force is internal (spring) or external. The \textit{angular mass} \( I \) is defined as

\[ I = mr^2 \]

where \( m \) is the mass at the end of the segment (the space between joints is considered massless). In the context of quaternions and vector space, torque has a somewhat simpler definition. Given a force \( F \) acting on a particle \( i \) rotating around the center of mass \( j \), and the vector from \( i \) to \( j \) is \( i\vec{j} \), the torque upon the point \( j \) is

\[ \tau = i\vec{j} \times F \]

according to Baraff [Bar97]. This leads us to the angular velocity, denoted as \( \omega \):

\[ \omega_{t+\Delta t} = \omega_t + I^{-1} * \tau \]

where \( \tau \) is the torque and \( I \) is the segment’s inertia matrix. Finally, the angular momentum is calculated through

\[ L_{t+\Delta t} = L_t + \tau \Delta t \]

### 3.3.4 The Spring Response

Each segment in a blade of grass functions as an individual torque spring. The exact implementation of this system is explained in detail later, but in essence it follows Hooke’s law, which states that

\[ F = -\varepsilon \Delta x \]

where the restoring force \( F \) of a spring is equal to the displacement \( \Delta x \) times the \textit{spring constant} \( \varepsilon \), a positive value that defines the stiffness of the spring [HRW01].

Our system contains no linear springs, as the distance between two points on a blade of grass is constant. Luckily, Hooke’s law holds true for angles just as well as for distances. We calculate the quaternion from \( q_{t+\Delta t} \) to the parent element’s direction.

\[ \delta q = \bar{q}_{t+\Delta t} * q_{\text{parent}} \]

The corrective velocity is calculated by applying Hooke’s Law with the spring constant \( \varepsilon \)

\[ \omega_{\text{correction}} = -\varepsilon * \delta q \]

and is then simply added to the object’s torque.

### 3.3.5 The Physics of Collisions

I chose a simple model for collisions, and chose to limit myself to collisions between spheres and spheres and planes. Detection is done in two steps, primarily by comparing axis aligned bounding boxes and after that spheres. A collision is in essence the penetration of a point into a surface, or the penetration of one surface.
3.3. PHYSICS

into another. Once a collision has been reduced to a matter of sphere/sphere or sphere/plane, the penetration in itself has to be classified as either approaching, withdrawing or resting. If the two bodies are receding from each other, the collision is ignored.

Given a point on the body \( a \) with the mass \( m_a \) colliding with the surface of the body \( b \) with the mass \( m_b \) at the point \( c \) in space, an letting the vector \( n \) represent the normal of \( b \) at the point \( c \), we have the following general values. The velocity of the point on the body \( a \), \( \dot{p}_a \) can be described by

\[
\dot{p}_a = v_a + \omega_a \times (p_a - x_a)
\]

We consequently calculate the velocity of the contact point on the surface of the body \( b \)

\[
\dot{p}_b = v_b + \omega_b \times (p_b - x_b)
\]

Next we calculate the relative velocity between these two points, and by calculating the dot product between that and the surface normal of \( b \) we get the relative velocity along the surface normal, i.e.

\[
\dot{p}_{relative}^a = \dot{p}_a \cdot \hat{n}^a
\]

and

\[
\dot{p}_{relative}^b = -\dot{p}_{relative}^a
\]

If \( \dot{p}_{relative}^a \) is below an arbitrary threshold, the contact is classified as resting. Baraff supplies a method for calculating resting contacts, but Baltman \textit{et al.} supplies an alternative method which is both simpler and incurs far less overhead (at the cost of accuracy). The penalty force on the body \( a \) is given by

\[
d_{penetration}^a = (c_i - e^a) \cdot \hat{n}^a
\]

\[
\dot{d}_{penetration}^a = \dot{v}_{relative}^a \cdot \hat{n}^a
\]

\[
F_{penetration}^a = (k_{spring}^a \dot{d}_{penetration}^a - k_{damping}^a \dot{d}_{penetration}^a) \hat{n}^a
\]

where \( e^a \) is the point on the surface of the collision sphere that passes through the contact point and the sphere’s center of gravity (presuming that the contact point will have passed within the sphere). Likewise, the force on the second rigid body \( b \) is calculated through

\[
d_{penetration}^b = (c - e^b) \cdot \hat{n}^b
\]

\[
\dot{d}_{penetration}^b = \dot{v}_{relative}^b \cdot \hat{n}^b
\]

\[
F_{penetration}^b = (k_{spring}^b \dot{d}_{penetration}^b - k_{damping}^b \dot{d}_{penetration}^b) \hat{n}^b
\]

The forces are finally calculated:

\[
F_{penetration}^a = \min \left( \frac{m_b}{m_a}, 1 \right) F_{penetration}^a
\]
 CHAPTER 3. THEORY

\[ F_{penetration}^b = \min \left( \frac{m_a}{m_b}, 1 \right) F_{penetration}^b \]

This force can be translated back into torque, and consequently linear and angular momentum. [BR04] If the contact is instead classified as colliding, the following method is used to calculate the collision forces, for a given collision. Using Baraff’s method for solving collisions, we solve the impulse \( j \)

\[ j = \frac{-(1 + \epsilon) p_{\text{relative}}^a}{m_a + m_b + \hat{n} \cdot \left( I_a^{-1} (r_a \times \hat{n}) \right) \times r_a + \hat{n} \cdot \left( I_b^{-1} (r_b \times \hat{n}) \right) \times r_b} \]

where \( r_a \) and \( r_b \) are the vectors from point \( c \) to the centers of mass of the bodies \( a \) and \( b \) and \( \epsilon \) is the coefficient of restitution, satisfying \( 0 \leq \epsilon \leq 1 \) and constituting

![Figure 3.1. Resting contact](image-url)
the amount of energy lost in the collision [Bar97]. A screenshot of a collision can be seen in figure 3.2

3.3.6 The Problem with Stiff ODE’s

In the context of any spring (or similarly stiff), the position $x(t)$ is given by the differential equation:

$$x''(t) = f(x(t), x'(t))$$

Defining $v = x'$ we get the following first order ODE:

$$\frac{d}{dx} \begin{pmatrix} x(t) \\ v(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ f(x(t), v(t)) \end{pmatrix}$$
The essence of the problem has to do with the size of the time step versus the spring constant $k$ in stiff equations. Large values for $k$ will give large jolts of acceleration to the particle, and if the time step is too big the entire equation blows up, i.e. the actual spatial coordinates being computed no longer correlate with reality. As Baraff likes to emphasize, it isn’t pretty. The trivial solution to this is to either arbitrarily constrain the size of a timestep (silly at best, not applicable at worst) or to limit the degree of accuracy (which will result in simulation errors) [BW99].

3.3.7 The Many Solutions to Stiff ODE’s

The above method of solving an ODE is known as the Explicit Euler method, which though simple and fast is in its trivial form is only $1^{st}$ order accurate, and may at its best in a modified form give $2^{nd}$ order accurate results [Dum05]. The order the error term for Euler’s method is $O(h^2)$, making it a first order method. Due to the equation’s simplicity (i.e triviality) it is considered efficient but essentially useless [PTVF92]. Though there are countless integration schemes, I will in the following subsections limit myself to describing the most common.

Implicit Euler

Instead Baraff suggests the Implicit Euler method, also known as the Backwards Euler, which attempts to solve the equation from the sought after point and going back to the origin rather than the other way around. By defining $h$ as the step size, $x_0 = x(t_0)$, $v_0 = v(t_0)$, $\Delta x = x(t_0 + h) - x(t_0)$, $\Delta v = v(t_0 + h) - v(t_0)$ and applying Barraff’s method to the above ODE yields the following equation system:

$$(\Delta x \Delta v) = h \left( f(x_0 + \Delta x, v_0 + \Delta v) \right)$$

Regrouping the above through several steps and letting $I$ represent the identity matrix finally yields

$$\left( I - h \frac{\partial f}{\partial v} - h^2 \frac{\partial f}{\partial x} \right) = h \left( f_0 + h \frac{\partial f}{\partial x} v_0 + \frac{\partial f}{\partial t} \right)$$

according to Baraff. Implicit Euler can solve an ODE with an unlimited time step, but suffers from the same accuracy issues as Explicit Euler and is of the common methods the most complex, both in computation and implementation.

Fourth Order Runge-Kutta [rk4]

The Fourth Order Runge-Kutta method, more commonly known as rk4 or simply Runge-Kutta, is an adaptation of the Second Order Runge-Kutta, also known as the Midpoint method, a (as subtly implied) second order method.

The midpoint method is based on using the initial derivative at each step to find a point halfway across the interval and then using the midpoint derivative across
3.3. PHYSICS

the full step of the interval, giving us a second order equation. \( rk4 \) keeps running
with this idea on a tangent by adding two additional steps and doubling the size of
the given interval, giving us a fourth order equation. The classical \( rk4 \) formula is
the following:

\[
\begin{align*}
  k_1 &= hf(x_n, y_n) \\
  k_2 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\
  k_3 &= hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\
  k_4 &= hf(x_n + h, y_n + k_3) \\
  y_{n+1} &= y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5)
\end{align*}
\]

The basis for \( rk4 \) is that each step is evaluated four times, once at each endpoint
and twice at trial midpoints. As a method that gives excellent accuracy at palatable
complexities, \( rk4 \) (often with adaptive stepsize control) is one of the most common
ODE-solvers. [PTVF92]

**Verlet Integration**

Verlet Integration method solves time stepping errors problem by simply eliminating
the velocity from the equation. Instead of Euler’s standard integration scheme we
store a particle’s position in \( x \) and its previous position in \( x_{i-1} \). The integration
step is then as follows [Jak01]:

\[
x_{i+1} = 2x_i - x_{i-1} + a * \Delta t^2
\]

The problem with the Verlet Integration scheme is that it requires two criteria not
commonly available in a real environment, namely fixed time steps and constant
acceleration. All Verlet schemes are inherently vulnerable to changing time steps,
though Dummer has solved this for the positional scheme. Dummer also cites that
error due to changing acceleration is dismissable [Dum05]. A common modification
to the standard verlet (often cited as Jacobsen’s scheme) is provided both by Balt-
man et al. and other authors, which introduces a damping factor \( \xi \) to the equation
[BR04]:

\[
x_{i+1} = x_i + \xi (x_i - x_{i-1}) + a * \Delta t^2
\]

There are also variants of the verlet scheme, such as the Leap-Frog algorithm and
Velocity Verlet.

**Reverse Euler**

In order to provide smooth movement, Baltman et al. use the following 'Verlet-
like'\(^1\) quaternion integrator. Worth noting is the fact that this solution does not
utilize SLERP, unlike the simpler solution provided in the same paper.

\(^1\)In truth a reverse Euler (nomenclature varies). It avoids exploding the equations by calculating
the velocity a step after the actual movement, but suffers from the standard Euler’s low resolution
\[ q_{t+\Delta t} = q_t + \dot{q}(q_t, \omega_t) \Delta t + \frac{1}{2} \ddot{q}(q_t, \omega_t, I^{-1}\tau_t) \Delta t^2 \]

and

\[ \omega_{t+\Delta t} = \xi \omega_t + I^{-1}\tau_t \Delta t \]

where \( \tau \) is the torque at the time \( t \), \( \omega \) is the rotational velocity, \( \xi \) is the dampening factor and where

\[
\dot{q}(q, \omega) = \frac{1}{2} q \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \\ 0 \end{bmatrix}
\]

and

\[
\ddot{q}(q, \omega, \dot{\omega}) = \frac{1}{2} q \begin{bmatrix} \dot{\omega}_x \\ \dot{\omega}_y \\ \dot{\omega}_z \\ -2 \dot{q} \cdot \dot{q} \end{bmatrix}
\]

Note that there are discrepancies between the formulas stated here and those found in [BR04] due to Baltman et al. preferring angle-axis-quaternions while DirectX prefers axis-angle.

### 3.4 CPU vs GPU

All computers and consoles have one or more central processing units, each containing one or more cores (CPUs which share the same internal memory). For example, the Microsoft XBox 360 ships with IBM’s Xenon chip containing three cores, while the Sony Playstation 3’s ships with six functioning processors (it actually ships with eight, but one is dedicated to OS and security functions and one is redundant to improve production yields). A CPU is a very general processor, and while that is appropriate for everyday needs, computer graphics are as a rule largely floating point operations (FLOPs), and are highly parallelizable. For that reason, all game-adapted PCs and gaming consoles also ship with a separate Graphics Processing Unit, a processor specialized for handling parallel computation in general, and computer graphics in particular. Due to the GPU’s specialized design, a lot of non-graphics-related applications such as scientific calculations and physics algorithms in games utilize the GPU as a processor in its own right.

### 3.5 Simulating Wind

#### 3.5.1 Dynamic Simulation

Simulating wind blowing across grass is in many ways similar to simulating water as both mediums are, in essence, fluids. A gust of wind can be seen as a wave moving across the field of grass, losing momentum and interacting with other waves as it
3.6. TEXTURE ATLASING

comes along. Forces applied to the field are often represented as a planar grid where each node (i.e. blade of grass) is connected to four neighbors [Ram05].

As a point of interest I’ll mention that wind simulation can also be implemented effectively in a three dimensional medium, e.g. Wei et al. who in 2003 implemented a Lattice Boltzman Model [WZF+03].

The main disadvantage to dynamic simulation is that even while it can simulate wind conditions perfectly, it needs memory to store the planar grid. In the case of simulating grass, where fluid motion is not as distinct as it would be in a body of water, this requirement is unwarranted and unacceptable to a memory intense application such as a game.

3.5.2 Procedural Simulation

In cases where memory is a constraint or when the level of detail given by dynamic simulation is simply unnecessary procedural wind simulation comes into its own. There are two applications for procedural wind, where the most basic is to create some form of basic movement in an object. In a simulation that demands little more realism than having the grass appear not to be static, a mere sinus function incorporated at vertex level in the shader gives sufficient results.

For more advanced wind simulation, use can be made of procedural wind primitive, a concept similar to a light primitive. Given a set of wind primitives and a time function, the effect of wind upon the fluid can be calculated at each point [PC01].

3.6 Texture Atlasing

A key concept in graphics programming is CPU efficiency, and minimizing draw calls by effective batching is a good start. The term batch-breaker describes an action that forces an application to cut a batch short of its optimal size, and foremost of these are state changes. A typical state change is a call to Direct3D’s SetTexture()-command, a typical action one implements when rendering a scene filled with diverse objects, or even just a scene with many similar objects using varied textures. The consequence of this is that we require one batch per texture.

But if we know that we are consequently going to use the same two textures for what might as well be one batch, why don’t we just combine them into a single texture? In the optimal case (n textures, n batches) we can halve our calls to the CPU. This is the essence of Texture Atlasing. The basic process simply involves first combining the two textures to a single texture and then adapting the application’s UV-coordinates to call for the appropriate texels. Triviality fades when one starts working with mip maps\(^2\), an ubiquitous method of minimizing scene complexity. Care must be taken that two or more mip-mapped textures don’t interfere with

\(^2\)The technique of including several versions of a texture in decreasing resolutions, allowing simpler textures to be used at ranges where detail is wasted.
each other. In applications where mip-maps are generated on the fly, this is a common source of artifacts. A simple solution is to specify that all texture sizes have to be powers of two and equal [Wlo04].
Chapter 4

Implementation

4.1 The Grass Model

Even a simple blade of grass is actually a quite complex physical object. The substance of a plant consists of water filled cells giving structure, which in turn are given tensile and rigid strength by fibers running length and crosswise along the body of the plant. Simulating an exact system is of course pointless, since we are merely trying to create a visual illusion and not an exact simulation. We subject it merely to whatever forces we choose ourselves. Hence we can make do with a far simpler (and easily calculated) model. This section describes my simulation model, from the atomic level and up, encompassing both its strengths and limitations.

4.1.1 From a spring...

Once we have decided to implement a true geometric model (foregoing any GPU based optical illusions), it becomes self evident that the smallest unit in the simulation has to be a spring. My first idea was to build a stable jelly model to represent a blade of grass, but was forced to dismiss it as impractical. A jelly model requires something in the range of 20 springs per segment to be considered stable, which is far too expensive to simulate in the volumes we require. My implementation instead uses torque springs, i.e, springs subject to differences in direction instead of position, with counterforces being applied through torque instead of linear force. All force applied to the base end of the spring is ignored, while force applied to the tip is translated into torque. The physics of such a spring is described in subsection 3.3.4.

4.1.2 ...to a blade

In my implementation, a full blade of grass is represented by a spring structure of interlocking torque springs (as can be seen in figure 4.1). Each individual spring derives its spatial position from its parent, while forces and linear momentum applied to the child are converted into torque and angular momentum. As mentioned
Figure 4.1. Screenshot showing two spring structures labeled A. and B. being subjected to force. Each has five interconnected torque springs, on structure A labeled \( \theta \) to 4.
4.2. INSTANCING

In the previous subsection, each individual spring is locked in linear space, meaning that it’s position always is a function of its parent’s (lying on a line parallel the parent’s direction, at a constant distance). In addition to this, each child also inherits its parent’s torque, a cosmetical touch which avoids obtuse angles. Note that interaction never occurs more than one step in either direction, a conscious decision to make improve scalability.

This system is based on the principle that each individual spring is self righting, and attempts to adjust its direction (and unwittingly, all its children) to the direction of its parent. The advantage of this scheme is simplicity, because it guarantees that the geometry always remains valid (a system based on linear springs would have to employ complex distance constraints to keep the system from stretching) while avoiding the complexity of a scheme based on Inverse Kinematics. The final result can be seen in figure 4.2.

4.1.3 ...to a field

Once an adequate spring structure has been established, a system must be found for running simulations on them en masse. The trivial solution of letting each spring structure represent an individual blade of grass and linking a geometric shape to the structure’s deformation is obviously poor.

Drawing inspiration from NVIDIA’s Nalu (see section 2.3, I instead chose to create a grid of spring structures, where any blade of grass (as represented by a geometric structure) interpolates its deformation from the surrounding four springs. This operation can be performed very effectively in the GPU’s vertex shader, taking load off the CPU. A screenshot of this grid can be seen in figures 4.3 and 4.4.

4.1.4 Geometric Representation

In my implementation I have chosen a relatively trivial structure to represent a blade of grass: two dual-sided textured planes placed perpendicular to each other. I prefer this method to billboarding because it looks better from an isometric perspective. A single blade of grass can be seen in figure 4.2.

4.2 Instancing

Of the methods mentioned in the previous chapter, the API based method might as well be tailored for our needs. For our field we will be drawing large volumes of a relatively simple mesh with small transformation. Using geometry instancing together with effective batching we can in other words produce a whole field of grass with the GPU as the only bottleneck. The effects of instance in my implementation can be seen in figures 4.5-4.7
Figure 4.2. Screenshot showing a single blade of grass, its deformation interpolated from the surrounding four spring structures. Note the cross shaped base.
4.2. INSTANCING

Figure 4.3. Screenshot showing an 8x8 grid of five sectioned spring structures at rest.
Figure 4.4. Screenshot showing an 8x8 grid of five sectioned spring structures in various stages of deformation while being subjected to forces from a sine based wind.
4.3 Physics

4.3.1 ODE Solvers

At first glance, Verlet integration seems to be the most logical, being by far the least complex in relation to accuracy. Sadly, the standard verlet scheme is unusable for any physics simulation involving springs or collisions since velocity is impossible to extract from the equation in realtime. Without velocity we lose the ability to track momentum, and thus any attempts at a collision scheme using standard Verlet fail.

While speed may seem like the most important criteria for an integration scheme, one needs to take into context the scope of this thesis. Since hardware is constantly evolving, any explicit measurements are pointless, and our only interest is for a
scheme which is reasonably accurate and scales well. Even dismissing the standard Verlet, we are left with several possible schemes to use. Of those stated above, the by far most well documented was RK4, which therefore became my scheme of choice.

4.4 CPU vs GPU

Nearly the entire simulation process consists of floating point operations so the advantage of outsourcing as much work as possible to the GPU is blatantly obvious. The GPU can handle physics simulation far faster than the CPU, which is in turn free to perform other tasks. The only cost is a slight overhead for transferring data
4.4. CPU VS GPU

Figure 4.7. Screenshot showing 2048 instanced blades. Note the lack of difference in framerate compared to 4.5.
back and forth from the GPU’s memory. In this light it should seem obvious for me to move all the physics work down to the GPU, so the reader may wonder why I have chosen to keep the simulation at CPU-level. The reason is as simple as the fact that GPU code is inherently difficult to debug. Since such an implementation wouldn’t affect the complexity of my solution, I chose to keep it in an environment where it was simple to develop. In the current solution the entire simulation is performed on the CPU. The results from the simulation are then processed into a map of float quadruplets which are sent down to the GPU. The GPU then applies this data to the instanced grass geometry in the vertex shader.

4.5 Simulating Wind

I chose to use a generalized procedural model for wind simulation based on the CPU, mostly to keep the solution open for future implementations. I have created basic models for a diffuse and a point wind with wind strength varying according to a simple sine curve. Should the user be inclined, both period and strength can be varied according to the user’s wishes for further realism, and the solution can easily be expanded to include explosions and slip streams. An example can be seen in figure 4.8.
4.5. SIMULATING WIND

Figure 4.8. Instanced grass being subjected to a sinus-curve wind. This implementation looks and acts like previous implementations while fully interactive with other objects in its environment.
Chapter 5

Analysis

5.1 Code Analysis

In spite of the statistical results, an analysis of the code is useful for finding potential bottlenecks and possible optimizations. Examining the code through a profiler reveals the following two bottlenecks:

5.1.1 Evaluation of the Spring System

In the process of evaluating the spring system, there is a slight performance penalty while iterating through the data structure containing the springs. The only performance drain after that point is the final evaluation function in each torque spring, which does the following:

1. Nulls all forces currently on spring.

2. Converts all forces and linear momentum on child-object into torque and angular momentum on spring.

3. Run ODE-solver on spring.

4. Update state

Although the number of possible improvements and optimizations on this scheme are far too many to enumerate, what we do witness is the fact that each cycle only involves the spring and it’s one child. Scaling should, in other words, be linear.

5.1.2 Rendering of the Spring System

Delving into the rendering of the spring system, the main bottleneck occurs when producing the large float array of instance deformations that gets transmitted down to the GPU. Compiling this data leads to a large amount of calls to std::vector::at, the elimination of which (through some debug work) releases the bottleneck.
5.2 Statistical Testing

With computer optimization being an art in itself, and with Moore’s law still in effect, pure results are of little interest. What is instead interesting to view is how my solution scales. I have chosen to examine two aspects: increasing spring complexity and the performance of DirectX’s instancing routines.

I performed the tests on my own computer, a Pentium4 running at 3.0 GHz with 1.0 GB of RAM and an NVIDIA GeForce 7800 GS GPU. Neither framerate nor timestep have been fixed in these simulations, meaning that the computer will always attempt to perform as many calculations as possible.

5.2.1 Spring Complexity

This test was performed by increasing the spring complexity, i.e. increasing the number of springs that constitute a blade of grass. I chose to lock the number of drawn instances to 0 and set the spring density to an 8x8 grid, meaning that the total number of active springs can be expressed as the product of $64 \times (n - 1)$. Even though the top vertex of each spring structure is represented by a string object, no calculations are ever performed on them. No optimizations have been implemented, meaning that the simulation performs collision detection on each and every spring, for every frame.

5.2.2 Spring Structure Density

The second test was to confirm that behavior was not affected by spring complexity. Hence the first test was performed again, but this time increasing the spring structure density instead of their complexity. The spring complexity was set to 5 and again the number of rendered instances was set to 0.

5.2.3 Instance density

The object of this test was to examine the performance of DirectX9c’s spring instancing routines, how its overhead affected performance. Locking the spring structure density to 8 and spring complexity to 5, I performed the test by increasing the number of instanced objects from 0 to 11264 in increments of 512.

5.3 Conclusions

5.3.1 The Spring System

The conclusion drawn from the code analysis is that my implementation, while clumsy and somewhat simplified, essentially performs the task of realistically simulating large volumes of grass and scales at a linear rate. The statistical analysis compiled in figures 5.1 and 5.2 clearly show a linear increase in rendering time per
5.3. CONCLUSIONS

frame. Although this unoptimized and clumsy implementation clearly is a worst-case scenario, this proves that my solution is viable.

5.3.2 Examining the Instantiation Scheme

As one might expect, a linear increase in the number of instances rendered yields a linear increase in rendering time, as can be seen in figure 5.3. The only curiosity is the saw-toothed shape of the curve, which suggests cache-hits/misses or a similar phenomena.
Figure 5.2. Spring structure density
5.3. CONCLUSIONS

Figure 5.3. Instance density
Chapter 6

Discussion

6.1 Difficulties

During the process of writing and implementing this thesis, several issues have cropped up which either caused me to reevaluate my design or question design choices. The more serious of these were taken care of, and aspects of my thesis were revised, but due to constraints in time and effort there are a number of points which I would like to improve upon given the time.

6.2 Further Work

6.2.1 An Improved Grass Model

As mentioned previously, my model for simulating an individual blade of grass lacks in realism what it gains in simplicity. Given a chance to improve my model, there are primarily two aspects I would like to add. The first and easiest to implement is making the model a true rigid body and not merely elastic. Moving the tip of a real blade of grass will effect the whole blade, while in my implementation it merely effects the relevant segment. By the same measure, raising the tip involves tensile forces from the whole length of the blade, and not only the affected segment. A more advanced model would give a more esthetically pleasing simulation while not adding complexity. The second aspect is the fact that many plants aren’t perfectly symmetrical, and will, instead of bending equally in all directions, instead bend to a certain degree before translating the bend into a twist. My model doesn’t support this functionality.

6.2.2 Improvements in Spring Resolution

One of the advantages to my model is the fact that resolution, that is to say the density of feelers within the field, can be varied after preference. But to take full advantage of this design the resolution should be varied dynamically, depending on need. There are two cases when the resolution can be decreased:
Distance

Just as the LOD of polygons is decreased with distance in order to limit the amount of polygons on the screen, the resolution can be decreased in order to limit the amount of spring operations that have to be calculated.

Variable spring density

A good way of saving resources and memory is adapting the density of the spring structures as needed, both based on viewing distance and by detail required (compare a tank to a soccer ball).

6.2.3 Improved Collision Detection

When it comes to collision detection, an important detail is to limit the number of actual checks needed. The natural solution to this problem is to sort the springs into quadtrees, as specified by Ferraris [Fer01]. A modification of the classical solution is to use loose quadtrees as specified by Ulrich whereby each quadtree overlaps its neighbors [Ulr00]. The reason for this is that the bounding sphere of each blade of grass is surprisingly large. When collision needs to be checked, the segments of a given blade may be anywhere within a radius of the total length of the blade from the blade’s root. My reason for not using octrees is that while the position on the xz-plane of a certain segment is static within the blade’s bounding sphere, a segment’s height can vary anywhere from the bottom of the y-axis to the maximum height of the blade. Simply put, while a quadtree gives an approximation of the segment’s position on the xz-plane, an octree would say nothing about its height.

6.2.4 Moving work down to the GPU

As I have mentioned before, my current implementation uses the CPU extensively to perform spring calculations before sending the results down to the GPU. Transferring these calculations to the GPU would result in faster calculation speed (due to the GPU’s efficiency at floating point operations) and less overhead, since the results don’t have to be transferred over the pipeline each cycle.
Chapter 7

Conclusion

As was hopefully proven in the introductory chapters, there were at the time of this thesis’ writing no techniques of simulating grass properly without resorting to the trivial approach, i.e. performing a mechanical simulation. By applying a wide range of techniques, both implemented in my solution and as suggested in the last section of this chapter, the computational severity of performing this simulation can be lessened to the degree of which it is feasible. Though my suggested solution introduced no new concepts to the world of computer graphics, it combined several existing concepts to produce an effect that at the time of this thesis’ writing has not been widely used in mainstream computer games.
Chapter 8

Acknowledgments

I would like to thank my supervisors at KTH and Avalanche Studios, Lars Kjelldahl and Linus Blomberg respectively. I would also like to thank my fellow interns Ragnar Svensson, Leif Jonsson, Olof Björk and Fredrik Zetterman for their support, suggestions and help. Lastly I would like to thank Patrik Broman for getting me interested in the world of computer graphics.

Lastly, a huge thanks to Adam Roigart for drawing a custom texture for me.

8.1 Additional Reference

The following works have been used throughout this thesis as general reference:

- *3D Math Primer for Graphics and Game Development* by Fletcher Dunn and Ian Parberry [DP02]
- *3D Game Engine Design* by David H. Eberly [Ebe01]
- *The C++ Programming Language* by Bjarne Stroustrup [Str97]
- *Effective C++, Third Edition* by Scott Meyers [Mey05]
Bibliography


BIBLIOGRAPHY


