Contents

List of Figures

Abstract

Over the last few years, the computer graphics hardware has evolved extremely fast from supporting only a few fixed graphical algorithms to support execution of dynamic programs supplied by a developer. Only a few years back all graphics programs were written in assembly language, a nonintuitive low level programming language. Today such programs can be written in high level, near written English, source code, making it easier to develop more advanced effects and geometric shapes on the graphics card.

This project presents a new way to utilize today's programmable graphics card to generate and render trees for real-time applications. The emphasis will be on generating and rendering the geometry utilizing the graphics hardware, trying to speed up the calculation of naturally advanced shapes for the purpose of offloading the systems central processing unit.

Preface

The autumn of 2004 I got into a project with Åsmund Nordstoga creating a parametric tree rendering system for real-time applications. I found this part of visualizing natural environments very interesting and satisfying and decided to write my thesis on this subject. For this new project I wanted to explore and utilize the revolutionary new features the pioneering computer graphics cards have to offer. I also wanted to improve the visual quality of the trees rendered using our previous system.

Acknowledgements

The amount of hard work and thoroughness of this report and the application implemented could not have been finished at the set dead-line without the help of some people.

I would especially like to thank my main supervisor, PhD candidate Jo Skjermo, for guiding me through the implementation phase and the entire phase of writing this report. Skjermo has given me ideas of what to include in the project and guidance on what may or may not work.

I would also like to thank Kristian Eide at Systems In Motion for valuable input and some interesting discussions.

Finally I would like to thank Ben Humphrey at GameTutorials.com for making his source code available to be able to jump start this project, getting into implementing the really interesting part of the code that really mattered for the results achieved.

Chapter 1

Introduction

Many papers have been written on rendering trees. *Creation and Rendering of Realistic Trees* by Weber and Penn [JW95] emphasize on speed and simplicity in defining tree types, while *Modeling the Mighty Maple* by Bloomenthal [Blo85] attach importance to making the trees look more real and natural. This project will emphasize on rendering the stems making up a tree as real as possible while still keeping a frame rate acceptable to real-time simulation. I will focus on drawing trees close-up, meaning that the observer is within a 100 meter radius of the tree being rendered. The project will look to techniques of utilizing the GPU (Graphics Processing Unit) to offload the workload on the CPU (Central Processing Unit). The project will try to combine the parametrical system proposed by Weber and Penn with the idea of representing branches by spline curves as proposed by Bloomenthal. Even though leafs at times is an essential part of a tree, this has not been implemented in this project. The reason for this is that I would like to focus on rendering curved stems without covering them with billboard leafs done in so many projects before. Further references to a tree in this report refers to all the stems making up the tree, meaning its trunk and branches. Implementing leafs is proposed as future work and should not be a difficult task to combine with the existing code.

The next chapter is meant to give the reader a presentation of some of the previous work done within rendering trees on a computer and other techniques to enhance visual realism of a virtually generated scene. In chapter 3 I will present my approach of implementing a real-time system for rendering trees using the techniques described in chapter 2, as well as some other techniques for optimizing the execution of the program. Chapter 4 renders the results from testing the application implemented and discusses the data collected. Chapter 5 will try to summarize the report and finally chapter 6 will propose future work relevant to this project.

Chapter 2

Previous work

This chapter is meant to give the reader a understanding of some of the work done in the field of simulating trees on a computer in three dimensions, as well as background information on other techniques within real-time visualization relevant to this project.

2.1 Creation and Rendering of Realistic Trees

One of the more famous and discussed articles within the field of rendering trees on a computer is *Creation and Rendering of Realistic Trees* by Weber and Penn [JW95]. This article presents a model to create and render trees based on a few parameters describing the features of a certain tree type. The advantage of this approach is the speed up of using parameters instead of advanced natural evolution algorithms when constructing a tree. Building a stem using parameters only requires a few calculations and can be done iteratively while trees constructed using for example L-systems[1] use time-consuming recursive algorithms. Hence systems such as proposed by Weber and Penn, with today's hardware, may be used for real-time rendering.

2.1.1 Skeletal parameters

A single stem in *Creation and Rendering of Realistic Trees* is created by gluing together smaller near-cylindrical segments, where each cylinder has its own reference frame rotated a certain degree from the previous cylinder in the stem (see figure 2.1). The center of each cylinder defines the skeleton of the stem. The number of near-cylindrical segments is defined by the parameter $nCurveRes$. The n in front of the parameter always refer to its recursive level, and if there is a V at the end of a parameter name it stands for variation. If $nCurveBack$ is zero the z-axis of each segment on the stem is

[1] Aristid Lindenmayer (1925-1989) introduced a string rewriting system for cellular interaction which was later applied to plants and trees. The notion of rewriting is central to L-systems where the basic idea is to define complex objects by successively replacing parts of a simple object using a set of rewriting rules or productions. [LA90]

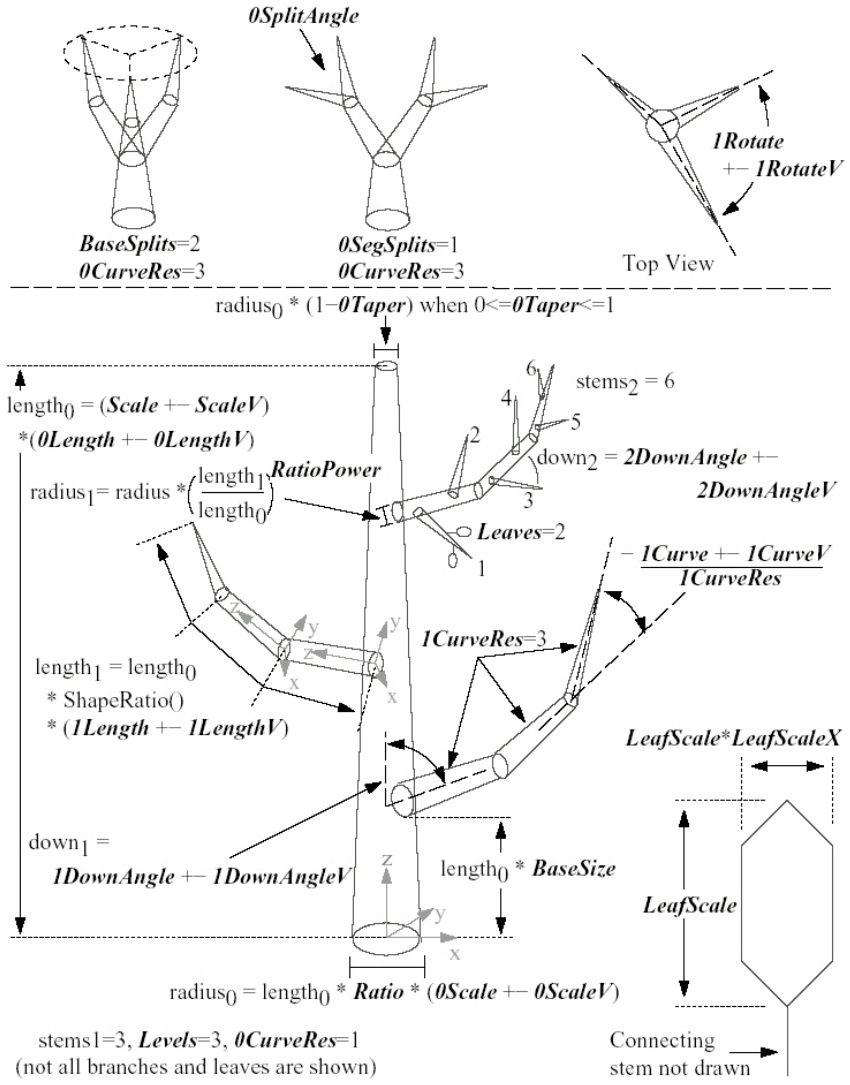


Figure 2.1: Tree Diagram [JW95]

rotated away from the z-axis of the previous segment by $(nCurve/nCuveRes)$ degrees about its x-axis. If $nCurveBack$ is nonzero, each of the segments in the first half of the stem is rotated $(nCurve/nCurveRes/2))$ degrees and each in the second half is rotated $(nCurveBack/nCurveRes/2))$ degrees. This makes it possible to form simple S shaped stems. In either case a random rotation $random(nCurveV/nCurveRes)$ is also added for each segment.

$nBranches$ defines the maximum number of child stems that a stem can create over the length of all its segments. In the article by Weber and Penn the number of successive child stems (really "grandchildren") is computed as

$$stems = stems_{max} * (0.2 + 0.8 * (length_{child}/length_{parent})/length_{child,max})$$

for the first level of branches, and

$$stems = stems * (1.0 - 0.5 * offset_{child}/length_{parent})$$

for further levels of branches.

The length of a child branch is calculated as:

$$length_{child} = length_{trunk} * length_{child,max} * \\ ShapeRatio(Shape, (length_{trunk} - offset_{child})/(length_{trunk} - length_{base})))$$

for the first level of branches and

$$length_{child} = length_{child,max} * (length_{parent} - 0.6 * offset_{child})$$

for further levels of branches. $offset_{child}$ is the position in meters for a child stem along the parents length from the base. $length_{base}$ is the length of the bare area at the base of the tree. The length of the trunk is calculated:

$$length_{trunk} = (0Length \pm 0LengthV) * scale_{tree}.$$

If $nDownAngleV$ is positive, the z-axis of a child will rotate away from the z-axis of its parent about the x-axis with the angle $(nDownAngle \pm nDownAngleV)$ degrees.

If $nRotate$ is positive, a child stem is rotated about the z axis, relative to the previous child by the angle $(nRotate \pm nRotateV)$ forming a helical distribution.

Using a normalized position along the trunk from 0.0 to 1.0 the *shape* parameter defines the overall tree shape according to the formula ShapeRatio(shape,ratio):

Shape	Result
0 (conical)	0.2 + 0.8 * ratio
1 (spherical)	0.2 + 0.8 * sin(p * ratio)
2 (hemispherical)	0.2 + 0.8 * sin(0.5 * p * ratio)
3 (cylindrical)	1.0

```
4 (tapered cylindrical) 0.5 + 0.5 * ratio
5 (flame)               ratio/0.7 if ratio<=0.7
                        (1.0 - ratio)/0.3 if ratio>0.7
6 (inverse conical)     1.0 - 0.8 * ratio
7 (tend flame)          0.5 + 0.5 * ratio/0.7 ratio<=0.7
                        0.5 + 0.5 * (1.0 - ratio)/0.3 ratio>0.7
```

2.1.2 Surface parameters

Flare (increased radius) exists along the base of many trees to support their massive weight and height and is computed with the value of y being a normalized position along the trunk:

$$flare_Z = Flare * (100^{y-1})/100 + 1$$

where $y = 1 - 8 * Z$, and limited to the value of zero.

Lobes specifies the number of peaks in the radial distance about the perimeter along the base. Preferably it should be a odd number to avoid symmetry along the base. $LobeDepth$ defines the depth of the lobes, using the formula: $lobe_Z = 1.0 + LobeDepth * sin(Lobes * angle)$.

The stem radius is given for the trunk and branches respectively:

$$radius_{trunk} = length_{trunk} * Ratio * 0Scale$$

$$radius_{child} = radius_{parent} * (length_{child}/length_{parent})^{Ratiopower}$$

The radius is also tapered along its length according to the $nTaper$ variable using the formulas:

```
nTaper Effect
0      Non-tapering cylinder
1      Taper to a point (cone)
2      Taper to a spherical end
3      Periodic tapering (concatenated spheres)
```

The periodic tapering is mainly used for modelling cactus. For a normalized position Z from 0 to 1 along the length of a stem, the following equations compute $radius_Z$, the tapered radius in meters:

$$\begin{aligned} unit\_taper &= nTaper & 0 \leq nTaper < 1 \\ unit\_taper &= 2 - nTaper & 1 \leq nTaper < 2 \\ unit\_taper &= 0 & 2 \leq nTaper < 3 \end{aligned}$$

$$taper_Z = radius_{stem} * (1 - unit\_taper * Z)$$

and when $0 \leq nTaper < 1$

$$radius_Z = taper_Z$$

or when $1 \leq nTaper \leq 3$

$$\begin{array}{l} Z_2 = (1 - Z) * length_{stem} \\ \quad depth = 1 \qquad\qquad (nTaper < 2) \quad or \quad (Z_2 < taper_Z) \\ \quad depth = nTaper - 2 \qquad otherwise \\ \quad Z_3 = Z_2 \qquad\qquad nTaper < 2 \\ \quad Z_3 = |Z_2 - 2 * taper_Z * \text{int}(Z_2/(2 * taper_Z) + 0.5)| \qquad otherwise \\ \quad radius_Z = taper_Z \qquad\qquad (nTaper < 2) \quad and \quad (Z_3 \geq taper_Z) \\ \quad radius_Z = \begin{array}{c} (1 - depth) * taper_Z + \\ depth * sqrt(taper_Z^2 - (Z_3 - taper_Z)^2) \end{array} \qquad otherwise \end{array}$$

Together the variables $radius_Z$, $lobe_Z$ and $flare_Z$ defines the radius at any point along the stem.

2.2 Geometric representation

Objects which have been formed by the randomness of natural evolution, often take the shape of curved forms. For this reason curves are often used to represent the skeleton of three-dimensional organic forms such as tree limbs. Probably the most common way to make a shape follow the trajectory of a certain curve is to let a two-dimensional shape make up the outer boundary of the object and then move the center of the shape along the curve. This section will describe the methods used for creating a geometric representation of a tree.

2.2.1 Spline curves

In the mathematical subfield of numerical analysis a spline is a special curve defined piecewise by polynomials. The term spline comes from the flexible spline devices used by shipbuilders and draftsmen to draw smooth shapes. [Wikb]

Curves defined by polynomials have a clear advantage over curves defined by functions as polynomials can be multivalued with respect to any dimension. For example the curve shown in figure 2.3 could never be defined by a function.

Cubic polynomials are on the form

$$x(t) = \sum_{i=0}^{n} a_i * t^i$$

Figure 2.2: Spline device used by shipbuilders and draftsmen to draw smooth shapes. [oV04]

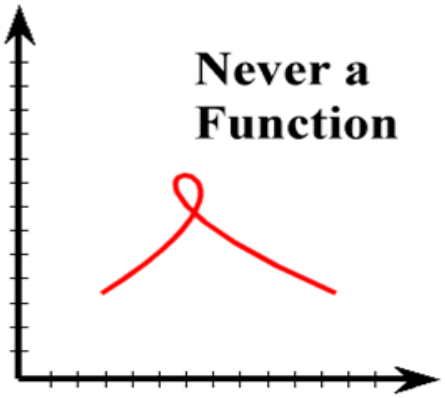


Figure 2.3: Example of a spline that could never be defined by a one dimensional function.

where t is limited to the range [0,1] and n defines the degree of the polynomial. Each polynomial is multiplied by a constant a_i called a control point. Any point on the curve is then determined by the polynomials, control points and the value of t. In three-dimensional space y(t) and z(t) have similar functions for defining a point.

2.2.2 The Hermite spline

A cubic Hermite spline, named in honor of Charles Hermite, is a third-degree spline with each polynomial of the spline in Hermite form. The Hermite form consists of two control points and two control tangents on each for each polynomial. On each subinterval, given a starting point p0 and an ending point p1 with starting tangent m0 and ending tangent m1, the polynomial can be defined by

$$p(t) = (2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)m_0 + (-2t^3 + 3t^2)p_1 + (t^3 - t^2)m_1$$

where

$$t \in [0, 1]$$

The four Hermite basis functions can be defined as blending functions between the control points

$$\begin{aligned} H_0(t) &= 2t^3 - 3t^2 + 1 \\ H_1(t) &= t^3 - 2t^2 + t \\ H_2(t) &= -2t^3 + 3t^2 \\ H_3(t) &= t^3 - t^2 \end{aligned}$$

to give the polynomial as

$$p(t) = H_0(t)p_0 + H_1(t)m_0 + H_2(t)p_1 + H_3(t)m_1$$

2.2.3 Generalized cylinders

Agin [Agi72] defines a generalized cylinder as a space curve (a class of curves including the spline) and a cross sectional contour perpendicular to the curve. As J. Bloomenthal [Blo85] suggests, the "surface of a tree limb, then, may be considered a generalized cylinder with a circular cross section ("disk") of varying radii", as illustrated in figure 2.5.

2.2.4 Framing a curve

When you want a shape to follow a curve, the shape has to be rotated and moved in order to follow the direction and orientation of the curve. Another way to see this is when the shape is moved to a position along the curve, the shape exists in its own local coordinate system, which is the original system in which it is defined, and in the coordinate system for the entire object. In order to represent the shape in this latter coordinate system, axial information regarding the shapes local coordinate system has to be calculated for every

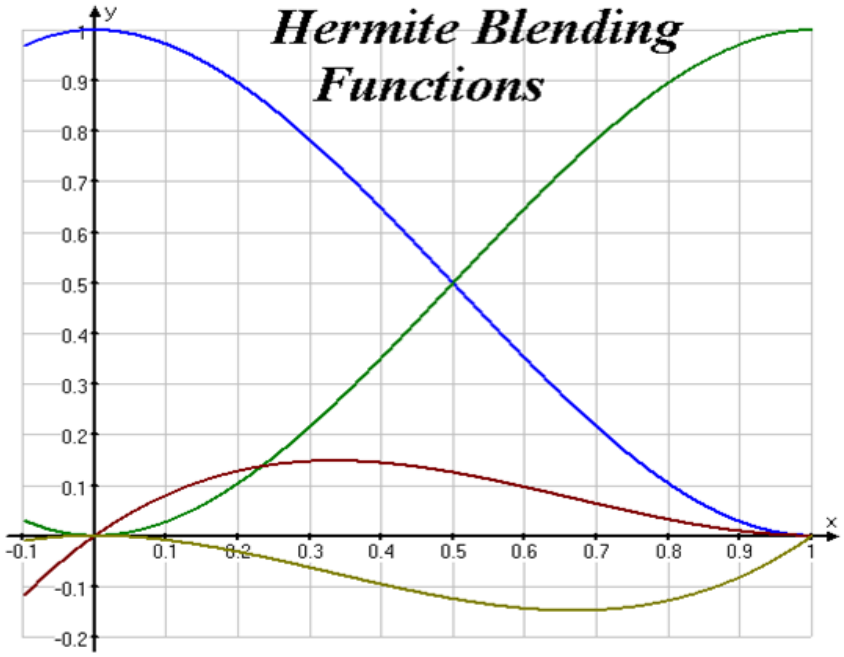


Figure 2.4: Hermite blending functions.

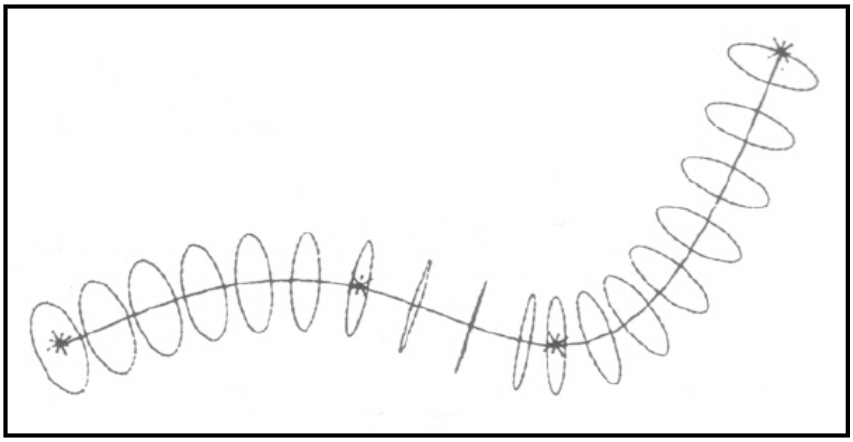


Figure 2.5: Several spline segments interpolate the data points (asterisks) with C2 continuity. A strobe captures a disk as it passes along the curve. [Blo85]

step along the curve. This is called framing the curve. The technique implies creating a reference frame at each sampling point along the curve, represented by three orthogonal vectors that define the position and orientation along the central axis of the cylinder [Blo90].

The Frenet frame is one of the more intuitive reference frames. It consists of the three vectors; a tangent to the curve, T, a principal normal, N, and a binormal, B. All these vectors can be computed analytically based on a three-dimensional cubic curve, like the Hermite curve. T is simply the velocity vector, i.e. the derivative of the curve. N is often computed as

$$N = (V \times Q \times V)/(|V \times Q \times V|)$$

where Q is the acceleration of the curve, i.e. the derivative of the velocity. The binormal B is then computed as $B = T \times N$.

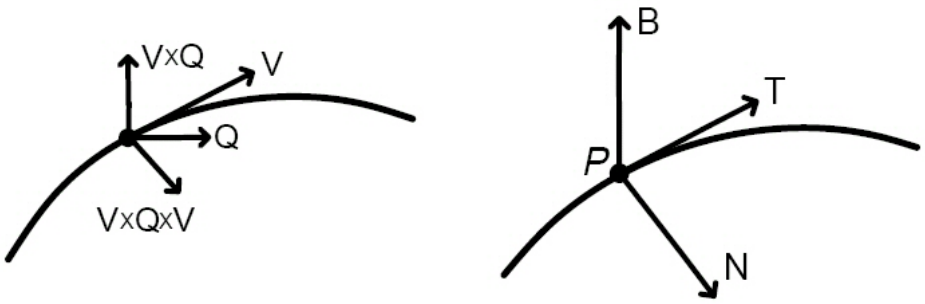


Figure 2.6: Curvature (left) and a Frenet frame (right). [Blo90]

2.3 Geometry instancing

When rendering large scenes consisting of many objects the bottleneck is often the drivers' throughput to the graphics card resulting in CPU overhead and poor performance. This happens because the Graphics API such as Direct3D and OpenGL are not designed to efficiently render a small number of polygons thousands of times per frame [Wlo04]. This is where the technique known as instancing becomes useful. Geometry Instancing takes advantage of the fact that many of the objects in the scene is similar, and therefore can be defined only once and then later referred to as instances of that object, only with different attributes. These attributes may be the model-to-world transformation matrix which places the instance in the scene, color information telling the fragment shader in which color to paint the object or small deforming information telling for example the vertex program to deform this instance slightly using a certain formula.

"A geometry packet is an abstract description of a piece of geometry, where the geometric entities are expressed in model space without any explicit reference to the context in which they will be rendered." [Car05] This packet may contain any kind of general data repre-

senting the object, but do not contain information describing any individual attributes. General data may be vertices, indices, bounding box a.s.o.

"A geometry instance is a geometry packet with the attributes specific to the instance." [Car05] In addition to contain a link to the general object of which it is an instance, it contains the additional attributes making it "unique" in the scene. This may be model-to-world matrix, color information a.s.o.

A geometry batch is a collection of geometry instances. The advantage of this comes obvious when such batching is supported in hardware, as it is by the GeForce 6 Series GPUs. This is called "Batching with Geometry Instancing API" and "offers a flexible and fast solution to geometry instancing." [Car05]

There exists several ways to upload the vertex data to the GPU. Static batching is the fastest, but least flexible way to instance geometry. The vertices are generated once during initialization of the program and stored in a vertex buffer and uploaded to the GPU. For every instance the whole or parts of this vertex buffer is drawn, setting the different attributes for each instance in between the draw-calls. Dynamic batching is slower, but more flexible as the vertex buffer is streamed to the GPU memory every frame. This makes it possible to modify the geometry information on the CPU before sending it, but may provide a transfer overhead if the data is large and/or slow to generate. A hybrid implementation exists called "Vertex constants instancing" where several instances are stored in a vertex buffer object (VBO), but with a different constant for each instance so the vertex program knows which instance the vertex belongs to, and can manipulate it according to the constant and other additional attributes stored in the GPU memory.

Geometry instancing is currently only supported by the Direct3D API (DirectX 9), but a fairly good substitute exists for OpenGL called Pseudo Instancing.

2.3.1 Pseudo Instancing

In Cg[2], external environment variables in the vertex and fragment programs are stored using the **uniform** type qualifier. It conveys that the variable's initial value comes from an environment that is external to the specified Cg program. When we render geometric instances using the static batching method we must provide each instance with its "unique" attributes through external variables. As with GLSL[3], setting these variables for each instance adds a lot of driver work since the driver must map abstract uniform variables into real physical hardware registers. Zelsnack [Zel04] states that constant updates on the hardware side can incur hardware flushes in the vertex processing engine. In OpenGL one can instead use persistent vertex attributes to store variables. Zelsnack says that these "API calls are very efficient on the driver side; they don't require validation or potentially

[2]Cg stands for "C for graphics" and is based on the popular C programming language. The Cg language makes it possible for you to control the shape, appearance, and motion of objects drawn using programmable graphics hardware. [FK03]

[3]The OpenGL Shading Language has been designed to allow application programmers to express the processing that occurs at programmable points of the OpenGL pipeline. [Ope04]

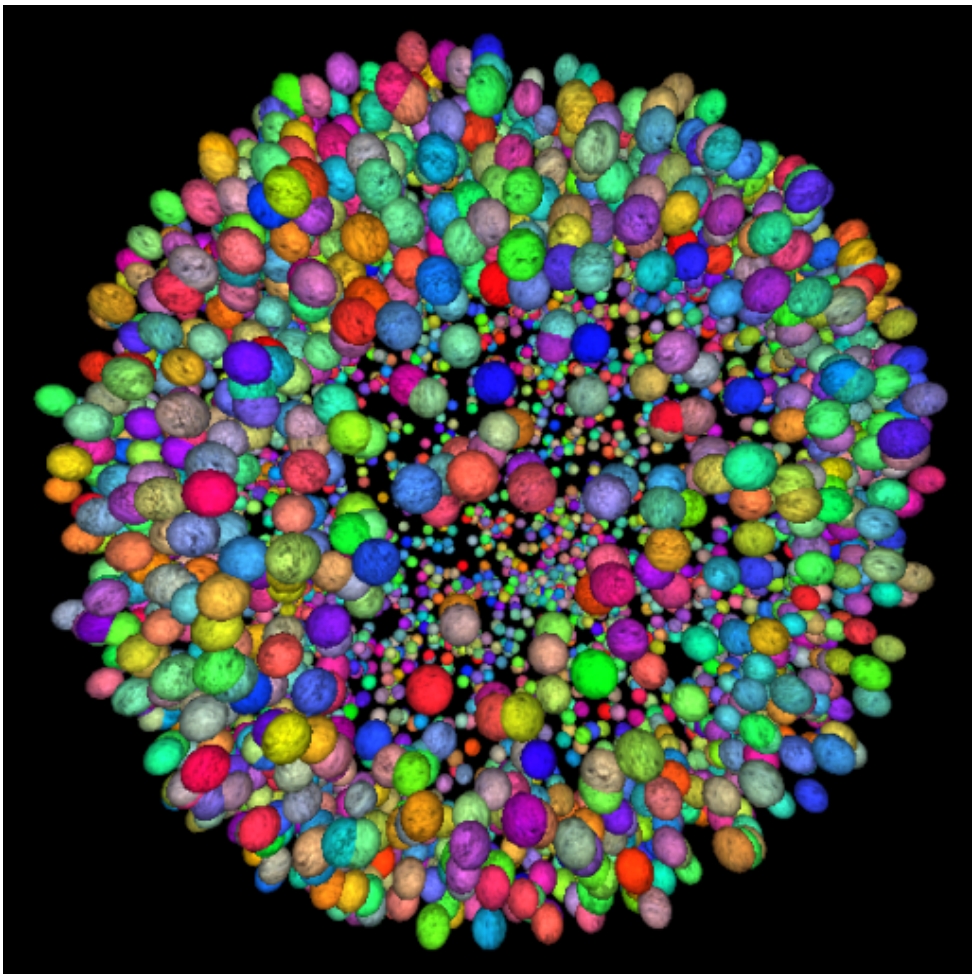


Figure 2.7: Pseudo instancing rendering each instance at an individual position and in an individual color. [NVI05]

complex remapping. They are also very efficient on the hardware side; they do not result in hardware flushes in the vertex processing engines."

One drawback however using this method is, as Zelsnack [Zel04] mentions, there are few such persistent vertex attributes and therefore not enough to store attributes for skinning as an example. The technique is not as good as the similar instancing technique supported by Shader Model 3.0 GPUs by Direct3D. "The major difference is that the Direct3D instancing API reduces the number of DrawIndexedPrimitive() calls from many to one. This DrawIndexedPrimitive() call reduction has a large performance benefit in Direct3D. In OpenGL, the application still calls glDrawElements() (or the like) for every instance. This isn't too much of a performance hit because glDrawElements() is very efficient in OpenGL." [Zel04] This method will however require more draw-calls and are more likely to produce CPU overhead when the number of instances increases.

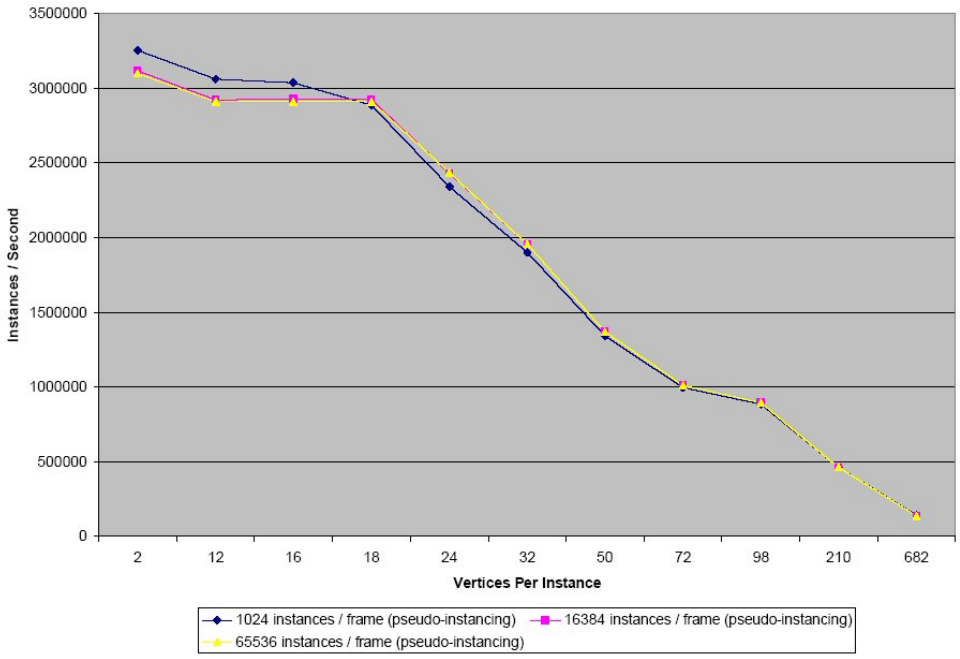


Figure 2.8: The GeForce 6800GT can hit over 3 million instances per second for tiny meshes. [Zel04]

As Zelsnack shows, the pseudo instancing technique works very well with instances consisting of a small number of vertices. This can have a clear advantage when drawing trees, as a single branch typically consists of a small number of vertices.

Figure 2.9: Purely geometric 2,000,000 triangle detail mesh in 3D Studio Max. [Tec04]

Figure 2.10: 5,287 triangle in-game mesh in 3D Studio Max. [Tec04]

2.4 Textures

2.4.1 Bump mapping

The main idea with bump mapping is making a flat surface look more detailed than it geometrically is. Bump mapping is a per-pixel (or per-texel) lightning calculation, calculating the light intensity based on the light model being used and the information in the given pixel on the texture (also often referred to as a texel). The great advantage lies in the speed of the light calculation compared to processing more geometrical detailed data.

There are many ways to represent bump mapping and normal mapping is one of the most popular ones.

2.4.2 Normal mapping

A detailed mesh might consist of several millions of triangles. This is not renderable in real-time. When reduced to a few thousand triangles it is renderable, but has lost a lot of detail (see figures 2.9 and 2.10). Instead we render a map of normals specifying only the change in the normal when the model was reduced to fewer polygons. When calculating the light for a texel on the model we use the normal map as a lookup table for adjusting the normal used in the calculation at that point (see figure 2.12).

2.4.3 MIP maps

A technique called MIP-mapping ('multum in parvo') is used to render many textures faster and more accurately at the same time. Instead of using one full-scaled texture

Figure 2.11: Resulting normal-mapped mesh in game.[Tec04]

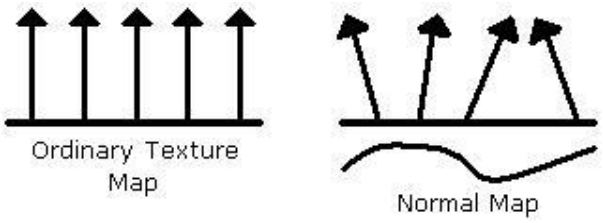


Figure 2.12: Normals on a plane surface (left). Normals adjusted according to the normal map (right). [Dre04]

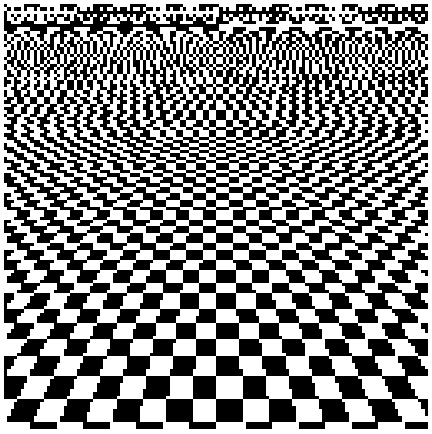


Figure 2.13: Aliased image [Wika]

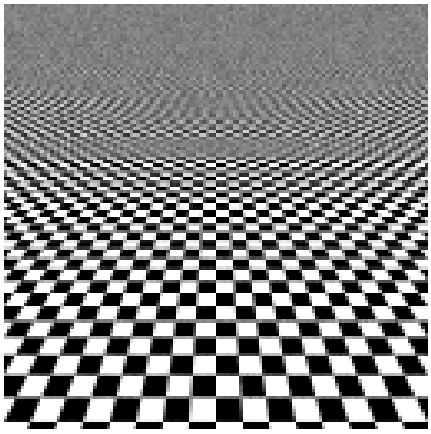


Figure 2.14: Anti-aliased image [Wika]

at all times, MIP-mapping makes a full-size version, a half-size version, a quarter-size version, etc, and uses the correct one corresponding to the distance from the observer to the object being rendered. This technique was developed as the result of having to put many textures out on the limited graphics card memory at the same time when rendering a scene. Since not all textures have to be rendered with the same amount of detail at the same time, some textures can be rendered with lower detail. For example when a object is far away from the viewer, the texture does not need to have the same level of detail as if the object was rendered up-close. Another purpose of MIP-maps is the undesired effect of aliasing when rendering a scaled texture. If we look at a chess board in perspective we get this undesired effect where the resolution is too low to render the detail of the texture.

Figure 2.13 illustrates the visual distortion which occurs when anti-aliasing is not used. Notice that near the top of the image, where the checkerboard is very distant, the image is impossible to recognize, and is displeasing to the eye. By contrast, figure 2.14 is anti-aliased. The checkerboard near the top blends into gray, which is usually the desired effect when the resolution is insufficient to show the detail. Even near the bottom of the image, the edges appear much smoother in the anti-aliased image. [Wika]

Chapter 3

Approach and implementation

This chapter will describe the approach taken for implementing the tree rendering system.

3.1 Creating a tree

Fully constructing a tree can be divided into two stages; generating the tree and rendering the tree. First we have to do the necessary calculations for individualizing each tree, like how many branches the tree has, how long each branch should be, where they will be placed along their parent branch a.s.o. When all this is defined, we can start rendering the tree by defining the vertices, faces and textures for the tree.

The tree generating system implemented is a parametric system based on some of the parameters proposed by Weber and Penn in *Creation and Rendering of Realistic Trees* [JW95]. This section describes the most important features for my approach on how a tree is constructed based on the parameters from Weber and Penn.

3.1.1 Generating a tree

The process of generating a tree is based on a set of calculations which can be divided into two separate groups. Some calculations only have to be calculated once for each tree, while others, depending on possible external changes, may vary during interaction with and execution of the application. Many parameters are fixed for a certain tree and only depend on the parameters defining the tree type. Parameters such as the start radius per stem ($radius_{stem}$), length of a stem ($length_{stem}$)[1] and number of branches ($stems$) are all examples of parameters calculated once when the application initializes the forest of trees. The article by Weber and Penn does not really specify the number of branches to spawn from the trunk. The decision fell on the intuitive formula:

$$stems_{trunk} = 0Branches \pm 0BranchesV$$

[1] either $length_{trunk}$ or $length_{child}$ depending on whether it's a trunk or branch

For level 1 and further I use the formulas presented in *Creation and Rendering of Realistic Trees* and quoted in section 2.1.1.

The second group of parameters are those calculated when external forces make an impact on the structure of the tree. These represent the placement and orientation of the tree and hence all its branches. Parameters which effect these transformations are variables such as downangle ($down_n$) and curvature ($nCurve$ and $nCurveBack$). These variables can for example be altered to simulate wind (see section 3.8), and therefore must be recalculated per frame (or as often as desired for the animation).

The skeleton for each tree in this system is represented by a Hermite spline (see section 3.2.1). Each spline is defined by 4 control points/vectors controlling the start tangent, start point, end point and end tangent of the spline. These control points are calculated somewhat differently for the trunk and the branches of the tree.

Generating the trunk

The start tangent for the trunk is calculated using the formula:

$$tangent_{start} = root_{tangent} * length_{stem}$$

where $root_{tangent}$ is the normal given as an input vector to the tree, identifying in what direction the tree should start growing. This feature will be explained in more detail in section 3.6. The vector is multiplied with the length of the trunk to get a significant effect.

The start point for the trunk is given by the position of the tree in world space coordinates. All coordinates for the control points are given in world space coordinates. The alternative would be to specify the coordinates in object space (meaning the local coordinate system for each branch) and sending the transformation matrix world-to-object to the vertex shader. The latter alternative has not been tested, but the reason for transforming them on the CPU is that we save one matrix operation per vertex on the GPU, which becomes significant when rendering many trees. We also save some time in the fragment shader, not having to transform both the eye position and the light vector into object space since we already have the matrix for transforming the tangent space normal into world space (see section 3.5.2).

The end point of the trunk spline is simply calculated using the formula:

$$position_{end} = [\sin(nCurve), \cos(nCurve), 0] * length_{stem} + tree_{position}$$

The end tangent is a bit more sophisticated since it has to take into account the $nCurveBack$ parameter. The formula adds the vector from the $nCurve$ parameter with a vector calculated from $nCurveBack$ and ends up with a scaled middle vector:

$$\begin{aligned} tangent_{end} &= Normalize([\sin(nCurve), \cos(nCurve), 0] + \\ &\quad [\sin(nCurveBack), \cos(nCurveBack), 0]) * length_{stem} \end{aligned}$$

A tree with $nCurve$ set to 50 and $nCurveBack$ set to 10 would look like the tree in figure 3.1.

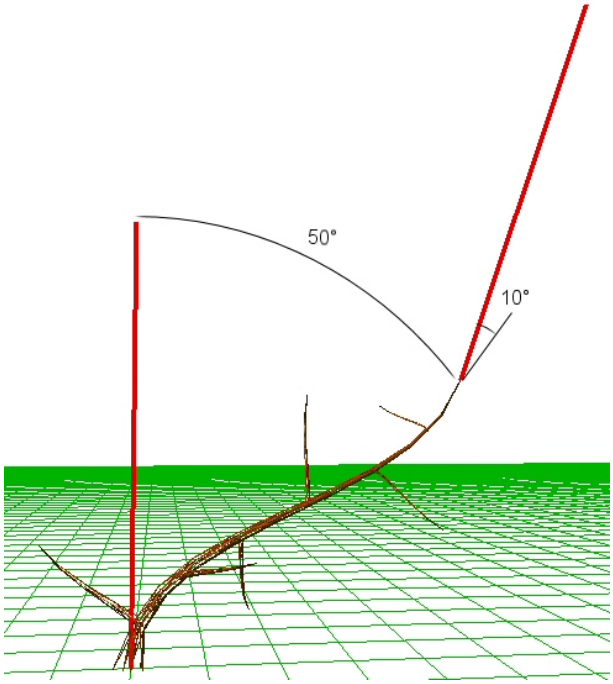


Figure 3.1: $nCurve = 50$, $nCurveBack = 10$

Generating the branches

The branches are generated recursively by spawning new children from their parent stem. The branch start tangent and position are calculated given the parametric (t) position at the parent stem where the child is spawned and the control points for the parent stem. Since the start tangent is set equal to its parent tangent at the position it is spawned, the new branch smoothly emerges out of the parent stem (see figure 3.2).

$$tangent_{start} = parent_{hermite\_tangent} * length_{stem}$$

To be able to generate the end point and tangent for a branch a orientation matrix has to be calculated. First a rotation matrix is generated in order to align the tangent-axis[2] to the Hermite tangent at position t at its parent stem. This matrix is then multiplied by a new rotation matrix rotating the branch around the parent stem (using $nRotate \pm nRotateV$) to the right angle where the stem should be placed (see figure 3.3).

The same matrix is then further multiplied by a rotation matrix lowering the branch so that it grows out of its parent stem at a certain angle (referred to as downangle in the article by Weber and Penn) as in figure 3.4.

When the orientation matrix is constructed the end tangent and position points are cal-

[2]In *Creation and Rendering of Realistic Trees* this axis is the z-axis following the skeleton of the tree. In this implementation I've chosen it to be the y-axis since most simulators use y as up in their virtual world.

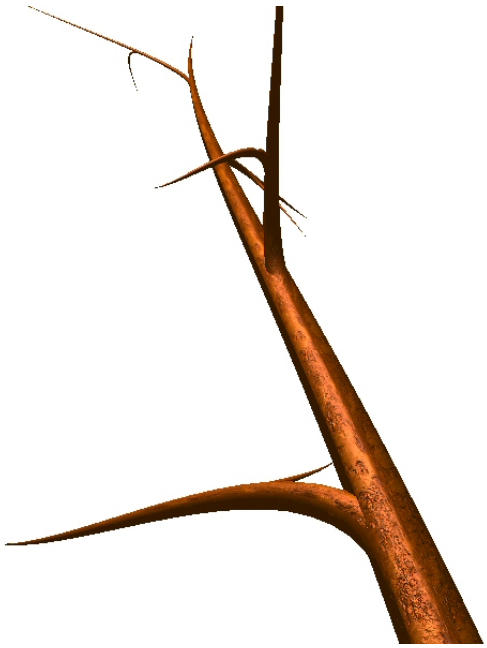


Figure 3.2: Branches emerging smoothly from their parent stem.

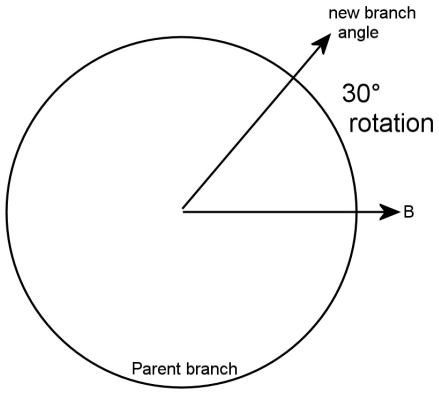


Figure 3.3: A branch is rotated by a specific angle from its initial position defined by the generated binormal (B) from section 3.2.2. Circle illustrates parent branch seen from above.

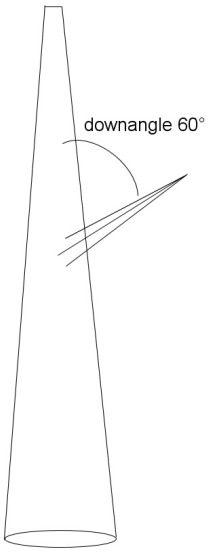


Figure 3.4: Showing a branch lowered from its parent stem by downangle.

culated using the formulas:

$$position_{end} = M_{orientation} * [[\sin(nCurve), \cos(nCurve), 0] * length_{stem} + parent_{hermite\_position}]$$

$$\begin{aligned} tangent_{end} &= M_{orientation} * Normalize([\sin(nCurve), \cos(nCurve), 0] + \\ &\quad [\sin(nCurveBack), \cos(nCurveBack), 0]) * length_{stem} \end{aligned}$$

This rotation matrix is then passed on to the next level of branches to be spawned from this particular stem.

Parameters not implemented

Creation and Rendering of Realistic Trees speaks about a special mode for the $nCurveV$ variable. When this variable is negative, the stem is formed as a helix. This is fairly easy to implement when using the technique proposed by Weber and Penn, as each segment along the stem is rotated individually. This is however not possible when the shape of the stem is determined purely by a single Hermite spline curve defined by the implemented approach. This feature would have to be implemented in Cg as an additional feature to the vertex program, creating an additional Frenet frame following the helix and combining this frame of reference with the frame following the spline curve.

Stem splits, leafs, pruning and vertical attraction are parts of *Creation and Rendering of Realistic Trees* which have not been considered in this project.

3.1.2 Rendering a tree

When drawing the tree, each stem is responsible for drawing itself and calling the draw routine for all its attached child stems. Each draw routine is responsible for uploading the control points and other information it should specify for drawing that particular branch or trunk to the GPU program. These parameters are precalculated variables like $radius_{stem}$, $length_{stem}$, $nTaper$, $Lobes$, $LobeDepth$, $Flare$ from [JW95] and the safe vector[3]. The draw routine then sends a draw call with the level of detail in which the stem should be rendered, which links to an offset in a vertex buffer object (VBO) for this particular stem. The GPU then calculates the position for every vertex based on calculations of the parameters $radius_Z$, $flare_Z$ and $lobe_Z$ from [JW95]. This will be explained in more detail in section 3.3.

3.2 Geometric representation

This section describes the approach taken for constructing a geometric representation of the splines defining the tree.

[3] explained in section 3.2.2

3.2.1 Using the Hermite spline

When defining the skeleton of a natural form, many types of splines could be used. Representing the tree skeleton as straight lines would look unnatural and Bloomenthal stresses the importance of the curve having "C2 continuity" meaning that the curve have continuous second derivative. [Blo85] This becomes useful when constructing the limbs surface and will be explained in more detail later.

Figure 3.5: Data points interpolated by straight lines (left) and by splines (right). [Blo85]

The most important feature is however that the curve is C1 continuous, meaning that the curve has a continuous tangent defined along the entire curve so that multiple curves can be merged at any position while keeping the basic shape smooth.

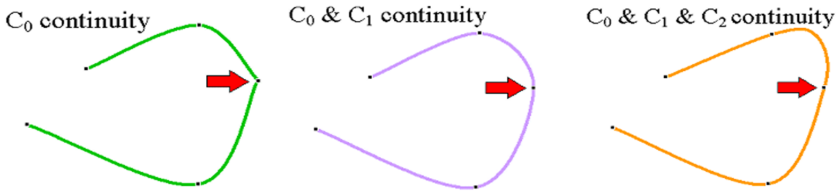


Figure 3.6: Curves that are C0, C1, C2 continuous. [oE04]

The reason for choosing the Hermite spline over splines like the Bezier spline is the simple and intuitive control over the tangents at the end points. When for example a new branch is spawned at some position along its parent stem, the parents' tangent can easily be calculated at this given position representing the start tangent for the new curve. The

two branches will then have C1 continuity and the transition will look natural as in figure 3.2.

3.2.2 Framing the curve

One important feature of framing a curve is the twisting of the reference frame along the curve. The orientation of the shape that is swept along the curve is determined by the normal and binormal. Some natural forms look more natural if twisted to some degree and others do not. Two problems with the standard method of calculating the Frenet frame arise. Wherever the curvature vanishes, such as at points of inflection or along straight sections of the curve, the double derivative is zero and hence the normal and binormal are undefined. Also, on either side of an inflection point the curvature vector can reverse direction, inflicting a violent twist in a progression of Frenet frames (see figure 3.8). [Blo90]

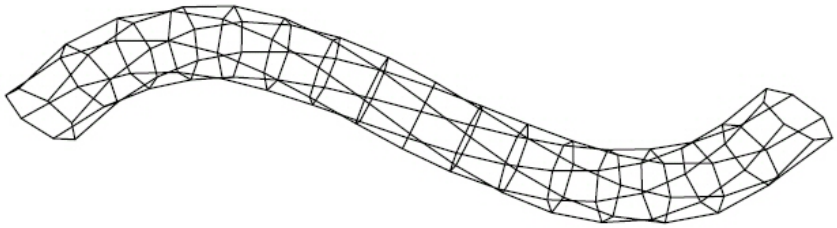


Figure 3.7: Polygons resulting from twisting reference frames. [Blo90]

A C2 continuous curve would solve the problem of the reversed curvature at inflection points, but would not solve the problem of the binormal not being defined wherever the acceleration is zero. As R. L. Bishop suggests, there is more than one way to frame a curve [Bis75]. In order to minimize the twisting of sequential disks Bishop introduces the idea of minimizing the rotation of sequential disks along the curve. The frame is defined by first picking a single unit vector $B_1(0)$ which is orthogonal to the unit tangent vector T(0), and then extending B_1 along the curve by the rule that $B_1'(t)$ is a linear combination of $B_1(t)$ and T(t). The second vector in the frame if the tangent plane is then simply defined by $B_2(t) = T(t) \times B_1(t)$. The next frame is then defined by applying a rotation which rotates the previous edge direction into the next edge direction by a rotation about an axis determined by their cross-product. [OKR04]

A clear constraint to this way of framing a curve is that this method does not work analytically, meaning it can not be calculated for any arbitrary point along the curve without having to calculate the propagating vectors from zero to that point.

One way to solve both these problems while keeping the analytical property of the frame is to use a so called ”safe vector” when generating the normal vector. The safe vector would always have to satisfy the rule $S \neq T$, meaning that the safe vector could never be parallel to the tangent vector. As long as we can prevent this, the normal vector can

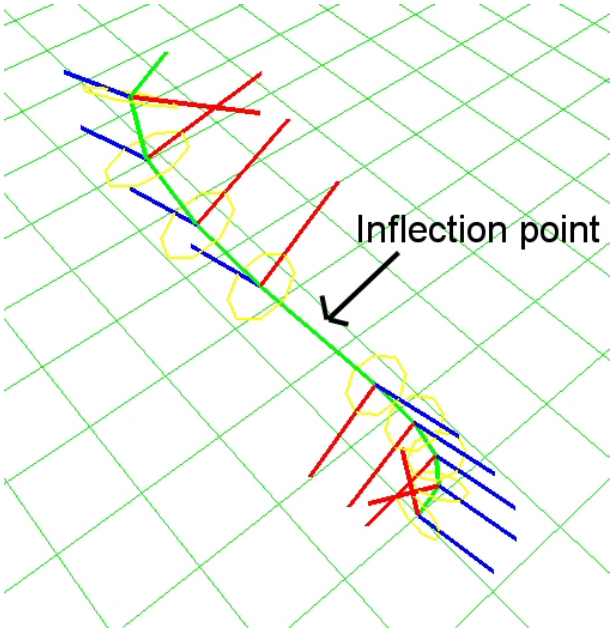


Figure 3.8: Reversed curvature vector at inflection point

be calculated by the cross product of the safe vector and the tangent generating a vector orthogonal to the tangent vector of minimal twist from the safe vector.

$$B(t) = T(t) \times safeVector$$

When generating the branches for the tree we can assume that the curve representing the branch will be rotated within a plane, so choosing a vector orthogonal to this plane would be safe since we can guaranty that the tangent will never be parallel to this vector (see figure 3.9).

One problem with this solution is however if we were to implement wind blowing from any angle. If the wind was strong enough to bend a branch so that its tangent coincide with the safe vector, we would be in trouble. One way to solve this would be to generate two safe vectors, one at the start of the branch and one at the end, and then in the vertex shader interpolate between these two vectors depending on where we are on the stem. This would remove the limitation of having to draw the branches in a plane. As an advanced wind model has not been implemented, this was not needed.

3.2.3 Defining the circle

Since its impossible to model a completely smooth circle using polygons, the circle which defines the outer boundary of each stem is defined by a certain number of vertices placed a certain distance from the center of the stem. The circle is constructed using the coor-

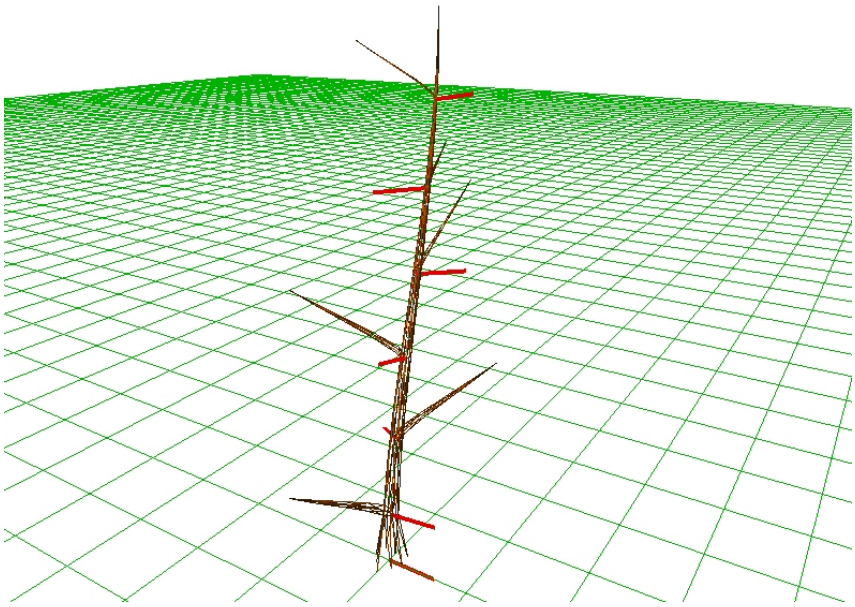


Figure 3.9: Tree skeleton with safe vectors for each branch

dinates of the Frenet frame plane using the following formulas:

$$p(x) = cos(rad)$$
$$p(y) = 0$$
$$p(z) = sin(rad)$$

where

$$rad \in [0, 2\pi\rangle$$

This circle is then sampled at equal intervals along the curve (see figure 3.10) and positioned using the current Frenet frame (see figure 3.11).

3.3 Geometry instancing

This project has implemented static batching as this is the most straight forward and intuitive method to do instancing, and also the fastest method supported by OpenGL. Instance attributes are stored in persistent vertex attributes as texture coordinates and uploaded from the client program to the GPU using the pseudo instancing technique. The per vertex parameters for each instance is stored as texture coordinates, vertex position information and vertex color information. The object properties for circulating the stem from 0 to $2 * \pi$ is stored in the x parameter in the texture coordinates, while the t value specifying at what position we are along the spline is stored in the y value (figure 3.12).

As an optimization, the blending functions for the Hermite curve can be stored as fixed values since we already know the t value. The only values we need are the ones for the

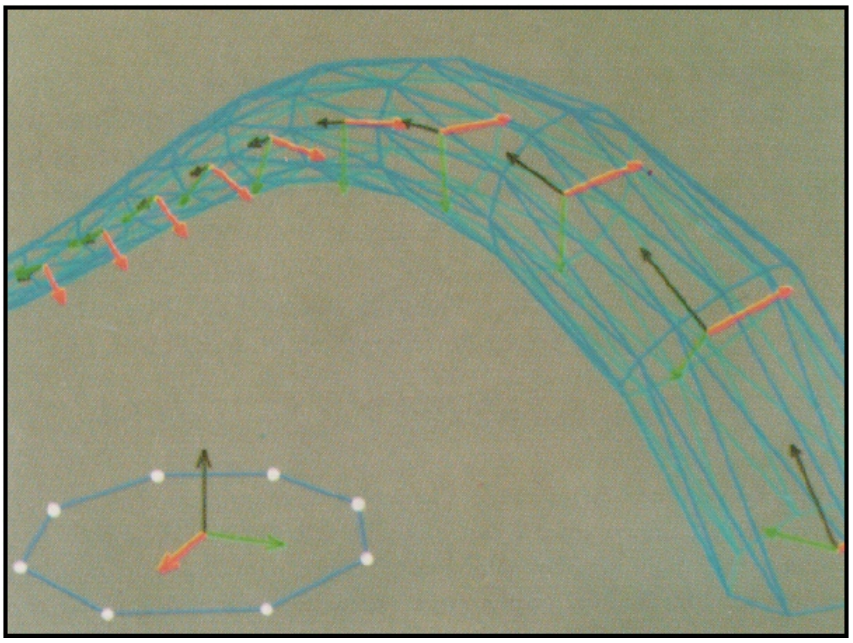


Figure 3.10: The Frenet frame at perimetrically equal distances along the curve [Blo85]

current value of Hermite and its derivative (figure 3.13 and 3.14). These are stored as float4[4] Cg parameter values (x,y,z,w) as vertex position and color information for each vertex.

The geometric packet sent to the GPU then looks like this:

Binding Semantics Name	Corresponding Data
TEXCOORD0	Per vertex parameters t and rad
POSITION	Hermite blending function
COLOR	Hermite derivative blending function

3.4 Optimizations

When rendering a scene consisting of several million vertices, not having to draw them all usually speeds up the frame rate. Storing multiple versions of the same geometry is often used since the objects do not have to be drawn in full detail in all circumstances. This kind of optimization if called level of detail (LOD).

[4]float4 is a predefined vector data type provided by the Cg Standard Library. This is a packed array which tells the compiler to allocate the elements of packed arrays so that vector operations on these variables are most efficient. If two input vectors are stored in packed form, programmable graphics hardware typically performs three-component or four-component math operations - such as multiplications, additions, and dot products - in a single instruction. [FK03]

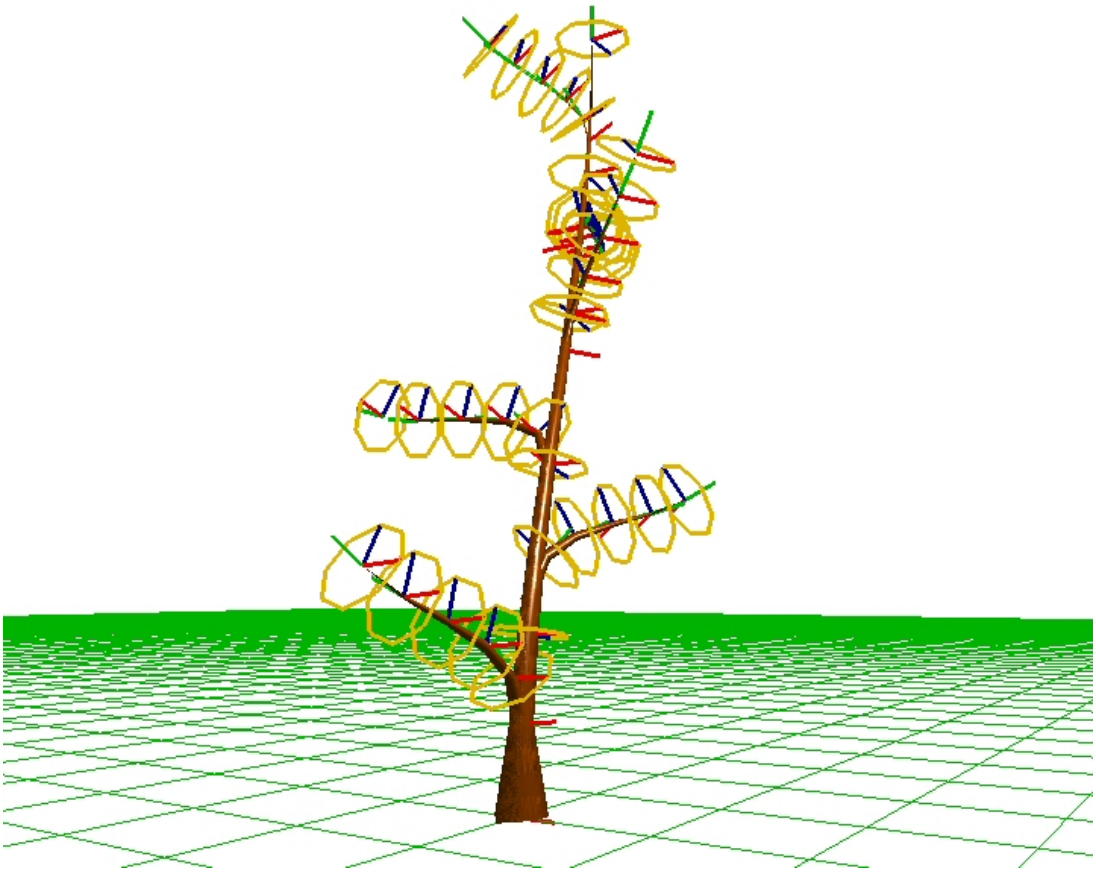


Figure 3.11: Frenet frames sampled along branches of a tree.

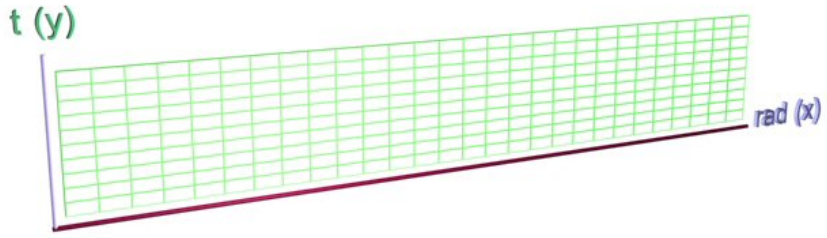


Figure 3.12: The t and radians values stored as texture coordinates per vertex

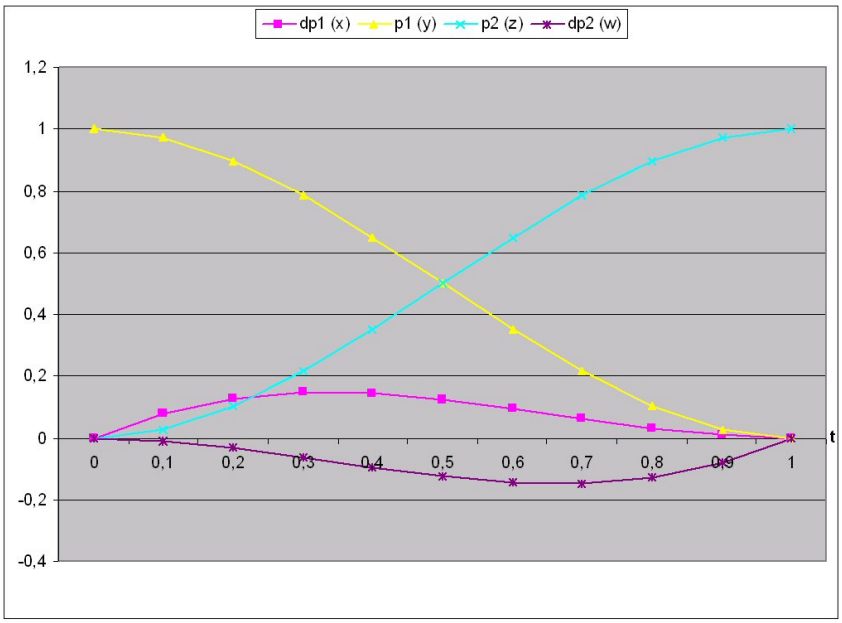


Figure 3.13: The original Hermite blending function stored as float4 position values per vertex

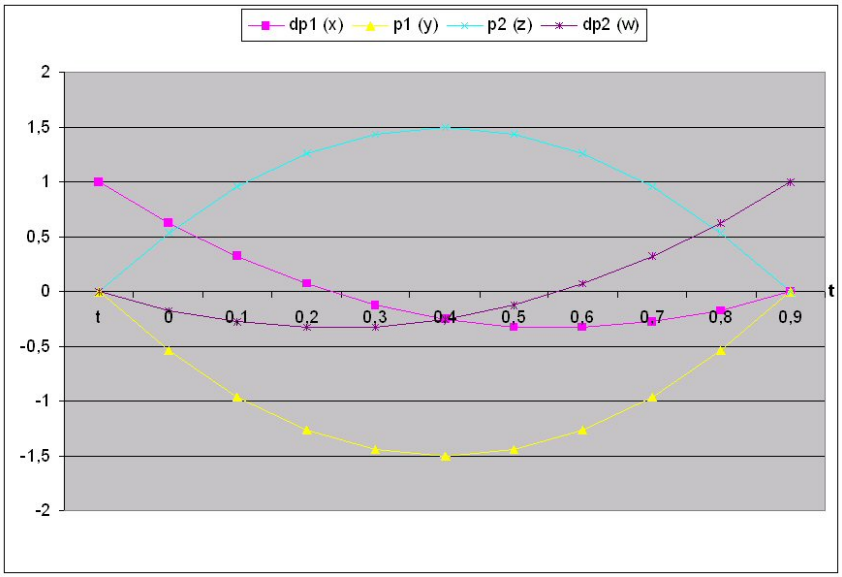


Figure 3.14: The derivative of the Hermite blending function stored as float4 color values per vertex

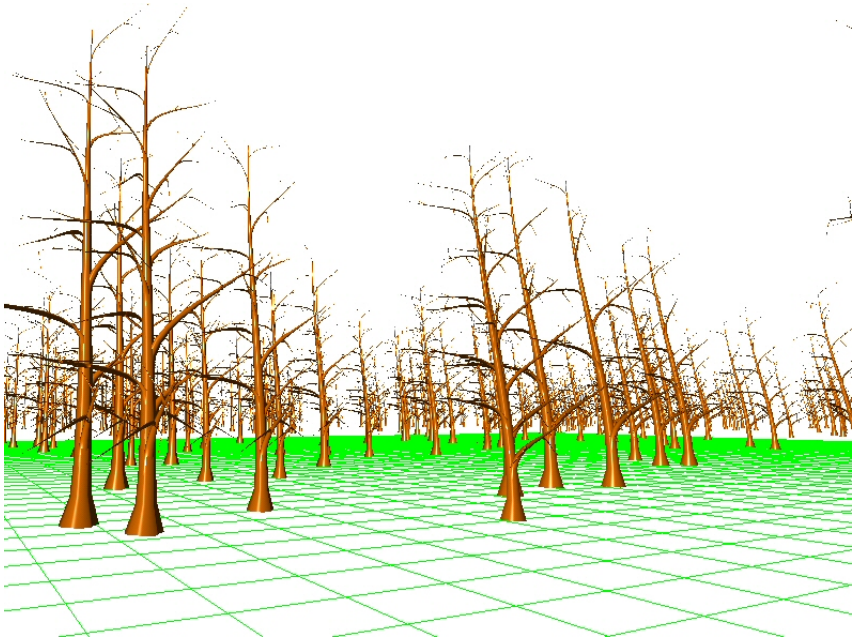


Figure 3.15: Branches rendered as instances of a general cylinder

3.4.1 Instance LOD

Normally a tree with many recursive branches, the outer branches are smaller and shorter than their parent branch. To represent their form we do not need as many vertices or faces to make them look good. Therefore we can distinguish between different levels of recursions when drawing the individual branches of a tree. The results show that this leads to faster rendering since the scene contains fewer vertices per tree and in total. To achieve this we store multiple versions of a standard stem in the vertex buffer, and the offset for each version in an array for later lookup.

The final VBO looks something like this

Instance level 0			
Instance level 1	Level 2	Level 3	...

The number of levels in the VBO is defined by the constant MAX_RECURSIONS at compile time. Each instance level contains enough vertices to be able to generate a stem (meaning at least 1 segment and 3 points in a circle) and each vertex is made out of the geometrical package described earlier. Both the number of segments per stem and points defining the circle of the stem is divided by 2 for every new level. This means that instances of level 0 consist of 4 times as many vertices as stems on level 1.

As a standard, each stem connected to a tree contains an identifier telling it at what level of recursion this stem is at. The trunk would be at recursion level 0 and all its child stems at level 1 a.s.o. When rendering an instance, the offset for this recursion level is fetched from an array and used as an offset for the VBO in the draw call.

3.4.2 Distance LOD

To make even better use of the concept of level of detail, I implemented a simple algorithm based on the distance from the observer to the tree. This distance is set as a parameter for the tree and taken into consideration when choosing the level of detail when rendering an instance.

The level of detail for a current tree is calculated using the formula

$$LOD = (integer)(distance^2)/(25^2)$$

where distance is the distance between the camera and the tree. When rendering a branch belonging to a tree, this LOD is added to the recursion level of the branch. The result is that all branches at recursion level 1 for a tree which LOD variable is calculated to be 2 are rendered using instance level 3 from the VBO.

The result is that every tree within a 25 unit lengths[5] radius is rendered in the highest level of detail (level 0) and ending at the lowest level of detail[6] at 50 unit lengths from the observer. These two lengths was chosen based on looking at the trees in my implementation and choosing the closest distance (25 units) from where the transition from one level to another was hardly noticeable. This implementation uses a screen resolution of 800x600. Rendering at a higher resolution would mean that the transition would be noticeable at a longer distance and would have to be redefined.

3.4.3 MIP Maps

Using MIP Maps is another way to speed up performance using different levels of detail. This project uses MIP-mapping (see section 2.4.3) to render the textures for the different stems. A typical MIP-map texture for the bark on a tree would look like in figure 3.16. The MIP-maps are precalculated and stored in DDS-files[7].

3.4.4 View-frustum culling

View-frustum culling is a way of optimizing rendering when not all objects in a scene is visible at all times. This technique consists of only drawing objects completely or partly visible to the user. Although this is usually part of any graphics package, this has not been implemented in this project to be able to run consistent benchmarking. Since we draw all objects at all times, what we are looking at in the simulator has little or no effect on the performance of the vertex shader. However if we look at an object close-up, a lot

[5] In *Creation and Rendering of Realistic Trees* and here measured in meters.

[6] The lowest level of detail is determined by the constant MAX_RECURSIONS and is set at compile time. For this compile it is set to 4.

[7] The Microsoft® DirectDraw® Surface (.dds) file format is used to store textures and cubic environment maps, both with and without mipmap levels. This format can store uncompressed and compressed pixel formats, and is the preferred file format for storing DXTn compressed data. [MSD05]

Figure 3.16: MIP-map of bark texture

more work is done in the fragment shader occupying the processor with texture and light calculations. To prevent any significant differences due to work done by the fragment shader, all benchmarks are done looking at all the trees at the same time from a distance and picking the lowest frame rate achieved.

3.5 Textures

3.5.1 Placing the texture patches

To make the texture of the bark more natural, the scaling for each individual stem has to be calculated and each vertex mapped to a set of texture coordinates. This mapping depends on the radius and length of the stem, and is calculated per vertex on the GPU as follows:

$$tex(x) = \frac{rad}{2*\pi*\lfloor radi+1 \rfloor}$$

$$tex(y) = t * \frac{length_{stem}}{2}$$

where rad is the property for circulating the stem, t the property specifying at what position we are on the curve and $radi$ the radius of the stem at this point on the curve calculated based on $lobe$, $flare$ and $radius_Z$ parameters from the article by Weber and Penn [JW95].

3.5.2 Normal mapping

The normal information in a normal map is for practical reasons often stored in a texture as RGB values. This leads to easier implementation since ordinary image formats are already supported by most APIs (like OpenGL). The change in the normal is stored as three vectors, X, Y and Z represented by the corresponding values of R, G and B in the texture.

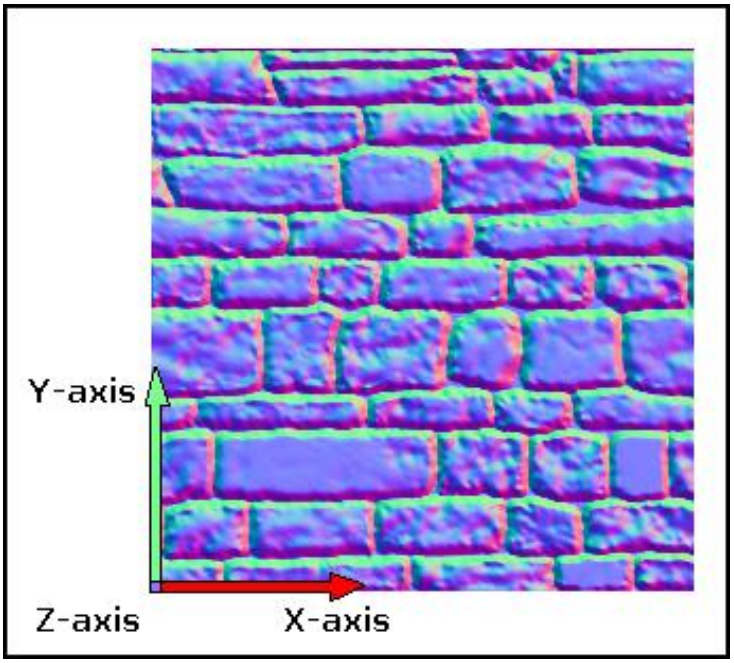


Figure 3.17: Illustration of a normal map texture.[Dre04]

Since the direction of a normal is defined in the range $[-1, 1]$ while the RGB values are defined in the range $[0, 1]$, the color value must be expanded to the range of the normal vectors by the formula:

$$vector = 2.0 * (colorValue - 0.5)$$

Since the normal vectors are defined in tangent space[8], meaning they are defined in the space represented by the face being rendered, the normal to be modified must exist in the same space. Normally vertices exist in either world- or object-space. For calculating Phong shading we need the position of the light, eye position and the normal. Two alternatives exists; one is to transform the light- and eye- coordinates to tangent space; the other alternative is to transform the normal map coordinates to world- or object-space.

[8]The tangent space of differential manifold M at a point $x \in M$ is the vector space whose elements are velocities of trajectories that pass through x. The standard notation for the tangent space of M at the point x is T_xM. [Pla05]

This all depends on what is more convenient. For this project the last method is the most effective since we already have the vectors representing tangent space (vectors T, N and B from section 2.2.4) and hence the matrix to convert any vector in tangent space to world space. All we have to do is construct the tangent-to-world matrix in the fragment shader based on the three interpolated vectors from the vertex shader and multiply the normal fetched from the normal map with this matrix. We can then use this new normal vector to calculate the Phong shading (since the light vector and eye position is given in world coordinates). Code for these calculations are rendered in appendix A.2.

3.6 Landscape

To be able to see the trees in a more natural environment, I added a landscape to the scene and placed the trees in the terrain. The implementation of the landscape is from a Height Map[9] 3 tutorial written by Ben Humphrey at GameTutorials.com [Hum02]. The tutorial shows how to effectively render a Height Map using VBOs and multitexturing[10] for adding detail to the ground. Although the landscape is implemented as a static VBO, it does have an impact on the frame rate when rendering the scene. However the vertices making up the landscape are not processed by the implemented vertex shader, so no calculations are done for these vertices. The landscape is not included in the benchmark tests in the next chapter, except when testing wind.

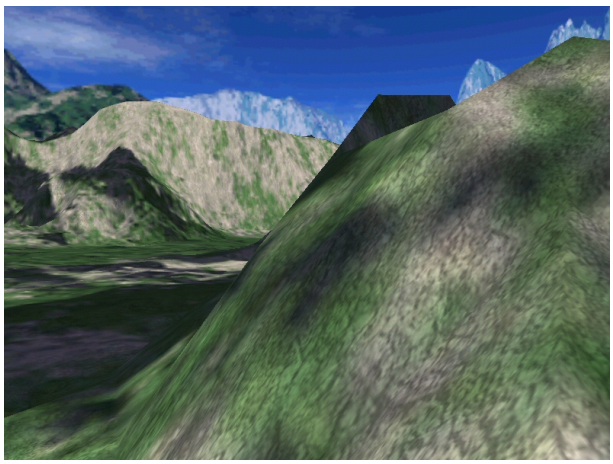


Figure 3.18: Landscape rendered with the Height Map 3 Tutorial. [Hum02]

[9] A Height Map is height data stored in a certain format where the values represent the height for consecutive points in a two dimensional plane.

[10] Multitexturing means that you are displaying several textures at the same time on triangles. The drawback of having to display several textures on the same triangle(s) using many passes is that the graphic card will transform many time the same triangles and vertices, whereas it can transform them only once and apply many texture on it at the same time using multitexturing, which is obviously faster. [Pd05]

To make the trees appear to grow naturally out of the ground their are standing on, the first tangent vector for the trunk of each tree is parallel to the normal at the point where the tree is placed. This makes the first disc of the trunk lie in the same plane as the terrain and slowly rotate towards the direction of the end tangent as shown in figure 3.20 and illustrated in figure 3.19.

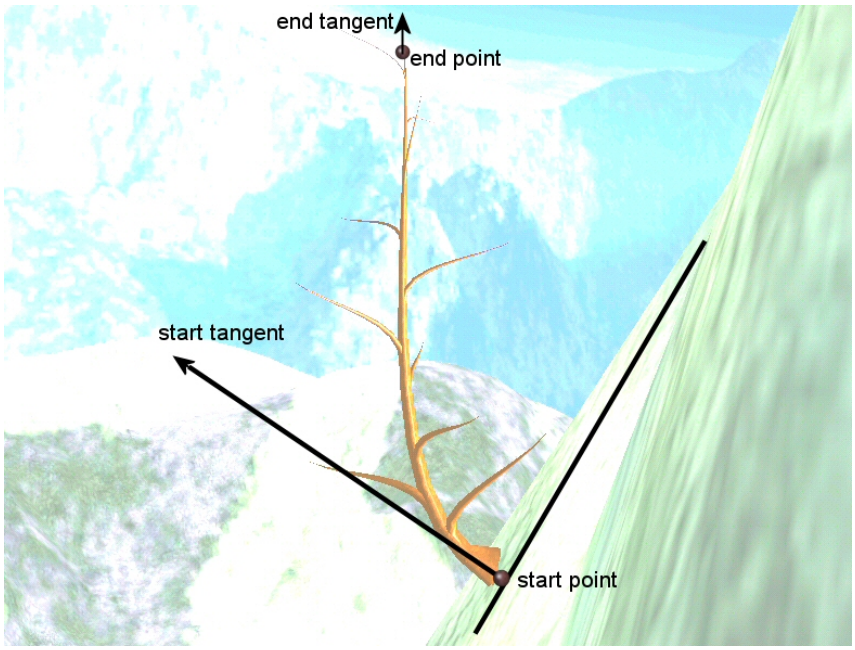


Figure 3.19: Tree growing out from a steep hillside with control points.

3.7 Different tree types

Using the parameters from *Creation and Rendering of Realistic Trees* its pretty easy to construct individual trees based on a variety of different tree types. By defining a set of parameters for a tree type, parameters such as $length_{stem}$, $radius_{stem}$ $nTaper$, $nCurve$ and $nCurveBack$ will be individual features for each tree. In addition we can define many sets of parameters to create completely different tree types. The application implemented supports such parameter sets as input to the program. The files containing tree type information must be stored in the Microsoft Windows .INI file format[11]. The implementation uses a class for reading .INI files written by Clauss and Hill [AC01]. The different files containing tree type information are then given as command line parameters to the program at start up, separated with space or comma. Example:

[11] Files with the extension .INI was originally invented by Microsoft for storing configuration information for MS-Windows. In addition, many applications have their own .INI files. In Windows 95 and Windows NT, .INI files have been replaced by the Registry, though many applications still include .INI files for backward compatibility.

Figure 3.20: Tree growing out from a steep hillside.

```
fasttreerendering ../treetypes/default.tsp ../treetypes/japan.tsp
```

An example on the content of a parameter file is given in appendix C.

Instead of calling the configuration files .INI they are renamed to .TSP (Tree System Parameters) without changing the basic .INI file format.

Apart from the standard parameters from the article by Weber and Penn, additional parameters are implemented. Among these are parameters for specifying the texture used for each tree type. The user can specify both the standard texture used for the tree type and the normal map texture (by parameters $material$ and $materialNormal$). The user can also specify the ambient, diffuse and specular color used in the Phong shading calculation for each tree type (by parameters Ka, Kd and Ks). See appendix C for an example.

3.8 Wind

Although not the main purpose of this project, I experimented with simulating wind by modifying the control points for each tree. The approach involves altering the $nCurve$ parameter for the trunk. Since all control points are defined in world space, when modifying the main trunk all other control points for the attached branches will have to be recalculated. Of course the main trunk would not move much unless in the case of a hurricane. This approach was chosen to see what impact having to recalculate all control

points of every tree had on performance. The new $nCurve$ value is calculated as follows:

$$nCurve = \cos(windCircle) * nCurve/2 + nCurve$$

where $windCircle$ is a variable augmented by a certain value every frame and randomly generated for each tree.

Preferably a more accurate and realistic wind model should be implemented. Such a model is proposed in the future work chapter in section 6.3.

Tests show that the proposed method of moving all per vertex transformations to the GPU has a clear advantage when dealing with animation. The results of this benchmark test can be seen in the next chapter.

Chapter 4

Results and discussion

This chapter will present the most important results found using the implementation described earlier.

For comparable benchmarks, all tests are run on a Pentium 4 3.0GHz with a GeForce 6600GT PCI Express x16 using NVIDIA graphics drivers version 71.84.

Screen resolution: 800x600x16
Maximum number of segments per instance: 20
Maximum number of points making up the circle: 10

These maximum values mean that each stem can consist of a maximum of 20 segments along its length, and a maximum of 10 vertices defining the circle of each stem. A trunk at instance level 0 will typically have 200 vertices (20 segments and 10 points) defining it, while a branch at level 1 will consist of only 25 vertices (10 segments and 5 points) when using the instance LOD method.

The benchmarks are presented in the order they were implemented, meaning that each stage includes the previously presented methods and techniques.

4.1 Storing vertices in a VBO

The upper graph in figure 4.1 shows the frame rate when we specify a memory area (filled with vertices) to be drawn once for every instance. The lower graph shows the frame rate when we calculate the values for each vertex individually for all instances during run-time and send them one-by-one to the GPU for processing. Even though the vertices are calculated on the CPU, their value is the same as in the vertex buffer object (VBO). So placing them according to the spline defining the branch is done on the GPU in both cases. The graph shows that storing vertices in a VBO and uploading it once to the GPU has a clear advantage over uploading each vertex separately. This means that we instead of executing commands hundreds of times per branch, we draw a branch with a simple command. Even though it might be difficult to see from the graph, the advantage of

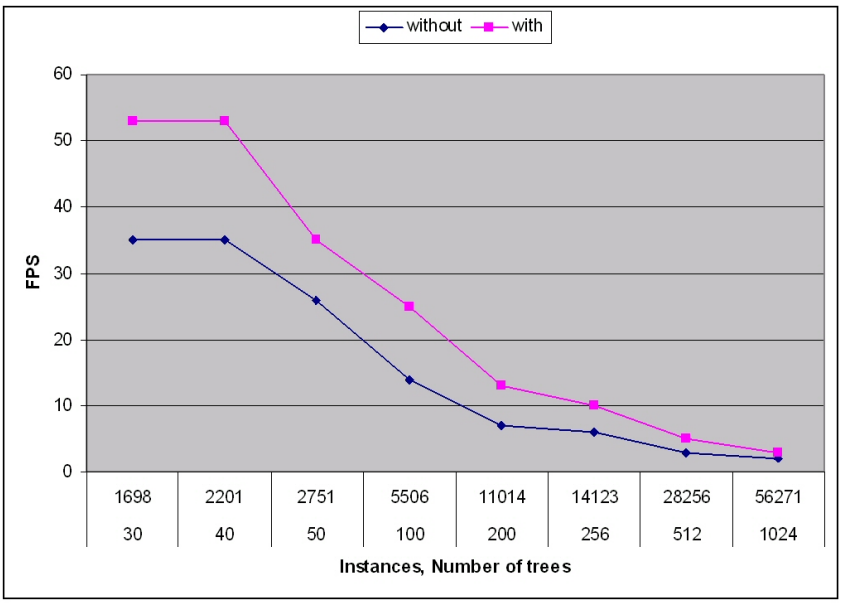


Figure 4.1: Performance graph when storing vertices in a vertex buffer object

this technique is actually larger when drawing many instances. When drawing around a hundred trees (equivalent to 11.014 instances for the tree type used in the benchmark) the frame rate is nearly doubled when using a VBO. When surpassing 56.271 instances the curves tend to coincide, which is due to driver bound for both methods, meaning that the CPU has trouble feeding the GPU with draw-calls.

4.2 Adding instance LOD

As expected the frame rate increases a great deal when drawing smaller branches with fewer details. The interesting part of the graph in figure 4.2 is the steep increase from 25 trees to 10 trees and then the graph goes flat. This is probably due to the bottleneck in the CPU, not able to feed enough draw-calls to the GPU. So if the driver was able to let through more commands we would see an even higher frame rate for a few trees with many branches. This means this approach works best for advanced trees with many branches, rather than many simple trees.

4.3 Pseudo Instancing

Pseudo Instancing has a clear advantage over uniform variables when updating the variables per instance. The speed up is nearly doubled when rendering around 8.000 instances (see figure 4.3).

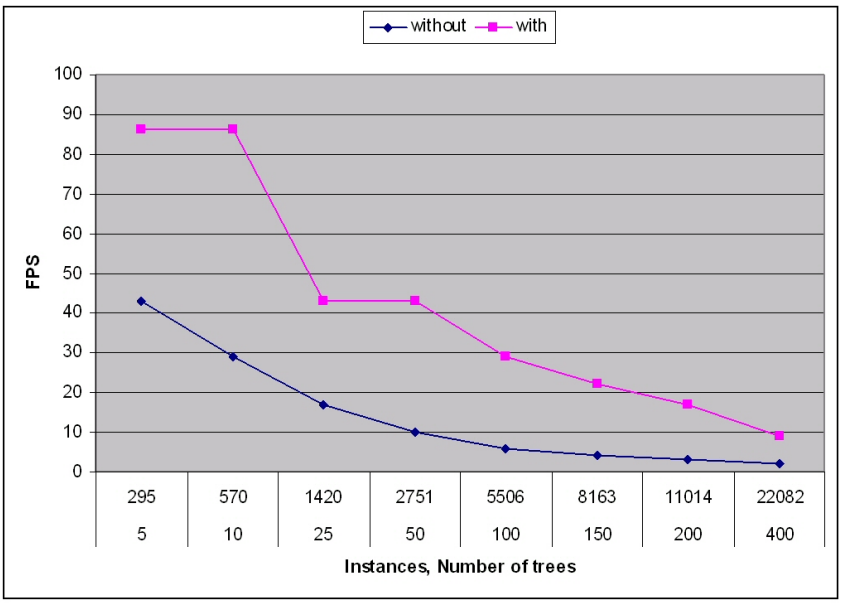


Figure 4.2: Performance graph when adding different level of detail to instances

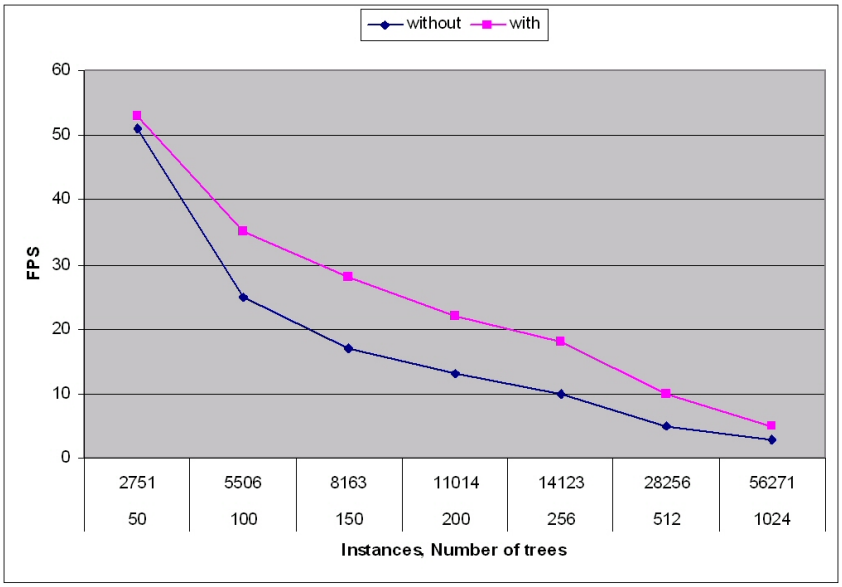


Figure 4.3: Performance graph when adding pseudo instancing

4.4 Precalculated blending functions

This benchmark only tests the gain in removing calculation of the blending functions from the vertex program. The two tests run identical code on the CPU, the only difference is in the vertex program where the calculation of the two Hermite blending functions are removed and replaced by the input from the vertex and color information for the current vertex.

Precalculating the blending functions on the CPU when generating the instances amounted to a speedup shown in the graph in figure 4.4. As expected the advantage is minor and only becomes significant when the number of vertices in the scene becomes sufficiently high.

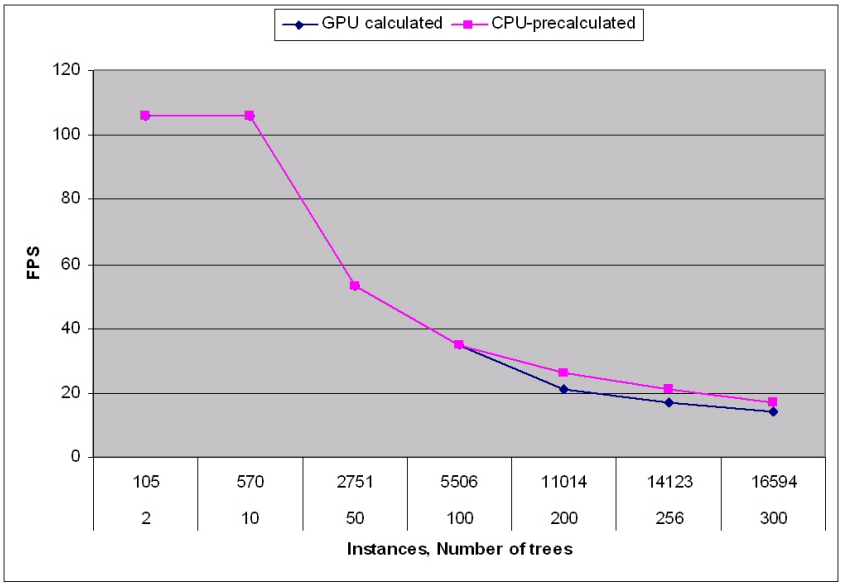


Figure 4.4: Performance graph when precalculating blending functions

4.5 Moving light calculation to fragment program

This benchmark test was performed to measure the speed-up of moving the light calculation to the fragment program instead of for simplicity running it in the vertex program. For calculating the light, the standard algorithm of Phong was implemented in both the vertex shader, calculating the light per vertex only and interpolating between the vertexes, and in the fragment shader, calculating the light based on interpolated normals from the vertex shader.

For a small number of instances the curves in figure 4.5 are almost concurrent. This is probably due to a bottleneck in the vertex shader since the CPU manages to send more draw-calls than the vertex program can process. Between 5.506 and 14.000 instances there

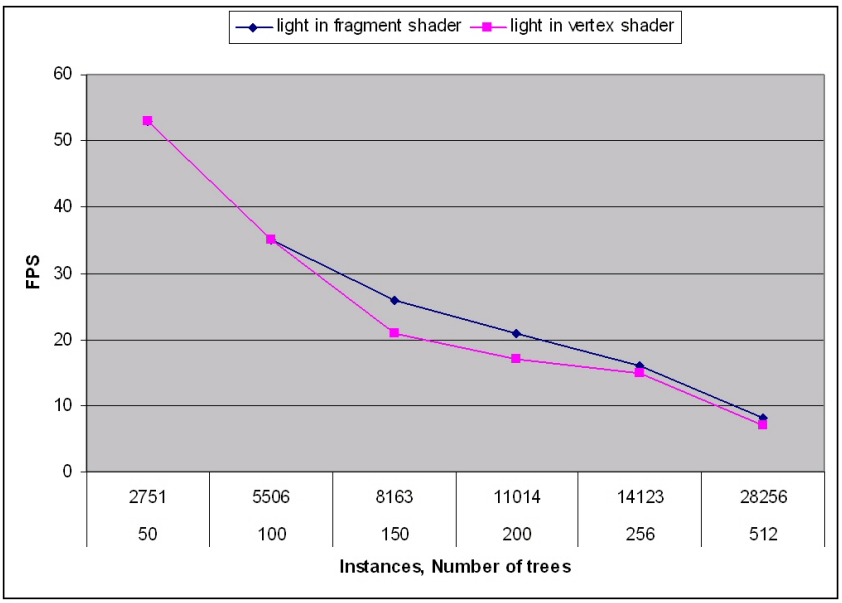


Figure 4.5: Performance graph when moving light calculation to fragment shader

is a clear advantage of letting the fragment program deal with the light calculation. The curves come together again at the end of the graph, which is due to CPU bound, meaning that the CPU fails to run as many draw-calls as the GPU can handle.

4.6 Adding normalsmaps

Normal mapping do come with a cost in performance, although its surprisingly small. Even so, calculating light has to be done anyhow, so adding one texture lookup and a matrix operation does not affect the shader much. If the algorithm for calculating the light had been more advanced, like ray tracing or advanced shadow algorithms, one could expect a bigger difference (see figure 4.6).

4.7 Adding distance LOD

The advantage of adding a simple level of detail algorithm based on the distance from the observer should be clear. This implementation is optimal when rendering around 150 trees (or around 8.000 instances) as shown in figure 4.7. If the trees were more spread in the landscape, this improvement would of course be more significant. For this benchmark the trees were placed semi-randomly using the formula

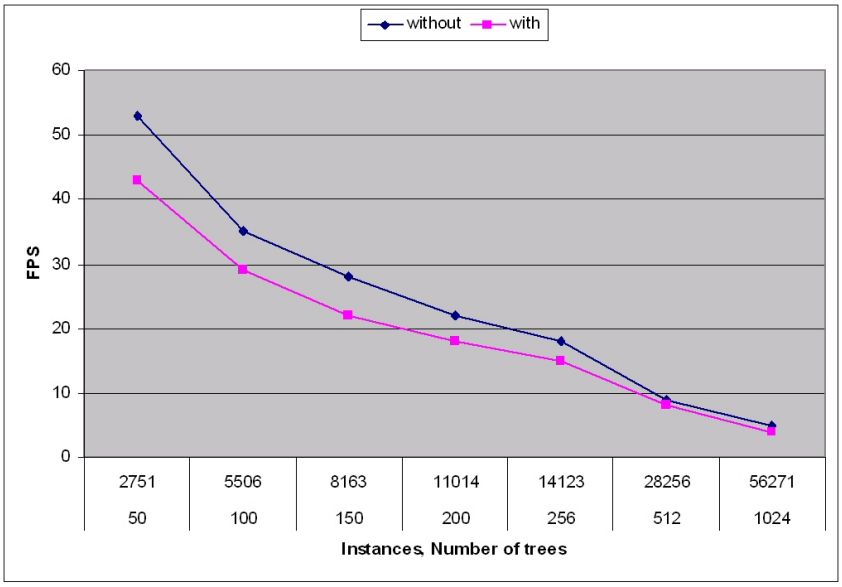


Figure 4.6: Performance graph when adding normal maps

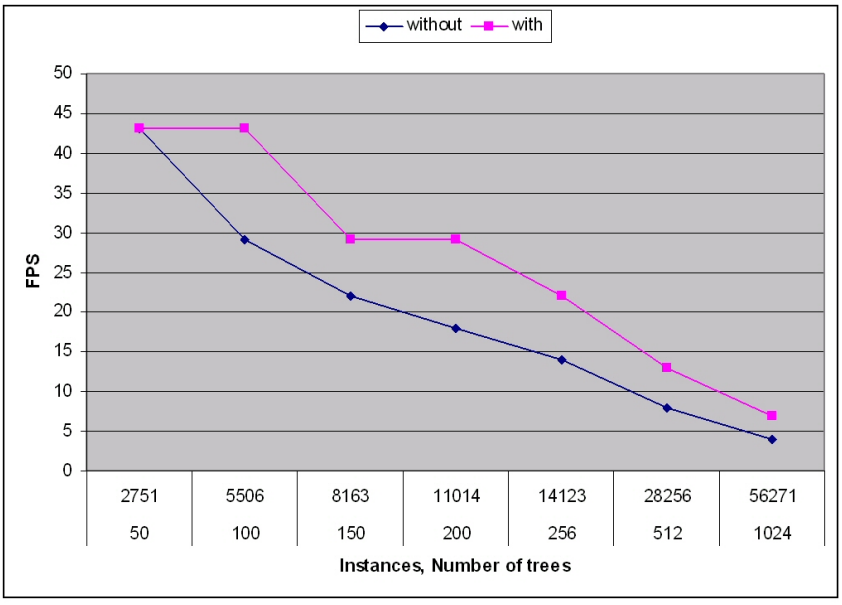


Figure 4.7: Performance graph when adding level of detail from observer

$$p(x) = random((int)(1.0/density * number\_of\_trees))) - 0.5/density * number\_of\_trees$$
$$p(y) = 0$$
$$p(z) = random((int)(1.0/density * number\_of\_trees))) - 0.5/density * number\_of\_trees$$

using a density of 0.9.

4.8 Adding wind

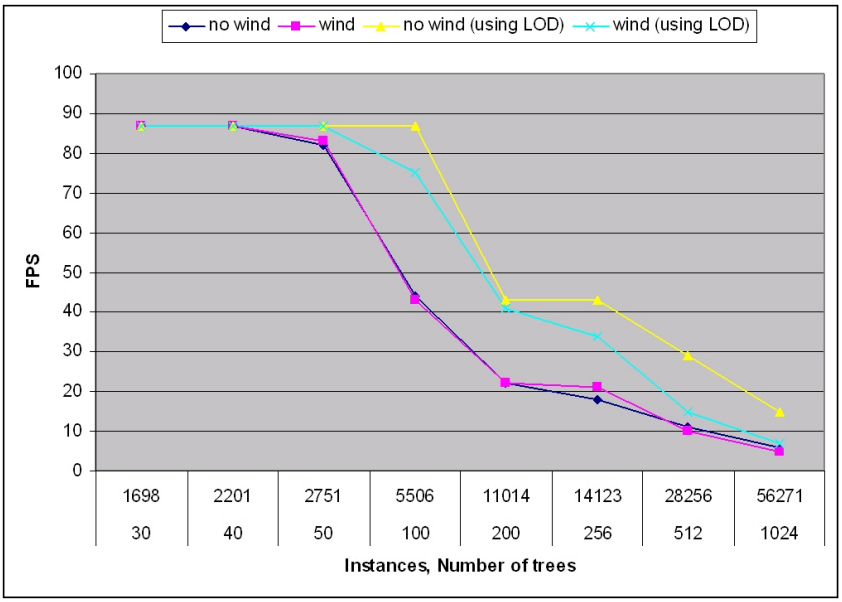


Figure 4.8: Performance graph when adding wind (with and without distance LOD)

We see from the graph in figure 4.8 that when removing the distance level of detail algorithm it doesn't seem to matter whether or not we recalculate the control points every frame. This is probably because the CPU is not the bottleneck in these cases, since the GPU has a lot more vertices to process than when using distance LOD. When we include the distance LOD algorithm, we see the real cost of recalculating the control points. This cost is however not that big when rendering a few trees, but as expected become more significant when we need to recalculate over 20.000 branches. This test shows that the technique of storing and recalculating the control points on the CPU is a good alternative to doing all the calculations on the GPU when we render under 20.000 instances.

For this benchmark, the landscape was included in the render. The highest frame rate achieved using this test bed when rendering the landscape alone was 88 FPS. This is why all the graphs go flat when exceeding 87 FPS.

4.9 CPU vs. GPU

A previous paper written by Nordstoga and myself called ”Parametric tree-rendering” [KN04] implements, among other things, some of the same functionality as this project based on the article by Weber and Penn. Kjær and Nordstoga however focuses purely on rendering trees using the CPU, calculating all vertex positions ”on-the-fly” while rendering. Using techniques such as display lists[1] the implementation reaches frame rates up to 159 FPS (on a modified GeForce 6800LE). Running the same benchmark in our test bed (using a GeForce 6600GT) the frame rate reaches 91 FPS. This is equivalent to rendering a hundred trees using this fast tree rendering system since the total number of vertices in the scene is the same in both cases. However display lists have the disadvantage of being static, meaning that the geometry can not be altered without having to regenerate whole or large parts of the geometry every frame. In comparison, using the implementation from Kjær and Nordstoga rendering a hundred trees, recalculating each tree every frame, gave a frame rate of 5 FPS in our test bed (see first pillar in figure 4.9). This is a more fair comparison to the implementation done in this project when we compare the performance of the CPU to the GPU since in both cases, calculating and placing the vertices in the scene is done for every vertex for each frame individually.

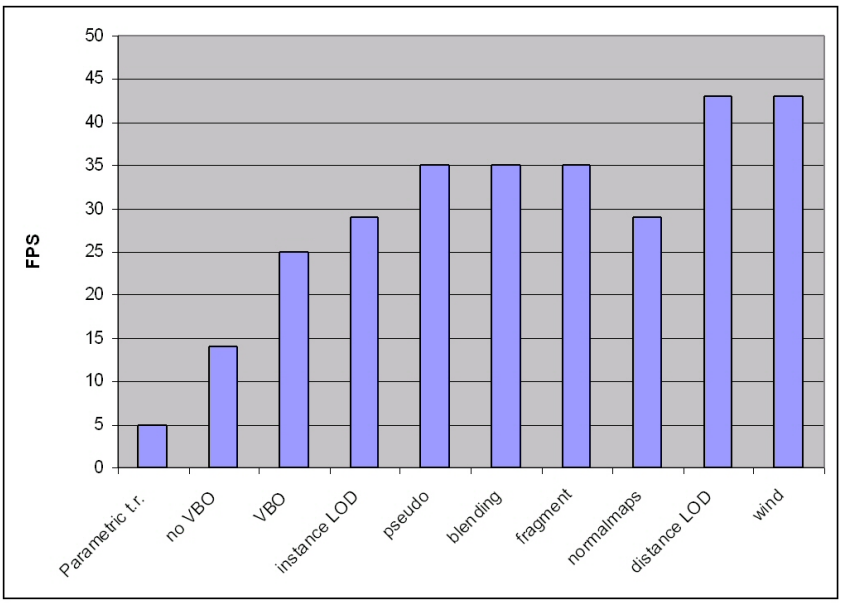


Figure 4.9: Summary graph comparing the different stages of implementation

The graph in figure 4.9 shows the frame rate of different stages of the implementation rendering 100 trees. From the graph it is easy to see that implementing VBO, instance

[1]OpenGL display lists are used to ”extract” and store only the OpenGL specific commands when drawing an object. This usually results in faster drawing of the object since all the code in between the OpenGL commands are omitted. One restraint however is that a display list is static and not dependent on other variables. Therefore when calling the display list the object(s) will be drawn in the same way as when it was created, i.e. the shape can not be animated.

LOD, pseudo instancing and distance LOD had the largest impact on improving the frame rate. Precalculating the blending functions and moving the light calculations to the fragment shader had a larger impact as the number of vertices grew. Implementing normal maps resulted in slower performance as more texture lookups have to be performed for every normal calculated. However using normal maps improved the visual quality a great deal when looking at the trees from a short distance.

Chapter 5

Summary and conclusion

Using today's GPU power this project has shown that it is possible to draw natural looking trees defined using a few simple parameters and formed by the smooth features of the Hermite spline. I have shown that using techniques such as pseudo instancing, vertex buffer objects, level of detail per instance, level of detail based on distance from observer and other minor optimizations are clearly applicable for speeding up real-time tree rendering.

Comparing with the implementation presented by [KN04], there is a clear advantage of drawing trees using the GPU when you want external forces, such as wind, to alter the geometry. In comparison, rendering a hundred trees, there is a speed up from 5 FPS to 43 FPS. The comparison also shows that display lists may be favorable when drawing static objects.

Drawing instances using VBOs instead of single vertices and using the Pseudo instancing technique had a great impact on performance, nearly doubling the frame rate for certain number of instances.

Adding level of detail based on the observers distance from each tree also had a great impact on performance, especially when rendering a large forest of widely spread trees.

Implementing precalculation of the Hermite blending functions emitted to a minor speed up when rendering a large number of vertices. What we can conclude from this is that the GPU calculates these type of functions extremely fast on its own, but as these calculations do not do need to be recalculated for every vertex, they can profitably be precalculated on the CPU in advance.

I've shown that using normal mapping greatly increases the viewing pleasure and realism with minor performance cost. Placing the trees in a landscape, the user is able to get a feel of walking around in a small forest of trees using the mouse and arrow keys. Using the flexibility of the Hermite spline the trees seem to be growing naturally out of the ground, no matter the angle of the ground.

Chapter 6

Future work

This chapter will propose and briefly discuss additions and enhancements relevant to the implementation presented in this report.

6.1 Segment Buffering

To achieve an ever higher frame rate, Jon Olick suggests a technique called ”Segment Buffering” [Oli05]. The problem with many static scenes containing many objects of the same material is that they are often bound by render-state changes such as transform changes, light map texture changes, or vertex stream changes. This is for example the case when drawing a tree with this fast tree rendering solution. Since all branches require individual draw-calls with texture changes, this produces driver and thus CPU overhead. Studies made by Wloka [Wlo04] shows that a 1 GHz CPU can render only around 10.000 to 40.000 batches per second in Direct3D or around 4.000 batches per frame on a modern CPU, meaning that without segment buffering we could only draw around 4.000 branches in a scene in real-time, which is also the limit using pseudo instancing in OpenGL. To make use of segment buffering, this project would have to be implemented using Direct3D. A single tree or maybe even an entire forrest of similar trees could then be drawn sending only one single draw call.

6.2 Vertex Constants Instancing

To make use of instancing to a further degree the project could be implemented using vertex constants instancing. This means buffering multiple instances into one vertex buffer, indexing each vertex with a number defining which stem/spline this vertex belongs to. The vertex program would then have to fetch the correct control points based on this index and move the vertex according to the remaining values of the vertex. Although this might be faster because you could draw an entire level of branches in one draw call, it requires a much more advanced vertex shader and memory lookups, which would result

in slower performance for the vertex shader.

6.3 Animation

One enhancement which usually make a scene more lively is adding animation. A widely discussed issue in real-time visualization of trees is animation based on influence from natural forces like wind.

Since all the trees are build out of a skeleton of control points, animating the trees only affects the CPU when having to calculate the new position for the control points. In a normal software rendering system one would have to move every vertex for every tree, occupying the CPU with vertex calculations. As these calculations are now moved to the GPU we could add more realistic animation features to run on the CPU, as more advanced wind models. A fast and efficient wind model was proposed in the article "Animating Trees" used in the animation of trees for the movie Shrek [Pet01]. Each branch is bend according to a force sampled at the tip of the branch. The bend factor has the same direction as the force and increases over the length of the branch.

$$b_i = b * (i/(n-1))^2$$

The idea is to preserve the distance between the control vertices for each branch so that the branch only changes direction and not form.

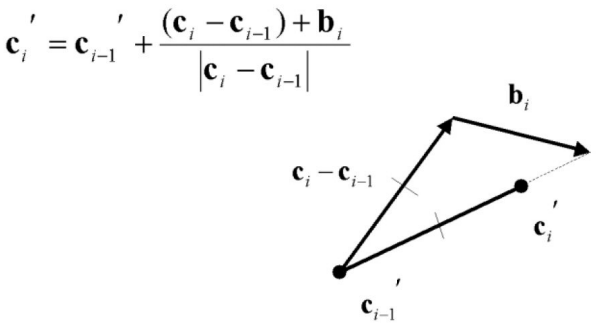


Figure 6.1: Deforming a Branch: equation [Pet01]

A way of implementing this would be to generate key-frame-trees (sets of different control points per tree) and then in the vertex shader interpolate between the control points by a time-variable specified by the program. This technique is called "Key-Frame Interpolation" [FK03] and the formula for interpolating between two key frames is:

$$blended_{Position} = position_A * (1 - f) + position_B * f$$

Figure 6.2: SGI Billboard [SG98]

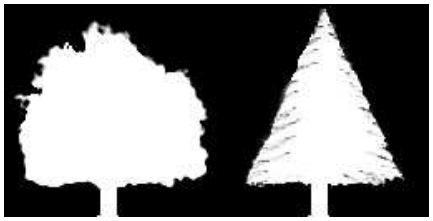


Figure 6.3: SGI Billboard mask [SG98]

6.4 Level of detail

The level of detail algorithm implemented in this project takes into account the level of recursion for each instance/branch and the distance from the viewer. The algorithm simply switches between vertex objects with different number of vertices associated with it. This does not lead to a smooth transition between the different levels of detail as the viewer moves closer and further away from the trees as it would using techniques such as geomorphing. However, defining the switch at a sufficient distance, the switch is hardly noticeable. Another improvement that could be done when the trees are viewed from a distance is the switch to billboards. This implies drawing a sprite representing the general shape of the tree instead of drawing the actual geometry defining the tree.

GPU Gems 2 presents a technique for better transition between billboards inside and outside our viewing frustum [Wha05]. By simulating alpha transparency via dissolve one could fade out the billboards at a distance near the boundary of the far clipping plane. The technique is basically to replace the normal billboard mask with a noise texture and then slide the alpha test for this noise texture from 1 to 0 when dissolving the billboard. This process is used in SpeedTreeRT, a C++ Application Programmer's Interface (API) package for real-time foliage creation [IDV].

6.5 Culling

The only culling done in this project is the standard culling performed by OpenGL by removing the faces which normal points away from the viewer. A natural improvement would be to implement culling algorithms to avoid drawing the trees outside the viewing frustum. Techniques such as octree culling with each tree (or even each branch) represented as a node with a bounding box would be a respectable candidate [MH98].

6.6 Continuity of texture patches

Currently there is a discontinuity in the texture patches between connecting stems. This might be fixed using a parameter for specifying where on the texture patch we should start drawing. The idea would be starting to paint the texture at the position t where

the stem emerges multiplied by the length of the parent stem:

$$parent_t * parent_{length}$$

This would however not be exactly accurate as the scaling of the texture for the child stem is smaller than that of the parent stem. Calculating the scaling factor would solve this problem, amounting to the formula:

$$parent_t * parent_{length} * (scale_{parent}/scale_{child})$$

6.7 Shadows

Shadows from a tree is not uncommon in nature and could be simulated by for example projecting the tree object onto the ground plane. Self shadowing could be simulated in the fragment shader by altering the diffuse color depending on what recursion level the stem rendered is at and what position (t) along the stem the face being rendered is at, making it darker as we approach the center and bottom of the tree. This effect would of course be more relevant after implementing leafs as the tree would cast more shadows. An interesting approach is proposed by [AMP01].

Bibliography

[AC01] Shane Hill Adam Clauss. Implementation of the cinifile class. 2001. 36

[Agi72] Gerald Jacob Agin. *Representation and description of curved objects*. PhD thesis, 1972. 9

[AMP01] Fabrice Neyret Alexandre Meyer and Pierre Poulin. Interactive rendering of trees with shading and shadows. In *Eurographics Workshop on Rendering 2001*, June 2001. 54

[Bis75] R. L. Bishop. *There is more than one way to frame a curve*, volume 82 of *Amer. Math. Monthly*, pages 246–251. March 1975. 25

[Blo85] Jules Bloomenthal. Modeling the mighty maple. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 305–311. ACM Press, 1985. v, vi, 1, 9, 10, 24, 28

[Blo90] Jules Bloomenthal. Calculation of reference frames along a space curve. *Graphics gems*, 1:567–571, 1990. v, vi, 11, 25

[Car05] Francesco Carucci. *GPU Gems 2*, chapter Inside Geometry Instancing, pages 47–67. Addison-Wesley, 1st edition, 2005. 11, 12

[Dre04] Søren Dreijer. Bump mapping using cg, 2004. http://www.blacksmith-studios.dk/projects/downloads/bumpmapping_using_cg.php. v, vi, 16, 34

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 12, 28, 52

[Hum02] Ben Humphrey. Height Map 3 Tutorial, 2002. http://www.gametutorials.com/. vi, 35

[IDV] IDV. SpeedTree. http://www.idvinc.com/. 53

[JW95] Joseph Penn Jason Weber. Creation and rendering of realistic trees. In *Computer Graphics*, pages 119–128. ACM Press New York, NY, USA, 1995. v, 1, 3, 4, 19, 23, 33

[KN04] Andreas Solem Kjær and Åsmund Nordstoga. Parametric tree-rendering. Desember 2004. 46, 49

[LA90] Przemyslaw Lindenmayer Aristid, Prusinkiewicz. *The algorithmic beauty of plants*. Springer, Verlag, 1990. 3

[MH98] Andreas Varga Markus Hadwiger. Visibility culling. *Proseminar Wissenschaftliches Arbeiten*, 1998. 53

[MSD05] Microsoft MSDN. Dds files, 2005. http://msdn.microsoft.com/archive/en-us/directx9_c_summer_03/directx/graphics/reference/ddsfilereference/ddsfileformat.asp. 32

[NVI05] NVIDIA. Nvidia sdk code samples, 2005. http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html. v, 13

[oE04] The University of Edinburgh. Computer graphics, lecture 10, curves and surfaces, 2004. http://www.inf.ed.ac.uk/teaching/courses/cg/lectures/lect10.ppt. vi, 24

[OKR04] John F. Hughes Olga Karpenko and Ramesh Raskar. Epipolar methods for multi-view sketching. *Eurographics Workshop on Sketch-Based Interfaces*, Grenoble France, 2004. 25

[Oli05] Jon Olick. *GPU Gems 2*, chapter Segment Buffering, pages 69–73. Addison-Wesley, 1st edition, 2005. 51

[Ope04] OpenGL.org. Opengl shading language, 2004. http://www.opengl.org/documentation/oglsl.html. 12

[oV04] University of Virginia. Cs 445 / 645 introduction to computer graphics, lecture 22, hermite splines, 2004. v, 8

[Pd05] Planet-d.net. Multitexturing with opengl, 2005. http://tfpsly.planet-d.net/english/3d/multitexturing.html. 35

[Pet01] Scott Peterson. Animating trees. Silicon Valley ACM SIGGRAPH, 2001. http://silicon-valley.siggraph.org/MeetingNotes/shrek/trees.pdf. vi, 52

[Pla05] PlanetMath.org. Definition of tangent space, 2005. http://planetmath.org/encyclopedia/TangentSpace.html. 34

[SG98] Inc. Silicon Graphics. Billboard, 1998. http://www.sgi.com/products/software/performer/brew/billboard.html. vi, 53

[Tec04] Unreal Technology. Unreal engine 3, 2004. http://www.unrealtechnology.com/html/technology/ue30.shtml. v, 15, 16

[Wha05] David Whatley. *GPU Gems 2*, chapter Toward Photorealism in Virtual Botany, pages 7–25. Addison-Wesley, 1st edition, 2005. 53

[Wika] Wikipedia. Anti-aliasing. http://en.wikipedia.org/wiki/Anti-aliasing. v, 17

[Wikb] Wikipedia. Spline curve. http://en.wikipedia.org/wiki/Spline_curve. 7

[Wlo04] Matthias Wloka. Batch, batch, batch: What does it really mean? *Presentation at Game Developers Conference 2003*, March 2004. 11, 51

[Zel04] Jeremy Zelsnack. Glsl pseudo-instancing. Technical report, NVIDIA Technical Report, http://download.developer.nvidia.com/developer/SDK/Individual_Samples/DEMOS/OpenGL/src/glsl_pseudo_instancing/docs/glsl_pseudo_instancing.pdf, 2004. v, 12, 14

Appendix A

Cg Shader code

A.1 Vertex shader

Listing A.1: Vertex Shader

```
// Function for calculating the flareZ variable from Weber and Penn
float getFlareZ(float flare, float Z) {
    float y = 1 - 8 * Z;
    if(y<0)
        y = 0;
    return flare * ( pow(100,y) - 1 ) / 100 + 1;
}

// Function for calculating the radiusZ variable from Weber and Penn
float getRadiusZ(float Z, float nTaper, float radiusStem, float
   lengthStem) {
    float unit_taper = 0;
    if(nTaper < 1)
        unit_taper = nTaper;
    else if(nTaper < 2)
        unit_taper = 2 - nTaper;
    else if(nTaper < 3)
        unit_taper = 0;
    float taperZ = radiusStem * ( 1 - unit_taper * Z );
    float radiusZ = 0;
    if(nTaper < 1)
        radiusZ = taperZ;
    else if(nTaper <= 3) {
        float Z2 = ( 1 - Z ) * lengthStem;
        float depth = 0;
        if((nTaper < 2) || ( Z2 < taperZ))
            depth = 1;
        else
```

```
28              depth = nTaper - 2;
29          float Z3 = 0;
30          if(nTaper < 2)
31              Z3 = Z2;
32          else
33              Z3 = abs(Z2 - 2 * taperZ * (int)( Z2 / (2 * taperZ) + 0.5
                    ));
34          if((nTaper<2) && ( Z3 >= taperZ))
35              radiusZ = taperZ;
36          else {
37              float sqrt1 = taperZ*taperZ - (Z3 - taperZ)*(Z3 - taperZ)
                    ;
38              float sqrt2 = sqrt(sqrt1);
39              radiusZ = (1-depth) * taperZ + depth * sqrt2;
40          }
41      }
42      return radiusZ;
43  }
44
45  void main (float4 btab : POSITION,     // Hermite blending function
46          float4 d_btab : COLOR,      // Hermite derivative blending
47                          // function
48          float2 objectProp : TEXCOORD0, // Variables t and rad
49          float3 ctrlp0 : TEXCOORD1,     // Start tangent
50          float3 ctrlp1 : TEXCOORD2,     // Start position
51          float3 ctrlp2 : TEXCOORD3,     // End position
52          float3 ctrlp3 : TEXCOORD4,     // End tangent
53          float3 safeVector : TEXCOORD5, // The safe vector
54          float3 param1 : TEXCOORD6,     // First set of parameters
55          float3 param2 : TEXCOORD7,     // Second set of parameters
56
57          out float4 oPosition : POSITION,  // Vertex position in
58                              // projection view space
59          out float3 objectPos : TEXCOORD0, // Vertex position in
60                              // world space
61          out float2 oTexCoord : TEXCOORD1, // Texture coordinate for
                the
62                              // current vertex
63          out float3 N_vec : TEXCOORD2,     // The normal vector
64          out float3 T_vec : TEXCOORD3,     // The tangent vector
65          out float3 B_vec : TEXCOORD4,     // The binormal vector
66                              //   used to calculate the
67                              //   tangent-to-world matrix
68
69          uniform float4x4 modelViewProj)   // The modelview
70                              // projection matrix
71  {
```

```
72      // The following variables are rendered for easier reading
73      // of the code and actually make the code a bit slower
74      float radiusStem = param1.x;
75      float lengthStem = param1.y;
76      float nTaper = param1.z;
77      float lobes = param2.x;
78      float lobeDepth = param2.y;
79      float flare = param2.z;
80
81      float t = objectProp.y;
82
83      //Make hermite
84      float3 her, d_her;
85
86      float4x3 control_pointsM =
87          float4x3(ctrlp0, ctrlp1, ctrlp2, ctrlp3);
88      her = mul(btab, control_pointsM);
89      d_her = mul(d_btab, control_pointsM);
90
91      // finding the normalized tangent:
92      // T pointing along the curve
93      T_vec = normalize(d_her);
94
95      // finding the normalized d cross safe vector:
96      B_vec = cross(T_vec,safeVector);
97      B_vec = normalize(B_vec);
98
99      // finding a third perpendiculat vector:
100     // N perpendicular to both T and B
101     N_vec = normalize(cross(B_vec, T_vec));
102
103     // we want to move to the correct point on the Hermite curve
104     // as tabulated in (her[ix][0],her[ix][1],her[ix][2])
105     float3 C_vec = her;
106
107     // at this point T,B,N describes the coordinate system
108     // we will use at the position C
109     // N will act as x-axes
110     // B will act as y-axes
111     // T will act as z-axes
112
113     // setting up the matix that will take us to the
114     // Frenet frame located at this t-point
115     // M=|N,B,T|
116
117     float3x3 normalM = float3x3(
118         N_vec,T_vec,B_vec
```

```
119     );
120
121     float rad = objectProp.x;
122     float cosS, sinS;
123     sincos(rad, sinS, cosS);
124
125     // Calculate the radius at the current t value
126     float radiusZ = getRadiusZ(t, nTaper, radiusStem, lengthStem);
127     // Calculate the lobe influence (if any)
128     float lobeZ = (lobes==0) ? 1 : 1 + lobeDepth * sin( lobes * rad )
            ;
129     // Calculate the flare influence (if any)
130     float addFlare = (flare==0) ? 1 : getFlareZ(flare, t);
131     // Calculate the final radius
132     float radi = radiusZ*lobeZ*addFlare;
133
134     // Generate the position of the vertex from where on the circle
            we are
135     float3 newPos = float3( cosS*radi,
136                 0,
137                 sinS*radi);
138
139     // Place the vertex according to the Frenet frame matrix
140     float3 normal = mul(newPos,normalM);
141
142     // Move the point along the spline to the right Hermite position
143     objectPos = normal + C_vec;
144
145     // Finally multiply the point with the modelview projection
            matrix
146     oPosition = mul(modelViewProj, float4(objectPos,1));
147
148     // Generate the texture coordinates to send to the fragment
            shader
149     oTexCoord = objectProp * float2(2/(3.14*((int)radi+1)),
150         lengthStem/2);
151
152     // Generate the vectors to send to the fragment shader
153     N_vec = normalize(normal);
154     B_vec = cross(T_vec, N_vec);
155 }
```

A.2 Fragment shader

Listing A.2: Fragment Shader

```
// Since the normals in the normal map are in the (color) range [0, 1] we need to uncompress them
// to "real" normal (vector) directions.
// Decompress vector ([0, 1] -> [-1, 1])
float3 expand(float3 v) { return (v-0.5)*2; }

void main(// The position of the vertex from the vertex shader
      float3 position  : TEXCOORD0,
      // The texture coordinates for the vertex
          float2 texCoord  : TEXCOORD1,
          // The normal vector from the vertex shader
          float3 N_vec : TEXCOORD2,
          // The tangent vector from the vertex shader
          float3 T_vec : TEXCOORD3,
          // The binormal vector from the vertex shader
          float3 B_vec : TEXCOORD4,

          out float4 color     : COLOR, // The color to paint the texel in

          uniform float3 globalAmbient, // Global ambient light in the scene
          uniform float3 lightColor,    // Color of the light source
          uniform float3 lightPosition, // Position of the light source
          uniform float3 eyePosition,   // Position of the observer
          uniform float3 Ke,          // The Phong emission
          uniform float3 Ka,          // The Phong ambient color
          uniform float3 Kd,          // The Phong diffuse color
          uniform float3 Ks,          // The Phong specular color
          uniform float  shininess, // The Phong shininess
          uniform sampler2D barkTexture,// The bark texture
          uniform sampler2D barkNormalTexture) // The normal map texture
{
  // Fetch and expand range-compressed normal
  float3 normalTex = tex2D(barkNormalTexture, texCoord).zyx;
  float3 normal = expand(normalTex);
  // Reconstruct the Frenet frame matrix
  float3x3 normalM = float3x3(N_vec,T_vec,B_vec);
  // Transform the tangent-space normal-map-texture
  // normal vector into world-space coordinates
  normal = normalize(mul(normal, normalM));

```

```
40    // Calculate direction vector needed for Phong shading
41    float3 objectPos = position.xyz;
42    float3 eyeDirection = normalize(eyePosition - objectPos);
43    float3 lightDirection = normalize(lightPosition - objectPos);
44    float3 halfAngle = normalize(lightDirection + eyeDirection);
45
46    // Compute emissive term
47    float3 emissive = Ke;
48
49    // Compute ambient term
50    float3 ambient = Ka * globalAmbient;
51
52    // Compute the diffuse term
53    float diffuseLight = saturate(dot(lightDirection, normal));
54    float3 diffuse = Kd * lightColor * diffuseLight;
55
56    // Fetch the color of the bark texture
57    float3 texelColor0 = tex2D( barkTexture, texCoord ).xyz;
58
59    // Compute the specular term
60    float specularLight = saturate(dot(halfAngle, normal));
61    specularLight = pow(specularLight, shininess);
62    if (diffuseLight <= 0) specularLight = 0;
63    float3 specular = Ks * lightColor * specularLight;
64
65    // Compute the final light at the current texel
66    color.xyz = (emissive + ambient + diffuse + specular) * texelColor0
          ;
67    color.w = 1;
68 }
```

Appendix B

Compiling the sourcecode

B.1 OpenGL, Cg and Glew

To compile the sourcecode for this project you need some additional files installed on your system. These include the library files for OpenGL and the NVIDIA Cg Toolkit. This toolkit can be downloaded from the NVIDIA web page http://developer.nvidia.com/. You also need the The OpenGL Extension Wrangler Library[1] which can be downloaded from http://glew.sourceforge.net/. The GLEW library is also included in the NVIDIA SDK, so it should be sufficient if you have version 9.0 or later installed already. To compile the project in Visual Studio .NET 2003 on a Win XP machine you need the following files in their respective directories:

HEADER FILES

The following files must be available and preferably exist in the following locations:

glew.h installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Include/Gl

wglew.h installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Include/Gl

glu.h installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Include/Gl

gl.h installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Include/Gl

glaux.h installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Include/Gl

LIBRARY FILES

The following files must be available and preferably exist in the following locations:

glew32.lib installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Lib

glew32s.lib installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Lib

glu32.lib installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Lib

[1]The OpenGL Extension Wrangler Library (GLEW) is a cross-platform C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. [http://glew.sourceforge.net/]

glaux.lib installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Lib
Opengl32.lib installed in ../Microsoft Visual Studio .NET 2003/Vc7/PlatformSDK/Lib

DLL FILES

The following files must be available and preferably exist in the following locations:
glu32.dll installed in C:/WINDOWS/system32
glew32.dll installed in C:/WINDOWS/system32
Opengl32.dll installed in C:/WINDOWS/system32

Appendix C

Tree type parameter file

```
[General]
shape=4
baseSize=0.1
scale=23
scaleV=2
ratio=0.015
ratioPower=0.13
lobes=3
lobeDepth=0.3
flare=1
recLevels=3
material=../materials/LondonPlaneBark.dds
materialNormal=../materials/LondonPlaneBarkNormal.dds
Ka={0.3,0.4,0.2}
Kd={0.8,0.8,0.8}
Ks={0.9,0.9,0.8}

[ParamLevel0]
0Curve=10
0CurveV=10
0CurveBack=0
0Rotate=140
0RotateV=0
0Length=1
0LengthV=0
0Taper=1

[ParamLevel1]
1Curve=0
1CurveV=90
```

```
1CurveBack=0
1Branches=10
1BranchesV=1
1Rotate=140
1RotateV=0
1DownAngle=60
1DownAngleV=0
1Length=0.3
1LengthV=0.05
1Taper=1

[ParamLevel2]
2Curve=-10
2CurveV=150
2CurveBack=0
2Branches=1
2BranchesV=1
2Rotate=140
2RotateV=0
2DownAngle=30
2DownAngleV=10
2Length=0.6
2LengthV=0.1
2Taper=1

[ParamLevel3]
3Curve=-10
3CurveV=150
3CurveBack=0
3Branches=0
2BranchesV=0
3Rotate=77
3RotateV=0
3DownAngle=45
3DownAngleV=10
3Length=0.4
3LengthV=0
3Taper=1
```

Appendix D

Binary distribution

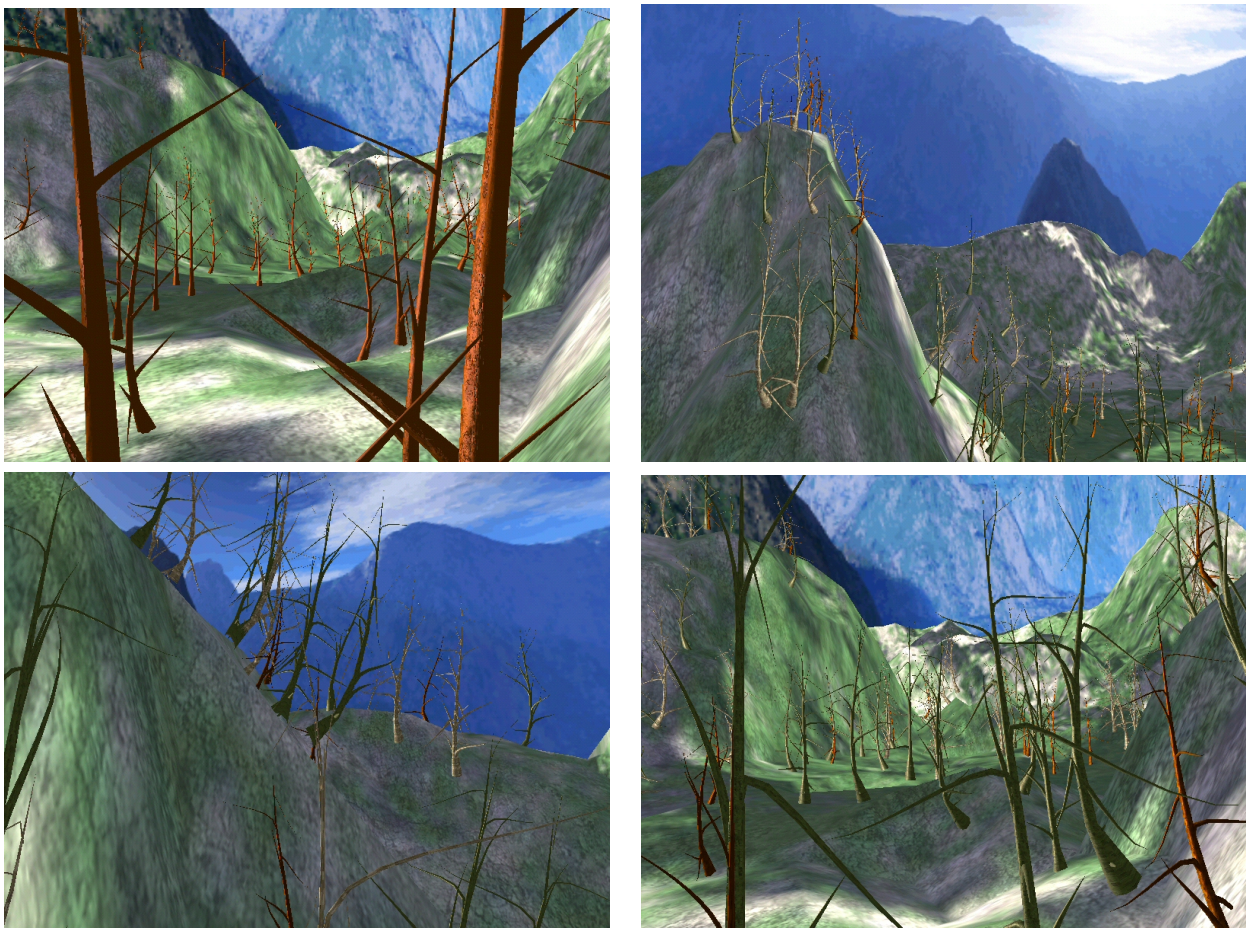
A binary distribution of the implemented application can be found on the accompanying compact-disc under the folder **/bin**. This distribution is compiled for Microsoft Windows XP and requires a graphics card supporting VBO and programmable shaders. To run the standard application simply run the .exe file. To run the demo including several tree types, run the .bat file.

The different tree types are stored under the folder **/treetypes** and all the textures are stored under **/materials**.

The observer view is controlled using the mouse, W, S, A and D keys on the keyboard for moving forward, backward, left and right and the up and down arrow-keys for moving up and down in the scene. By tapping space-bar twice the observer is taken for a ride circling the landscape of trees.

Appendix E

Screenshots and videos



Video demonstrations[1] can be found on the accompanying compact-disc under the folder **/videos**.

[1]Videos using low detail consists of a maximum of 20 segments and 10 circle points per stem. High detail has 24 segments and 32 circle points.