Jo Skjermo

Generation, Rendering and Animation of Polygon Tree Models

Thesis for the degree of Philosophiae Doctor

Trondheim, September 2008

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

NTNU
Norwegian University of Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

Faculty of Information Technology, Mathematics
and Electrical Engineering
Department of Computer and Information Science

© Jo Skjermo

ISBN 978-82-471-1701-9 (printed ver.)
ISBN 978-82-471-1702-6 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2009:155

Printed by NTNU-trykk

To my family near and far.

Abstract

This thesis contains a set of contributions that focuses on challenges in visualization of polygon models of trees, in particular models of tree stems and branches. These challenges cover a wide range of the visualization research field, including generation of polygon models of trees, hardware accelerated visualization and animation of trees from parametric systems, and surface modelling and tessellation for tree stems.

To generate polygon models of trees, an existing approach for automatically generating polygon models of branching vascular transportation systems, typically blood vessels, is extended to handle tree stems. The approach allowed for automation of the generation process, while maintaining smoothness at branching areas. The approach is used to generate polygon models of trees.

For hardware accelerated visualization and animation of trees from parametric systems, methods for visualization of trees described by parametric descriptions are adapted and extended to run on dedicated graphic processing hardware. To allow for animation of trees visualized in such a manner, an animation process that is hardware accelerated is developed using knowledge gained from work on hardware acceleration approaches in the field of image processing.

For surface modelling and tessellation for visualization of trees, concepts, ideas and early work are presented. The basic concept is that by working with a coarse polygon model of a tree during animation (or by generating a coarse polygon model of a tree after animation is done); one can tessellate this coarse model in hardware in real-time to a much higher degree then earlier approaches. This will then enable one to visualize trees stems of a higher Level of Detail (LOD) and at higher speeds than were previously achievable.

Although the contributions in this thesis cover a wide range of challenges from the visualization research field, the main focus has been on generation, animation and visualization of tree stems and branches for use in real-time applications. Especially the concept of hardware acceleration has become more of a focus, as graphic processing hardware has become more and more powerful both in speed and especially in functionality. The contributions will hopefully help in future development of even better methods for visualizing trees and plants.

Preface

This thesis is submitted to the Norwegian University of Science and Technology in partial fulfilment of the requirements for the degree of *philosohiae doctor (PhD)*. The work reported herin has been conducted at the Department of Computer and Information Science, NTNU, Trondheim. The work has been funded by the Faculty of Information Technology, Mathematics and Electrical Engineering, NTNU, Trondheim, through a PhD scholarship position.

Acknowledgments

The work presented in this thesis is funded by the Norwegian University of Science and Technology. Firstly i would like to thank the Faculty of Information Technology, Mathematics and Electrical Engineering, and the Department of Computer and Information Science, for funding and supporting my work.

I am grateful to many people that have provided inspiration and support during the course of this work. In particular, I would like to thank my supervisors Professor Keith Downing, Assistant Professor Torbjørn Hallgren and Associate Prof. II, Bjørn Sæther.

I would also like to thank Anders Kofod-Petersen and Odd Erik Gundersen for fruitful discussions and comments.

Finally, I would like to thank my family, whose steadfast support has been truly inspirational.

Jo Skjermo
Trondheim, 16/06, 2009

Contents

List of Figures

Part I

Context

Chapter 1

Introduction and Main Focus

Botanical trees are beeing modelled extensively for the portrayal of natural scenes in the animation, gaming and simulation industry. Common for the uses of botanical tree models in these industries, are that increased realism for both appearance and animation, are fast becoming a factor for success.

Other factors of success are the speed of which one can draw the tree models and the computational cost of drawing a tree model. Especially, in the gaming industry it is of importance to give the impression that the user is in a world that updates in real-time. Because of this, one wants to be able to render as many tree models as possible, using as few computational resources as possible. However, this limits the complexity of the models one can use, and thus the realism one can achieve.

1.1 Motivation and Aim

To control the complexity when rendering realistic tree models, one can, with success, target the rendering approach toward a needed *scale*. In the work by Garcia et al. [2005] three distinct scales were defined: *insect scale*, *human scale* and *vehicle scale*. At the vehicle scale, the viewing distance are generally large and individual trees are usually not focused on. On the other end of the scale, at the insect scale, a single tree has to show individual branches and leaves with great realism.

For rendering of realistic trees, there are two general approaches: image-based and geometry based. The first approach, image-based methods, works by using images of trees to give the appearance of trees inside the virtual world. The images are generally applied as textures on very simple polygons thereby achieving low rendering cost. This approach does, however, trade precision and consistency for increased rendering speed.

For geometry based methods, a tree model is rendered directly with a classic approach using polygon meshes. When using polygon meshes, one can more easily maintain precision and consistency to get increased visual quality. Generally, this has a higher computational cost then using image-based methods.

As the image-based methods lack precision and consistency, such approaches are best used at the *vehicle scale*. Geometry based methods will generally give better realism when approaching the *human scale* and *insect scale*.

In Section 1.1.1 an overview of image-based approaches in the context of tree rendering is presented. This overview shows a trend toward using hybrid methods consisting of both image-based and geometry based methods for tree rendering. In the most recent cases, image-based methods are completely exchanged with geometry based methods for parts of the trees when approaching the human scale

From this, the aim of the work presented here has been to improve the state of generation, rendering and animation of geometry-based tree models, especially for the main stem and branches of the trees.

1.1.1 Image-based methods

This section gives a short overview of the history of image-based methods in the context of tree rendering.

The first image-based approach was *billboards*, as defined by Jones [1994]. A billboard is a quad (a four-sided polygon) that has a partial transparent texture mapped on to it (analogous to cardboard cut-outs). Each individual billboard has a position and orientation, and the orientation of a billboard can be changed to always face the observer. Another alternative is to use several quads to represent the object for different directions, and then blend them together to generate a good result from the observer's point of view.

A *billboard cloud*, as defined by Décoret et al. [2003] is when a triangle mesh is approximated using a set of billboards. A billboard cloud is usually generated by rendering a triangle mesh into a set of billboard planes that are nearly co-planar with the triangles in the mesh. Billboard clouds have a range of improved properties compared with normal billboards. For instance, in the work by Lacewell et al. [2006], one could have both good motion parallax and smooth transition between recursive level of details.

Billboards and billboard clouds are generally good at rendering at a vehicle and toward human scale, but often struggle when going towards the insect scale. One should note the conceptual difference between a classic billboard and a billboard cloud when rendering. A billboard **is** the object, while billboard clouds **represent/depict** the object. Polygon mesh models of trees are often used when generating the images to use as billboards, but also when generating billboard clouds. One should note that one can generate the images used as a billboard or billboard cloud when needed. For instance, one can generate a new image to use whenever the angle or distance between the observer and an object changes more than a given limit. Such an approach is called *impostors*, and is, for instance, used in Décoret et al. [2003].

The other general approach is real-time geometry based rendering. Here a tree model is rendered directly with a classic approach using polygon meshes, compared to using it as starting point for billboards or billboard clouds. As a tree model might require hundreds of thousands of triangles to be realistic (especially as one approaches insect scale), some form of LOD is therefore required to achieve any form of real-time performance.

In computer graphics, mesh simplification methods have historically been used to generate different LOD for geometry based rendering (see the work of Luebke [2001] for a survey of different mesh simplification methods). However, for naturally looking tree models, mesh simplification approaches are usually not adequate to get good enough results because of the models' high complexity, especially for leaves. However, in the works of Bromberg-Martin et al. [2004] and Rebollo et al. [2007] hybrid approaches were suggested, using both billboard clouds (especially for leaves) and mesh simplification (especially for the stems and branches).

In summary, as one moves close to the human scale and even toward the insect scale, using polygon meshes will become more and more important. Image-based approaches suffer from a limitation in that it is hard to achieve a believable parallax effect in these ranges. Although, newer image-based approaches somewhat overcome this problem as the meshes are not rotated as the camera moves, the effect falls apart when one approaches the human scale.

1.2 Dissertation Statement and Problem Definition

As graphic hardware becomes more and more powerful, rendering of complex objects such as trees as part of the underlying simulated world, and not just part of the scenery, are fast becoming an option for real-time applications. With the increased computational power available through the introduction and subsequent improvements of the programmable graphic processor unit (GPU), the visual quality one can achieve in real-time application has also increased. One approach for increasing the quality of tree rendering in real-time applications could therefore be to utilize the computational power of the

GPU for this area.

This thesis takes a closer look at tree rendering with respect to the visual aspects when approaching the human scale and goes toward the insect scale. For polygon based tree rendering, can the GPU be used to improve the rendering quality or offload computations from the CPU? To simplify the process of tree rendering, the overall process is here divided into three focus areas, namely: tree generation, animation and rendering.

The first area is tree object generation. The tree objects should appear as close to real botanical trees as possible when rendered. To achieve this, the tree model generation approach should produce results that mimic the natural world. On the other hand, a real tree has very large complexity with hundreds to thousands of branches, and thousands to millions of leaves. If the generated objects are of such high complexity, animating and rendering fast become too computationally expensive even if using LOD, especially when considering a whole forest.

As earlier stated, using billboard techniques does not generate adequate results when trees are viewed up close. A hybrid approach that uses billboard based approaches for small scale branch structures and leaves, and polygon meshes for the main stem and branch structure is therefore preferable. Such an approach enables fast rendering while still maintaining a good appearance, as the stem and the largest branches have the most visual impact.

The second main area is animation. A tree model should appear to behave naturally when subjected to influence from external forces to maintain immersion. Especially wind influence is commonly found in the natural world, and as such wind animation is an important tool for increasing the immersion of users. Influence from other forces are also becoming more and more important, as tree objects are beginning to become part of the underlying simulated application world, instead of merely part of the scenery. Such influences can for instance be collision with vehicles, or more destructive actions like being cut or falling due to explosive blasts.

The final area is rendering. Even with the power of today's graphic hardware, rendering a large amount of polygon meshes of trees is limited by its complexity. As cost of animating increases with the resolution and complexity of an object, very simple polygon meshes are usually used for animation in real-time application. As such, the polygon meshes generated of the main stems and branches should have as small a number of polygons as possible, to achieve fast animation, but high enough complexity so that the animation looks convincing. Simple polygon meshes with few triangles do not produce good results when rendering, especially when close to the object (human toward insect scale). However, approaches such as tessellation algorithms generate good visual results from simple polygon meshes and are therefore of interest.

In summary, the main problem definition is: *how to generate, animate and render polygon mesh objects of natural looking tree stems for use in real-time applications?*

From the main problem definition, the overall hypothesis explored in this thesis is: *A programmable graphical processing unit (GPU) can be employed, both directly and indirectly (by offloading the CPU) to improve the quality of three key aspects of polygon-based tree visualization: mesh generation, animation and rendering.*

1.3 Research Contributions

The contributions presented in this thesis all focus on the three areas identified, that is, generation, animation and rendering of polygon models of trees. However, only the stems and main branches of a botanical tree model are considered. Other parts of a tree, like leaves, fruits, flowers and smaller branches are not considered, as such details can for instance be added using similar approaches as used in the commercial product SpeedTreeRT [2008]. In SpeedTreeRT fine details are added using textures for smaller branches (fronds) and leaf cluster billboards, as described in Section 2.1.2.

In Figure 1.1, one can see a tree mesh structure both with and without any leaves and small branches. Adding fronds increases the visual aspect of the inner structure, while adding leaf clusters as textures flesh out the final trees.

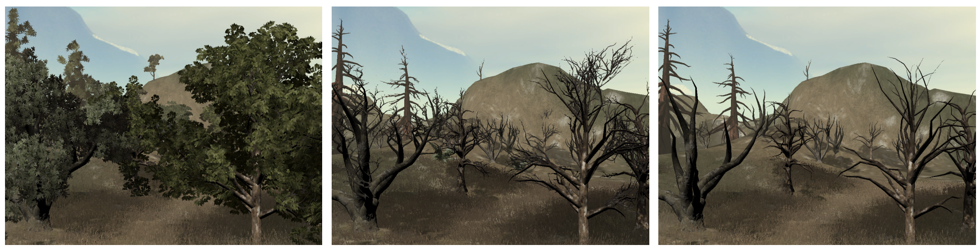

Figure 1.1: Screenshots from SpeedTreeRT demo application. From left: full tree, removed leaf clusters, removed leaf clusters and fronds

The presented contributions will hopefully help toward new solutions for rendering of naturally looking trees closer to insect scale then presently done.

One should note that the approaches utilized in the presented contributions, are in concept opposite to mesh simplification. For mesh simplification one starts with a high resolution model that is simplified. The presented approaches start with a low resolution model that is increased in resolution and amount of detail as needed.

The main contribution presented in this thesis is contained in five published peer-reviewed conference papers. The following section gives an overview of the papers, the abstracts and their place of publication. Note that the publication of Eidheim et al. [2005] is added as an appendix, as this paper is only partially relevant to the main focus of this thesis.

In Figure 1.2 an indication of the connections between the different contributions are given for an even better overview. In the figure each box depicts a single article, while the arrowed lines between the boxes depict the influences one paper has on another. The thickness of the lines gives an indication of the amount of influence a paper had on another, and can therefore be considered as a suggestion in which order to read the contributions in order to obtain as good an understanding as possible.

1. Skjermo, J. and Eidheim, O. C. (2005). Polygon mesh generation of branching structures. In Kälviäinen, H., Parkkinen, J., and Kaarna, A., editors, *SCIA, Image Analysis, 14th Scandinavian Conference, SCIA 2005, Joensuu, Finland, June 19-22, 2005, Proceedings*, volume 3540 of *Lecture Notes in Computer Science*. Springer

   Abstract: We present a new method for producing locally non-intersecting polygon meshes of naturally branching structures. The generated polygon mesh follows the object‘s underlying structure as close as possible, while still producing polygon meshes that can be visualized efficiently on commonly available graphic acceleration hardware. A priori knowledge of vascular branching systems is used to derive the polygon mesh generation method. Visualization of the internal liver vessel structures and naturally looking tree stems generated by Lindenmayer-systems is used as examples. The method produces visually convincing polygon meshes that might be used in clinical applications in the future.

2. Eidheim, O. C., Skjermo, J., and Aurdal, L. (2005). Real-time analysis of ultrasound images using gpu. In Lemke, H. U., Inamura, K., Doi, K., Vannier, M. W., and Farman, A. G., editors, *CARS 2005 - Proceedings of the 19th International Congress and Exhibition*, pages 284–289

   Abstract: Analyzing the increasing number of medical images stemming from various imaging modalities is a time consuming task for the clinicians. Presently, medical images are mainly hand-segmented and manually analyzed. Automatic tools are needed to make medical image analy-

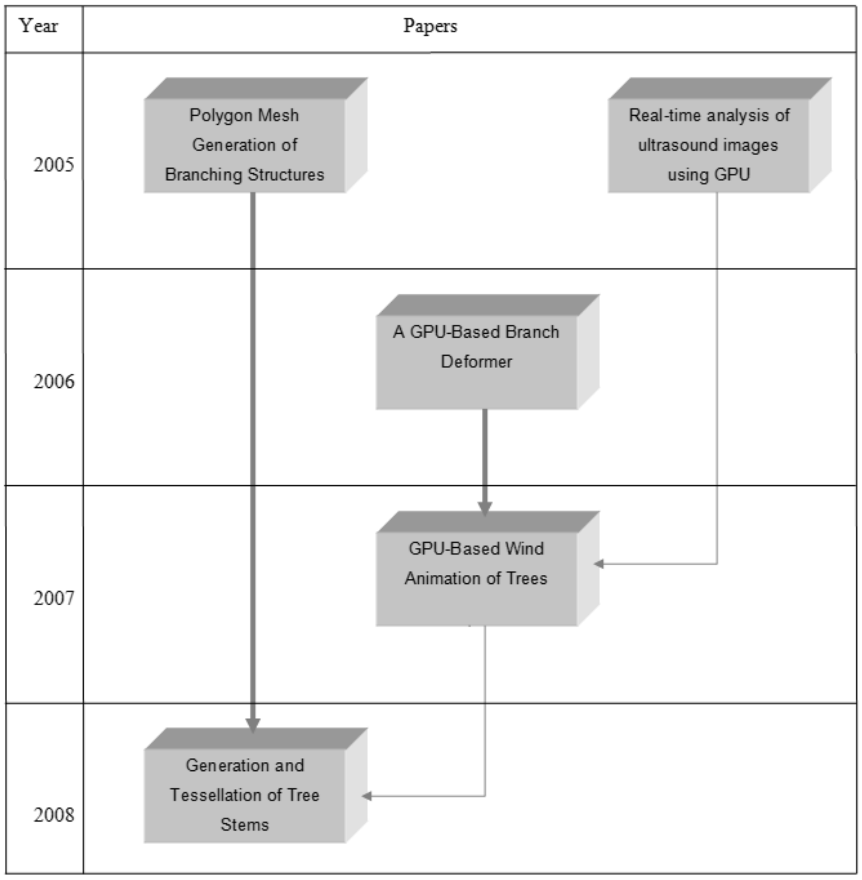
Year
Papers
2005
2006
2007
2008
Polygon Mesh Generation of Branching Structures
Real-time analysis of ultrasound images using GPU
A GPU-Based Branch Deformer
GPU-Based Wind Animation of Trees
Generation and Tessellation of Tree Stems

Figure 1.2: Overview of published articles and their influence on each other

sis easier, and to create better presentations of information needed prior to and during surgery. In this article, we focus on real-time analysis of ultrasound images, and on how inexpensive graphics hardware can be utilized to accelerate current image processing techniques. Algorithms evaluated on the GPU were mathematical morphology, gradient vector flow, and the snake model. The results are promising and show that a large number of iterations can be run in 1/24 of a second.

3. Skjermo, J. (2006). A gpu-based branch deformer. In *Proceedings of the 22nd spring conference on Computer graphics*, pages 120–127. Comenius University, Comenius University, Bratislava

Abstract: This paper present a fast algorithm for generation and visualization of natural looking tree stems and branches. The outlined approach harness the power of the latest programmable consumer graphic cards to achieve real-time performance while still being able to visualize a large number of individual branches, with a high level of detail.

4. Skjermo, J. (2007). GPU-Based Wind Animation of Trees. In Lim, I. S. and Duce, D., editors, *Theory and Practice of Computer Graphics*, pages 29–36, Bangor, United Kingdom. Eurographics Association

   Abstract: This paper present a simplified approach to wind animation of natural looking tree stems and branches. The presented approach is composed from several earlier works by a number of authors, each adapted to increase its suitability for processing on a Graphic Processing Unit (GPU). The outlined approach uses two passes through the GPU. The first pass samples from a simple wind force simulator based on sine sums. It then animates the parameters and the control points defining each branch using the sampled force, taking advantage of the parallel nature of GPU‘s. The second pass uses a previously presented GPU-based deformer to generate and render actual models of each branch, using the animated control points.

5. Skjermo, J. (2008). Generation and Tessellation of Tree Stems. In Lim, I. S. and Tang, W., editors, *Theory and Practice of Computer Graphics*, pages 163–166, Manchester, United Kingdom. Eurographics Association

   Abstract: When visualizing tree stems and branches for use in interactive applications, the polygon models resolution are usually as low as possible to achieve a high frame rate. Also, to ease animation and mesh generation, each branch of a tree model is often considered as a distinct mesh. However, by using a single watertight mesh for a tree, together with (GPU-based) tessellation, both the resolution and appearance of a tree can be greatly improved while maintaining a high frame rate. This paper presents concepts, ideas and early work on generating watertight polygon meshes of animated trees stems suitable for refinement and tessellation of such meshes.

1.4 Thesis Outline

This thesis consists of three parts. In the remainder of the first part, Chapter 2, related work in the field of tree rendering is considered in light of the problem definition given in Section 1.2.

In Chapter 3, a research summary on the contributed papers is given, and considered in the context of the related works.

In Chapter 4, the results presented in the contributions are summarized in the context of the problem definition and related works, and some propositions for future work are given.

Part two contains each of the actual contributions, while part three contains an Appendix. The appendix contains the paper Eidheim et al. [2005]. This paper is only partially relevant to the work presented in this thesis, as described in Section 2.4.

Chapter 2

Related Works and State-of-the-art

The main focus areas identified in Section 1.2 were tree polygon mesh generation, tree animation and tree rendering. This chapter presents related works and concepts with a focus on these three areas, especially in the context of real-time applications.

2.1 Tree Polygon Mesh Generation

For generation of plausible polygon mesh models of botanical trees one could consider using modelling software. However, if one wants to generate a large number of distinct tree models, to increase the realism of the virtual world, one needs an underlying model to automatically generate the tree models. The two most used approaches for this are Lindenmayer Systems (L-systems) and parametric systems. As these systems form the basis for the polygon mesh generation approaches used in the contributions presented, they are briefly presented here for clarification.

2.1.1 L-system

L-systems, as presented by Lindenmayer [1968] is basically a variation of formal grammars, that is, a set of symbols, a set of rules operating on these symbols and a string containing the initial symbols. The first L-systems were so-called *context-free systems.* For context-free L-systems, each symbol in a string could only be changed based on the symbol itself. For instance, an often used example of an L-system that generates a Fibonacci sequence can be seen in Equation

2.1, were n is the recursion number.

$$\begin{aligned} symbols &: AB \\ rules &: (A \rightarrow B), (B \rightarrow AB) \\ start &: A \\ \\ n = 0 &: A \\ n = 1 &: B \\ n = 2 &: AB \\ n = 3 &: BAB \\ n = 4 &: ABBAB \\ n = 5 &: BABABBAB \end{aligned} \tag{2.1}$$

In *context-sensitive systems*, one can also take the previous and/or next symbol into account. This makes it possible to simulate propagation of signal through the string of symbols, for instance to simulate growth modelled as activity of buds at discrete time intervals.

Another variation is a *stochastic L-system* that enables probability based variation. This is done by applying a probability for each of a set of rules that can be applied for a symbol, enabling more variation to the produced sequence of symbols. In *parametric systems*, the symbols can have parameters associated with them. One can then have rules that change the parameters in addition to only changing the symbols, introducing even finer control.

For tree mesh generation, the resulting string from an L-system can be interpreted geometrically by using LOGO style turtle graphics (see Abelson and DiSessa [1981] for details). The basic concept, as presented by Prusinkiewicz [1986], is to interpret the tokens in the L-system generated string as commands to a LOGO style turtle. Typical interpretations are, for instance, 'rotate present state by a given angle', or 'move in the present direction by a given length'. For parametric L-systems, the parameters are used to give greater control over the interpreter, for instance to allow change in diameter or lengths when moving the turtle.

The general concept of turtle graphics can easily be extended into 3D by interpreting tokens as commands that operate in 3D instead of on a plane. These approaches to using L-systems and turtle graphics, for tree generation and rendering were popularized in the book *The Algorithmic Beauty of Plants* by Lindenmayer and Prusinkiewicz [1990].

To get branching structures like trees, the notion of brackets are used. A bracket is interpreted as either a push command to push the present turtle state onto a stack, or a pop command to set the present turtle state from the stack.

To simulate the bending of a plant's branches due to gravity, wind or even increased growth toward a light source, a *tropism vector* can be introduced. When drawing each segment (while interpreting the produced string), the turtle is rotated slightly in the direction of a predefined tropism vector, and thereby simulating bending due to influences such as growth towards light, wind and simple obstacles.

The concept of *query modules* was introduced in the work of Prusinkiewicz et al. [1994]. When interpreting a produced string, a query module would return values depending on the turtle's present position and rotation in the environment, making the process sensitive to its environment. This was used to make environmentally sensitive L-systems that simulated pruning of trees and shrubs.

The query module concept was further extended by Mech and Prusinkiewicz [1996], to enable environmentally sensitive systems that would compete for space. Such competition could for instance be roots competing for water in the soil, or competition within and between trees for access to light.

One should note that in the literature referred to here, the turtle graphic approach is used to produce polygon meshes during the interpretation. However, the token string can also be interpreted to produce a centreline data definition for a tree instead of a mesh. A branch segment is then typically given as a line and its radius, and all the segment data for a tree are ordered in a directed acyclic graph (to give branching).

2.1.2 Parametric systems

In the work of Honda [1971], a model that defined the skeleton of a tree from parameters was introduced. This was used to produce one of the first computer simulations for the modelling of trees where branches were drawn as lines. In this model, a tree was defined by a limited set of parameters for branch lengths, change in lengths as one traverses further out in the tree and angles to use when a branch splits into 2 child branches.

Similar, but somewhat more complex parameters were used by Weber and Penn [1995] to generate polygon models for a wide range of trees. The main concept was that a tree consisted of several *levels*. The main stem was level 0, monopodial branches growing out of the main stem was level 1, and so on. In addition, a stem or branch could split into two or more dichotomous branches.

A tree species was defined from parameters such as branch lengths, radius, curvature, child branches distribution. These parameters were given for each level. One would also define parameters for leaf distribution, pruning envelopes, tropism parameters and envelopes giving the global overall appearance of the trees. For each parameter one would also give in a variation range that was used when generating individual trees. In all, over 30 parameters were used to define a tree species. Individual trees with unique appearance could then be generated by sampling from within parameters given variation ranges using random values. LOD was handled by drawing pixels and lines instead of polygons of leaves and branches when the tree was far away from the observer.

A commercial middleware solution for use in real-time applications like games, SpeedTreeRT [2008], has been used in several hi-profile game titles the latest years. In SpeedTreeRT the concepts used by Weber and Penn are further refined by giving more control over the parameters. Control is increased by letting the user that defines a tree species use Bezier curves to control several aspects of the parameter space. To increase the visual aspects of the trees inner structure, textures depicting smaller branches (called *fronds*) are attached to the main branches. Leaves are handled by applying billboards containing textures of leaf clusters to the main branching structure and the fronds.

In the SpeedTreeRT software, LOD is not handled by mesh simplification, exchanging the mesh with lines and pixels, or billboard clouds. Instead LOD is handled in an approach that is somewhat similar to the concept of a hybrid approach between mesh simplification and billboard clouds. The mesh generated is usually already of low resolution if targeted for real-time applications. LOD is handled by removing fronds and single small branches as the distance to the observer increases. Leaves are rendered using billboards that depict several leaves (a so-called leaf cluster). The LOD algorithm fades out these billboards and replaces them with fewer billboards that are scaled up as the distance increases. The fading is done while simulating alpha transparency via dissolve as described by Whatley [2005]. Using such an approach drastically simplifies animation of the trees compared to using billboard clouds or hybrid approaches, as one do not have to re-generate the billboard clouds for each timeframe.

One should note that although a polygon mesh is directly generated by the methods presented here, they can also be used to produce strictly tree centreline data, as for L-systems.

2.1.3 Polygon mesh generation

Early work on tree rendering usually focused on branching patterns. The generated trees therefore usually lacked details, especially when viewed at a close distance. For instance, trees generated by L-systems were usually rendered by drawing a cylinder for each branch. If a branch consisted of several segments, each segment was rendered by drawing a cylinder. To avoid gaps between branch segments, one would draw a sphere at the segments joints, with the same radius as the largest cylinder. This resulted in trees that had straight branches and poor appearance when viewed up close.

The first attempt to rectify this was presented in "Modeling the Mighty Maple" by Bloomenthal [1985]. In this work, branches were handled as generalized cylinders. A circular cross section was swept along the curves of an underlying tree skeleton, while varying the cross section's radius. At a point of bifurcation, a ramiform (a free-form surface) was used to connect the branches together. By using a bark texture and a bump-map generated from x-ray of real bark, a polygon mesh model could be generated that could be rendered in good detail even when viewed at a close distance. However, to get good results, the underlying tree skeleton had to be handled as interpolating curves that had to be C^2 continuous at any branch point. Also, the method was not easily extended to handle arbitrarily complex ramiform structures (for an arbitrary number of branches).

For parametric systems, the situation has been somewhat different. When using the approach by Weber and Penn [1995], one can produce polygon meshes that can have quite a high resolution and have good details even when rendered at a human scale. However, the approach produces distinct polygon meshes per branch. A child branch is attached to its parent branch simply by positioning the start of it inside the parent branch. This results in that the polygons of the branch intersect the polygons of its parent branch.

Some approaches have been developed to generate a single polygon mesh of a tree, that could handle more arbitrary branching structures. One such approach was rule based mesh growing with L-systems, as introduced in the work of Maierhofer [2002]. This algorithm used polygon mesh connection templates for adding new parts of a tree to the mesh model, as L-system productions were selected during the generation phase. The mesh connection templates were made in such a way that they produced a coarse polygon mesh that would serve as a basis mesh for Catmull-Clark subdivision (Catmull and Clark [1978]). The coarse mesh was fully watertight and produced smooth branching areas after a few steps of subdivision. However, the approach could only grow the mesh by rules, not generate a mesh from a finished branch centreline data set.

Another method for generating a single polygon mesh of a tree is the *refining by intervals* approach presented by Lluch et al. [2004]. In this method the area around a branching is refined by adding contours at the meeting point of a ramification (between a branch segment and its child segments), equidistantly spaced in height, until the segments are completely separated. The contours are then connected by polygons, generating a mesh for the ramification area that is watertight. The approach was used to generate a single polygon mesh of a tree generated by an L-system. However, the approach did not work well with all angles between a parent and a child segment, especially if a child segment was at a 90 degree angle from the parent segment's direction.

A more recent approach is based on Constructive Solid Geometry (CSG) as presented in the work of Galbraith et al. [2004a]. In this approach, named *BlobTree*, a basic tree skeleton (generated using an L-system) is used to produce implicit surface primitives and operations. Using different operations one could generate effects such as bark ridges around branching areas, as often seen in nature, producing good results even if operating on a near insect scale. The BlobTree approach was later used to render parametric defined trees, where the growth could be simulated by inverse modelling as shown by Galbraith et al. [2004b]. Although the BlobTree approach generated impressive visual results, solving the implicit surfaces is computationally expensive, so the approach is less suitable for real-time applications.

2.2 Animation

Most works on animation of trees for real-time applications have focused on the influence of wind. A typical approach is to apply physics simulation techniques based on the equations of motion to generate the branch motions. In this section some important works on animation of trees are presented, both approaches based on physical animation techniques and others.

In early L-system based tree rendering, wind was sometimes simulated by looking at influence from wind as a changing tropism vector. However, this only works if the L-system string is re-interpreted for each change in the wind vector.

In the work of Jirasek et al. [2000], biomechanical concepts were introduced into L-systems. This enabled one to capture the impact on the shape of branches due to gravity, tropism, contact with obstacles and breakage due to excessive stress, in a more controllable way then through the classic tropism vector based approach. This was further examined by Taylor-Hell [2005], together with some preliminary investigation on the dynamics problem of representing oscillatory motion of tree branches for wind animation, by interpreting the branch centrelines as joined springs.

Another approach for animation of L-system based trees was presented by Giacomo et al. [2001], where both a procedural and a physics based approach were used. The procedural approach was used for most of the trees being rendered, while the physics based approach was used for trees where the user should be able to interact with the tree (pull on a branch and it should bend). These methods were blended between as required. For the physics-based approach, branches were represented as rigid rods, connected by joints where torques were modelled using linear damped angular springs.

In the work by Kanda and Ohya [2003], an approach was presented that animated tree models using branches that consisted of several segments. Instead of calculating the motion on each branch segment, motion was calculated only for a single *representative* segment per branch. Calculating the orientation of the representative segment was done by simulating a branch as a pendulum in the plane. The rest of the segments in a branch were orientated based on the assumption of uniform orientation difference between adjacent segments.

In Stam [1997] it was noted that the nature of wind influence on trees can be seen as chaotic in nature. Periodic motions for trees can therefore be synthesized directly instead of synthesizing the wind field to drive the motion. The pre-calculated motions could then be stored in simple lookup tables and applied as needed, removing the need for time-based integrating. Specific frames of an animation could then be calculated directly without simulating everything from the beginning.

A similar approach was used by Ota et al. [2003]. A noise function generating noise of a type sometimes observed in natural blowing winds was proposed used to pre-calculate noise. Motion for the individual branches and leaves was calculated based on interpolation of the generated noise data. The final branch animation was handled by a simple simulation method based on the spring model and the noise.

2.2.1 Wind animation using the GPU

An early attempt to utilize the improved calculation performance introduced with the GPU for wind animation of trees was presented by Wesslén and Seipel [2005]. In this approach wind influence on trees generated with the parametric approach of Weber and Penn was animated on the GPU using two simple methods, swaying of the trunk and fluttering of leaves. Leaf fluttering was animated by quick rotation around the spine of the individual leaves. Swaying was a periodic rotational motion around the origin of the tree trunk. The amount of rotation was relative to the distance from the ground. However, the swaying was confined to a single direction, and the branches did not move independently of each other as the swaying was uniform over the whole tree.

Another approach was presented by Singh et al. [2005], who animated palms using the GPU. In this work, classic matrix skinning was used to animate wind influence on the trees. For matrix skinning, the stem and each branch were simulated using three bones each. Animation was applied by skinning matrices procedurally calculated based on the wind force. The approach could also be applied to planar billboards as long as the billboard objects consisted of enough vertices to bend with skinning techniques. This enabled easy LOD not only on tree models of different mesh resolution, but parts of the model or even the whole model could in concept be switched with billboards while still being animated using the same approach. The method did however only consider the wind influence on palm trees of simple complexity, that is, one main stem with a few palm leaves attached at the top of the stem.

One of the latest works on tree animation using the GPU was presented by Sousa [2007], the approach used in the popular computer game *Crysis*. In this work animation was applied as a two-step process. The first step was to animate the entire vegetation object along a wind vector's direction. This was done by displacing the vertices along the direction of the wind vectors x-y component. The deformation was scaled using the mesh height, and the wind's strength. Also, the distance to the vegetation object centre was used to rescale the displacement of each vertex, resulting in a spherical limited motion. The other step was to animate the vegetation object's leaves. For leaf displacement only the strength of the wind was considered. The displacement was scaled based on leaf stiffness, which was encoded into the leaf textures. One channel of the texture was used to give the stiffness for displacing in the x and y directions, another channel gave stiffness used when displacing in the y direction. The resulting approach gave good visual results for a limited range of wind forces. However, turbulence could not be simulated in a single vegetation object, because of the simplified approach.

2.3 Rendering and Level of Detail

The main motivation for rendering trees in real-time applications is to immerse the users in an environment that looks as natural as possible. Especially if the goal is to give the appearance of a forest environment, the number of trees to render can easily overwhelm even the most modern graphic engine. Efficient

LOD algorithms are therefore of great importance for rendering trees in real-time applications.

For image-based approaches efficient LOD can often be achieved by reducing the number of polygon planes used for the billboard clouds, for instance as shown by Garcia et al. [2005]. If image based approaches are of interest, one commercial product of interest could be *Bionatics NatFX and RealNAT* (see NatFX [2008]). This software specializes in tree generation for image-based approaches.

However, as the focus is on trees rendered as polygon meshes, other approaches are needed. In the work on parametric generation and rendering of trees by Weber and Penn, level of detail was handled by drawing branches and leaves as lines and pixels instead of polygon mesh as the distance to the observer increased. However, these days GPUs are actually faster at rendering triangles then drawing lines, so such an approach is of less interest then previously.

Another approach would be to use a polygon mesh simplification method. Although for tree rendering this is not optimal, as the approaches for generating polygon meshes of tree stems and branches usually can give out meshes of different resolutions directly. For instance, the Xfrog software, Xfrog [2008], as described by Deussen and Lintermann [1997] and Lintermann and Deussen [1999], can generate tree meshes of desired resolutions. This software differs from others mentioned, in that they only try to get 'visually correct' shapes of plants and is not based on a biological model.

For hybrid approaches, an early work on LOD for real-time rendering of a biological environment by Perbet and Cani [2001] is of interest. In this work grass swaying in the (procedural simulated) wind on large areas of prairie was successfully rendered using different approaches for LOD. Basically, the environment was divided into three regions where the grass in each region was handled by individual methods. For the area closest to the observer the grass was handled as geometric models. For the medium area grass patches were handled as a volumetric texture mapped onto polygon strips (2.5D), and for the far area static 2D billboards were used.

To get smooth transitions between the areas as the observer moved around, two separate approaches were used. For the transition between the near and middle area each individual blade of grass was linearly interpolated with its position in the 2.5D map. The transition between the middle and far area was done by an image dissolve at the same time as the 2.5D map shrunk into the ground, while growing the billboards out of the ground. Opposite if the distance to the observer was increasing. This approach has similarities with the approach used in the SpeedTreeRT middleware, where some leaves are dissolved out while the size of other is increased to maintain the overall leaf coverage.

Another take on LOD is to de-couple the representation used for animation from the appearance of the trees, usually called Simulation Level of Detail (SLOD). When using distinct levels in a LOD model one might introduce large noticeable changes when switching level, that is, part of the model might 'pop' in or out. For SLOD the same holds for sudden changes in the simulation level, introducing large positional changes. An early use of SLOD in tree rendering was shown by Beaudoin and Keyser [2004]. In this work, the simulation was done using a tree structure in addition to the structure that described the tree's appearance. The main structure used for rendering was simplified to produce simpler structures for different detail levels to use for animation, and interpolation between these structures was used to avoid simulation *popping*.

As earlier stated, when working with polygon mesh based approaches for tree rendering in real-time, one usually uses polygon meshes of very low resolution. This implies that instead of looking at classical LOD were one starts with a high resolution polygon mesh and simplify this, one could look for approaches that improve rendering quality when using simple low-resolution meshes, as one goes toward the human and even insect scale. Some early work on this has been done by Kharkamov et al. [2007] for the SpeedTreeRT software. Texture based approaches like parallax mapping was used to increase the visual quality when approaching human and insect scale. Although this seems to generate good results, the contributions and proposed directions for future work presented later in this thesis instead look at approaches that allow increased mesh surface detail. Texture based approaches could then be applied in addition, to increase the appearance even more as one goes toward insect scale.

2.4 General Purpose Calculations on the GPU

In the contribution of Skjermo [2006], as presented in Section 3.2.1, a general purpose stream-based approach is used to animate trees swaying in the wind. The contribution in Eidheim et al. [2005], an example of general purpose calculation using the GPU, is given in Appendix A. A short explanation is also included here for clarification, as the approach and methods are directly applied to the work presented in Section 3.2.1.

General purpose calculation on the GPU was first developed for use in medical image analysis. Using CT, MR, ultrasound, and other image acquisition equipment has revolutionized diagnosis of numerous diseases. However, the amount of data produced by such sources can be staggering, so efficient methods for analysis of such data are of interest. Some tasks of such an analysis are of a parallel nature, and some can even be solved using the *stream computer paradigm* on a GPU, as described by Venkatasubramanian [2003].

The GPU can be seen as a stream processor that can achieve a high degree of parallelism when used appropriately. The main concept, as described by Göddeke [2005] is to use a one-to-one mapping between input data size (in textures) and the output fragments written. The GPU can then compute operation on each fragment using fragment shaders and output into a new texture. If needed, one can then *ping-pong* the output and input textures so the output becomes input for the next iteration.

Using the GPU as a stream co-processor for suitable task has gained popularity for offloading the CPU, decreasing computation time or usually both. The main problem with this approach is to identify what algorithms or part of algorithms in medical image analysis that are suitable for being handled by stream architectures. An additional problem is that a GPU cannot scatter information during (texture) writes, only gather information during reads, thus only pure iterative parallel stream computation without the need for intercommunication can be done. However, this and other limitations have been removed on newer hardware, and new programming languages like Nvidia's CUDA have simplified the process of writing parallel applications running on GPU.

One example of general purpose computation on the GPU is the work in Eidheim et al. [2005], as given in Appendix A. In this work, the main contribution was twofold.

First, by stating that the two most computationally expensive stages in medical image analysis are commonly pre-processing and segmentation, several commonly used image analysis algorithms are identified as candidates for an implementation where the GPU is used as a stream co-processor.

Secondly, as an example, it showed how to implement the *snake* algorithm that incorporates several of the identified candidate algorithms. The implementation is then used for analysis of both ultrasound images of the heart (locating the left ventricle wall) and ultrasound images of the liver (segmenting a tumour). This particular partial stream implementation was shown to perform up to 34 times faster than a normal single CPU implementation (using 512x512 pixel images).

Chapter 3

Research Contributions

In this chapter the contributions are presented in the context of the problem definition given in Section 1.2, and related to earlier works in the field as presented in Chapter 2.

3.1 Polygon Mesh Generation

One problem common for many algorithms that produce mesh models of biological tree stems is the way branching is handled. Typically, a simple workaround is used where a branch is considered to start inside its parent branch, and the branch is not part of the same polygon mesh as the parent branch. The visual result of this approach is interpenetration between meshes, as seen in Figure 3.1. Not only does this approach not generate visually convincing results for static tree models, but it gets even worse if a child branch moves relative to its parent branch. This is a common occurrence when doing wind animation. The visual result of this problem is that the visible surface area of a child branch, near the branching point, will change as the branches are moving with respect to each other. It also appears as if the child branch does not grow out of the parent branch as it will 'wander' over its surface.

3.1.1 Polygon mesh generation of branching structures

One approach to avoid such problems is to actually generate polygon tree stem meshes that are fully connected or *watertight*. Several methods for generating watertight polygon meshes of tree stems have been proposed by others, as noted in Section 2.1.3. The approach of *rule based mesh growing* as presented by Maierhofer [2002], used a template based L-system that connected mesh

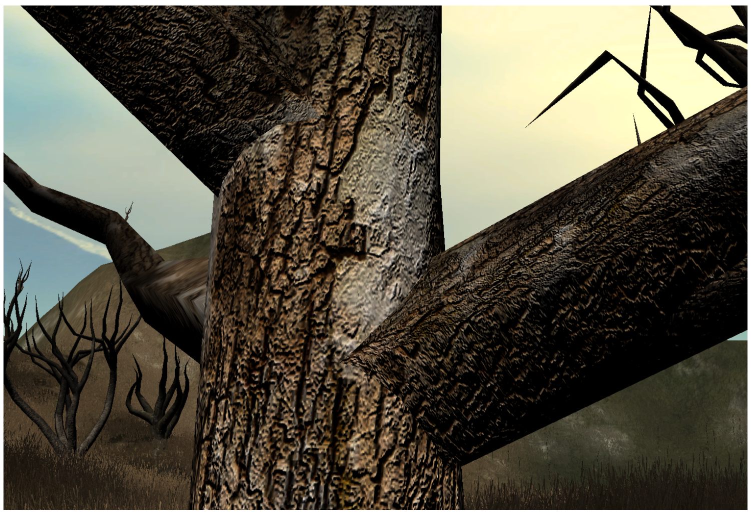

Figure 3.1: From *The Valley* SpeedTreeRT demo application SpeedTreeRT [2008], coarse tree mesh where child branches starts inside parent branch

segments (the templates) of the stems into a single polygon mesh for the whole tree.

This mesh was then further refined using subdivision [1]. This approach was however limited by the templates themselves. In other words, as the approach connected together selected templates to build the final mesh, the number of possible appearances were limited by the possible permutations of the templates. A more general, less limited approach that could be applied as a post-process step based on a tree description would be preferable.

An approach somewhat inspired from rule based mesh growing was the work presented in Felkel et al. [2004]. This was a fast and simple approach for generating a polygon mesh of a branching tube structures suitable for further refinement using subdivision. This approach generated template like mesh structures that was connected into a single mesh, based on a centreline data set. However, this approach was limited in that it would in some cases generate polygon meshes that intersected locally near branching areas if not manually tuned.

Starting with the method proposed in Felkel et al. [2004], a closer look at the generation approach was taken with respect to the need for manual tuning. The original approach considered vascular transportation systems (blood vessels), but biological trees show resemblance with vascular transportation systems.

[1]See Zorin et al. [2000] for a comprehensive introduction to subdivision methods.

When considering vascular transportation systems, Leonardo Da'Vinci defined a simple rule describing how branching would naturally behave in such systems, as described by Richter [1970]. The basic rule was that the cross-section area of a segment just before branching is equal to the combined cross-section area of the child segments just after the branching. The Da'Vinci rule was further refined in Murray [1927], to directly calculate the angles child segments diverge from the parent segment after branching.

Based on Murray's refined rules, simple trigonometry was used to generate a formula for calculating lengths and angles needed during mesh generation, to avoid local intersection. This does however imply that one will avoid generating self intersecting meshes only as long as the centreline data set used more or less conforms to the rules mentioned.

The method proposed in Skjermo and Eidheim [2005] generates the mesh templates using the Da'Vinci rules based on the given centreline data. The mesh templates are then connected into a single mesh for the whole structure and can be subdivided using Catmull-Clark subdivision. An example of this approach can be seen in Figure 3.2 and 3.3.

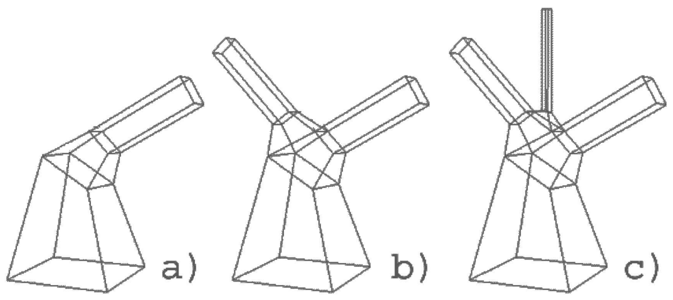

Figure 3.2: Adding branch mesh templates together to produce a single polygon mesh. a) Adding continuing branch segment. b) Adding first child branch segment. c) Adding another child branch segment.

The approach was tested by visualizing tree stems using centreline data from a tree descriptions generated by a simple L-system (blood vessels using centreline data generated from CT scans of liver was tested by the papers co-author).

For both the blood vessel and the tree rendering, only static models were considered. For both approaches, it was assumed that the refined rules hold true. An example of an L-system that generates tree descriptions that follows such rules can be found in Linsen et al. [2005].

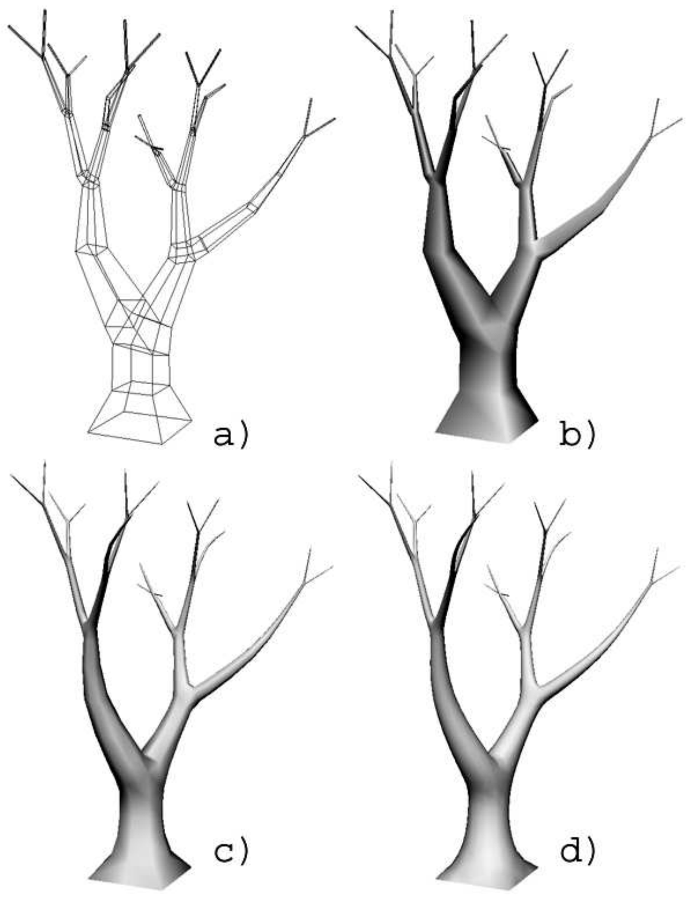

Figure 3.3: a) final connected polygon mesh, b-d) shaded mesh subdivided to level 0-2.

In this contribution texture mapping onto the polygon mesh was not considered, since the main focus was on the construction of the polygon mesh.

3.1.2 A GPU-based branch deformer

As the computational power and functionality of GPU increases, new methods and approaches to take advantage of this are needed. Also, many already existing approaches can be adapted to take advantage of the massive parallelism of the GPU.

However, taking advantage of the possible increase in performance might not be straightforward. Several limitations must be taken into account, such as limited transfer speed and bandwidth, latency when reading from texture memory, limited performance when using conditionals, no data scatter approaches and so on. How to actually take advantage of this increased power for mesh generation of tree objects might not be obvious, especially with the limited functionality of the early GPU.

A single watertight polygon mesh of the stem and main branches of a tree can give significant visual improvements, when compared to using separate meshes for each branch. However, being able to render a high quantity of branches in real-time is also of importance. Usually, for use in real-time applications, one would generate tree polygon meshes as a pre-processing step before rendering. However, in some cases one could have need for on-the-fly generation of tree meshes. For instance if one wants to have individually unique trees, pre-generated meshes would require more storage for each additional tree.

Considering the approach of using on-the-fly generated distinct meshes for each branch, finding methods to offload the CPU using the GPU seems somewhat simpler. The parametric system proposed by Weber and Penn, as described in Section 2.1.2, produced trees with separate meshes for each branch and stem. Also, the approach gave very good control of the appearance of each individual branch, and therefore seemed as a good candidate for adaption to run on the GPU.

In Skjermo [2006] such an attempt to optimize the approach of Weber and Penn is presented. The parameters used to generate the mesh remains as originally defined. However, the interpretation of the parameters is restructured to simplify the process. By looking at the mesh generation step as a deformation of a planar template mesh into a cylinder, then into a branch shape, the resulting approach can be optimized for the GPU.

From the tree description parameters, a set of four control points are generated for each branch, that defines its centreline. The control points are stored in textures for use in a vertex shader program. Each set of control points is used to deform an input mesh using functions describing the shape of a branch. The shape is defined from given parameters for flare, bulges, taper and lobes. The approach to generate flare, bulges, taper and lobes is adapted from the original work by Webber and Penn, as their approach was heavily dependent upon conditional branching.

After the shape of a branch is defined, it was deformed to bend naturally using the generalized de-Casteljau approach as described by Chang and Rockwood [1994]. This approach is basically an early approach to *free-form deformation*, that was found to cover the need for control while not being too computationally costly.

When using the generalized de-Casteljau approach, a normal will not transform correctly as the mesh deforms. Therefore, for an input vertex, a tangent was generated using an estimate for the partial derivatives in the direction of the growth, and a bi-tangent was generated using the partial derivatives in the direction of the circumference of the branch at each vertex. The Jacobian of the de-Casteljau was then used to transform the tangent and bi-tangent as the mesh deformed. This follows the overall approach for a 'Deformer' as described by d'Eon [2004].

The final result was a vertex shader that efficiently generated meshes of the bent branches with flare, bulges, taper and lobes from per-branch parameters. This was used to draw trees with a high amount of branches, while most of the workload was handled by the GPU. This initial work did however not support splitting of the stem. In Figure 3.4 a screenshot of trees with a large number of branches can be seen, while Figure 3.5 shows rendering performance for number of branches versus the number of vertices (using a Nvidia GeForce 6800 graphic card).

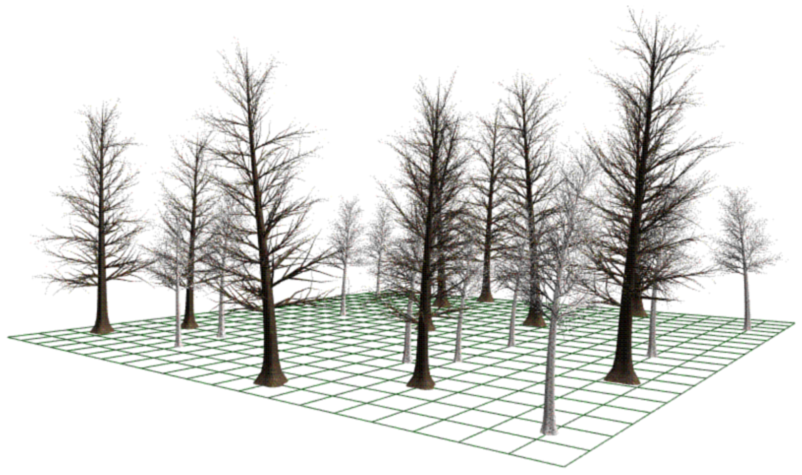
Figure 3.4: A small forest with large number of branches per tree. From Skjermo [2006].

3.2 Wind Animation

When animating natural occurring phenomena, animation by physics-based simulation methods usually gives good results. Mostly it is such that the better the physical approximation is of the real world, the better the results

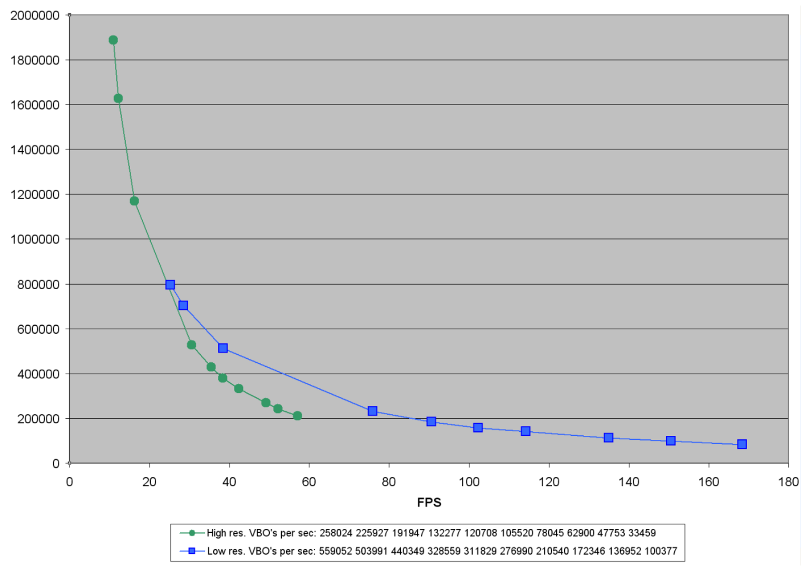

Figure 3.5: Rendering performance for number of branches (vertex buffer objects) versus the resolution (number of vertices. From Skjermo [2006]. The height gives the number of vertices.

appear to the human eye, as we are well trained in observing nature. However, simplification of the rules of physics are required both from a complexity point of view, as we do not have the required processing power, and from the fact that we often do not have a complete picture of how the real world actually works.

In real-time applications one might divide between *effect physics* and *game-play physics* (see Luban [2007] for a good discussion on the need for game-play physics). With effect physics one usually means that physically based rules are applied to generate aesthetic/cosmetic effects. After these effects are applied they do not influence anything else in the virtual world. This often gives a clear avenue for simplification of the rules of physics, in contrast to a game-play physic-based approach where changes can influence other entities in the virtual world.

In the context of tree rendering, an example of effect physics would be trees that sway in the wind without influencing the game world in any way other then purely aesthetic. An example of game-play physics could be a tree that falls over (typically from a player or game AI generated event). The falling tree would then impact the game world. For instance, in the 2007 game *Crysis* published by Electronic Arts, a player can cut tree stems, and get the trees to fall over enemy soldiers (damaging them) or block roads for enemy vehicles. However, when animating trees swaying in the wind, one can often limit oneself to effect physics and still achieve visually plausible results, as long as the tree objects do not need to be destructible.

3.2.1 GPU-based wind animation of trees

In the contribution presented in Skjermo [2006], a vertex shader for generating and visualizing polygon meshes of tree branches and stems is presented. However, this work did not consider animation of the produced trees. For the original approach to parametric generation of trees by Weber and Penn [1995], the idea was to animate each branch during mesh generation. The authors discuss simulating tree motion as a system of oscillators, but do not provide direct examples.

As the contributed work was based on parametric system as described by Weber and Penn for the branch and stem generation, one could maybe consider the approach by Wesslén and Seipel for wind animation. In that approach wind based animation was handled on the GPU as a periodic rotational motion around the origo of the tree trunk. However, as stated in Section 2.1.2, the swaying was uniform over the whole tree, so this approach did not give an appearance of turbulence between branches inside a tree.

Using noise-based approaches, as proposed by Stam [1997] or Ota et al. [2003] could be an alternative. However, although good wind based animations with internal turbulence can be generated with such methods, it is not clear how to include sudden changes to the wind influence by game-related events.

To animate wind influence on trees where one also gets the appearance of turbulence internal in a tree, and a good control over the actual wind generators, a semi-physically approach was explored. The results of this contribution are presented in Skjermo [2007].

In this contribution, the centreline of a branch was defined by the control points and as such they define the polygon mesh for a branch. These control points were used to define the mesh deformation, and used to animate the trees. As the control points were stored in texture memory on the GPU, a GPU based approach that animated trees by translating the branches control points were developed to simulate wind influence.

By calculating the simulation on the GPU one gains several improvements. First, using the GPU to update the control point's position for each time-step off-loads the CPU.

Also, as the branch meshes are deformed into their final shape in a single pass on the GPU, the wind animation could be added as a separate pass.

This new pass is done before the deformation pass. Because of this one does not need to transfer the control points between the CPU and the GPU for rendering after each time-step of the wind animation, and thereby avoid issues with limited transfer speed and bandwidth.

To calculate the next time-step in the simulation, new positions for the control points were generated using the initial position of each control point, its position in the last time-step (and the position of its parent branch), and a wind force vector. To achieve this on the GPU, a ping-pong texture based approach was used. Such texture memory based approaches were originally developed to utilize the GPU for general purpose calculations. A typical area where such general use of the GPU has had a great impact is in medical image analysis. The actual implementation was based on the work presented in Eidheim et al. [2005], as described in Section 2.4.

The actual simulation of wind forces was based on a Sums of Sine approach. This approach is mostly used to simulate waves on water as mentioned by Finch [2004]. When one simulates waves, Sums of moving Sine curves are used to sample wave heights at a given position. However, for wind simulation one is interested in a force vector instead of height, so the approach was adapted to take this into consideration and produce force vectors. One should note that the approach only generated forces in a single 2D plane, so the wind force at a position was the same at all heights. Another simplification was that the force was only sampled at the tip of each branch.

The animation itself was calculated by the same approach as used for tree animation in the movie Shrek, as described by Peterson [2001]. In this approach, the force sampled at the tip of the branch is distributed over all control points, scaled down as one gets closer to the start of the branch. A control point is then translated in the direction of its contribution of the force, while maintaining the distance between the control points. A screenshot from a test application that uses the proposed sums of sine approach for animation can be seen in Figure 3.6.

One should note that for the calculation of the control points for a branch at time n, the position of the parent branch at time $n - 1$ was used. This introduced animation lag were the branches position was one time-step per level behind where it should have been. The further out in the tree, the larger the time lag between branches.

Even with the simplified wind model and the time lag, the resulting wind animation was quite believable for small to medium wind strengths, especially as one could introduce internal turbulence inside a tree. However, for large scale wind forces the effect was that branches did not seem to be firmly attached or

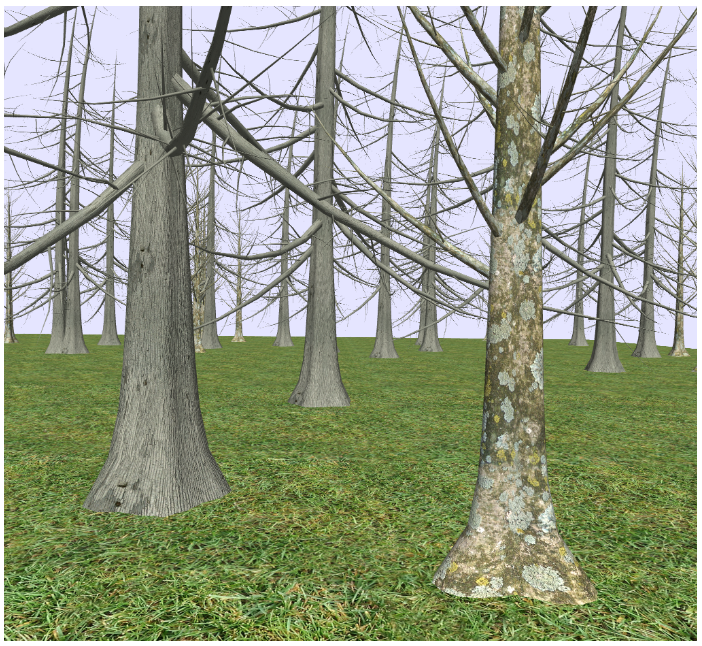

Figure 3.6: A few trees animated using the sums of sine approach. From Skjermo [2007].

part of a tree, but only attached loosely as if by rubber bands.

3.3 Rendering

In the contribution presented in Section 3.1.2, the parametric system based on the findings of Weber and Penn was used to define tree branches and stems. This approach can generate individual branches with enough details and high enough mesh resolution so it might look convincing even when close to human scale. However, this and similar approaches do not generate a polygon mesh that look convincing near the branching areas.

To get better results for the branching areas one could consider using the BlobTree approach of Galbraith et al. [2004a]). However, as noted in Section 2.1.3, this approach is less suitable for real-time applications. Solving the implicit surfaces is computationally expensive, and it is less than obvious how one could use the GPU to increase performance on a large scale. For instance, GPU based implementations of marching cubes would not scale well to handle a large number of trees.

In the contribution on rendering trees animated by wind, Skjermo [2007], the generated results that were purely of the nature of *effect physics*. As the need for realism increases one would like to use game-play physics for the simulation of vegetation, for instance to introduce vegetation that is both influenced by, and influences the virtual world. A discussed in Section 3.2, an example of effect physics would be destructible trees.

Considering this, a problem is how to generate polygon meshes of trees that look good at branching areas when rendered, and at the same time behave as close to reality as possible not only when animated by wind influence, but also when animated with destruction. This chapter considers several different ideas, concepts and approaches for solving this problem. The considerations stem from the contribution given in Skjermo [2008]. The main concepts considered are animation of destructible tree meshes as either rigid body systems or as spring based point-mass systems. The last concept that is considered is tree mesh rendering using tessellation for increased visual quality and LOD performance.

3.3.1 Generation and tessellation of tree stems

In the contribution given in Skjermo [2008] some ideas and preliminary results on generation and rendering of tree stems for use with *game-play physics* are presented. As the paper consists mostly of ideas and preliminary results, some areas of this paper are explained in more depth here for clarification. The paper considers generation and rendering of trees both in the context of rendering around branching areas, wind animation and a first look at destruction.

Using the same physics engine for all the physics simulation in an application is considered an advantage, as this simplifies interaction between different elements of the simulation. When simulating game-play physics for vegetation, it would therefore be an advantage to use the same physics simulator engine as the rest of the application. One such physics engine is the *Nvidia PhysX engine*, PhysX [2008], previously named *Ageia PhysX*.

Two approaches for the simulation were considered using the PhysX engine. The first approach was using rigid body animation based on the tree description before the mesh generation. An introduction to rigid body animation can be found in Witkin et al. [1995].

The other approach was animation of the generated mesh, using methods normally used for animating cloth, as shown by Müller et al. [2006].

Rigid body approach

The rigid body animation approach was in concept similar to the physics based part of the approach used by Giacomo et al. [2001], where each branch was represented as a rigid rod. However, for the contributed work, full rigid body animation was tested using a tree structure made from capsules, as this enabled better collision detection than using simple rods. Each capsule was connected together using joints with springs, which could break if exposed to forces over a given strength, thereby introducing destruction.

After each time-step a polygon mesh suitable for tessellation was generated by taking square cross sections at the beginning and end of the capsules, and connecting these together into a polygon mesh. As such, the polygon generation could be considered as very low-resolution skinning. This resulted in a polygon mesh similar to the one generated by the approach used in Skjermo and Eidheim [2005].

It is noted that using a full rigid body approach with joints between each branch segment quickly becomes computational expensive. A large number of joints do not scale well even if hardware accelerated using the Nvidia PhysX accelerator or Nvidia GPUs. As such, one could consider adding several branch segments (i.e., capsules) together into one single rigid body, reducing the needed number of joints. Even with such a simplification, one would typically only be able to simulate a very limited number of trees. As such, hybrid approaches similar to the work by Giacomo et al. [2001], could be considered to handle more trees.

Cloth-based approach

For the cloth-based approach, a polygon mesh of a tree stem was generated once, and then animated as a piece of cloth using constraint-based physics. In the work by Jakobsen [2003], practical approaches for constraint based physics with Verlet integration for character animation was considered (this approach is called *ragdoll physics*). Jakobsen also mentioned that the IO Interactive game,"Hitman: Codename 47", from 2000 used a similar approach for animation of trees. In Hitman, the centreline of a branch segment was considered a length constraint between the positions of the end points of the line. Also, some additional anchor points were needed to control the simulation. The presented approach differs in that it uses the polygon mesh itself as constraints to maintain the overall structure of the tree and branches during simulation.

In the PhysX physics engine, cloth animation is solved using position based simulation as described by Müller et al. [2006]. In the approach suggested by

Jakobsen [2003] the velocity of each particle was implicitly stored by current and previous position, whereas in the PhysX approach explicit velocities are used, making damping and friction simulation much easier.

For the presented approach each vertex in the mesh is considered a mass particle in a constrained particle system, and each edge of the mesh is considered length constrains, the angles between the edges are considered angle constrains, and an internal pressure is applied. As the generated mesh is watertight, these constraints will work to keep the tree polygon mesh overall structure intact, while still allowing swaying from wind forces and bending from collisions. This is called the *cloth approach* from now on, as the cloth solver in the PhysX engine is used for the simulation.

This approach can be seen as an attempt to simulate a tree model based on the polygon mesh of the model, instead of first simulating based on a tree description and then having to generate the polygon mesh. This differs from earlier approaches in that one only has to generate a polygon mesh once. Although this approach, like the rigid body approach, is limited by complexity, it will hopefully scale somewhat better.

One should note that both the concept of rigid body simulation and of cloth simulation can support destruction through joint breaking and cloth tearing in the physics engine used. However, only some very simple scenarios were actually tested for the rigid body case. One should also note that although the cloth approach assumes polygon mesh models in a form similar to the ones generated by the approach presented in Skjermo and Eidheim [2005], most of the actual models used were hand crafted in 3DS Max for speed of implementation.

Mesh rendering and tessellation

As a virtual observer approaches a tree at the human scale or even goes toward the insect scale, polygon models of large resolutions are needed to render tree stems and branches of a good visual quality (for instance to get smooth silhouettes).

In Skjermo [2008], a coarse single polygon mesh of the tree stem and branches is generated using a similar approach as the one presented in Section 3.1.1. This low resolution mesh is then further tessellated to increase the resolution. One should note that in the original approach, several steps of Catmull-Clark subdivision were used to increase the resolution of the mesh as a pre-processing step before visualization. In the approach discussed here, only 0 or 1 level of subdivision is used. Tessellation can be used to increase the resolution of such a coarse mesh, even if animated, as indicated in the left part of Figure 3.7.

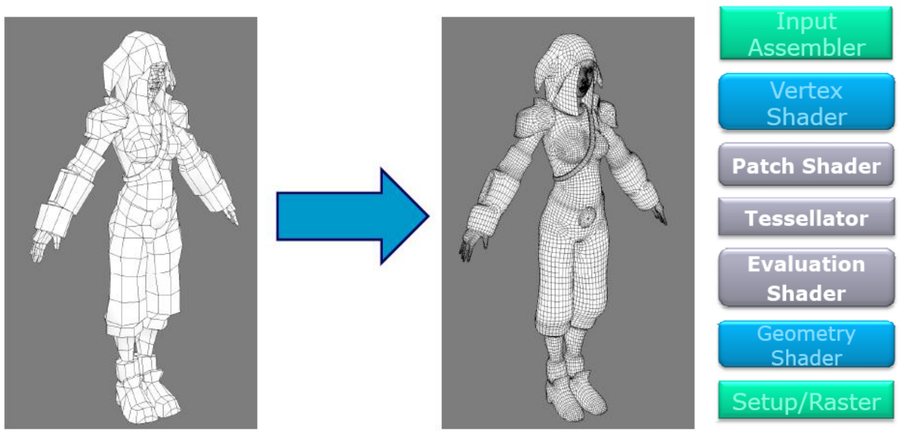

Figure 3.7: Left: Tessellation increases mesh resolution and smoothness. Right: Future pipeline including tessellation shaders as presented by Tatarinov [2008]. Figures with permission from NVidia.

As stated by Antal and Szirmay-Kalos [2008] subdivision can be done on the GPU in a single pass per subdivision level, using the new geometry shader on a modern GPU. Using such an approach with Catmull-Clark or Loop subdivision (Loop [1987]) could at first glance seem to be a good idea for offloading this step from the CPU. However, using the geometry shader in the proposed way would output triangles serially. The serial output and one pass per level of subdivision limits the number of levels one can hope to subdivide while still maintaining any sort of performance. Other schemes for GPU-based subdivision also suffer from a range of similar limitations, as discussed by Antal and Szirmay-Kalos, so other methods to increase the mesh resolution and increasing the details on the mesh should be considered.

In the relative near future with the release of DirextX 11, new GPUs will support a wide range of tessellation methods, as indicated in the talk by Tatarinov [2008] on instanced tessellation at GDC 2008 [2]. In this talk, the upcoming

[2]Both Nvidia and ATI supported hardware based tessellation for personal computers already in 2001. Nvidia with the GeForce 3 Series, and ATI with the release of the Radeon 8600 graphic card based on the Radeon R200 chip design. However, the functionality of the tessellation was limited. For instance, ATI only supported PN-Triangles, while Nvidia supported Bezier patch evaluation. In the latest ATI graphic cards based on the R600 chip design, a more general tessellation unit is available. This tessellation unit is based on the Xenos GPU implementation from the Xbox 360 game console. Common for all these are that they are not standardized in either OpenGL or DirectX/Direct3D, and are in fact proprietary

shader stages that handle tessellation as shown in the right hand side of Figure 3.7 were discussed.

The pipeline assumes a new input primitive, a *patch*. In the *Patch Shader*, one would do basis conversions for easier evaluation and calculate a level of detail factor for each edge. In the *Tessellator* (configurable fixed function) stage, one would generate UV coordinates, and finally, in the *Evaluation Shader* one would evaluate the surface using the given parametric UV coordinates, and apply vertex displacement mapping, space transformations and so on [3].

In Andrei Tatarinov's talk, a general approach for efficient approximation of such a new tessellation pipeline was also presented, called *instanced tessellation*. The presentation only considered quads, but the general approach also works with triangles. This formed the basis for the tessellation approach used in the contribution, and can be seen as a simulation of the future pipeline model that supports hardware tessellation on the GPU.

The approach follows five steps:

1. Set the entire mesh as constant buffer data. This uploads the mesh data to the GPU memory for fast access.

2. Generate a pre-tessellated patch (UV-map) of the needed resolution.

3. Set the entire pre-tessellated patch as instance data. This uploads the patch to the GPU memory for fast access.

4. Render the pre-tessellated patch with instancing, one instance per patch in the mesh

5. Compute refined vertex positions in Vertex Shader using the chosen evaluation algorithm

For the rigid body approach to simulate tree motion, a coarse mesh is generated after each simulation step. This mesh is further tessellated by the GPU, and is therefore faster than generating a high resolution mesh after each step. However, a skinning approach as suggested by Zioma [2007] could be equally effective, as one would not have to regenerate the mesh after each step if using skinning.

technology

[3]In the November 2008 DirectX/Direct3D 11 Technical Preview SDK from Microsoft, the *Patch Shader* is now known as the *Hull shader*, and the *Evaluation Shader* is known as the *Domain shader*. The SDK included a software tessellator that could be used to test a subset of the upcoming tessellation API.

For the particle cloth approach to simulate tree motion, it is even more obvious that it is faster to generate only a coarse mesh which is then further tessellated by the GPU. The particle cloth approach uses the mesh itself to do the simulation, and the mesh resolution will therefore determine the complexity of the simulation. Also, after the initial generation of the coarse mesh, one does not need to do this again, as the coarse mesh is used for both animation and tessellation.

PN-Triangle and PN-Quad tessellation

For this contribution, PN-Triangles as presented in Vlachos et al. [2001] or PN-Quads as presented in Theyer [2003], is used for tessellation, dependent on triangle or quad input. In a point-normal (PN) approach, the vertices and the per-vertex normal are used to calculate control points that gives a cubic Bezier surface. As the vertices and normals are part of the data given for each triangle or quad, the approach does not need or consider data from neighbouring triangles or quads. As such, the approach is considered to be local (compared to subdivision approaches).

One should note that the surface one can generate with PN-Triangles or PN-Quads is generally not even C^1 continuous between patches. To correct this, the normals are interpolated separately from the tessellation, using either linear interpolation, or quadric interpolation to take into account surface influx. This gives the appearance of a single continuous surface when applying lightning. However, one will still see the geometry discontinuity over patch edges, especially if seen against the silhouette.

For point-normal methods, two new control points have to be calculated for each polygon/patch edge. Calculating these for an edge is done by projecting the edge into the tangent plane defined by the closest vertex and its normal. For example, as seen in Figure 3.8, for an edge between two vertices P_1 and P_2, one can project a point at $\frac{1}{3}$ along the edge into the tangent plane at P_1 (defined by P_1 and N_1). Equation 3.1 shows this calculation for a single control point along an edge.

$$\frac{2 * P_i + P_j - ((P_j - P_i) \cdot N_i)N_i}{3} \tag{3.1}$$

Repeating this for all edges and also using the original vertices as control points gives all the edge control points needed. Only one inner control point is then needed for PN-Triangles or four inner control points for PN-Quads. For PN-Triangles the inner control point is found by translating the average position of the three original vertices of the triangle. First a vector to translate

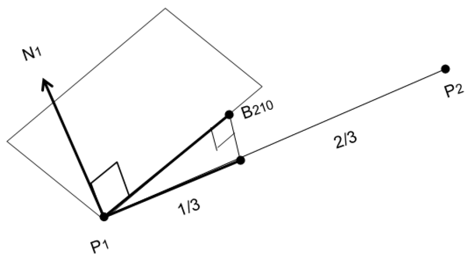

Figure 3.8: Calculating a new control point as the projection of a point at $\frac{1}{3}$ of the edge length, onto the tangent plane at P_1. Reproduced from Vlachos et al. [2001].

along is found as the average of the 6 new edge control points, subtracted by the average position of the three original vertices. Then the position of the inner control point is found by translating the average position of the three original vertices, along the found vector by 3/2 of the length of the vector.

For PN-Quads the missing four inner control points can be a bit harder to calculate. In the presented contribution an approach is used that simply assumes zero twist at the corners. Using the naming conversion shown in Figure 3.9, and assuming zero twist, a inner control point P_{11} closest to the corner P_{00} can be calculated using $0 = 9(P_{00} - P_{01} - P_{10} + P_{11})$, and similar for the other inner control points.

However, as the assumption of zero twist of the surface is generally not correct, one does get better results using the original PN-Quad approach as suggested by Theyer [2003], where the inner control points were calculated using the same equation as for the edges (Equation 3.1), but this time over the diagonals.

A simple change to the way one calculated the control points for the PN-Triangles (and PN-Quads) that produced an overall rounder surface then the normal PN-Triangle approach was also considered. Basically, by considering a cross section of the box definition of a branch, and assuming this is square; one can generate Bezier control points that would approximate a circle around this square cross section.

From the literature on cubic Bezier curves, it is known that one can only approximate a quarter of a circle segment using a normal cubic Bezier curve.

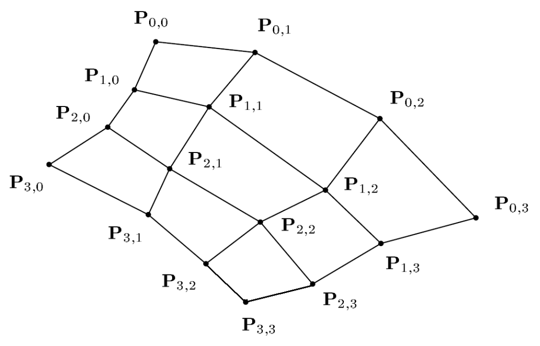

Figure 3.9: Control point naming

A cubic Bezier curve is defined by four control points, usually named P_0, P_1, P_2 and P_3. To approximate a quarter of a circle with a Bezier curve, one sets P_0 and P_3 as the start and end point of the segment. The position of P_1 is set along the tangent of the circle at P_0, at the radius of the circle times 0.55228 along the tangent. The position of P_2 is set in a similar way, but then moving in the opposite direction along the circle tangent at P_3 [4].

When calculating edge control points, the usual approach is to project a point $\frac{1}{3}$ along the edge from a vertex onto the tangent plane. As we are considering a square cross section, the tangent plane becomes a tangent vector. Finding edge control points can now be compared to finding control points that approximate a circle, as seen in Figure 3.10.

By using $\frac{1}{2}$ along the edge instead of $\frac{1}{3}$ for the projection onto the tangent plane, one gets a control point that is $\frac{1}{2}$ times the radius of the imagined circle segment. Although this differs slightly from the usually used value of 0.55228, one still gets the visual effect of a more rounded surface as seen in Figure 3.11. Equation 3.1 is then reduced by one multiplication, as seen in Equation 3.2.

$$\frac{P_i + P_j - ((P_j - P_i) \cdot N_i)N_i}{2} \tag{3.2}$$

[4]This approach is based on defining a point on the circle at $u = 0.5$. The point can then be used to calculate the position for P_1 and P_2, that gives the point for $u = 0.5$ when evaluating the Bezier curve. A good explanation of this approach can be found at http://www.whizkidtech.redprince.net/bezier/circle/kappa/ (link valid 16/01/2009)

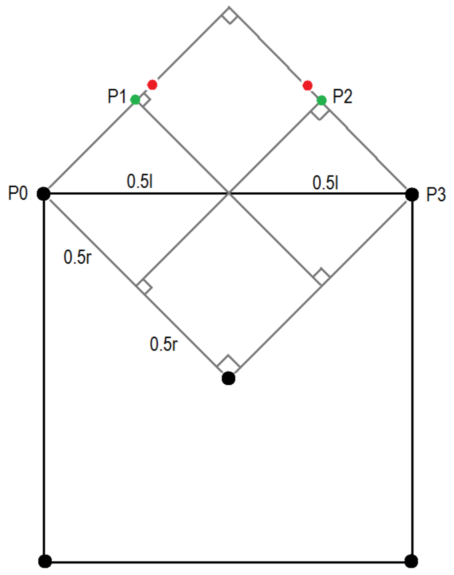

Figure 3.10: Green points: control points P_1 and P_2 found by projecting a point at the centre between P_0 and P_3 onto the tangents. Red points: control points P_1 and P_2 found using 0.55228 times the radius along the tangents.

In the contribution of Skjermo [2008], a *limited normal calculation* was used. Any vertex common for both a parent and a child branch, was considered only part of the parent branch when calculating its normal. In other words, the normal for a vertex was calculated using only the neighbouring vertices that were part of the same branch level.

This was considered as a method to control and limit the influence of attaching a child branch, so the surface deformation from the tessellation only directly influences parts of the mesh that are common for both a branch and a child branch at a branching area. This approach should however be used only with great care, as this has a tendency to increase the lack of continuity when tessellating the polygon mesh. Especially when using the proposed method for control point generation leading to rounder shapes, ones loses continuity, as seen in the right hand sides of Figure 3.14 and 3.15. Generating inner control points by assuming zero twist will generally lead to a deformed surface that is too far away from continuity to be of practical use in most cases, as seen in Figure 3.12 and 3.13.

Finally, it is noted that point-normal based approaches works best when

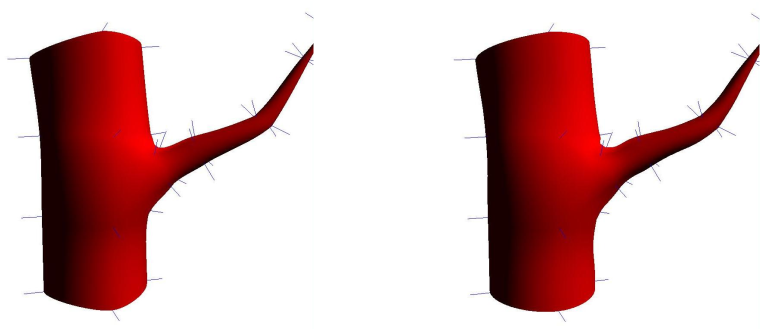

Figure 3.11: From left: PN-Quads, PN-Quads using $\frac{1}{2}$ instead of $\frac{1}{3}$ for control point calculation.

the edges of the polygons are almost similar in length. For long branches of low resolution this is generally not the case, so care should be taken when generating the tree meshes to ensure that the resolution is large enough, for the point-normal approaches to generate natural looking models.

Some results for different inner control point calculation approaches, using both $\frac{1}{2}$ and $\frac{1}{3}$ for edge control point calculation, and with or without the *limited normal calculation* approach can be seen in Figure 3.12 to 3.17 for PN-Quads rendered with triangles.

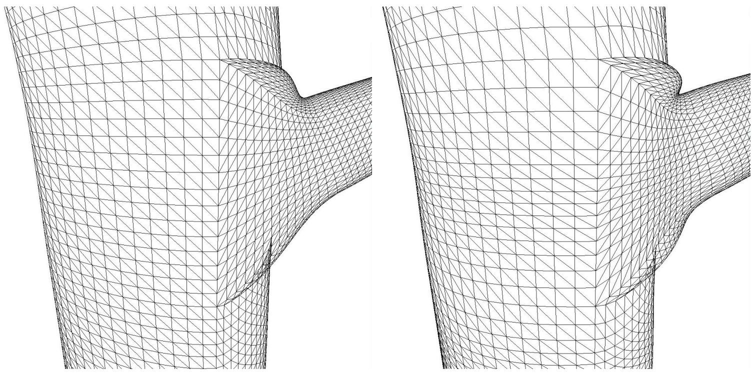

Figure 3.12: Left: PN-Quads with zero twist inner control points. Right: PN-Quads with zero twist inner control points, using $\frac{1}{2}$ instead of $\frac{1}{3}$ for control point calculation.

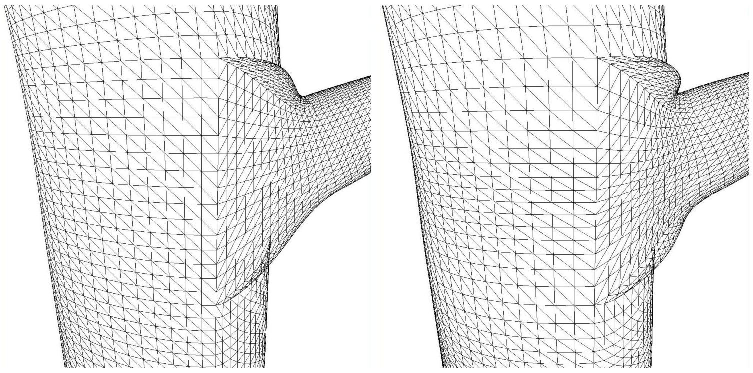

Figure 3.13: Left: PN-Quads with zero twist inner control points using adapted normals. Right: PN-Quads with zero twist inner control points, using adapted normals and $\frac{1}{2}$ instead of $\frac{1}{3}$ for control point calculation.

Subdivision and tessellation

Although using PN-Triangles or PN-Quads for tessellation gives good results, even better results can be achieved by subdividing the mesh by one step, before doing tessellation. One can for instance use Catmull-Clark subdivision if working with quads, or Loop subdivision if working with triangles.

The down side of tessellation is that it will introduce more constraints when simulating using the *cloth approach*, as this uses the vertices as mass points. However, early results were promising, as seen in Figure 3.18. Note that using the proposed method for control point calculation that leads to rounder shapes leads to a less uniform tessellation then the PN-Quad original approach.

Another approach one could consider is to generate a control mesh that enables one to actually approximate the limit surface of the subdivision directly with Bezier patches, as proposed by Loop and Schaefer [2008].

Displacement mapping

Using hardware tessellation enables true vertex displacement mapping in real-time, for instance to generate bark like ridges on the tree stem. However, one should note that parametric based displacement is also possible. Such effects as ridges, bulges and flare, as described in Skjermo [2006] could be considered for such displacements.

Finally, one should note that the texture-based approach as used in Kharkamov et al. [2007], still can be used to even further increase the appearance of the tree, even if the mesh is improved using both tessellation and vertex displace-

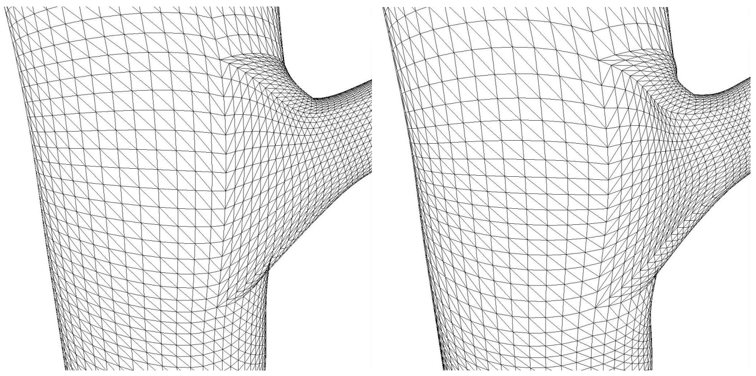

Figure 3.14: Left: PN-Quads. Right: PN-Quads using $\frac{1}{2}$ instead of $\frac{1}{3}$ for control point calculation.

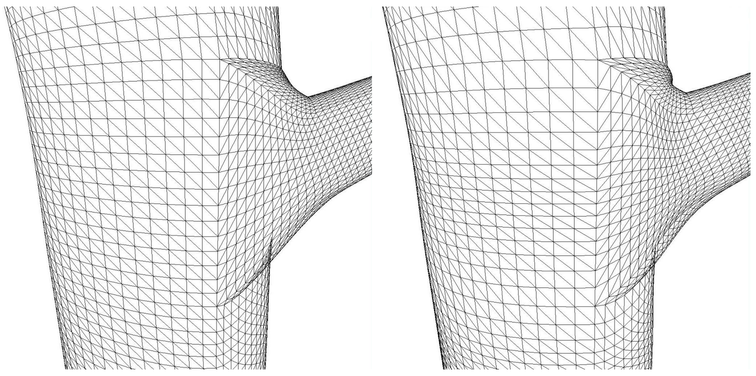

Figure 3.15: Left: PN-Quads using adapted normals. Right: PN-Quads using adapted normals and $\frac{1}{2}$ instead of $\frac{1}{3}$ for control point calculation.

ment mapping.

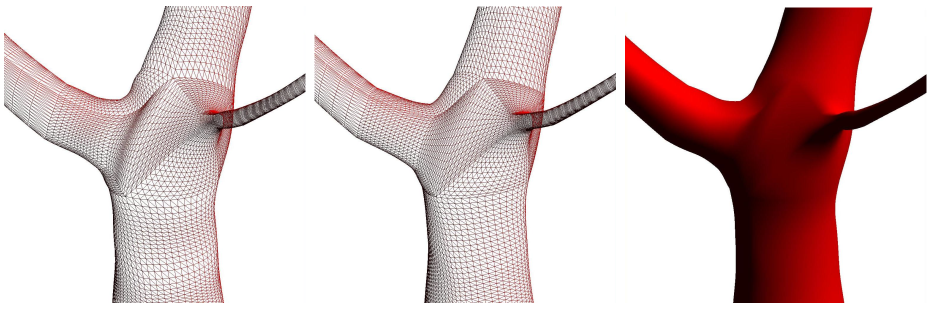

Figure 3.16: Using PN-Quads to level 15 without normal adjustment for a more complex branching, from left: zero twist inner control points using $\frac{1}{2}$ instead of $\frac{1}{3}$, original PN-Quad approach using $\frac{1}{2}$ instead of $\frac{1}{3}$, original PN-Quad approach using $\frac{1}{2}$ instead of $\frac{1}{3}$ shaded using quadric interpolated normals.

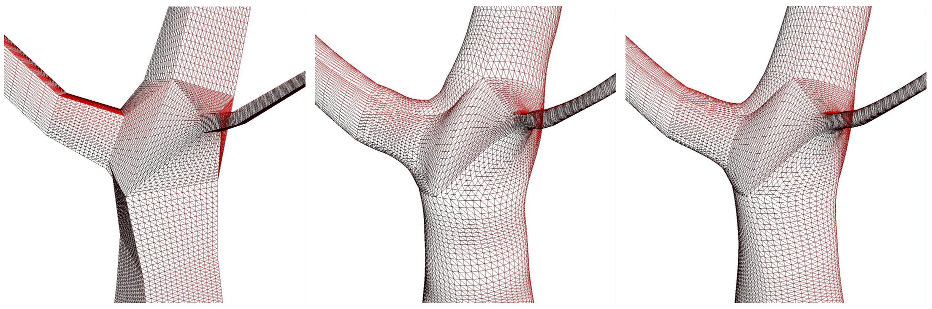

Figure 3.17: Using PN-Quads to level 15 without normal adjustment for a more complex branching, from left: bilinear interpolation, zero twist inner control points, original PN-quad approach.

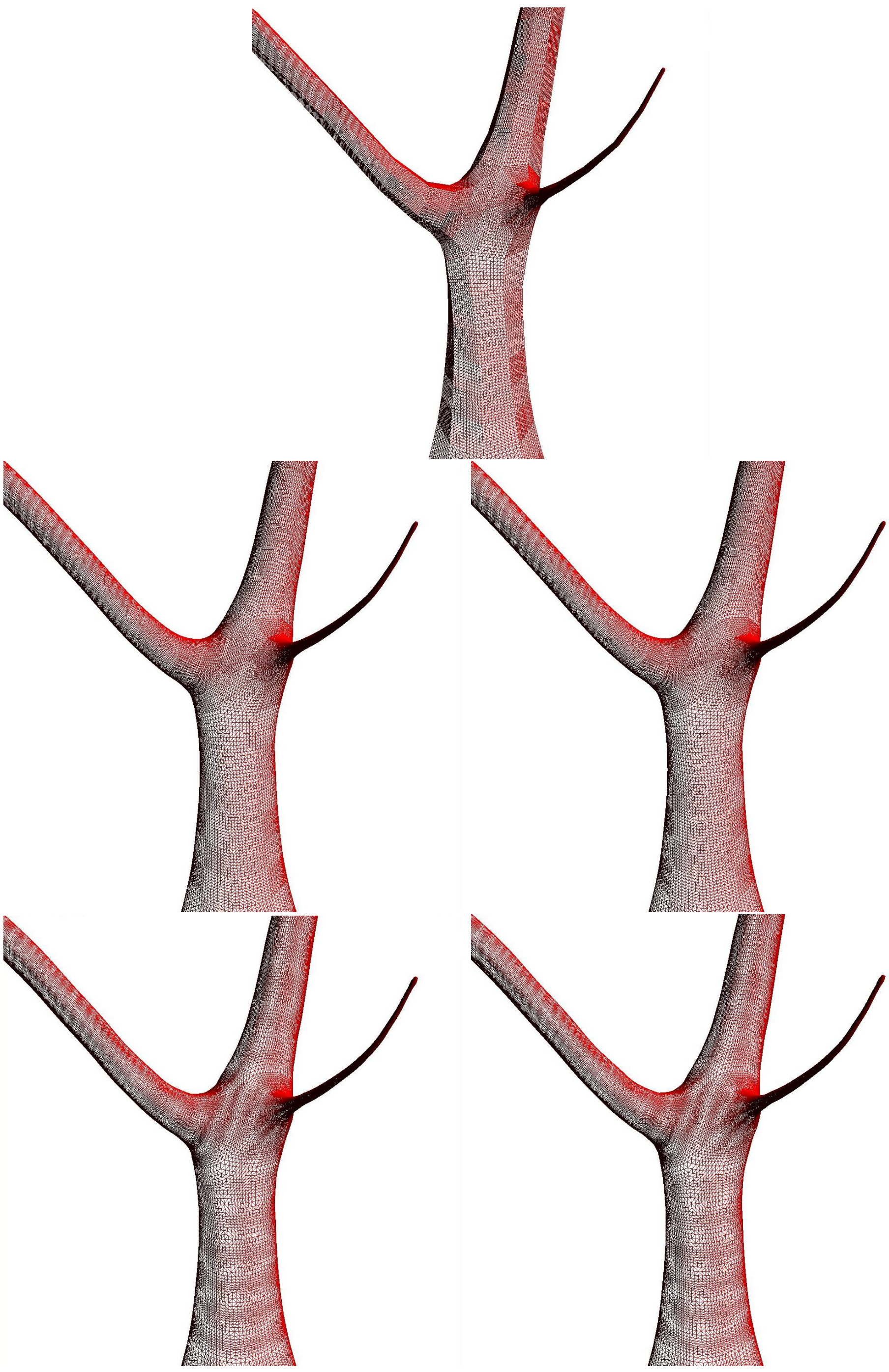

Figure 3.18: Using PN-Quads to level 15 after one step of Catmull-Clark subdivision, top: bilinear interpolation, Middle from left: zero twist inner control points, original PN-Quad approach. Bottom from left: zero twist inner control points using $\frac{1}{2}$ instead of $\frac{1}{3}$, original PN-Quad approach using $\frac{1}{2}$ instead of $\frac{1}{3}$.

Chapter 4

Conclusions and Future Work

The main problem definition describing the work contributed by this dissertation was: *How to generate, animate and render polygon mesh objects of naturally looking tree stems for use in real-time applications* (Section 1.2).

This chapter first summarizes the main conclusions, then consider each of the areas of generation, animation and rendering of polygon mesh objects of tree stems, in the context of each of the contributions. Finally, some considerations and pointers to future work related to this thesis are presented.

4.1 Conclusions

The overall hypothesis of this dissertation was: *A programmable graphical processing unit (GPU) can be employed, both directly and indirectly (by offloading the CPU) to improve the quality of three key aspects of polygon-based tree visualization: mesh generation, animation and rendering* (Section 1.2).

In this thesis, several contributions to the field of generation, rendering, animation and simulation of tree stems and branches were presented. In addition to the main focus areas and the problems defined in Chapter 1, the focus has been on methods that use the GPU to either offload the CPU or to achieve performance suitable for real-time applications. Also, a partial focus has been on improving the appearance of tree stems and branches for viewing distances towards the 'human scale' or even 'insect scale'.

However, the main aim for the works presented have been to improve on the areas of generation, rendering and animation of polygon mesh based tree models. Although the problems defined in Section 1.2 for the main focus areas Although the problems in our main focus areas still remain open, this dissertation has provided several improvements. Also, several ideas and directions for future work in these fields have been presented. These ideas and directions, will hopefully lead toward further improvements.

Finally, approaches presented in this thesis, take advantage of the programmable GPU to improve the fields of mesh generation, animation and rendering of polygon-based tree models. The presented ideas and directions for future work also show how to take even further advantage of modern GPUs computational power and features. Also, during the time spent on this work, other authors have also shown improvements in these fields by using the programmable GPU. This could be seen in our discussion on related works and state-of-the-art in Chapter 2. As such the overall hypothesis can be said to have been confirmed, and that further research into the defined problems can lead to even further future improvements.

4.2 Generation

One problem considered was how to generate polygon meshes of tree models suitable for use in real-time applications.

In Skjermo and Eidheim [2005], an approach is presented that generates watertight polygon mesh models of tree stems based on descriptions generated by L-systems. As the generated meshes had a rather coarse resolution, they were further refined using Catmull-Clark subdivision to increase the resolution and smoothness of the mesh. Because of the subdivision steps, this approach did not perform in real-time when considering more than a few tree models. Also, generation of texture mapping coordinates is not investigated as part of this work. Having good texture mapping is an obvious requirement in modern real-time graphical applications, and as such this is a clear limitation. Compared to the approach presented in Lluch et al. [2004], our contribution is more general and can produce tree models of a larger variety, but also produces coarser meshes of less resolution.

In Skjermo [2006], the approach of Weber and Penn was refined and simplified to allow hardware accelerated generation of parametric defined tree polygon models. The presented approach generates a mesh model for each branch in a tree, using a deformation of a parametric plane around a quadric Bezier curve. As each branch was handled separately, the approach did not produce a single watertight mesh model of a tree. Note that stem splitting was not considered. Stem splitting could be added by splitting a Bezier curve into two curves at a given branching point, and using one of the new curves that starts at the branching point for the split branch. By moving only the last two control points of the new curve one would maintain C^1 and G^1 continuity for the branching area.

In this article, a comparison with other methods was not done; as no other approaches that generate polygon mesh objects of trees on the fly using the GPU were presently known. The approach would generate a medium number of tree polygon models in real-time, and as such is considered best used for trees close to the observer. Trees further away can for instance be blended out to billboards.

4.3 Animation

Another problem considered was how to animate polygon mesh models of trees in real-time applications.

In Skjermo [2007], animation of wind was shown for tree models generated with the approach in Skjermo [2006].

In the proposed approach wind forces were simulated using the sum of sine curves. For each branch a wind force vector is scaled and used directly to displace the control points of the Bezier curves that define the branch. The control points are stored in texture memory, and updated for each step using methods from the field of general purpose calculation on the GPU. As such, the constant reading and writing from texture memory made the approach slower then simpler approaches for GPU-based wind animation of trees, such as the method in Wesslén and Seipel [2005]. The approach was therefore shown to be best suited for a relative small number of trees close to the observer.

The presented approach managed to simulate internal turbulence in a tree which simpler approaches could not. This was visually comparable with the noise-based approaches of Stam [1997] and Ota et al. [2003]. However, for the approach in the contribution it is conceptually simpler to include sudden wind influence by game-related events.

If one wants full interaction between tree objects and other objects in the world, using the same physics engine for all objects is preferable. In Skjermo [2008] some ideas and early results for tree animation using a commercial physics engine was presented. First a somewhat naive full rigid body approach was considered, before showing a method that considers a tree polygon mesh as a constrained particle system. One idea that was considered in this contribution was how to introduce destructible tree objects, something that none of the previous animation approaches presented in Section 2.2 did. However, only some early preliminary work on this was presented.

4.4 Rendering

Another problem considered was how to increase the quality when rendering polygon mesh models of trees in real-time applications. In this thesis, mainly subdivision and tessellation has been considered as methods for increasing rendering quality.

In Skjermo [2008] the approach in Skjermo and Eidheim [2005] was considered further as basis for hardware tessellation instead of offline Catmull-Clark subdivision for mesh refinement. The approach was considered a simulation of future hardware tessellation in DirectX 11, that hopefully will make this approach valid for a large number of tree models in real-time. Using tessellation one got LOD more or less for free, while other approaches like fading to billboards at far ranges can still be considered.

In this article a coarse mesh is tessellated directly, and several limitations with this approach were considered (as described in Section 3.3.1). However, it was shown that using a single step of Catmull-Clark subdivision before tessellation produced similar if not better results, at the cost of increased resolution for the initial mesh.

In Kharkamov et al. [2007] some work on increasing rendering quality in the human to insect scale, using texture based approaches like parallax mapping was presented. While improving quality with tessellation, our contributions can still incorporate such methods. When using tessellation one can still use methods like parallax mapping, but also methods like texture or procedural based displacement mapping can be used.

4.5 Future Work

For each of the main problem areas of this dissertation: generation, animation, and rendering, suggestions for future work are presented in the following sectons.

4.5.1 Generation

Constructing an L-system that generates good tree data can be somewhat confusing for a content developer of real-time applications. It is considered by many that other approaches are better suited for fast generation of model description. For instance, the approach chosen in Xfrog to get *visually correct shapes of plants* is by many considered to be an easy to use graphical interface, as described by in Deussen and Lintermann [1997].

Parametric systems similar to Weber and Penn [1995], can be considered as a middle ground. Parametric approaches are based on a (somewhat simplistic) biological description, while at the same time allowing for both good user control and ease of use, as seen in the SpeedTree software.

During the initial phases of the work presented in Skjermo [2008], a somewhat modified version of our approach given in Skjermo and Eidheim [2005] was tried on tree data generated from parametric tree descriptions (using the approach presented in Skjermo [2006]). However, as the underlying parametric model does not follow the branching rules of Da'Vinchi, the generated mesh did have a tendency to include self-penetrations at the branching areas.

In recent work on simulation of tree models by Weber [2008], a variation of the Weber and Penn approach is used that follows the branching rules. No details on the implementation were given, but the source code is available. As such this could form the basis for a future look at how to generate a watertight polygon mesh model, based on a parametric tree description. In Weber [2008], some interesting ideas on simulation of trees by decomposition of the motion equation into several 1D cases in 2D space were also presented.

One direction for future work would be to search for an algorithm that is able to generate a watertight polygon mesh from a parametric tree description. For instance, such an approach would enable mesh generation for both L-systems and parametric systems for further refinement with tessellation. Tessellation can be used to increase both the quality of LOD, and give better looking branching areas. This holds true even if animating trees, as the animation can be done on the coarse mesh before tessellation.

A good starting point for such an approach could be to use a parametric system that follows the Da'Vinchi rule, as mentioned in Weber [2008]. The descriptions generated from such a system could be used to make watertight polygon tree meshes, using the method in Skjermo and Eidheim [2005]. The watertight meshes could then be the starting point for DirectX 11 based hardware tessellation approximating Catmull-Clark subdivision.

Another approach could be to do one level of actual Catmul-Clark subdivision on the meshes as a pre-processing step, before using PN-Triangles, as mentioned in Skjermo [2008]. The PN-Triangle approach would require less computation for the tessellation, but would have more vertices to translate if the polygon mesh before tessellation was to be used for animation.

A third approach could be to base the tessellation on the new work by Fünfzig et al. [2008]. In this work, an approach similar to PN-Triangles is presented, suitable for calculation on the geometry shader, which is G^1 continuous across triangle edges. What approach to use will have to be considered taking both any animation approaches and the speed of future tessellation hardware into consideration, and is therefore unknown.

In Skjermo and Eidheim [2005], texture mapping was not considered. A suggested starting point for texture coordinate generation could be the texture scheme used in Maierhofer [2002], Another approach one can consider is to blend together textures on different branches around the branching areas using texture arrays.

4.5.2 Animation

In Skjermo [2007], the wind animation was generated by deforming a branch around a Bezier curve. As the Bezier curve chang due to the wind animation, the length of the branch will change. In Habel et al. [2009], nonlinear deformation were used to maintains the length of a branch during wind animation. A similar approach, or reposition the control points for the Bezier curve based on curvature, could be considered for maintaining the branches' lengths.

From the findings in Skjermo [2008], a further look at physical based animation utilizing hardware acceleration could be considered. For a further look at the cloth-based approach, obvious areas to look closer at would include the physics simulation. When using the cloth mass-spring approach, the need for stiff springs to maintain the overall shape requires a high number of iterations to maintain stability. A closer look at destruction could also be considered. One will need to fill in the mesh with new polygons when it tears/destructs, to avoid holes in the mesh.

Also, it is noted that in the PhysX manual it is suggested to use 'soft bodies' for tree rendering. For soft bodies, i.e., a volumetric deformable object, an object is divided into tetrahedral volumes that are constrained. In the proposed 'cloth based' approach, the whole tree object is considered a single constrained volume.

Soft bodies do have very good behaviour under simulation in PhysX. For instance, simulation LOD levels are automatically pre-calculated and applied. However, destruction of soft body objects is not possible with the PhysX engine at this time. A closer look at the methods used for soft bodies with respect to destruction could therefore be considered.

4.5.3 Rendering and tessellation

Skjermo [2008] considers the use of tessellation for increasing polygon density as one approaches a tree model. However, the contribution did not consider adaptive or continuous tessellation for LOD handling. With continuous tessellation one can remove any popping effects when changing LOD levels. Adaptive tessellation can be used to control the tessellation level of each patch while still maintaining a watertight mesh, thereby one can tessellate only as needed.

When tessellating triangles or quads, one can add details to the surface using true per-vertex displacement by vertex texture fetches from a height-map. Another approach for adding high-frequency detail is to use parametric functions as seen in Boubekeur and Schlick [2005]. Adding bulges and lobes in a similar approach as in the contribution presented in Skjermo [2006] could therefore be an option.

For future work in this area it is recommended to look at how to best utilize new upcoming hardware support in DirectX11 for tessellation, in the context of tree rendering. Especially such approaches as texture and parametric vertex displacement mapping, and both adaptive and continuous tessellation for LOD handling, could be considered.

Bibliography

Abelson, H. and DiSessa, A. A. (1981). *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* MIT Press.

Antal, G. and Szirmay-Kalos, L. (2008). Fast evaluation of subdivision surfaces on direct3d 10 graphics hardware. In Engel, W., editor, *ShaderX6*, chapter 1. Course Techbology, Cengage Learning.

Beaudoin, J. and Keyser, J. (2004). Simulation levels of detail for plant motion. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 297–304, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Bloomenthal, J. (1985). Modeling the mighty maple. *SIGGRAPH Comput. Graph.*, 19(3):305–311.

Boubekeur, T. and Schlick, C. (2005). Generic mesh refinement on gpu. In *ACM SIGGRAPH/Eurographics Graphics Hardware.* http://www.labri.fr/publications/is/2005/BS05.

Bromberg-Martin, E., Árni Már Jónsson, Marai, G. E., and McGuire, M. (2004). Hybrid billboard clouds for model simplification. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Posters*, page 30, New York, NY, USA. ACM.

Catmull, E. and Clark, J. (1978). Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer Aided Design*, 10(6):350–355.

Chang, Y. K. and Rockwood, A. (1994). A generalized de casteljau approach to 3d free-form deformation. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994*, pages 257–260. SIGGRPAH, ACM Press.

Décoret, X., Durand, F., Sillion, F. X., and Dorsey, J. (2003). Billboard clouds for extreme model simplification. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 689–696, New York, NY, USA. ACM.

d'Eon, E. (2004). Deformers. In Fernando [2004], chapter 42, pages 723–732.

Deussen, O. and Lintermann, B. (1997). A modelling method and user interface for creating plants. In *Proceedings of the conference on Graphics interface '97*, pages 189–197, Toronto, Ont., Canada, Canada. Canadian Information Processing Society.

Eidheim, O. C., Skjermo, J., and Aurdal, L. (2005). Real-time analysis of ultrasound images using gpu. In Lemke, H. U., Inamura, K., Doi, K., Vannier, M. W., and Farman, A. G., editors, *CARS 2005 - Proceedings of the 19th International Congress and Exhibition*, pages 284–289.

Felkel, P., Wegenkittl, R., and Buhler, K. (2004). Surface models of tube trees. *Computer Graphics International, 2004. Proceedings*, pages 70–77.

Fernando, R., editor (2004). *GPU Gems*. Addison-Wesley Profesional.

Finch, M. (2004). Effective water simulation from physical models. In Fernando [2004], chapter 1, pages 5–29.

Fünfzig, C., Müller, K., Hansford, D., and Farin, G. (2008). Png1 triangles for tangent plane continuous surfaces on the gpu. In *GI '08: Proceedings of graphics interface 2008*, pages 219–226, Toronto, Ont., Canada, Canada. Canadian Information Processing Society.

Galbraith, C., MacMurchy, P., and Wyvill, B. (2004a). Blobtree trees. In *CGI '04: Proceedings of the Computer Graphics International*, pages 78–85, Washington, DC, USA. IEEE Computer Society.

Galbraith, C., Mündermann, L., and Wyvill, B. (2004b). Implicit visualization and inverse modeling of growing trees. *Computer Graphics Forum*, 23(3):351–360.

Garcia, I., Sbert, M., and Szirmay-Kalos, L. (2005). Tree rendering with billboard clouds. Third Hungarian Conference on Computer Graphics and Geometry, Budapest, 2005.

Giacomo, T. D., Capo, S., and Faure, F. (2001). An interactive forest. In Cani, M.-P., Magnenat-Thalmann, N., and Thalmann, D., editors, *Eurographics Workshop on Computer Animation and Simulation (EGCAS)*, pages 65–74. Springer. Manchester.

Göddeke, D. (2005). Gpgpu–basic math tutorial. Technical report, FB Mathematik, Universität Dortmund. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300, `http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu`.

Habel, R., Kusternig, A., and Wimmer, M. (2009). Physically guided animation of trees. In *Comput. Graph. Forum*, pages 523–532.

Honda, H. (1971). Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology*, pages 331–338.

Jakobsen, T. (2003). Advanced character physics. Online Article. Valid as of the 8th of Oct. 2008, http://www.gamasutra.com.

Jirasek, C., Prusinkiewicz, P., and Moulia, B. (2000). Integrating biomechanics into developmental plant models expressed using l-systems. *Plant Biomechanics*, pages 615–624.

Jones, M. (1994). Lessons learned from visual simulation. Siggraph 94 Course Notes: Designing Real-Time 3D Graphics for Entertainment.

Kanda, H. and Ohya, J. (2003). Efficient, realistic method for animating dynamic behaviors of 3d botanical trees. *Multimedia and Expo, 2003. ICME '03. Proceedings. 2003 International Conference on*, 2:II–89–92 vol.2.

Kharkamov, A., Cantlay, I., and Stepanenko, Y. (2007). Next-generation speedtree rendering. In Nguyen [2007], chapter 4, pages 69–92.

Lacewell, J. D., Edwards, D., Shirley, P., and Thompson, W. B. (2006). Stochastic billboard clouds for interactive foliage rendering. *journal of graphics tools*, 11(1):1–12.

Lindenmayer, A. (1968). Mathematical models for cellular interaction in development, parts i and ii. *Journal of Theorectical Biology*, 18:280–315.

Lindenmayer, A. and Prusinkiewicz, P. (1990). *The Algorithmic Beauty of Plants*. Springer-Verlag.

Linsen, L., Karis, B. J., McPherson, E. G., and Hamann, B. (2005). Tree growth visualization. In *WSCG 2005*. UNION Agency - Science Press.

Lintermann, B. and Deussen, O. (1999). Interactive modeling of plants. *IEEE Comput. Graph. Appl.*, 19(1):56–65.

Lluch, J., Vivo, R., and Monserrat, C. (2004). Modelling tree structures using a single polygonal mesh. *Graphical Models*, 66:89–101.

Loop, C. (1987). *M.S. Mathematics Thesis, Smooth Subdivision Surfaces Based on Triangles*. PhD thesis, University of Utah.

Loop, C. and Schaefer, S. (2008). Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Trans. Graph.*, 27(1):1–11.

Luban, P. (2007). Physics in games: A new gameplay frontier. Online Article. Valid as of the 8th of Oct. 2008, http://www.gamasutra.com.

Luebke, D. (2001). A developer's survey of polygonal simplification algorithms. *Computer Graphics and Applications, IEEE*, 21(3):24–35.

Maierhofer, S. (2002). *Rule-Based Mesh Growing and Generalized Subdivision Meshes*. PhD thesis, Technische Universitaet Wien, Technisch-Naturwissenschaftliche Fakultaet, Institut fuer Computergraphik.

Mech, R. and Prusinkiewicz, P. (1996). Visual models of plants interacting with their environment. In *Proceedings of SIGGRAPH 96*, pages 397–410.

Murray, C. D. (1927). A relationship between circumference and weight in trees and its bearing in branching angles. *Journal of General Phyiol.*, 9:725–729.

Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. (2006). Position based dynamics. In *Proceedings of Virtual Reality Interactions and Physical Simulations (VRIPhys)*, pages 71–80.

NatFX (2008). Bionatics natfx, tree modelling software. Commersial product, http://www.bionatics.com.

Nguyen, H., editor (2007). *GPU Gems 3*. Addison Wesley.

Ota, S., Fujimoto, T., Tamura, M., Muraoka, K., Fujita, K., and Chiba, N. (2003). 1/eta noise-based real-time animation of trees swaying in wind fields. *cgi*, 00:52.

Perbet, F. and Cani, M.-P. (2001). Animating prairies in real-time. In *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 103–110, New York, NY, USA. ACM.

Peterson, S. (2001). Visual effects in shrek. Silicon Valley ACM SIGGRAPH, http://silicon-valley.siggraph.org/MeetingNotes/Shrek.html.

PhysX (2008). Nvidia physx engine sdk. Physic Engine and Software Developer Kit. http://www.nvidia.com/object/physx.html.

Prusinkiewicz, P. (1986). Graphical applications of l-systems. In *Proceedings of Graphics Interface '86 / Vision Interface '86*, pages 247–253.

Prusinkiewicz, P., James, M., and Mech, R. (1994). Synthetic topiary. In *Proceedings of SIGGRAPH 94*, pages 351–358.

Rebollo, C., Remolar, I., Chover, M., Gumbau, J., and Ripollés, O. (2007). A clustering framework for real-time rendering of tree foliage. *JOURNAL OF COMPUTERS (JCP)*, 2(4):57–67.

Richter, J. P. (1970). *The notebooks of Leonardo da Vinc Vol. 1.* Dover Pubns.

Singh, P., Zhao, N., Chen, S.-C., and Zhang, K. (2005). Tree animation for a 3d interactive visualization system for hurricane impacts. *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, pages 598–601.

Skjermo, J. (2006). A gpu-based branch deformer. In *Proceedings of the 22nd spring conference on Computer graphics*, pages 120–127. Comenius University, Comenius University, Bratislava.

Skjermo, J. (2007). GPU-Based Wind Animation of Trees. In Lim, I. S. and Duce, D., editors, *Theory and Practice of Computer Graphics*, pages 29–36, Bangor, United Kingdom. Eurographics Association.

Skjermo, J. (2008). Generation and Tessellation of Tree Stems. In Lim, I. S. and Tang, W., editors, *Theory and Practice of Computer Graphics*, pages 163–166, Manchester, United Kingdom. Eurographics Association.

Skjermo, J. and Eidheim, O. C. (2005). Polygon mesh generation of branching structures. In Kälviäinen, H., Parkkinen, J., and Kaarna, A., editors, *SCIA, Image Analysis, 14th Scandinavian Conference, SCIA 2005, Joensuu, Finland, June 19-22, 2005, Proceedings*, volume 3540 of *Lecture Notes in Computer Science*. Springer.

Sousa, T. (2007). Vegetation procedural animation and shading in crysis. In Nguyen [2007], chapter 6, pages 105–122.

SpeedTreeRT (2008). Tree modelling and rendering middleware. Commersial product, http://www.speedtree.com.

Stam, J. (1997). Stochastic dynamics: Simulating the effects of turbulence on flexible structures. *Computer Graphics Forum*, 16:159–164.

Tatarinov, A. (2008). Instanced tessellation in directx10. At Game Developer Conference 2008., http://developer.download.nvidia.com/ presentations/2008/GDC/Inst_Tess_Compatible.pdf.

Taylor-Hell, J. (2005). *Biomechanics in Botanical Trees. M.Sc. thesis.* PhD thesis, University of Calgary.

Theyer, M. (2003). Higher-order surfaces using curved point-normal (pn) triangles. In Lander, J., editor, *Graphics Programming Methods*, chapter 2.6. Charles River Media.

Venkatasubramanian, S. (2003). The graphics card as a stream computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams.*

Vlachos, A., Peters, J., Boyd, C., and Mitchell, J. L. (2001). Curved pn triangles. In *SI3D*, pages 159–166.

Weber, J. and Penn, J. (1995). Creation and rendering of realistic trees. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 119–128, New York, NY, USA. ACM Press.

Weber, J. P. (2008). Fast simulation of realistic trees. *IEEE Comput. Graph. Appl.*, 28(3):67–75.

Wesslén, D. and Seipel, S. (2005). Real-time visualization of animated trees. *The Visual Computer*, 21:397–405.

Whatley, D. (2005). Toward photorealism in virtual botany. In Pharr, M. and Fernando, R., editors, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, chapter 1, pages 7–25. Addison-Wesley Professional.

Witkin, A., Baraff, D., and Kass, M. (1995). An introduction to physically based modeling. SIGGRAPH 1995 Course Notes. http://www.cs.cmu.edu/ baraff/pbm/pbm.html.

Xfrog (2008). Greenworks xfrog, tree modelling software. Commersial product, http://www.xfrogdownloads.com.

Zioma, R. (2007). Gpu-generated procedural wind animations for trees. In Nguyen [2007], chapter 6, pages 105–122.

Zorin, D., Schröder, P., Levin, A., and Sweldens, W. (2000). Subdivision for modeling and animation. SIGGRAPH 2000 Course Notes. http://www.mrl.nyu.edu/ dzorin/sig00course/.

Author Index

Part II

Research Results

Polygon Mesh Generation of Branching Structures

Skjermo, J. and Eidheim, O. C. (2005). Polygon mesh generation of branching structures. In Kälviäinen, H., Parkkinen, J., and Kaarna, A., editors, *SCIA, Image Analysis, 14th Scandinavian Conference, SCIA 2005, Joensuu, Finland, June 19-22, 2005, Proceedings*, volume 3540 of *Lecture Notes in Computer Science*. Springer

Abstract

We present a new method for producing locally non-intersecting polygon meshes of naturally branching structures. The generated polygon mesh follows the object's underlying structure as close as possible, while still producing polygon meshes that can be visualized efficiently on commonly available graphic acceleration hardware. A priori knowledge of vascular branching systems is used to derive the polygon mesh generation method. Visualization of the internal liver vessel structures and naturally looking tree stems generated by Lindenmayer-systems is used as examples. The method produce visually convincing polygon meshes that might be used in clinical applications in the future.

Polygon Mesh Generation of Branching Structures

J. Skjermo, O.C. Eidheim
Algorithm and Visualization Group and the
Artificial Intelligence and Learning Group
Department of Computer and Information Science
Norwegian University of Science and Technology
Sem Sælands vei 7-9, NO-7491 Trondheim, NORWAY
{Jo.Skjermo,Ole.Christian.Eidheim}@idi.ntnu.no
http://www.idi.ntnu.no/

Abstract

We present a new method for producing locally non-intersecting polygon meshes of naturally branching structures. The generated polygon mesh follows the object's underlying structure as close as possible, while still producing polygon meshes that can be visualized efficiently on commonly available graphic acceleration hardware. A priori knowledge of vascular branching systems is used to derive the polygon mesh generation method. Visualization of the internal liver vessel structures and naturally looking tree stems generated by Lindenmayer-systems is used as examples. The method produce visually convincing polygon meshes that might be used in clinical applications in the future.

1 Introduction

Medical imaging through CT, MR, Ultrasound, PET, and other modalities has revolutionized the diagnosis and treatment of numerous diseases. The radiologists and surgeons are presented with images or 3D volumes giving them detailed view of the internal organs of a patient. However, the task of analyzing the data can be time-consuming and error-prone.

One such case is liver surgery, where a patient is typically examined using MR or CT scans prior to surgery. In particular, the position of large hepatic vessels must be determined in addition to the relative positions of possible tumors to these vessels.

Surgeons and radiologists will typically base their evaluation on a visual inspection of the 2D slices produced by CT or MR scans. It is difficult, however, to deduce a detailed liver vessel structure from such images. Surgeons at the Intervention Centre at Rikshsopitalet in Norway have found 3D renderings of the liver and its internal vessel structure to be a valuable aid in this complex evaluation phase. Currently, these renderings are based on a largely manual segmentation of the liver vessels, so we have explored a way to extract and visualize the liver vessel structure automatically from MR and CT scans.

The developed procedure is graph based. Each node and connection corresponds to a vessel center and a vessel interconnection respectively. This was done in order to apply knowledge based cost functions to improve the vessel tree structure according to anatomical knowledge. The graph is used to produce a polygonal mesh that can be visualized using commonly available graphic acceleration hardware.

A problem when generating meshes of branching structures in general, is to get a completely closed mesh that does not intersect itself at the branching points. We build on several previous methods for mesh generation of branching structures, including methods from the field of visualization for generation of meshes of tree trunks.

The main function of a tree's trunk can be explained as a liquid transportation system. The selected methods for the mesh generation can therefore be extended by using knowledge of the branching angles in natural systems for fluid transportation. This enables us to generate closed and locally non-intersecting polygon meshes of the vascular branching structures in question.

2 Previous work

In the field of medical computer imagery, visualization of internal branching structures have been handled by volume visualization, as the data often was provided by imaging systems that produced volume data. However, visualization of polygon meshes is highly accelerated on modern commonly available hardware, so we seek methods that can utilize this for our visualization of branching vascular transportation structures.

Several previous works have proposed methods for surface mesh generation of trees that handles branching. We can mention the parametric surfaces used in [Blo85], the key-point interpolation in Oppenheimer [Opp86], the "branching ramiforms" in [BW90] (that was further developed by Hart and Baker in [HB96] to account for "reaction wood"), and the "refinement by intervals" method in [LVF*01].

In [Mai02] and [TMW02], *rule based mesh growing* from L-systems was introduced. The algorithm used mesh connection templates for adding new parts of a tree to the mesh model, as L-system productions was selected during the generation phase. The mesh connection templates were produced to give a final mesh of a tree, that could serve as a basis mesh for subdivision. This method could only grow the mesh by rules, and could not make a mesh from a finished data set.

The work most similar to our was the *SMART* algorithm presented in [FFKW02], that was further developed in [FKFW02]. This algorithm was developed for visualization of vascular branching segments in the liver body, for use in a augmented reality aided surgery system. The algorithm produced meshes that could be used with Catmull-Clark subdivision [CC78] to increase surface smoothness and vertex resolution.

The SMART algorithm defined local coordinate axis in a branching point. The average direction of the incoming and outgoing segments was one axis, and an *up* vector generated at the root segment was projected along the cross sections to define another axis (to avoid twist). The child closest to the average direction was connected with quads, at a defined distance. The *up* vector defined a square cross section, and four directions, at a branching point as seen in Figure 1. All remaining outgoing segments were classified into one of these directions according to their angle compared with the average direction. The child closest by angle in each direction was connected with the present tile, and this was recursively repeated for any remaining children.

Furthermore, the SMART algorithm did not include any method for automatic adjustment of the mesh with respect to the areas near forking points, and could produce meshes that intersected locally if not manually tuned. Our method automatically generates meshes without local intersection as long as the underlying structures loosely follows natural branching rules.

3 Main Algorithm

The proposed algorithm is loosely based on the extended *SMART* algorithm. It also uses knowledge of the branching angles in natural systems for fluid transportation as described in section 3.1.

3.1 Natural branching rules

Leonardo Da Vinci presented a rule for estimating the diameter of the segments after a furcation in blood vessels, as stated in In [Ric70]. The *Da Vinci* rule states that the cross-section area of a segment is equal to the combined cross section area of the child segments, as seen in the following section.

$$\pi r_0^2 = \pi r_1^2 + \pi r_2^2 + ... + \pi r_n^2 \tag{1}$$

A generalization, as seen in Eq. 2 was presented by Murray in [Mur27]. Here, the *Da Vinci* rule has been reduced so that the sum of the diameters of the child segments just after a furcation is equal to the diameter of the parent just before the furcating, where d_0,d_1, and d_2 are the diameters of the parent segment and the child segments, respectively. α was used to produce different branching. α values between 2 and 3 are generally suggested for different branching types.

$$d_0^\alpha = d_1^\alpha + d_2^\alpha \tag{2}$$

From this Murray could find several equations for the angle between 2 child branches after a furcation. One is shown in Eq. 3, where x and y are the angles between the direction of the parent and each of two child segments. As seen from the equation, the angles depend on the diameter of each of the child segments. Murray also showed that the segments with the smallest diameter have the largest angle.

$$\cos(x + y) = \frac{d_0^4 - d_1^4 - d_2^4}{2d_1^2 d_2^2} \tag{3}$$

Thus, we assume that the child segment with the largest diameter after a furcation, will have the smallest angle difference from the direction of the parent segment. This forms the basis for our algorithm, and it will therefore produce meshes that do not locally intersect as long as the underlying branching structure mostly follows these rules that are based on observations from nature. We now show how this is used to produce a polygon mesh of a branching structure.

3.2 New Algorithm

The algorithm is based on the extended *SMART* algorithm, but uses the *Da Vinci* rule to ensure mesh consistency. It uses data ordered in a DAG (Directed Acyclic Graph) where the nodes contains the data for a segment (direction vector, length and diameter). The segments at the same level in the DAG are sorted by their diameters. An *up* vector is used to define the vertices at each segments start and end position. The vertex pointed to by the *up* vector, is set to be a corner of a square cross section of a segment. The sides of this square defines the directions used in sorting any child segments as seen in Figure 1. The sorting results in four sets of child segments (one for each direction of the square), where the child segments in each set are sorted by the largest diameter.

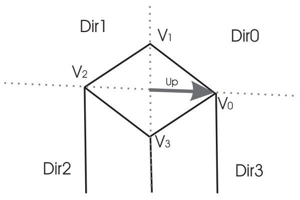

Fig. 1. Square cross section defined by *up* vector. The vertices act as control point for subdivision, producing a circular cross section. The directions $DirX$ are defined by the vertices for each edge (for example, V_0 and V_1 defines the direction $Dir0$).

To connect segments, we basically sweep a moving coordinate frame (defined by a projected *up* vector) along a path defined by the segments data. However, at the branching points we must build another type of structure with vertices, so we can add the child segments on to the polygon mesh. This is done by considering the number of child segments, and their diameters and angles compared to the parent segment.

Starting at the root node in the DAG we process more and more child segments onto the polygon mesh recursively. There are four possible methods for building the polygon mesh for any given segment. If there are one or more child segments in the DAG, we must select one of the methods described in section 3.3 (for one child segment), section 3.5 (for more then one child segment), or in section 3.4 (for cases where the child segment with largest diameter has an angle larger then 90 degrees with respect of the parent segment).

If there are no child segments, the branch is at its end. The segment is closed with a quad polygon over four vertices generated on a plane defined by the projected *up* vector and the segments diameter at the segments end.

3.3 Normal Connection

If there is one child segment (with angle between the present and the child segment less then 90 degrees), we connect the child segment to the present segment, and calculates the vertices, edges and polygons as described in this section. Each segment starts with four vertices on a plane at the segments start position. As the algorithm computes a segment, it finds four vertices at the segment's end. It then closes the sides of the box defined by these eight vertices (not the top and bottom).

The first step is to calculate the average direction between the present segment, and the child segment. This direction is the *half direction.* Next, the up vector is projected onto the plane defined by the *half direction* and the segments end point. A square cross section is then defined on the plane at the segment's end position, oriented to the projected *up* vector to avoid twist. The length of the *up* vector is also changed to compensate for the tilting of this plane compared to the original vector.

The corners of the cross section are the four end corner vertices for the present segment. These vertices, along with the four original vertices, defines the box that we close the sides of with quad polygon faces. In Figure 2, the mesh of a stem made of four segments connected in sequence can be seen. After processing the segment, the child segment is set as the present segment.

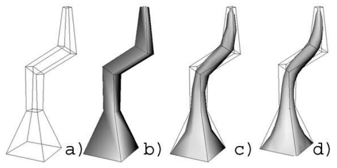

Fig. 2. Simple mesh production. a) the produced mesh, b) shaded mesh, c) one subdivision, d) two subdivisions.

3.4 Connect Backward

When the first child segment direction is larger than 90 degrees compared to the present segments direction. special care has to be taken when producing the mesh (the main part of the structure bends backward). We build the segment out of two parts, where the last part is used to connect the child segments onto the polygon mesh.

The first step is to define two new planes. The *end plane* is defined along the direction of the segment at the distance that equal to the segments' length, plus the first child's radius (from the segments start position). The *middle plane* is defined at a distance equal to the diameter of the first child, along the negative of the present segments direction (from the segments end position).

Two square cross sections are defined by projecting the *up* vector into the two newly defined planes. The cross section at the segments top can be closed with a quad surface, and the sides between the segments start and the middle cross section can also be closed with polygons. The sides between the middle and the top cross sections that has no child segments in its direction, can also be closed with polygons.

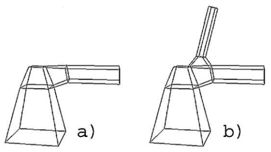

Fig. 3. Mesh production for direction above 90 degrees. a) first child added (direction of 91 degrees), b) next child.

All child segment (even the first one) should be sorted into sets, defined by their direction compared to four direction. The directions are defined by the *middle* cross section, and each set should be handled as if they were sets of normal child segments, as described in section 3.5 Vertices from the newly defined *middle* and *end* cross sections are used to define the start cross sections (the four start vertices) for each of the new directions. An example can be seen in Figure 3.

3.5 Connect Branches

If there is more than one child segment, we start with the first child segment. The first segment has the largest diameter, and should normally have the smallest angle compared to its parent segment.

To connect the first branching child segment onto the mesh, we first use the same method as in section 3.3. We make a square cross section at the end of the present segment, and its sides are closed by polygons. The distance from the segments start to the new cross section can be decreased to get a more accurate polygon mesh (for instance by decreasing the length by half of the calculated *l+x* value found later in this section). In the example where vessels in a liver was visualized (section 4.2), the length was decreased as we just described.

A new square cross section is also defined along the *half direction* of the first child, starting at the center of the newly defined cross section. These two new cross sections defines a box, where the sides gives four cross sections (not the top or bottom side). The first child segment (and its children) are recursively added to the top of this box (on the cross section along the *half direction*), while the rest are added to the four sides. Note that the end position of a segment is calculated by vector algebra based on the parent segment's end position, and not on the cross section along the *half direction*. This means that the segment's length must be larger than the structure made at the branching point, to add the child.

When the recursion returns from adding the main child segment (and its child segments), the remaining child segments are sorted into four sets. The sorting is again done by the segments angle compared to the sides of the cross section around the present segment's end point. One must remember to maintain ordering by diameter while sorting.

The vertices at the corners of the two new cross sections defines a box where the sides can be used as new cross sections for each of the four directions (not the top and bottom sides). For each of the four directions, a new *up* vector is defined as the vector between the center of the directions cross section, and a corresponding vertex on the present segment's end cross section. Figure 4 shows the *up* and *half direction* when adding a child segment in one of the four directions.

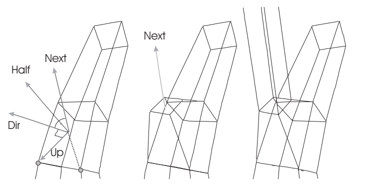

Fig. 4. Adding a second child segment. a) new *up*, *direction* and *half direction* vectors for this direction, b) first part of child segment, c) resulting mesh.

The main problem one must solve is to find the distance along the *half vector* to move before defining the start cross section for the main child segment. This to ensure that there is enough space for any remaining child segments. If the distance moved is too small, the diameter of any remaining child segments will seem to increase significantly after the branching. A too large distance will result in a very large branching structure compared to the segments it connects.

Our main contribution is the automatic estimation of the distance to move, to allow space for any remaining child segments. In Figure 5, we can see the situation for calculating the displacement length m for a given half angle.

The length of m must at least be as large as the root of the $d_1^2 + d_2^2 ... + d_n^2$, where $d_1, d_2..$ are the diameter of the child segments. This because we know from the *Da Vinci rule* and murray's findings that every child segment at this point will have equal or smaller diameter then the parent segment (hence the sorting by diameter of child segments). Note that the *half angle (h)* will be less or equal to 45 degrees, as any larger angle will lead to the segment being handled as in section 3.4 (as the main angle then will be larger then 90 degrees).

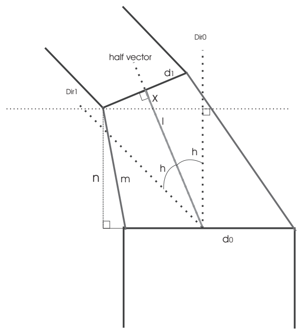

Fig. 5. Finding minimum distance m to move along the half vector to ensure space for any remaining branches (as seen from a right angle).

We could find the exact length of m, but observe that as long as the length of n is equal to d_1, we will have enough space along m. Setting $n = d_1^2 + d_2^2 ... + d_n^2$ gives Eq. 4 for calculating the length to move along the *half vector*.

$$l + x = \sqrt{d_1^2 + d_2^2 + ... + d_n^2}/cos(h) + tan(h) * d_1/2 \tag{4}$$

The error added by using x+l instead of m, will introduce a small error in the mesh production. However, we observe that setting $n = d_1$ seems to give adequate results for most cases. An example result from using the algorithm can be seen in Figure 6.

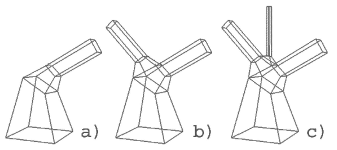

Fig. 6. The mesh production in a branching point. a) First child added, b) next child added, c) third child added.

4 The examples

A preliminary application has been produced in OpenGL to test the algorithm. This section shows this application at work. We have used it to produce polygon meshes for both naturally looking tree stems from a L-system generator, as well as meshes of the derived portal vein from a CT scan of a liver.

4.1 Lindenmayer Generated Tree Stems

An extension to the application accepted an L-system string, representing a tree stem after a given number of Lindenmayer generation steps, as input. The extension interpreted the L-system string into a DAG that the application used to produce a base polygon mesh from. The application then subdivided this mesh to the level set by the user. Normal Catmull-Clark subdivision was used for this step, so the original polygon mesh looked somewhat blocky before subdivision. An example with a shaded surface can be seen in Figure 7.

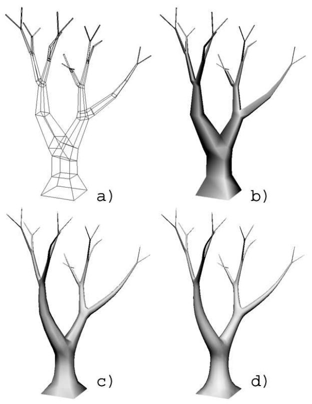

Fig. 7. A tree defined by a simple L-system. a) the produced mesh, b) shaded mesh, c) after one subdivision, d) after two subdivisions.

4.2 Delineation of Hepatic Vessels From CT Scans

Several processing steps has to be completed in order to visualize the hepatic vessels from a CT scan. In the preprocessing phase, histogram equalization [GW02] is first conducted to receive equal contrast on each image in the CT scan. Next, the blood vessels are emphasized using matched filtering [CCK*89]. After the preprocessing phase, the blood vessels are segmented using entropy based thresholding [KSW85] and thresholding based on local variance with modifications using mathematical morphology [Soi03]. A prerequisite to our method is that the liver is segmented manually beforehand.

After the vessel segments are found, the vessels' centers and widths must be calculated. These attributes are further used in a graph search to find the most likely vessel structure based on anatomical knowledge. First, the vessel centers are derived from the segmentation result using the segments' skeletons [Soi03]. The vessels' widths are next computed from a modified distance map [GW02] of the segmented images.

The last step before the vessel graph can be presented is to make connections between the located vessel centers. Centers within segments are interconnected directly. On the other hand, interconnections between adjacent CT slices are not as trivial. Here, as previously mentioned, we use cost functions representing anatomical knowledge in a graph search for the most likely interconnections [EAOJ*04]. The resulting graph is finally visualized using the outlined algorithm in this paper.

A few modification to the existing graph is made in order to make it more visually correct. First, nodes with two or more interconnected neighbors have their heights averaged since the resolution in the y-direction is normally lower than that in the image plane. Second, if two interconnected nodes are closer than a predefined limit, the two nodes are replaced by one node positioned between them. Figure 8 shows the resulting visualization of the derived portal vein from a CT scan of a liver.

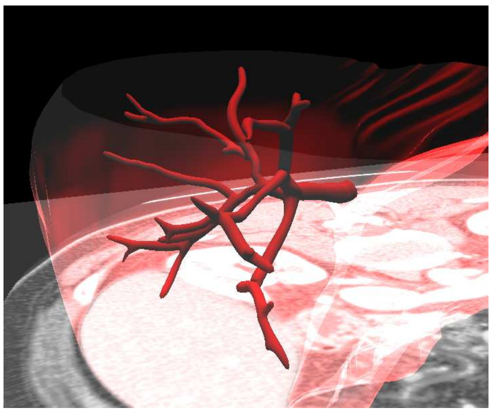

Fig. 8. Portal vein visualized from a CT scan of a liver (the CT scan data can be shown at the same time for any part of the liver).

5 Findings

Our method for automatically calculating the distance for sufficient space for any remaining child segments after the first child segment has been added, seems to produce good results. The preliminary results from our method applied to visualization of hepatic vessels in the liver gives good results when compared with the CT data they are based on, but these results have only been visually verified (however the first feedbacks from the Intervention Centre at Rikshsopitalet in Norway has been promising).

The algorithm is fast and simple, and can be used by most modern PC's with a graphic accelerator. The meshing algorithm mostly does its work in real-time, but the subdivision steps and any preprocessing slow things down a bit. Graphic hardware support for subdivision will hopefully be available in the relative near future. When this happens, the subdivision of the branching structures may become a viable approach even for large amounts of trees or blood vessels in real-time computer graphics.

References

[Blo85] Bloomenthal J.: Modeling the mighty maple. *Computer Graphics 19*, 3 (July 1985), 305–311.

[BW90] Bloomenthal J., Wyvill B.: Interactive techniques for implicit modeling. *Computer Graphics 24*, 2 (Mar. 1990), 109–116.

[CC78] Catmull E., Clark J.: Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer Aided Design 10*, 6 (1978), 350–355.

[CCK*89] Chaudhuri S., Chatterjee S., Katz N., Nelson M., Goldbaum M.: Detection of blood vessels in retinal images using two-dimensional matched filters. *IEEE Transactions on Medical Imageing 8*, 3 (1989), 263–269.

[EAOJ*04] Eidheim O. C., Aurdal L., Omholt-Jensen T., Mala T., Edwin B.: Segmentation of liver vessels as seen in mr and ct images. *Computer Assisted Radiology and Surgery* (2004), 201–206.

[FFKW02] Felkel P., Fuhrmann A., Kanitsar A., Wegenkittl R.: Surface reconstruction of the branching vessels for augmented reality aided surgery. *Analysis of Biomedical Signals and Images 16* (2002), 252–254. (Proc. BIOSIGNAL 2002).

[FKFW02] Felkel P., Kanitsar A., Fuhrmann A. L., Wegenkittl R.: *Surface Models of Tube Trees.* Tech. Rep. TR VRVis 2002 008, VRVis, 2002.

[GW02] Gonzalez R. C., Woods R. E.: *Digital Image Processing*, second ed. Prentice Hall, 2002.

[HB96] Hart J., Baker B.: Implicit modeling of tree surfaces. *Proc. of Implicit Surfaces '96* (Oct. 1996), 143–152.

[KSW85] Kapur J. N., Sahoo P. K., Wong A. K. C.: A new method for gray-level picture thresholding using the entropy of the histogram. *Computer Vision, Graphics, and Image Processing 29* (1985), 273–285.

[LVF*01] Lluch J., Vicent M., Fernandez S., Monserrat C., Vivo R.: Modelling of branched structures using a single polygonal mesh. In *Proc. IASTED International Conference on Visualization, Imaging, and Image Processing* (2001).

[Mai02] Maierhofer S.: *Rule-Based Mesh Growing and Generalized Subdivision Meshes.* PhD thesis, Technische Universitaet Wien, Technisch-Naturwissenschaftliche Fakultaet, Institut fuer Computergraphik, 2002.

[Mur27] Murray C. D.: A relationship between circumference and weight in trees and its bearing in branching angles. *Journal of General Phyiol. 9* (1927), 725–729.

[Opp86] Oppenheimer P. E.: Real time design and animation of fractal plants and trees. *Computer Graphics 20*, 4 (Aug. 1986), 55–64.

[Ric70] Richter J. P.: *The notebooks of Leonardo da Vinc Vol. 1.* Dover Pubns., 1970.

[Soi03] Soille P.: *Morphological Image Analysis.* Springer-Verlag, 2003.

[TMW02] Tobler R. F., Maierhofer S., Wilkie A.: A multiresolution mesh generation approach for procedural definition of complex geometry. In *Proceedings of the 2002 International Conference on Shape Modelling and Applications (SMI 2002)* (2002), pp. 35–43.

A GPU-Based Branch Deformer

Skjermo, J. (2006). A gpu-based branch deformer. In *Proceedings of the 22nd spring conference on Computer graphics*, pages 120–127. Comenius University, Comenius University, Bratislava

Abstract

This paper present a fast algorithm for generation and visualization of natural looking tree stems and branches. The outlined approach harness the power of the latest programmable consumer graphic cards to achieve real-time performance while still being able to visualize a large number of individual branches, with a high level of detail.

A GPU-Based Branch Deformer

Jo Skjermo[*]
Norwegian University of Science and Technology.
Department of Computer And Information Science.

Abstract

This paper present a fast algorithm for generation and visualization of natural looking tree stems and branches. The outlined approach harness the power of the latest programmable consumer graphic cards to achieve real-time performance while still being able to visualize a large number of individual branches, with a high level of detail.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Line and Curve Generation I.3.7 [Computer Graphics]: Three Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: graphics hardware, geometric modeling, model deformation, tree rendering

1 Introduction

As computational power increases, the use of highly detailed polygonal models becomes a viable approach for rendering natural looking trees in real-time computer graphics when close to the virtual observer.

This paper focuses on the generation and visualization of individual tree branches and stems on the graphics processing unit (GPU). The presented method takes advantage of the computational power of the programmable pipeline in the latest consumer level graphic cards available on today's market. A simple geometric level of detail (LOD) algorithm is also supported.

The algorithm generates and deforms polygonal meshes into meshes of tree stems or branches, given a set of parameters that fully defines each single branch. The algorithm works by deforming each single vertex in an input mesh, generating a new position. At the same time one calculates a tangent and binormal on the new surface, so higly detailed texturing methods can be utilized. The presented algorithm for visualizing branches runs fully on the GPU, and is specially designed to be used when other parts of a overall tree visualization system (like wind animation) also run's on the GPU.

There has been a number of proposed methods for generating polygon meshes for visualization of tree stems and branches. Historically, when using a tree generator such as L-systems [Lindenmayer and Prusinkiewicz 1990], polygon cylinders were used for each segment of a branch, sometimes stitched together with spheres in branching points.

In [Maierhofer 2002], a complete polygon mesh model for a tree was generated at the same time as the tree was grown using L-systems. This approach was further developed to be used with a predetermined centerline definition in [Felkel et al. 2002]. These methods uses Catmul-Clark subdivision [Catmull and Clark 1978], and do not easily map to implementation on the GPU, as updating the mesh when changed (for example when influenced by wind) will require the mesh to be generated again.

In [Bloomenthal 1985], Bezier curves were used to define a centerline for a branch. The polygon mesh was then generated by sweeping a reference frame along this centerline, while changing the radius. In[Weber and Penn 1995], a polygon mesh for a branch was generated by stitching together cylinder segments whose appearance was defined by parameters. In this method, the number of branches and leafs to display changed with the trees distance to the observer. Also, when a branch or a leaf became small enough, it was drawn as a line or a point instead of a polygon mesh.

When rendering trees for use in real-time computer animation, like games and simulators, an approximation by using billboards [JAKULIN 2000] [Meyer et al. 2001] has often been used. However, this approach will not produce adequate results when the virtual observer is close to the viewed object. In [Candussi et al. 2005] and in the commercial product SpeedTree [Spe], polygon meshes are switched out with texture billboards. Thus, interactive rates can be achieved even when using polygon meshes when close to the observer.

The proposed method can be explained in its simplest form as a sweep along a Bezier curve. Although similar methods has been used for visualizing tree branches in the past, our approach is especially tailored for running on the GPU, overcoming the limitations a GPU based implementation imposes.

1.1 GPU Programming and Cg

In 3D computer graphic, the process of drawing a scene is often explained as a pipeline. Historically, there has only been partial support for the programmer to change the behavior of this pipeline. When using libraries such as OpenGL or Direct3D, parts of the pipeline was hardwired in the GPU for improved performance.

However, in modern GPUs, parts of the pipeline can be fully replaced with small programs where the vertex and fragment processors are programmable (as seen in figure 1).

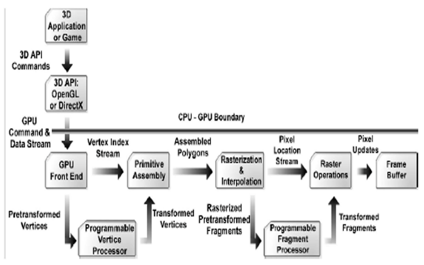

Figure 1: The programmable pipeline, from ([Fernando 2004])

[*]e-mail: jo.skjermo@idi.ntnu.no

A programming language for writing such programs, is Nvidia's Cg language. Cg provides the programmer with a standard library with highly efficient functions for many vector, matrix, texturing and lighting methods. Also, one can define the interfaces between the CPU and vertex programs, and between vertex programs and fragment programs.

One should take special note of the fact that when using the programmable vertex processors, one does not have any information about the other vertices being processed at the same time, as vertices are processed in parallel.

Together with Cg, the presented work uses OpenGL to handle the drawing and cpu-gpu communication.

2 Overall Approach

The proposed algorithm generates fully shaded and textured tree stems and branches on the GPU. A parametric description of each branch is the basis for the algorithm.

Parametric tree description was first introduced in [Honda 1971], while Weber and Penn first used a parametric tree description to visualize trees in computer graphics in [Weber and Penn 1995]. Our algorithm assumes such an approach, where a tree and its individual branches are positioned, oriented and fully described by a set of parameters.

2.1 Tree Description

A tree consists of several *levels* of branches, where the main stem is *level 0*, the branches growing out of the stem is of *level 1*, and so on. We use a simplified version of Weber and Penn's system for the generation of the parameters, positions and orientation for each branch, but note that any similar system can be used for this step as long as it can be used to produce the required parameters.

The presented approach uses a cubic Bezier curve to describe the overall shape of a branch, but also uses a set of other parameters that describes different aspects such as radius, flare, bulges and taper. Using a cubic Bezier curve, gives at the most 2 rotational centers for a single branch, enabling the definition of S-shaped branches, as often seen in nature [Jirasek et al. 2000].

To generate a tree and its branches, an extended version of the parametric generator described in [Weber and Penn 1995] is used. For a branch, the parameter set given to the generator includes angles used to generate Bezier control points for a branch. Using branch length, and the given angles, one can compute four control points, defining a Bezier curve that describes the overall shape of the branch. One should note that the arc length of the generated Bezier curve differs from the branch length parameter (increasingly as the angles increase).

To generate the control points, the branch length is first divided by 3, to get the length of the three segment vectors S_0 - S_2. For a given coordiante system, segment vector S_0 is defined to point along the X axis. The coordinate system is then translated to the end of S_0 (defining P_1). Then the coordinate system is first rotated around the direction of S_0, then around the new Z axis, with the given angles. This process can be seen in figure 2. This procedure is repeated for segment vector S_2, continuing from the present coordinate system.

The control points for the branch to be generated is then transformed and rotated into its final position in the tree, as given by the tree generation algorithm of choice.

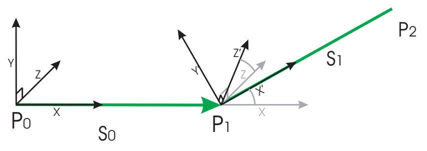

Figure 2: Part of the generation of the Bezier control points given bending angles and branch length

The presented algorithm only handles branches defined by 4 control points. Also, the presented algorithm imposes a limitation that the angle between two sucessive segments has to be less than 90 degrees (as discussed in section 3.3.2. This means that a branch can not bend more than 180 degrees overall.

2.2 The Deformer

The presented approach is basically a vertex program *deformer*, as described in [Fernando and Kilgard 2003]. The deformer works on each input vertex in three steps - input to unbend branch, unbend branch to bend branch and finally finding the needed surface coordinate system for the present vertex. The overall approach of the deformer can be seen in figure 3

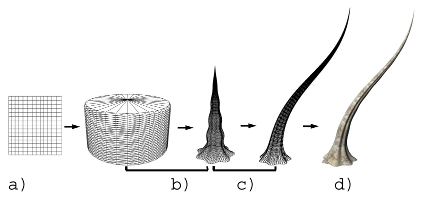

Figure 3: a) Input vertices. b)Deform to unbent branch. c) Deform to bent branch. d) Shaded branch

2.3 Vertex Buffer Object and Data Transfer

For a branch, the input to the deformer is a grid mesh of vertices, where a vertex has values going from 0 to 2π in one direction (θ), and from 0 to 1 in the other (υ). The resolution of the grid determines the resolution of the generated branch with a 1-to-1 relation. For each vertex, texture coordinates are also defined.

Several input polygon meshes with different resolution is stored in a vertex buffer object(VBO), together with the indexes needed to draw any of the grid meshes. This minimizes the data transfer between the CPU and the GPU for each draw command.

The indexes for a specific polygon mesh, is defined so that a whole grid is described by the indexes at the same time (using degenerated triangles). This way, one can draw a whole grid at the same time using the OpenGL draw command *glDrawRangeElements*. Drawing a spesific mesh grid is done by offsetting into the index array.

By using different resolutions on the grid meshes, and using different VBO's for each *level* of branches for a tree type (needed if one uses different textures on the different branch levels), one can implement a simple geometric lod algorithm, selecting on the fly which LOD level (grid resolution) to draw. An example can be seen in figure 4.

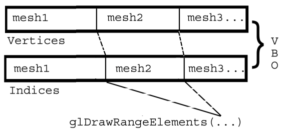

Figure 4: VBO indices and vertices layout.

2.4 Parameters

In addition to the grid mesh value (υ, θ and the texture coordinates for the color texture), the deformer also requires some other input. The values for these inputs are the same for all vertices on a specific branch, and therefore only needs to be set once per branch (as setting input values is the same as setting states in the OpenGL pipeline).

Of special interest is the four control points for the Bezier curve describing the overall shape of a branch. These values are not given as input values directly, but stored in a set of four separate 32 bit float textures. Control point P_0 is stored in texture 0, point P_1 in texture 1 and so on. For a texture, the *rgb* values are used to store the control point, while the *a* value is used to partly store a direction vector (in the direction of the parent branch, used later in the algorithm) as shown in figure 5. The input to the Vertex program is therefore the texture coordinate for its control points.

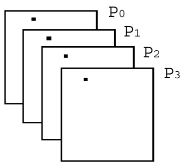

Figure 5: Four control points at a given texture address

The values in the textures are stored as follows (for rgba values):

$$\begin{array}{lcllllll} texture0 & : & P_{0(u,v)} = & P_{0,x} & P_{0,y} & P_{0,z} & O_x \\ texture1 & : & P_{1(u,v)} = & P_{1,x} & P_{1,y} & P_{1,z} & O_y \\ texture2 & : & P_{2(u,v)} = & P_{2,x} & P_{2,y} & P_{2,z} & O_z \\ texture3 & : & P_{3(u,v)} = & P_{3,x} & P_{3,y} & P_{3,z} & 0 \end{array}$$

The reason four textures are used to hold the Bezier control points, is that if using the new OpenGL *Framebuffer* extension, and the *Multiple Render Targets* functionallity introduced in pixel Shader 3.0, one can write into four output buffers at the same time in a fragment program, where the buffers can be textures. This means that changing the position of the Bezier control points (if for example one wants to simulate wind), can be done fully on the GPU using methods developed for general purpose computations on the GPU.

3 Algorithm

In this section we present the algorithm, with special attention to the fact that it is implemented as a vertex program for use on a GPU.

3.1 Deformer - Input Polygon Mesh to Cylinder

A cylinder can be described parametricly as shown in equation 1.

$$\begin{array}{rcl} x & = & \upsilon \\ y & = & \sin(\theta) radius \\ z & = & \cos(\theta) radius \end{array} \qquad (1)$$

Where υ defines the present position along the length of the cylinder, θ defines the present position as an angle around the center of the cylinder, while *radius* define the radius of the cylinder. Using the υ and θ values given as input from our vertex buffer object, one can find the corresponding position on the cylinder's surface, given the cylinder's *radius*.

3.2 Deformer - Cylinder to Branch

A cylinder is further deformed by varying the *radius* as υ and θ changes, to give the cylinder the shape of an idealized unbent branch.

This part of the deformer is defined by four functions, $f_{taper}, f_{flare}, f_{bulges} and f_{lobes}$, giving the radius of an unbent branch. The formulas used here are derived from [Weber and Penn 1995]. For all the formulas, θ is the angle around the center line, while υ is the position along the centerline defined to be between 0 and 1, as shown in equation 1

3.2.1 Taper

The f_{taper} function (equation 2) controls how fast the final radius goes toward 0.

$$f_{taper}(\upsilon) = r(1 - \upsilon^t) \qquad (2)$$

Where t is the amount of taper (how fast the branch end goes toward 0). r is the original radius of the branch. If t is 1, the final radius goes toward 0 in a linear fashion, while higher values will delay the taper until closer to the end of the branch. Some examples can be seen in figure 6.

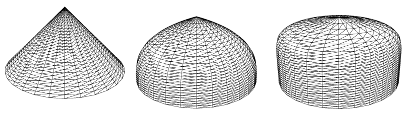

Figure 6: From left, a taper of 0, 3 and 10

3.2.2 Flare

The f_{flare} function (equation 3) adds an increased radius near the base of a branch, and is mostly used when the branch is in fact the main stem of a tree.

$$f_{flare}(\upsilon) = \frac{f}{100}(100^{(1-8\upsilon)}) + 1 \qquad (3)$$

Where f is the $flare$ parameter, defining the amount the branch should flare out near the base. Some examples can be seen in figure 7.

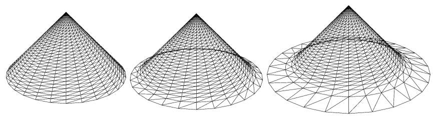

Figure 7: Flare: from left, a flare of 0, 0.1 and 0.33 (using f_{taper} with $t = 1$)

3.2.3 Bulges

The f_{bulges} function (equation 4) defines a sin-curve added to the surface radius, along the direction of the branch centerline. This can be used to generate bulges from the defined frequency and amplitude.

$$f_{bulges}(\upsilon) = 1 + b_d \sin(b_n 2\pi\upsilon) \qquad (4)$$

Where b_n is the number of bulges (frequency), and b_d is the depth of the bulges (amplitude). Some examples can be seen in figure 8.

One must remember not to set the frequency too high compared to the geometric resolution along the branch length (of the closest LOD level), as this would introduce geometric undersampling (not enough geometric resolution to sample the overall appearance of the true defined surface).

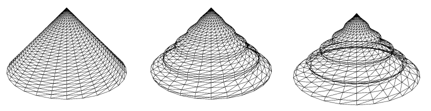

Figure 8: Bulges: from left, 5 bulges with depth of 0, 0.05 and 0.1 (using f_{taper} with $t = 1$)

3.2.4 Lobes

The f_{lobes} function (equation 5) basically does the same as the f_{bulges} function, but on the radius, as defined by the angle around the branch's centerline. In addition, an interpolation step is added, so that it is possible to decrease the lobes as ones moves along the centerline of the branch.

$$f_{lobes}(\upsilon, \theta) = 1 + l_d \sin(l_n\theta) - \upsilon l_d l_t \sin(l_n\theta) \qquad (5)$$

Where l_n is the number of lobes (frequency), l_d is the depth of the lobes (amplitude), and l_t defines how fast the lobes goes away (0 for no lobes near tip, 1 for full amount of lobes near tip). Again, it is necessary to be careful when setting values for the frequency to avoid geometric undersampling. Some examples can be seen in figure 9.

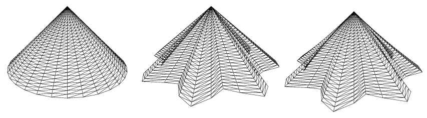

Figure 9: Lobes: from left, 7 lobes with depth of 0 and taper 0, depth 0.2 and taper 0, depth 0.2 and taper 1 (using f_{taper} with $t = 1$)

3.2.5 Deformed Cylinder

Using $f_{taper}, f_{flare}, f_{bulges} and f_{lobes}$, to define the $radius$ in equation 1, the deformer that generates an unbent branch is defined as the result of three functions, one for each axis, as shown in equation 6.

$$\begin{aligned} f_u &= \upsilon \\ f_v &= \sin(\theta) f_{taper}(\upsilon) f_{flare}(\upsilon) f_{bulges}(\upsilon) f_{lobes}(\upsilon, \theta) \\ f_w &= \cos(\theta) f_{taper}(\upsilon) f_{flare}(\upsilon) f_{bulges}(\upsilon) f_{lobes}(\upsilon, \theta) \end{aligned} \qquad (6)$$

3.3 Deformer - Branch to Bent Branch

After applying the $f_{taper}, f_{flare}, f_{bulges} and f_{lobes}$ functions, a surface of an unbent branch of length 1 is described.

The last step is to deform this surface along a qubic Bezier curve. This is done using the generalized de Casteljau approach to 3D free form deformation defined by Chang and Rockwood [Chang and Rockwood 1994].

3.3.1 Generalized de-Casteljau Approach to Deformation

The generalized de-Casteljau approach is a function $\phi[p,q] : R^3 \rightarrow R^3$ defining an affine transformation from parametric space into affine space, as defined in equation 7.

$$\phi[p,q]\begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} = \begin{bmatrix} q_x-p_x & s_x & t_x & p_x \\ q_y-p_y & s_y & t_y & p_y \\ q_z-p_z & s_z & t_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} u \\ v \\ w \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{7}$$

Where p and q are two control points (for the first iteration), while s and t are *handles* defined for the line segment between p and q, as seen in figure 10.

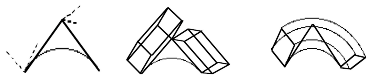

Figure 10: The generalized de Casteljau deformer, from [Chang and Rockwood 1994]

The generalized de Casteljau approach can be seen as an iterative approach to deform space around a Bezier curve, in the same way that the normal de Casteljau approach defines a Bezier curve (if s and t is 0, the generalized approach reduce to the normal de Casteljau approach for solving Bezier curves).

Also, if the *handles* are unit vectors orthogonal to the line defined by P_i and P_{i-1}, and each other, for the first level, and zero vectors for all following levels, the space will be warped with the most natural bending. This is the method used in the presented branch deformer.

3.3.2 Handle Definition

As the control points for a cubic Bezier curve are given to the GPU vertex program, only *handles* are needed to use the generalized de Casteljau approach. These can be generated using Ken Sloan's approach for a moving frame on a Bezier curve as described by Jules Bloomenthal in [Glassner 1990].

Basically, using the four control points P_0, P_1, P_2 and P_3, together with the given unit vector O in the parent branch's direction (at the banching point), Sloans approach gives us the reference frames at position P_0, P_1 and P_2 as seen in equation 8. Figure 11 shows the handle generation for a single branch.

$$\begin{aligned} L_0 &= normalize(P_1-P_0) \\ L_1 &= normalize(P_2-P_1) \\ L_2 &= normalize(P_3-P_2) \\ \\ S_0 &= L_0 \times O \\ T_0 &= S_0 \times L_0 \\ \\ T_1 &= S_0 \times L_1 \\ S_1 &= L_1 \times T_1 \\ \\ T_2 &= S_1 \times L_2 \\ S_2 &= L_2 \times T_2 \end{aligned} \tag{8}$$

Using the parent branch direction at the branching point for the present branch, means that one can not have branches growing in the same exact direction as the parent, at the branching point. Also, Sloan's method is only valid when the angle between the vectors L_0 and L_1, and L_1 and L_2 are less then 90 degrees.

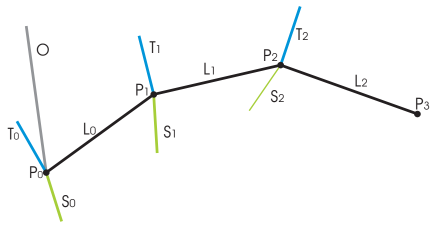

Figure 11: Generating handles for a given branch

Applying the generalized de Casteljau deformation using the f_u, f_v and f_w from equation 6 as υ, ν and ω in equation 7, gives us the final deformed branch that follows the given Bezier curve.

In Nvidias Cg, this can be done by using the *lerp* mathematical function provided in the Cg Standard Library (Cg Linear interpolation: (1-f)*a + b*f where a and b are matching vector or scalar types. Parameter f can be either a scalar or a vector of the same type as a and b). Using the generalized de Casteljau approach then gives us the point on the surface at a given υ and θ as seen in equation 9.

$$\begin{aligned} R_0^0 &= lerp(P_0,P_1,\upsilon)+S_0\nu+T_0\omega \\ R_1^0 &= lerp(P_1,P_2,\upsilon)+S_1\nu+T_1\omega \\ R_2^0 &= lerp(P_2,P_3,\upsilon)+S_2\nu+T_2\omega \\ \\ R_0^1 &= lerp(R_0^0,R_1^0,\upsilon) \\ R_1^1 &= lerp(R_1^0,R_2^0,\upsilon) \\ \\ R_0^2 &= lerp(R_0^1,R_1^1,\upsilon) \end{aligned} \tag{9}$$

R_0^2 gives the final world position for the present vertex defined by a υ and θ value. To get the projected position one must multiply this with the model-view projection matrix that is defined at the start of each frame.

Some examples of deformed branches can be seen in figure 12.

3.4 Shading and Texturing

To add lighting and texture to the deformed branch, the tangent and binormal at each surface point must be found. This also enables us to use texture normal mapping to add even further detail to a branch surface. Finding a tangent and a binormal is a two-step process. First, the tangent and binormal on the unbent branch is directly calculated using the partial derivatives of the deformer shown in equation 6. Then, the final tangent and binormal is calculated using the Jacobian of the generalized de Casteljau deformer.

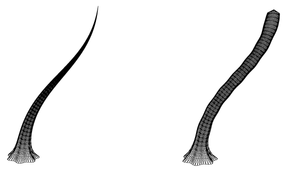

Figure 12: Two deformed branches, with the same control points, but different shape parameters

3.4.1 Partial Derivatives

The partial derivatives of the f_{taper}, f_{flare}, f_{bulges} and the f_{lobes} equations, with respect to υ is given in equation 10.

$$\begin{aligned}
\frac{\partial f_{taper}(\upsilon)}{\partial \upsilon} &= tr\upsilon^{t-1} \\
\frac{\partial f_{flare}(\upsilon)}{\partial \upsilon} &= \frac{f}{100} - 8log(100)100^{1-8\upsilon} \\
\frac{\partial f_{bulges}(\upsilon)}{\partial \upsilon} &= b_d b_n 2\pi \cos(b_n 2\pi \upsilon) \\
\frac{\partial f_{lobes}(\upsilon,\theta)}{\partial \upsilon} &= -l_d l_t \sin(l_n \theta) \qquad (10)
\end{aligned}$$

The partial derivatives of the *lobes* equation (equation 5), with respect to θ is given in equation 11.

$$\frac{\partial f_{lobes}(\upsilon,\theta)}{\partial \theta} = l_d l_n \cos(l_n\theta) - l_n l_t \cos(l_n\theta)\upsilon \qquad (11)$$

The partial derivative of the $taper, flare$ and $bulges$ with respect to θ all gives 0.

Using the product rule for derivation, the partial derivatives of equation 6 with regard to υ and θ, will give the tangent and binormal at the unbent branch surface. We define the tangent Ts as shown in equation 12.

$$\begin{aligned}
Ts_x &= \cos(\theta)(\frac{\partial f_{lobes}(\upsilon,\theta)}{\partial \upsilon} f_{bulges}(\upsilon) f_{flare}(\upsilon) f_{taper}(\upsilon) \\
&+ f_{lobes}(\upsilon,\theta)\frac{\partial f_{bulges}(\upsilon)}{\partial \upsilon} f_{flare}(\upsilon) f_{taper}(\upsilon) \\
&+ f_{lobes}(\upsilon,\theta) f_{bulges}(\upsilon)\frac{\partial f_{flare}(\upsilon)}{\partial \upsilon} f_{taper}(\upsilon) \\
&+ f_{lobes}(\upsilon,\theta) f_{bulges}(\upsilon) f_{flare}(\upsilon)\frac{\partial f_{taper}(\upsilon)}{\partial \upsilon}) \\
Ts_y &= 1 \\
Ts_z &= \sin(\theta)(\frac{\partial f_{lobes}(\upsilon,\theta)}{\partial \upsilon} f_{bulges}(\upsilon) f_{flare}(\upsilon) f_{taper}(\upsilon) + \\
&+ f_{lobes}(\upsilon,\theta)\frac{\partial f_{bulges}(\upsilon)}{\partial \upsilon} f_{flare}(\upsilon) f_{taper}(\upsilon) + \\
&+ f_{lobes}(\upsilon,\theta) f_{bulges}(\upsilon)\frac{\partial f_{flare}(\upsilon)}{\partial \upsilon} f_{taper}(\upsilon) + \\
&+ f_{lobes}(\upsilon,\theta) f_{bulges}(\upsilon) f_{flare}(\upsilon)\frac{\partial f_{taper}(\upsilon)}{\partial \upsilon}) \qquad (12)
\end{aligned}$$

The binormal Bs on the surface of the unbent branch is given in equation 13.

$$\begin{aligned}
Bs_x &= -\sin(\theta) f_{lobes}(\upsilon,\theta) f_{bulges}(\upsilon) f_{flare}(\upsilon) f_{taper}(\upsilon) \\
&+ \cos(\theta)\frac{\partial f_{lobes}(\upsilon,\theta)}{\partial \theta} f_{bulges}(\upsilon) f_{flare}(\upsilon) f_{taper}(\upsilon) \\
Bs_y &= 0 \\
Bs_z &= \cos(\theta) f_{lobes}(\upsilon,\theta) f_{bulges}(\upsilon) f_{flare}(\upsilon) f_{taper}(\upsilon) \\
&+ \sin(\theta)\frac{\partial f_{lobes}(\upsilon,\theta)}{\partial \theta} f_{bulges}(\upsilon) f_{flare}(\upsilon) f_{taper}(\upsilon) \qquad (13)
\end{aligned}$$

3.5 The Bent Branch

A method for generating a tangent and binormal (for normal generation) on a parametricly defined surface, suitable for use in GPU programs, is presented in [Fernando 2004].

Given $f(x,y,z) = (f_x, f_y, f_z)$ the Jacobian matrix is defined as shown in equation 14

$$J(x,y,z) = \begin{bmatrix} \frac{\partial f_x}{\partial x} & \frac{\partial f_x}{\partial y} & \frac{\partial f_x}{\partial z} \\ \frac{\partial f_y}{\partial x} & \frac{\partial f_y}{\partial y} & \frac{\partial f_y}{\partial z} \\ \frac{\partial f_z}{\partial x} & \frac{\partial f_z}{\partial y} & \frac{\partial f_z}{\partial z} \end{bmatrix} \qquad (14)$$

If the unit tangent and binormal vectors T and B are given on the surface before deformation, multiplying these vectors with the Jacobian will give the deformed tangent and binormal. Also, one can calculate a deformed unit normal N by using:

$$n` = normalize[(J(x,y,z)t \times (J(x,y,z))b)]$$

Using this approach, one first takes the partial derivative by υ and θ on the equations for each axis in equation 6. This gives a tangent and binormal on the surface of the cylinder deformed to an unbent branch. Multiplying these with the Jacobian of the de-Casteljau deformation, will give us the final deformed tangent and binormal.

Calculating the Jacobian can be done as shown in equation 15.

$$\begin{aligned}
\frac{\partial Q}{\partial \upsilon} &= 3\upsilon^2(P_3 - 3P_2 + 3P_1 - P_0) \\
&+ 6\upsilon(P_2 - 2P_1 + P_0) + 3(P_1 - P_0) \\
&+ 2\upsilon\nu(S_2 - 2S_1 + S_0) \\
&+ 2\upsilon\omega(T_2 - 2T_1T_0) \\
&+ 2\nu(S_1 - S_0) + 2\omega(T_1 - T_0) \\
\frac{\partial Q}{\partial \nu} &= \upsilon^2(S_2 - 2S_1 + S_0) + 2\nu(S_1 - S_0) + S_0 \\
\frac{\partial Q}{\partial \omega} &= \upsilon^2(T_2 - 2T_1 + T_0) + 2\omega(S_1 - T_0) + T_0
\end{aligned}$$

$$J(\upsilon,\nu,\omega) = \begin{bmatrix} \frac{\partial Q}{\partial \upsilon} \\ \frac{\partial Q}{\partial \nu} \\ \frac{\partial Q}{\partial \omega} \end{bmatrix} \tag{15}$$

For a Bezier curve of degree n with control points P_i, the first parametric derivative can be expressed as a curve of degree $n-1$ with control points D_i where $D_i = n(P_{i+1} - P_i)$. This curve is known as the *hodograph*.

Using the knowledge that the derivative of a Bezier curve can be calculated by using the *hodograph*, and rearranging equation 15 somewhat, gives us a new approach for finding the Jacobian of the generalized de-Casteljau deformer, that is more efficient to calculate on a GPU. Equation 16 shows this approach.

$$\begin{aligned}
L_0^J &= 3(P_1 - P_0) \\
L_1^J &= 3(P_2 - P_1) \\
L_3^J &= 3(P_3 - P_2) \\
\\
S_0^J &= 2(S_1 - S_0) \\
S_1^J &= 2(S_2 - S_1) \\
\\
T_0^J &= 2(T_1 - T_0) \\
T_1^J &= 2(T_2 - T_1) \\
\\
\frac{\partial Q}{\partial \upsilon} &= lerp(lerp(L_0^J, L_1^J, \upsilon), lerp(L_1^J, L_2^J, \upsilon), \upsilon) \\
&+ \nu lerp(S_0^J, S_1^J, \upsilon) \\
&+ \omega lerp(T_0^J, T_1^J, \upsilon) \\
\frac{\partial Q}{\partial \nu} &= lerp(lerp(S_0^J, S_1^J, \upsilon), lerp(S_1^J, S_2^J, \upsilon), \upsilon) \\
\frac{\partial Q}{\partial \omega} &= lerp(lerp(T_0^J, T_1^J, \upsilon), lerp(T_1^J, T_2^J, \upsilon), \upsilon)
\end{aligned}$$

$$J(\upsilon,\nu,\omega) = \begin{bmatrix} \frac{\partial Q}{\partial \upsilon} \\ \frac{\partial Q}{\partial \nu} \\ \frac{\partial Q}{\partial w} \end{bmatrix} \tag{16}$$

Using the Jacobi matrix, the tangent, normal and binormal can be calculated, and a TBN matrix can be constructed. The TBN matrix is used to calculate the light direction, the view direction (and the half direction between those), in texture space (from input values to the vertex shader program giving light direction and view direction in global space).

The light direction, view direction and the half direction (together with the position) is the output from the vertex program for each vertex. This is also the input to the fragment program, that does the texture lookup in the normal and color textures, and calculates the final color for each fragment.

Using the tangent and binormal on the unbend branch surface defined in equation 12 and equation 13, we calulate the TBN matrix as follows:

$$\begin{aligned}
T &= normalize(mul(Ts, J(\upsilon,\nu,\omega))) \\
N &= normalize(T \times mul(Bs, J(\upsilon,\nu,\omega))) \\
B &= N \times T
\end{aligned}$$

$$TBN = \begin{bmatrix} T \\ B \\ N \end{bmatrix}$$

Where *mul* is a vector-matrix multiplication provided in the Cg Standard Library. To get the light direction and view direction one simply multiply with the TBN matrix respectively.

4 Results and Future Works

Some examples of the algorithm using normal textures and ligtning can be seen in figure 13, while figure 14 shows the branche generation method used to draw a smal number of trees with a high number of branches.

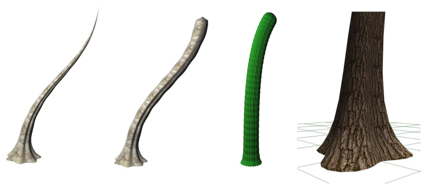

Figure 13: From left: shaded branches with texturing and lightning, shaded branch with different parameters, the stem of a cacti, a closer look at the start of a stem

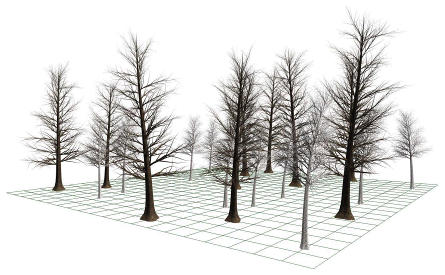

Figure 14: 20 trees of 2 different species (Black Tupelo and Quaking Aspen), 335000 vertices in 17400 branches at 15 fps

Early testing shows that the proposed algorithm manages to process up to 165000 vertices while maintaining a framerate of 30 frames per second, when drawing up to 10000 branches, with lightning and texturing enabled on an Nvidia 6800GT GPU. Some results from drawing with different number of branches and vertices can be seen in figure 15.

The presented approach reads the Bezier control points and the parent branch direction from a texture. Another use is to feed the GPU

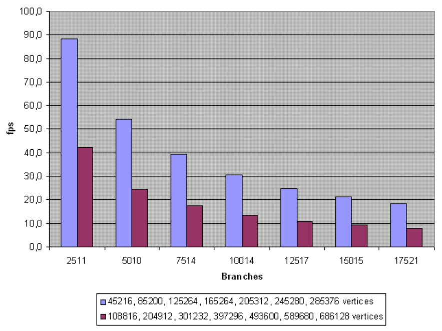

Figure 15: Frames per secound for different number of branches and vertices

these values as texture coordinates for each drawing call (once per branch). This approach could for instance be used if wind animation of the Bezier branches is done on the CPU (only working with the Bezier centerline curves). The GPU is then used for offloading the calculation of the deformation for the vertices (compared to doing all the work on the CPU).

Adding wind animation to the trees on the GPU can be done using previously mentioned OpenGl extensions, enabling one to update the Bezier control points for all branches of a specific level in one pass. Also, it is planned to add leaves by using texture billboards to simulate leaf clusters, done in similar ways as in the SpeedTree software (including rotation around the view-axis to simulate small wind influence).

Another area of interest is to add another layer of detail to the branches. Although one can produce a number of distinct branch types with the proposed method, all the branches of a type will look quite similar. Adding more control to the deformation formulas is one option for allowing small changes to the overall look of branches.

Even with the option of rendering branches with different geometric LOD levels, adding another LOD approach to increase performance is planned. Simply switching the polygon meshes in a tree into a billboard as the lowest LOD level, seems promising for increased performance.

References

BLOOMENTHAL, J., AND WYVILL, B. 1990. Interactive techniques for implicit modeling. *Computer Graphics 24*, 2 (Mar.), 109–116.

BLOOMENTHAL, J. 1985. Modeling the mighty maple. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 305–311.

CANDUSSI, A., CANDUSS, N., , AND HLLERER, T. 2005. Rendering realistic trees and forests in real time. *Eurographics journal, Computer Graphics Forum 24*, 73–76. EG Short Presentations.

CATMULL, E., AND CLARK, J. 1978. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer Aided Design 10*, 6, 350–355.

CHANG, Y. K., AND ROCKWOOD, A. 1994. A generalized de casteljau approach to 3d free-form deformation. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1994*, ACM Press, SIGGRPAH, 257–260.

FELKEL, P., FUHRMANN, A., KANITSAR, A., AND WEGENKITTL, R. 2002. Surface reconstruction of the branching vessels for augmented reality aided surgery. *Analysis of Biomedical Signals and Images 16*, 252–254. (Proc. BIOSIGNAL 2002).

FERNANDO, R., AND KILGARD, M. J., Eds. 2003. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison Wesley Professional, 218–226.

FERNANDO, R., Ed. 2004. *GPU Gems*. Addison-Wesley Profesional.

GLASSNER, A., Ed. 1990. *Calculation of Reference Frames along a Space Curve*. Academic Press, 567–574. By Jules Bloomenthal.

HART, J., AND BAKER, B. 1996. Implicit modeling of tree surfaces. *Proc. of Implicit Surfaces '96* (Oct.), 143–152.

HONDA, H. 1971. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. *Journal of Theoretical Biology*, 331–338.

JAKULIN, A. 2000. Interactive vegetation rendering with slicing and blending. In *In Proceedings of Eurographics 2000*. Short Presentations.

JIRASEK, C., PRUSINKIEWICZ, P., AND MOULIA, B. 2000. Integrating biomechanics into developmental plant models expressed using l-systems. *Plant biomechanics*, 615–624.

LINDENMAYER, A., AND PRUSINKIEWICZ, P. 1990. *The Algorithmic Beauty of Plants*. Springer-Verlag.

LLUCH, J., VICENT, M., FERNANDEZ, S., MONSERRAT, C., AND VIVO, R. 2001. Modelling of branched structures using a single polygonal mesh. In *Proc. IASTED International Conference on Visualization, Imaging, and Image Processing*.

MAIERHOFER, S. 2002. *Rule-Based Mesh Growing and Generalized Subdivision Meshes*. PhD thesis, Technische Universitaet Wien, Technisch-Naturwissenschaftliche Fakultaet, Institut fuer Computergraphik.

MEYER, A., NEYRET, F., AND POULIN, P. 2001. Interactive rendering of trees with shading and shadows. In *In Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, Springer-Verlag, 183–19.

OPPENHEIMER, P. E. 1986. Real time design and animation of fractal plants and trees. *Computer Graphics 20*, 4 (Aug.), 55–64.

Speedtree, idv inc. Commersial product, http://www.speedtree.com.

WEBER, J., AND PENN, J. 1995. Creation and rendering of realistic trees. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 119–128.

GPU-Based Wind Animation of Trees

Skjermo, J. (2007). GPU-Based Wind Animation of Trees. In Lim, I. S. and Duce, D., editors, *Theory and Practice of Computer Graphics*, pages 29–36, Bangor, United Kingdom. Eurographics Association

Abstract

This paper present a simplified approach to wind animation of natural looking tree stems and branches. The presented approach is composed from several earlier works by a number of authors, each adapted to increase its suitability for processing on a Graphic Processing Unit (GPU). The outlined approach uses two passes through the GPU. The first pass samples from a simple wind force simulator based on sine sums. It then animates the parameters and the control points defining each branch using the sampled force, taking advantage of the parallel nature of GPU's. The second pass uses a previously presented GPU-based deformer to generate and render actual models of each branch, using the animated control points.

Generation and Tessellation of Tree Stems

Skjermo, J. (2008). Generation and Tessellation of Tree Stems. In Lim, I. S. and Tang, W., editors, *Theory and Practice of Computer Graphics*, pages 163–166, Manchester, United Kingdom. Eurographics Association

Abstract

When visualizing tree stems and branches for use in interactive applications, the polygon models resolution are usually as low as possible to achieve a high frame rate. Also, to ease animation and mesh generation, each branch of a tree model is often considered as a distinct mesh. However, by using a single watertight mesh for a tree, together with (GPU-based) tessellation, both the resolution and appearance of a tree can be greatly improved while maintaining a high frame rate. This paper presents concepts, ideas and early work on generating watertight polygon meshes of animated trees stems suitable for refinement and tessellation of such meshes.

Part III

Appendices

Appendix A

Real-time analysis of ultrasound images using GPU

Eidheim, O. C., Skjermo, J., and Aurdal, L. (2005). Real-time analysis of ultrasound images using gpu. In Lemke, H. U., Inamura, K., Doi, K., Vannier, M. W., and Farman, A. G., editors, *CARS 2005 - Proceedings of the 19th International Congress and Exhibition*, pages 284–289

Abstract

Analysing the increasing number of medical images stemming from various imaging modalities is a time consuming task for the clinicians. Presently, medical images are mainly handsegmented and manually analysed. Automatic tools are needed to make medical image analysis easier, and to create better presentations of information needed prior to and during surgery. In this article, we focus on real-time analysis of ultrasound images, and on how inexpensive graphics hardware can be utilised to accelerate current image processing techniques. Algorithms evaluated on the GPU were mathematical morphology, gradient vector flow, and the snake model. The results are promising and show that a large number of iterations can be run in 1/24 of a second.

Real-time analysis of ultrasound images using GPU

O.C. Eidheim[a,*], J. Skjermo[a], L. Aurdal[b]

[a]*Norwegian University of Science and Technology*
[b]*Norwegian Computing Center*

Abstract. Analysing the increasing number of medical images stemming from various imaging modalities is a time consuming task for the clinicians. Presently, medical images are mainly hand-segmented and manually analysed. Automatic tools are needed to make medical image analysis easier, and to create better presentations of information needed prior to and during surgery. In this article, we focus on real-time analysis of ultrasound images, and on how inexpensive graphics hardware can be utilised to accelerate current image processing techniques. Algorithms evaluated on the GPU were mathematical morphology, gradient vector flow, and the snake model. The results are promising and show that a large number of iterations can be run in 1/24 of a second.

Keywords: GPU; ultrasound; segmentation; snake

1. Introduction

Medical imaging using CT, MR, ultrasound, PET, and other modalities have revolutionised the diagnosis and treatment of numerous diseases. The radiologists and surgeons are presented with 2D or 3D images giving them a detailed view of the patient's anatomy. However, the task of analysing the data can be time-consuming and error-prone.

Automatic image analysis can partially solve the previously mentioned problems. Using parallel computing most practically occurring image processing tasks may be performed in reasonable time given adequate computing power. Furthermore, the results would be identical independent of who executed the analysis.

Another challenge arises during ultrasound analysis where the user is presented with a video stream, and analysis of the stream is performed concurrently. Here, image analysis tasks must be run in real-time and the delay between the input devices and the output devices must be minimal.

Since image analysis techniques typically are resource demanding, a cluster of computers is typically needed to perform all the operations necessary in a medical application in real-time. This solution is often impractical, however, recent advances in GPU (Graphics Processing Unit) performance on inexpensive graphic cards has made it possible to solve very complex image processing tasks efficiently.

[*] Corresponding author. Tel.: +47 73591717; fax: +47 73594466.
E-mail address: eidheim@idi.ntnu.no.

We have looked at two ultrasound applications. The first application consists of extracting the left ventricle walls for ischemia diagnosis. By locating the ventricle walls during heart cycles, analysis of the heart movement can be performed. In the second application, we automatically detect lesions in the liver. The purpose here is to more accurately determine the positions of previously detected lesions. These lesions are typically found prior to surgery through CT or MR scans.

1.2 Previous work

Programmable GPUs became available in 2000 to let developers get more control over the rendering pipeline on graphics hardware. The main motivation was increased performance, and the technology has been continuously pushed forward by the gaming industry.

Modern GPUs are highly optimised for *stream* computations as described in [1], and using the GPU as a *stream co-processor* is gaining popularity as it often outperforms the CPU for such tasks. In addition, todays GPUs also include *spatial parallelism* in that each pixel on the screen can be seen as a stream processor. This results in a high degree of parallelism when used appropriately.

According to [2], GPUs may increase the speed of the algorithms previously run on the CPU by more than 10 times. Furthermore, the development of GPUs does not follow Moore's law, but advances even more rapidly than the development of CPUs.

Several advanced image processing methods have been implemented using GPGPU (General-Purpose computation on GPUs) for speed improvements. Some examples can be seen in [2], including segmentation by the level-set method, the Hough Transform, and motion estimation by eigenvector analysis of spatio-temporal tensors. Another example is segmenting the brain surface from an MRI data set by the level-set method [3].

2. Methods

All operations described in this section were implemented on the GPU. The image processing tasks were implemented on the fragment shader using Nvidia's Cg programming language in combination with C++ and OpenGL. The graphics card used was a GeForce 6600 GT on a 2.8 GHz Intel Pentium 4 computer.

The two most computationally expensive stages in medical image analysis are frequently preprocessing and segmentation. The procedures are typically iterative, and calculations are commonly executed for each pixel in the image. Fortunately, one procedure is often repeatedly executed on each pixel for each iteration. This matches the stream computation model and spatial parallelism in GPGPU.

As seen in figure 1, the border of the sought regions may be challenging to find. In the first image, showing a left ventricle, the motivation is to locate the ventricle walls in order to calculate the ventricle volume for ischemia diagnosis. Similarly, in the second example, the goal is to segment the tumour in the liver.

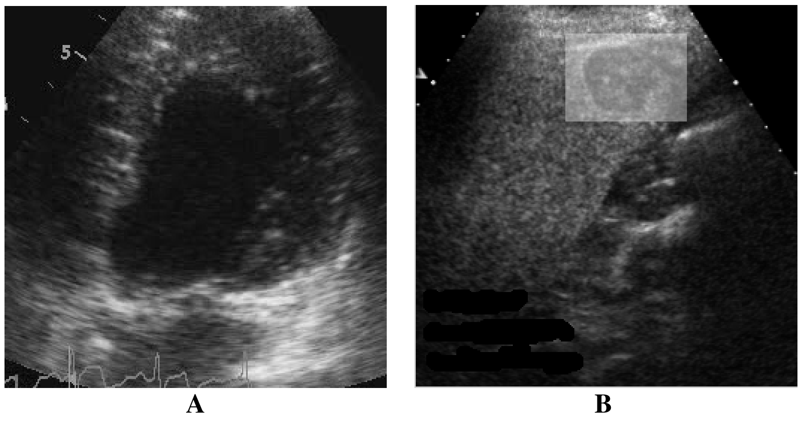

Fig. 1. This figure shows two examples of ultrasound images to be processed and segmented. The border of the sought regions may be fuzzy and undefined. a) Ultrasound image of a left heart ventricle. b) Ultrasound image showing a liver tumour.

During the preprocessing step in both applications, grey level mathematical morphology [4] was used in to reconstruct the fuzzy borders. Since the regions to be segmented were darker than the surrounding tissue, we first used grey level morphological dilation in order to make the borders physically larger. After this, grey level morphological erosion was computed to reduce the borders to original size, although, as a result some border elements are now connected. The described processing steps also corresponds to what is commonly called a morphological closing [4].

The implementation of grey level morphological operators on the GPU is similar to the implementation of convolution on the GPU as described in [5]. Morphological operators are based on morphological erosion and dilation, which is simply local minima and maxima filters over specific domains respectively. However, in order to run these operators effectively on the GPU, one has to take advantage of the built-in min and max functions of the shading language. This is due to the fact that branching, e.g. if-branches, are very expensive operations on the GPU.

After having implemented the two basic morphological operators, namely erosion and dilation, a wide range of morphological methods can be run easily. A few examples are thinning, skeletonising, thickening, geodesic erosion, geodesic dilation, reconstruction, and the distance transform [4]. Still, morphological reconstruction can be more efficiently implemented on the CPU using the algorithm described in [6].

After the borders of the sought regions have been emphasised, the next step is to apply a segmentation algorithm. Our method of choice was the active contour model described in [7]. This model is popularly called the snake model, and is basically a set of connected nodes that are moved according to internal snake forces as well as forces external to the snake arising from the image itself or from specific user constraints. The internal forces are commonly proportional to the first and second spatial derivative, representing tension and rigidity of the snake respectively. Image and external forces are typically forces set to guide the snake towards sought regions of an image. An example of image forces is the image gradient, while balloon forces [8] is an example of external forces.

To attract the snake towards the left ventricle walls or the walls of the tumour, we need to compute forces from the image that will achieve this. Our solution was to calculate the gradient vector flow [9, 10] of the ultrasound images. This method attracts the snake towards edges with increased capture range, and also attracts the snake into concavities. The process of calculating the gradient vector flow, however, is time-consuming and resource demanding. This is due to the need of an algorithm that has to be run for each pixel iteratively. This led to an implementation on the GPU, which greatly reduced the processing time of the procedure.

The general position of the sought regions of both applications is assumed known, either from centring the ultrasound device on the left ventricle, or from previous segmentations of the liver tumours from CT or MR scans. Thereby, we used this general position, assumed to be inside of the sought region, to create a small initial snake. By the means of balloon forces [8] we then expanded the snake until stability, and the snake's final position was used to segment the region.

3. Results

The most time-consuming procedures of our methods are the preprocessing steps. By implementing these procedures on the GPU, we received a running time reduction of 34 times when processing 512x512 images compared to the CPU. The runtime of the whole procedure is 54 ms (resulting in a frame rate of 18.5 fps), and thus the processing steps outlined in the previous section can be run in real-time using the GPU. Final segmentation results can be seen in figure 2.

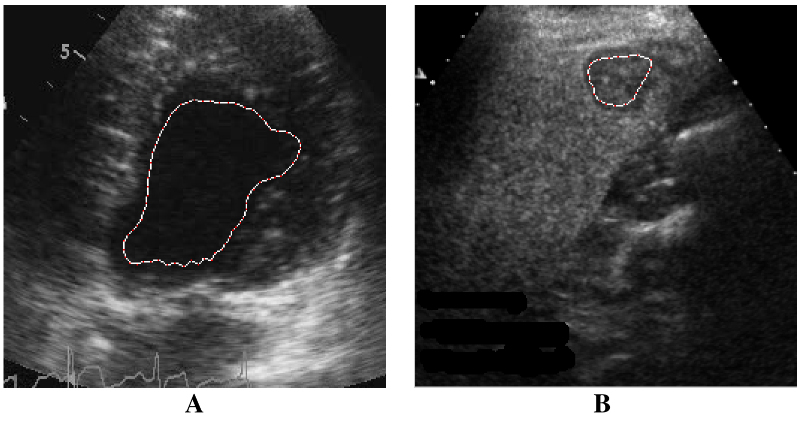

Fig. 2. The final segmentation results can be seen in this figure. a) Outlined left ventricle wall. b) Segmented liver tumour.

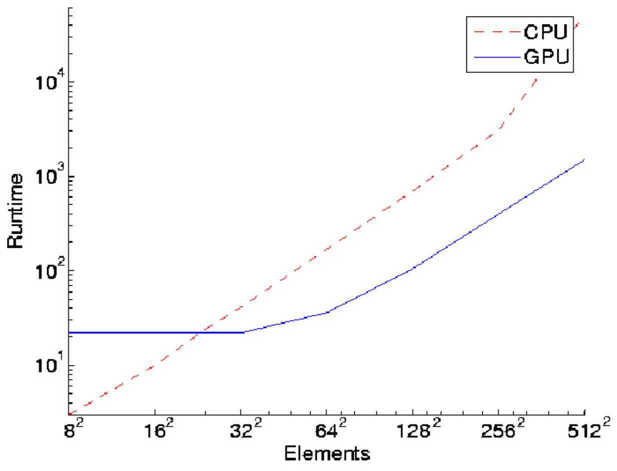

Fig. 3. This figure shows a comparison of procedures running on the GPU and the CPU. The number of iterations run on each test were 2400 (100 iterations 24 times a second). If the number of elements exceeds 24^2, the processing time of the GPU is less than that of the CPU. For instance, the processing time of an image of size 512x512 is 34 times faster on the GPU than the CPU on our test platform.

Figure 3 shows the runtime of GPU programs compared to CPU programs using our setup. If the number of elements are lower than approximately 24^2, it is more efficient to run the procedure on the CPU. In our applications, the number of nodes in our snake model did not exceed this number, and we therefore chose to run the snake model on the CPU instead of the GPU.

Calculations on the GPU are not as exact as calculations on the CPU due to inherent difficulties with data types larger than 32-bit. In our applications, this drawback resulted in only minor artifacts, and we argue that our results are adequately accurate for medical usage.

A possible processing bottleneck is the AGP bus when moving data from CPU to GPU memory. The AGP bus is asymmetric, meaning data is sent faster to the graphics card than back. A solution to this problem is to some extent offered by the new PCI express cards that are full duplex and sends data to the graphics card at twice the rate of 8x AGP cards. A data stream of up to approximately 4 Gbps is attainable on this bus presently.

4. Conclusion

We have successfully implemented several complex image processing techniques for real-time processing of ultrasound video. The computations were executed on the GPU of an inexpensive modern graphics card. Consequently, the computations were run in parallel without the expense of a large computer cluster or a multiprocessor computer. This can provide attainable and practical automatic solutions that offer valuable assistance in ultrasound analysis.

References

[1] S. Venkatasubramanian, The Graphics Card as a Stream Computer, SIGMOD Workshop on Management and Processing of Data Streams (2004).

[2] R. Strzodka, GPGPU: General-Purpose Computing on Graphics Processors, IEEE Visualization (2004).

[3] A. E. Lefohn, J. M. Kniss, C. D. Hansen, R. T. Whitaker, Interactive Deformation and Visualization of Level Set Surfaces using Graphics Hardware, IEEE Visualization (2003).

[4] P. Soille, Morphological Image Analysis, Springer-Verlag (2003).

[5] R. J. Rost, The OpenGL Shading Language, Addison-Wesley Professional (2004).

[6] L. Vincent, Morphological Grayscale Reconstruction in Image Analysis: Applications and Efficient Algorithms, IEEE Transactions on Image Processing 2 (2) (1993) 176-201.

[7] M. Kass, A. Witkin, D. Terzoploulos, Snakes: Active Contour Models, International Journal of Computer Vision (1988) 321-331.

[8] L. D. Cohen, On active contour models and balloons, CVGIP: Image Understand. 53 (1991) 211-218.

[9] C. Xu, J. L. Prince, Snakes, Shapes, and Gradient Vector Flow, IEEE Transactions on Image Processing, 7 (3) (1998) 359-369.

[10] C. Xu, J. L. Prince, Gradient Vector Flow Deformable Models, Academic Press, Handbook of Medical Imaging, (2000) 159-169.