NTNU
Norwegian University of
Science and Technology

Parametric Generation of Polygonal Tree Models for Rendering on Tessellation-Enabled Hardware

Jørgen Nystad

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Torbjørn Hallgren, IDI
Co-supervisor: Jo Skjermo, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Recent movements in the graphics hardware field provide direct hardware support for tessellation which should improve the performance of GPU-based subdivision. To take advantage of this in visualization of trees, a system for generating single body polygonal models of these subjects is needed. These should be generated with a high degree of subdivision in mind.

The goal is to define a parametric system for generating tree models, taking into account natural rules of growth. The models must be suitable for and exhibit a high level of realism after subdivision. Accompanying rendering methods for achieving near-photorealistic appearance should be explored.

Assignment given: 15. January 2010
Supervisor: Torbjørn Hallgren, IDI

Abstract

The main contribution of this thesis is a parametric method for generation of single-mesh polygonal tree models that follow natural rules as indicated by da Vinci in his notebooks [Ric70]. Following these rules allow for a relatively simple scheme of connecting branches to parent branches. Proper branch connection is a requirement for gaining the benefits of subdivision. Techniques for proper texture coordinate generation and subdivision are also explored.

The result is a tree model generation scheme resulting in single polygonal meshes susceptible to various subdivision methods, with a Catmull-Clark approximation method as the evaluated example.

As realistic visualization of tree models is the overall objective, foliage appearance and the impression of a dense branching structure is considered. A shader-based method for accurately faking high branch density at a distance is explored.

Preface

This master's thesis is the final part of a five year programme leading to the degree Master of Science in Computer Science, specialized in visualization. The programme was completed in entirety at the Norwegian University of Science and Technology, in Trondheim, Norway.

This thesis was conducted at the Department of Computer and Information Science during the spring semester of 2010. My supervisor was Assistant Professor Torbjørn Hallgren, aided by my co-supervisor, Associate Professor Jo Skjermo.

I would like to thank both my supervisors for valuable support and guidance throughout the semester.

Trondheim, Norway
June 2010

Jørgen Nystad

Contents

List of Figures

List of Listings

Glossary

CPU Central Processing Unit.

DAG Directed Acyclic Graph.

DirectX 11 Microsoft's API for handling multimedia related tasks, including graphics. DirectX 11 is the latest of the series.

GPU Graphics Processing Unit.

LOD Level of detail. In computer graphics, typically denotes a scheme for decreasing the complexity of a mesh based on distance, viewing angle or other metrics to avoid rendering details that will not be visible for the viewer, and thus increasing the efficiency of rendering.

Mesh In computer graphics, an interconnected network of vertices, defining edges and faces, typically representing a 3D model.

MIP-level When using MIP maps for textures, the texture is stored in different resolutions. The originally sized texture is MIP-level 0, the next level is typically half the width and height of the previous level till the last, which is 1x1 pixels. MIP-levels are chosen based on occupied screen space to reduce flickering.

Polygon Closed path of straight lines defining a surface, usually planar, but not necessarily so in computer graphics.

Shader A program designed to run on a GPU, usually performing operations on vertex primitives or fragments.

Subdivision In computer graphics, the process of calculating a smooth surface based on a coarse mesh.

Tessellation In computer graphics, the process of dividing a polygon face into many smaller faces.

Vertex The fundamental primitive for describing meshes/graphs. In computer graphics, a vertex describes a corner of a surface (three vertices describes a triangle, etc.).

Chapter 1

Introduction

1.1 Motivation

Generation of tree models is not a new field. Several models and methods of generation already exist, one of which is the basis for the method in this thesis. However, a common shortcoming of these methods is that they either ignore the geometric side or let the resulting geometry be unconnected at branching points. Instead, child branches (usually represented by tapered cylinders) tend to originate inside its parent branch's geometry.

This causes several problems. Firstly, animation (e.g. wind displacements) usually makes the child branch's point of attachment appear to move around on the parent branch. Secondly, the texture mapped to these surfaces have no continuity at branching points, making the attachment seem unrealistic, even without animation.

With recent evolutions in the field of graphics hardware, another problem have also become more relevant for the casual gaming or simulation industry aiming for high realism. The lack of proper branch connections makes the tree models unsuitable for subdivision.

1.1.1 Target Scale

When talking about tree rendering, it is important to define which scale is targeted. In the work by Garcia et al. [GSSK05], three scales are defined; *vehicle scale*, *human scale* and *insect scale*. As may be deduced from the names, *vehicle scale* refers to relatively large viewing distances, where the trees are background objects never viewed up close. *Human scale* refers to relatively close distances, usually viewed from the ground, but with the ability to walk around a tree and viewing it from different angles. At the *insect scale*, one may imagine walking around on the tree itself, tripping in ridges and leaves. The geometry at this scale must naturally be highly detailed and natural-looking for a realistic experience.

In this report, the target scale is usually closest to the *human scale*, moving towards *insect scale*.

1.1.2 Tessellation-Enabled Hardware

Let us define two terms for this context.

Tessellation is the process of taking a geometric primitive and dividing it into many smaller primitives, increasing the density, or resolution, of the original geometry.

Subdivision is the process of taking a coarse mesh and tessellating it into a finer mesh, adding smoothness and/or detail in the process. Subdivision surfaces are common in 3D content creation software, allowing the artist to manipulate a small set of points relative to the generated smooth surface.

The new generation of graphics hardware have built-in support for tessellation. Looking at the new DirectX 11 [Gee08] pipeline (see figure 1.1), we have the addition of three stages. The hull shader follows the traditional vertex shader, setting up parameters for tessellation in the fixed tessellator. The tessellator outputs the geometry with increased resolution to the domain shader, which determines the positions and attributes of the generated geometric points (vertices). These are then passed to the geometry shader or pixel shader as before.

The result is the ability to perform subdivision directly in hardware, without the need of sending the high resolution geometry from CPU to GPU. Sending geometry may be a costly process, and could very well become the bottleneck for high quality rendering of polygon meshes.

It is also possible to animate the mesh on the graphics hardware, prior to tessellation. This gives the benefits of cheap animation of a coarse mesh, and the visual quality of a high resolution subdivision mesh.

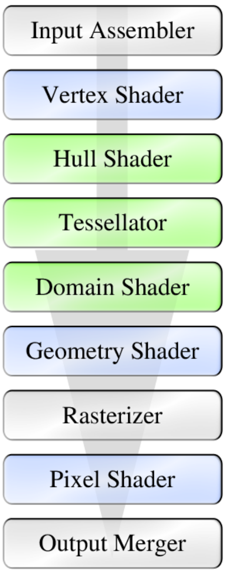


Figure 1.1: *DirectX 11 rendering pipeline*

1.1.3 Subdividing Tree Models

Subdivision of tree models may be beneficial in several ways. First of all, the increased resolution together with smoothing should make the tree model more visually appealing, hiding geometric discontinuities and enabling support for a high level of detail, by for example displacement mapping.

Second of all, reducing the amount of geometry that needs to be transferred from the CPU to the GPU may increase performance of high quality rendering of trees. This may in the future enable entire forest scenes to be rendered with subdivision.

Another benefit of subdivision in hardware is the innate LOD (Level of Detail) scheme present due to the dynamic control of tessellation factors. By setting the tessellation factors based on distance from viewer, the amount of subdivision can be reduced or turned off when the gained detail or quality would be wasted.

Subdivision requires properly connected geometry. This is not usually the case with existing methods for tree generation, and they will therefore not obtain one of the main benefits of subdivision, namely smoothing the coarseness at branching points as needed.

1.2 Problem Area

The primary focus of this project is the generation of polygonal tree meshes suitable for subdivision on tessellation-enabled hardware. This entails the need for properly connected geometry. The visual appearance should also be close to realistic. Therefore, texture coordinate generation and treatment at subdivision is an important aspect of the process.

Foliage and the appearance of a dense branch structure is another aspect of achieving realism. Techniques for generation of these elements and related visual improvements are therefore included as a natural extension to the problem area.

1.3 Overview

Chapter 2 describes some earlier work in this and related fields.

Chapter 3 is a summary of the work conducted by myself in the specialization project, the autumn of 2009, on which the work in this thesis is heavily based.

Chapter 4 and 5 describes the methodology and the implementation thereof, as conducted during this spring.

Results are shown and discussed in chapter 6, and the report is concluded in chapter 7, together with some suggestions for future work.

Chapter 2

Related Work

2.1 Subdivision

In computer graphics, subdivision is the process of converting a coarse mesh to a smooth, higher resolution mesh. Numerous methods exist, some of which are described here.

2.1.1 Catmull-Clark Subdivision

Catmull and Clark [CC78] proposed a method for subdividing a mesh consisting of quadrilateral faces, or quads. The process is known as Catmull-Clark subdivision.

One level of Catmull-Clark subdivision will replace each quad with four. Each created quad has one corner vertex of the original quad, a common face vertex, and two edge vertices. The face vertex is the centroid of the original quad (average of all four vertices). The vertices created at each edge are defined as the average of its edge's two vertices, and the two face centroids of the adjacent quads. At boundaries, edge vertices are set to the edge midpoint, to avoid degeneration.

After the five new vertices have been calculated, the four vertices defining the original quad are adjusted to the location resulting from the following formula:

$$P_{adjusted} = \frac{\bar{F} + 2\bar{E} + (n-3)P}{n} \tag{2.1}$$

Where F is the set of centroids of the faces connected to the vertex, E is the set of midpoints (plain average of the two end points) of the connected edges, n is the number of connected edges (valence) and P is the original point.

The result is a smoothed mesh with four times the number of quads as the original mesh. The traditional example of a cube subdivided into a sphere approximation is shown in figure 2.1.

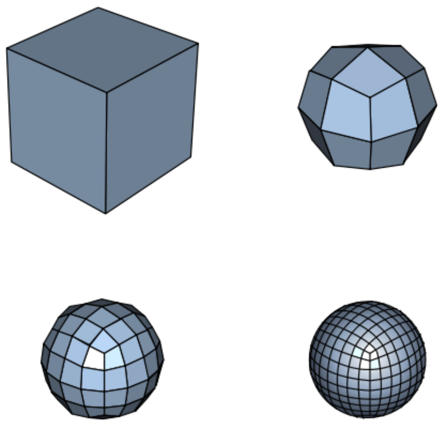


Figure 2.1: *Catmull-Clark subdivision of a cube down to level three*[a]

[a]Image shared by Wikipedia user Romainbehar under the Creative Commons Attribution ShareAlike 3.0 License

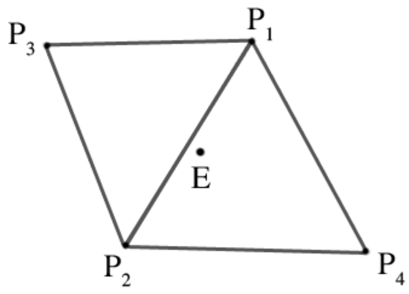


Figure 2.2: *Edge point and points used for the weighted average in Loop subdivision*

2.1.2 Loop Subdivision

Loop subdivision as presented in [Loo87] works on a mesh consisting of triangles. Each triangle is replaced by four triangles.

Each edge of a triangle is assigned an edge vertex (E). This vertex is calculated based on the three vertices in the triangle (P_1, P_2 and P_3), in addition to the final vertex of the triangle at the opposite side of the edge (P_4). See figure 2.2. These vertices are weighted as follows, where P_1 and P_2 are the points defining the edge in question.

$$E = \frac{3}{8}(P_1 + P_2) + \frac{1}{8}(P_3 + P_4)$$

The resulting edge vertices are used together with the original vertices to define four triangles. Three of which consist of two edge vertices and an original vertex, and the fourth defined exclusively by the the new edge vertices.

The original vertices are then adjusted to the position of a weighted average of the old position and the originally connected edges' end points. The weighted formula is as follows:

$$V_{new} = (1 - n\beta)V_{old} + \beta \sum_{i=0}^{n} P_i$$

Where n is the number of edges containing V, P_i is the opposing endpoint of edge i, and a possible choice for β is:

$$\beta = \begin{cases} \frac{3}{16} & \text{if } n = 3 \\ \frac{3}{8n} & \text{otherwise} \end{cases}$$

An example of Loop subdivision is shown in figure 2.3.

2.1.3 Approximation

Performing a full subdivision level by level as described above is not directly mappable to the new graphics pipeline. Instead, approximation techniques are necessary.

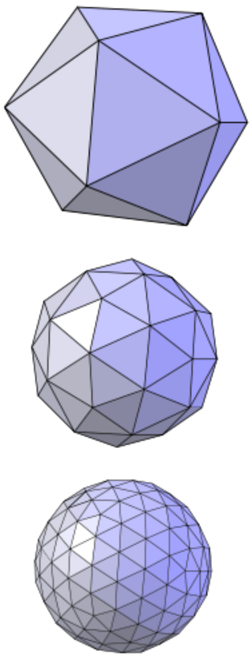


Figure 2.3: *Loop subdivision of an icosahedron to level one and two. Image by Simon Fuhrmann.*

One is created by Loop and Schaefer [LS08], where Catmull-Clark subdivision is approximated with bicubic patches. Patches with an arbitrary number of control points are converted to regular bicubic patches before they are evaluated directly in the desired resolution.

The result is a fairly fast and highly accurate approximation of the Catmull-Clark limit surface. The process is also mappable to the new graphics pipeline, where the conversion to the regular bicubic patch may be performed at the vertex or hull shader stage, and the direct evaluation can be performed at each generated point in the domain shader. See section 5.4.2 for details.

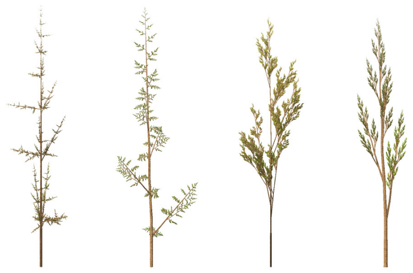


Figure 2.4: *"Weeds" generated using an L-system in 3D. Image released into the public domain.*

Other approaches have been available prior to hardware supported tessellation [BS05, SJP05, PEO09, KKM07, KP05]. ATI/AMD graphics hardware have also included non-standard tessellation support for some time, but this was implemented as a stage prior to the vertex shader [TBB09].

2.2 Lindenmayer-systems

Lindenmayer-systems (L-systems) [Lin68, PL90] are commonly used for generating complex models, especially for simulating phenomena exhibiting self-similar fractal patterns or repetitiveness in growth.

L-systems take rule strings combined with symbol replacement to recursively expand an initial string to a predefined depth. The result is then interpreted in some manner to generate output of the desired type.

An example of interpretation is generating geometry with the transformation scheme called turtle graphics [Ad81], known from the programming language Logo. The concept is to treat the rules as commands to a "turtle robot", which always executes the commands relative to its current transformation. Generating geometry can be based on turtle moves and spawns due to recursion.

Using L-systems is one of the traditional tree generation methods. An example of weed-like models are seen in figure 2.4. However, to avoid the obvious self-similarity that often arise in L-systems, and achieve realistic trees, the rule set can become very complex. This makes it cumbersome to find where a change is required to achieve a certain outcome, or predict the outcome of any alterations, leading to a try-fail approach

A more intuitive approach with easily observable cause-effect relationships might then be desirable.

2.3 Parametrized Tree Models

Weber and Penn [WP95] designed a parametric system for generating tree models. The parameters were selected to be intuitive rather than represent botanical phenomena.

This is opposed to some predecessors of Weber and Penn. De Reffye et al. [dREF^{+}88], among others, created strict botanical models, which lack the intuitive relationship between parametric input and desired output. Oppenheimer, on the other hand, used a fractal based method in [Opp86] to generate tree models.

Honda's parametric system for describing the form of trees [Hon71] may be one of the more intuitive approaches prior to Weber and Penn. Honda did however differentiate between dichotomous (branch split) and monopodial (primary branch continues the parent branch's direction) branching, somewhat increasing the complexity.

Weber and Penn's model allows for great flexibility, with parameters selected for intuitive recognition, even for botanically ignorant users. They include example sets of parameter values that may be used as starting points to create and tweak real-looking tree models.

Parameters include tree branching depth, deviation angles, split factors, length ratios, curvature, rotation and various parameters for describing branch shapes (taper, lobes, trunk flare), pruning and foliage.

The freedom provided by the chosen parameters does not impose the restriction of generating realistic tree models. Unnatural angles, growth rates and rotations may be chosen freely. This may or may not be a positive side effect, depending on intended usage.

The parametric model in itself does not impose a certain way of interpreting the result into a tree model. However, the solution proposed by Weber and Penn is to generate cylinders per branch segment, with no direct connection at branch splits or child branch attachments.

The resulting mesh of this approach cannot benefit from subdivision, as connection points would not be smoothed. Animation (e.g. wind movements) will likely result in unnatural displacement of connection points, as the cylinders typically are animated from the starting points placed inside parent branches' geometry.

2.4 Creating a Single Polygonal Mesh

To benefit from subdivision, the generated tree models should be properly connected at branching points.

Lluch et al. [LVM04] proposed a solution for generating properly connected meshes from models created using L-systems. Their focus was on branching points, where they suggested a stepwise interpolation of each of the child branches' connection points and the parent's connection point. At each step, an outline of all the interpolations is used as the basis for the generated geometry volume.

The generated mesh from this method will have dense geometry at connection points. This is not desirable when subdivision should be applied later, as one goal is to reduce the complexity of the base geometry to reduce the load on GPU memory and CPU-GPU bandwidth. However, the resulting mesh could be simplified using other methods as a post-process.

Maierhofer [Mai02] presented an extension to L-systems, a rule based system

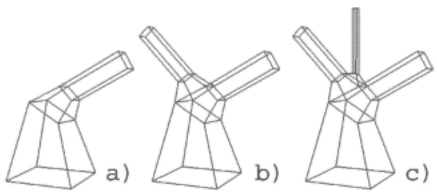


Figure 2.5: *Connection of several child segments. Figure from [SE05]*

for generating single polygonal meshes of arbitrary complex geometry. Tree model generation was one of the demonstrated usages.

2.4.1 Connection at Branching Points

Skjermo and Eidheim [SE05] presented an algorithm based on the extended *SMART* algorithm [FKFW02, FFKW02]. The article suggested how to treat connection points when converting a tree structure – stored as a DAG (Directed Acyclic Graph) – into a polygonal mesh.

The DAG is processed from the root, generating four-sided geometry along each tree segment's length. The end points are then squares, and the problem in question is how to connect several child segments appropriately to these end points.

At each level in the DAG structure, the child segments are treated according to the following cases.

If there are no child segments, the parent segment is concluded with a single cross section. If there is only one child segment, it is connected directly, with only a single cross section generated between the parent and child segments.

If there are more than one, the child segments are sorted by their diameter in descending order. A box is appended to the parent segment, and the first child segment is connected to its top. The next segment is then attached to the most appropriate of the four remaining sides of the box, creating another box in the process. The next child is then attached to the appropriate side of this box, and so on. This is demonstrated in figure 2.5.

If the primary child segment (regardless of count) has a direction that deviates more than 90 degrees from its parent's direction, a backward connection is made. A box is appended to the parent segment, but instead of connecting the primary child segment to the top, it is connected to the most appropriate side, generating another box for connecting any remaining child segments.

2.5 Texture Coordinate Generation

Maierhofer [Mai02] proposed a solution for generating texture coordinates for branching structures. The proposal was to simply increase the v-coordinate together with the distance from the root. The u-coordinate is not so straightforward.

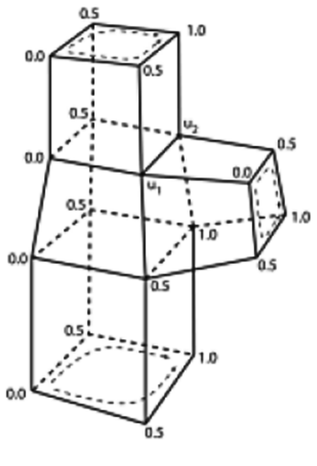


Figure 2.6: *Suggested generation scheme for texture coordinates in the u-dimension at branching points.* $u_1 = \frac{0.0+0.5}{2}$, $u_2 = \frac{0.5+1.0}{2}$. *Figure from [Mai02]*

Maierhofer suggests using a mirrored wrap of the u-coordinate space from front to back of a branch. Working with square cross sections, this means assigning the front vertex a u-value of zero, the back vertex a u-value of one, and the two side vertices a u-value of 0.5.

The result is the complete texture wrapped around from front to back on both sides of the branch.

At branching points, the attached branch should inherit the u-values from its parent for the bottom two vertices. This dictates the remaining two values to complete the dual wrapping. To smooth the resulting texture distortion, the u-values at the parent segment's vertices are set to an average of their neighbouring vertices' and connected vertices' u-values, as shown in figure 2.6.

Chapter 3

Previous Work

This Master's Thesis follows a project entitled *Parametric Generation of Trees for Rendering on Next Generation GPUs*, conducted the autumn of 2009 by myself, with supervisor Torbjørn Hallgren, and co-supervisors Jo Skjermo and Odd Erik Gundersen. This project will be referred to as the *specialization project*.

The specialization project resulted in a method for polygonal mesh generation using a parametric system based on Weber and Penn's parametric tree generation [WP95] with additional traits taking natural branching rules into account.

An investigation of texture coordinate generation schemes suitable for subdivision, with the goal of reducing artefacts to improve realism, resulted in a custom method based on an approach presented by Maierhofer [Mai02].

The resulting meshes were subjected to tests of subdivision schemes using CPU-based subdivision implementations. Catmull-Clark subdivision [CC78] was the preferred method in the end. Catmull-Clark limit surfaces are also commonly used as a target reference for approximation techniques.

This chapter will highlight and summarize the project results. Additionally, some involved explanations are included where it is necessary in order to achieve a full understanding of the following chapters.

3.1 Natural Branching Rules

Da Vinci noted in [Ric70] that the sum of all branches' cross sectional area is equal at any distance from the root. In other words, the sum of the cross-sectional area of the children at a branching point is equal to the parent's cross-sectional area, or:

$$r_{parent}^2 = \sum_{i=0}^{n} r_i^2 \tag{3.1}$$

Where r_{parent} is the parent branch's radius and r_i is the ith child branch's radius.

Murray [Mur26, Mur27] suggested that the correct exponent for the formula is approximately 2.49, with the exception of *small trees*, where an exponent of 3 seems to hold true.

Due to the variability of this exponent, it was introduced as a new parameter of the tree generation method proposed in the specialization project, called the *tree shape exponent*.

Another observation of da Vinci was the relationship between the radii distribution at branching points with the deviation angles away from the parent branch's direction. When a branching point results in two child branches, the child branch with the greatest radius will deviate less from the parent branch's direction than the child branch with the lesser radius.

Murray explored this relationship in artery branching patterns in [Mur26], applying the physiological principle of minimum work. This was then referenced in [Mur27], under the assumption that tree branching pattern follows a similar logic. He suggested the following relationship:

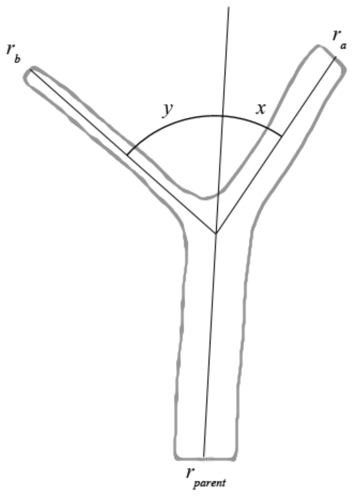


Figure 3.1: *The distribution of angles x and y for a child branch pair with radii r_a and r_b*

$$cos(x+y) = \frac{r^4_{parent} - r^4_a - r^4_b}{2r^2_a r^2_b} \tag{3.2}$$

Where r_a and r_b are child branches' radii, and r_{parent} is the parent branch's radius. $x + y$ is the sum of the deviation angles of the two child branches. See figure 3.1.

After a thorough investigation in the specialization project, a number of relationships were stated. E denotes the introduced *tree shape exponent* and S_s denotes another introduced parameter, called the *radius ratio*. S_s is the ratio between the biggest child branch's radius and the smallest child branch's radius (a *radius ratio* of one means the two child branches' radii are equal).

First of all, the following constants for the primary (largest radius) and secondary (smallest radius) child branches are defined, denoted by subscripts p and s, respectively. These constants also states the ratio between the child branches' radii and the parent branch's radius.

$$C_p = \frac{1}{\sqrt[E]{1+S_s^E}} \qquad C_s = \frac{S_s}{\sqrt[E]{1+S_s^E}}$$
$$r_p = C_p r_{parent} \qquad r_s = C_s r_{parent}$$

To calculate the deviation angles, the following formula is provided:

$$\angle_p : \angle_s = \angle_{ratio} = S_s \frac{1-(1-S_s^4)Cs^4}{1+(1-S_s^4)Cs^4} \tag{3.3}$$

From which we can calculate the primary and secondary angles as follows:

$$\begin{aligned} x+y = \angle_{total} &= cos^{-1}(x+y) \\ \angle_s &= \frac{\angle_{total}^2}{(1+\angle_{ratio})\angle_{total}} \\ \angle_p &= \angle_{ratio}\angle_s \end{aligned} \tag{3.4}$$

It should be noted that the constant factors and angle factors may be computed once per *tree shape exponent/branch ratio*-pair.

The complete investigation of da Vinci's rules and Murray's research as performed in the specialization project is presented in appendix B.

3.1.1 Child Branch Offset

Calculating needed space between parent branch geometry and child branch end point is necessary to ensure a natural-looking connection. Due to the introduction of natural rules, the solution proposed in [SE05] could be utilized.

However, an interpretation of the presented results lead to the more simplistic conclusion that the needed space between the parent branch's axis and the child branch's central end point must be greater than the sum of the parent branch's radius and the child branch's radius. This would accommodate for any deviation angle.

The resulting offset could then be calculated by the following formula:

$$l_{dist} = (1 + S_s) r_{parent} \tag{3.5}$$

Where r_{parent} is the radius of the parent branch at the connection point, and S_s is the introduced *branch ratio*-parameter.

3.2 Generating Tree Data

Since the tree generation phase has been revised to some extent, only a brief explanation of the generation method used in the specialization project is included.

The method for generating tree data was heavily based on the algorithm proposed by Weber and Penn [WP95]. The parameters used were mostly the same, apart from the addition of parameters related to the natural rules, and the removal of some parameters rendered obsolete by these additions.

Tree data was generated recursively in a depth-first order. Leaves and fronds were generated at the appropriate levels whenever they were reached.

A limitation by this approach was that generating a tree model with the same seed, but less levels, or without leaves/fronds, would not result in the same base tree structure.

Global transformations were used throughout the generation process. The deviation resulting from an attached branch could therefore not be applied in a simple way. Due to time constraints, this was ignored, and generated trees with high branch ratios would not follow the natural rules completely.

3.3 Producing a Single Polygonal Mesh

A problem with existing tree generation methods is the lack of a continuous mesh output. Usually, child branches are "attached" to parent branches by originating inside the parent branch geometry, effectively hiding the end point. This leads to a problem with inconsistent animation, where the apparent point of connection to the parent branch seems to move. Also, when subdividing the mesh, no smoothing appears at connection points, since the geometry is not really connected.

To solve these problems, the geometry must be connected in an appropriate way at branching points, while avoiding extensive twist, distortion or other artefacts that may decrease the realism of the rendered tree.

3.3.1 Converting From Data to Mesh

In the specialization project, the transformation from the generated tree node data to a polygonal mesh went via a DAG (Directional Acyclic Graph).

Tree node data consisted of node transformation (matrix), radius and any descendants (defined as nodes continuing the branch) and/or children (defined as nodes of the next branch level attached at some point along the parent node's length). Each tree node was converted into a DAG node, with additional DAG nodes at final nodes' end point, where children nodes were attached, or a split resulted in more than one descendant.

The resulting DAG consisted of nodes with transformation data (position, direction, up vector), branch level, radius and children (not distinguishing between children and descendants).

The final step was converting the DAG into a polygonal mesh. Each DAG node

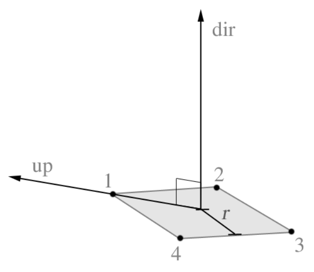


Figure 3.2: *A cross section generated from a DAG node*

was converted to a cross section of a tree branch consisting of four vertices defining the square to be subdivided into the branch's circumference (see figure 3.2).

Vertices were then connected to the vertices of the previous node, texture coordinates generated together with a flip coordinate denoting the "side" of the branch, as well as storing tangent data, radius data and branch level as additional vertex attributes.

3.3.2 Connection at Branching Points

Where branching occurs (e.g. where a DAG node has more than one child), certain rules are followed when connecting the cross sections. These rules were derived primarily from a SMART-like algorithm [FFKW02] proposed by Skjermo et al. in [SE05] (see section 2.4).

In short, three cases were considered. Backward connection (deviation angle more than 90 degrees), split connection (two child branches of equal radius) and ordinary connection (branching with various radii).

The first case was determined to be unreachable as soon as the branching structure conformed with the natural rules (maximal deviation angle will then be 90 degrees), and handling of this case was never implemented.

Split connections were treated as a special case, adding intermediate geometry to allow a symmetric connection for the two same-size child branches. Ordinary connections were performed in line with an interpretation of the algorithm presented in [SE05].

The algorithm is presented as pseudo code in appendix D.

3.4 Texture Coordinate Generation

There is no obvious way to map a texture to a branching structure. Various approaches exist, but the approach presented in the specialization project was a variation of the scheme presented by Maierhofer [Mai02] (see section 2.5).

3.4.1 Stretch Distortion Artefacts

Maierhofer proposes averaging the u-coordinate at two vertices in the parent cross section, let's call this parent branch averaging. Using Maierhofer's method, the stretching artefacts were reduced, but not to a satisfactory degree.

Two alternatives were explored. Averaging the two connected vertices at the child cross section instead, let's call this child branch averaging, was found to produce a similar amount of stretching artefacts.

Combining this with Maierhofer's method, performing the child branch averaging followed by the parent branch averaging, was found to produce the best results after subdivision. See figure 3.6.

3.4.2 u-space Mirroring

The problems with mirroring artefacts was addressed by including a flip coordinate which denoted which side of the branch the vertex was located (the vertices at seams have no side). See figure 3.3.

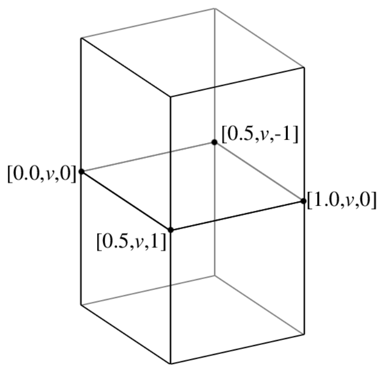


Figure 3.3: *The generated $[u, v, w]$ coordinates at a branch cross section.*

The additional coordinate was utilized in shaders, flipping the texture at one side of the trunk. This resulted in a full wrap around the trunk instead of mirrored texture progression at each side (see figure 3.4). Problems at seams arose due to the discontinued texture coordinates, but were addressed by manually computing texture derivatives in the fragment shader.

Another problem arose when combined with bump mapping. As the tangents and bi-tangents depend on appropriate derivatives of the texture coordinates, the sampled normal displacement must be flipped where the textures are sampled in a mirrored fashion.

3.4.3 Texture Space Degeneration from Subdivision

To maintain correctness and smooth the texture coordinates during subdivision, an approach based on a method proposed by DeRose et al. [DKT98] was developed. DeRose et al. suggest to treat texture coordinates together with the spatial coordinates, effectively subdividing in five dimensions (for two-dimensional texture coordinates). The result was a degeneration of the u-coordinate space. To avoid this, the u-coordinate was not adjusted at existing vertices during subdivision. At generated edge points, a simple average of the

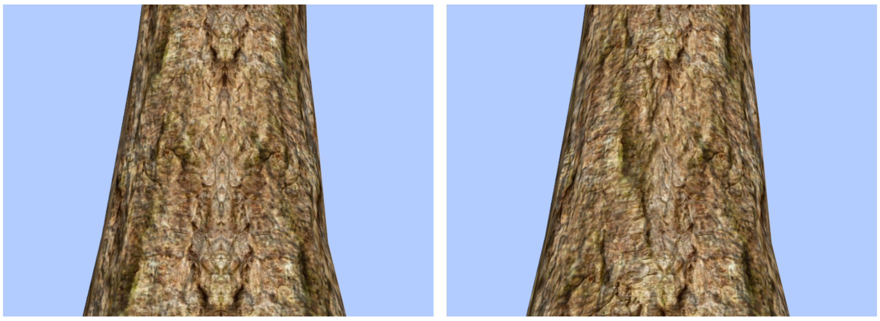

(a) *Branch without u-flipping.* **(b)** *Branch with u-flipping.*

Figure 3.4: *The effect of texture-flipping on a trunk.*

edge vertices' u-coordinate was used (see figure 3.5).

The w-coordinate (flip coordinate) was treated separately. At generated vertices the w-coordinate was set to the sign (-1 or 1) of the weighted average of all related w-coordinates.

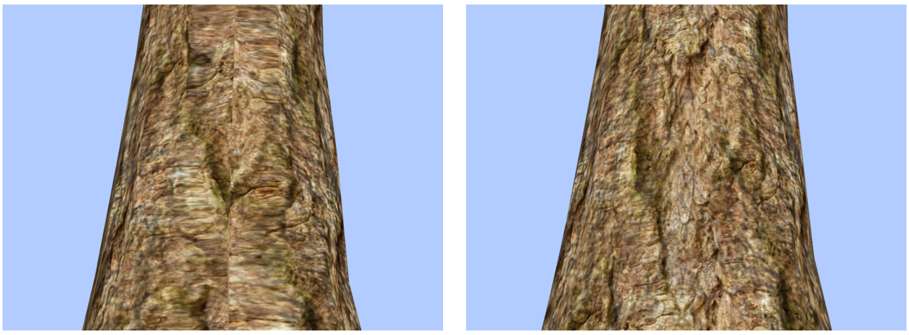
3.5 Rendering

To increase the realism when rendering the generated tree models, some techniques were investigated.

3.5.1 Displacement Mapping

When tessellation hardware is capable of producing increasingly dense geometry, real displacement mapping becomes a potential replacement for the usual faux methods (bump mapping, parallax occlusion mapping).

Direct displacement of geometry is perhaps the easiest solution for displace-



(a) *Texture space degenerated by subdivision* **(b)** *Texture space preserved after subdivision*

Figure 3.5: *When subdividing texture coordinates uniformly (a) the texture space is degenerated, resulting in visible seams and stretching. If the u-coordinate is unaltered at existing vertices during subdivision (b) the texture space is preserved.*

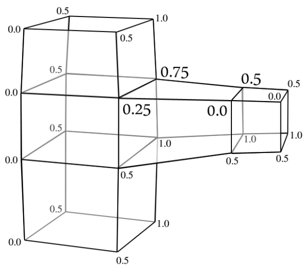
ment mapping, but this necessarily requires the density to match the complexity of the displacement.

The specialization project only had a CPU-implementation of Catmull-Clark subdivision available, but tests with simple tree models were performed to achieve a high density mesh. Displacement mapping of this kind were found to improve realism drastically, without the need for contour fixes and other tricks. See figure C.4 for an example.

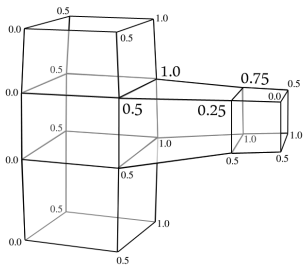
3.5.2 Leaf Back-lighting

A technique described in [KCS07] for improving leaf realism was employed. The technique involves colouring of the leaves based on view angle. If the light relative to the viewer is located directly behind a leaf, its colour is shifted towards yellow. This appears in figure C.5.

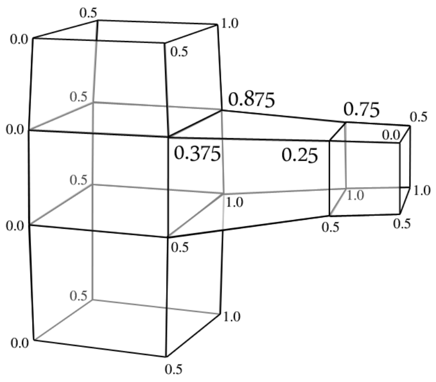
Specular lighting is also added when leaves reflect the light directly.



(a) *Maierhofer's approach*



(b) *Averaging at child branch*



(c) *Combination*

Figure 3.6: *The generated* u*-coordinates using the various methods.*

3.6 Results

Some rendered trees resulting from the specialization project may be seen in appendix C.

Chapter 4

Method

4.1 Process Outline

The complete process from parametric input to geometric output is outlined:

1. Generate tree node data
   - Generate branches
   - Generate leaves

   Tree node data is generated from a parametric model.

2. Add intermediary nodes

   Intermediary nodes are added prior to attached child branches, if the closest node is beyond a certain threshold.

3. Post-apply primary branch deviation

   Primary branches where child branches are connected are deviated according to natural rules.

4. Convert to DAG (Directed Acyclic Graph)

   Tree node data is normalized and child branches and descendant segments are merged into single lists of child nodes. Nodes are added to ensure uniform geometric output.

5. Convert to desired geometric output

   DAG nodes are converted to cross sections of the resulting geometry, adding interconnections between vertices according to the desired output topology.

4.2 Parametric Generation of Tree Data

The generation of tree node data uses the parameters described in appendix A. These are heavily based on Weber and Penn's [WP95] parametric model for tree generation, and the algorithm presented is the basis for generating the tree node data.

Some alterations are made to enforce conformance to the natural rules as presented and explored by da Vinci [Ric70] and Murray [Mur26, Mur27]. These manifest themselves in the removal of some manual angle parameters (*DownAngle* and *SplitAngle*), and the addition of two parameters controlling the behaviour of the natural rule restrictions, *TreeShapeExponent* and *BranchRatio*. The added parameters are described in section 3.1.

The algorithm is recursive, generating branches and their segments – starting with the trunk – followed by the generation and attachment of child branches. Child branches may be attached at any point of a branch segment.

Leaves, if any, are attached to branches at the final branch level.

Detailed information about the parameters, as well as interpretations deviating from Weber and Penn's definitions, are listed in appendix A. The detailed algorithm for generating tree data may be reviewed in Weber and Penn's article [WP95].

4.2.1 Incorporating Natural Rules During Generation

One of the goals of this project is to fully incorporate the effects of the natural rule restrictions. An encountered problem is the primary branch deviations that should occur when a child branch is attached. This is problematic as the deviation must be applied to the attachment point, and propagate to any descendant nodes.

Making the transformation propagate to any descendants is solved by making the transformation data at each node relative to the parent node. The full transformation is then calculated by traversing the tree structure from the target node, requesting the parent node's transformation recursively down to the root node.

Another problem is the application of the primary branch deviation at an arbitrary point of a branch segment. This requires new nodes to be inserted at attachment points, without affecting other parts of the resulting tree structure. To avoid major modifications to the original generation algorithm, step two and three of the process are introduced.

Intermediary nodes are added after the initial data generation, followed by the post-application of deviations to the primary branches at the newly added nodes.

The angle of deviation is stored at child branches during generation, which allows the post-process to retrieve the deviation angle directly, avoiding the full recalculation. The deviation angle of the primary branch is the deviation angle of the child branch multiplied by the angle ratio ($\angle_{ratio}$) as defined in equation 3.3.

4.2.2 Adding Intermediate Nodes

Intermediate nodes are added after the main tree structure is generated. The tree structure is then traversed. When a branch segment has attached child branches, an intermediate node is added before (prepended node) and after (appended node) the point of attachment. The prepended node is then set as the parent node for the child branch root segment.

The intermediate nodes are positioned the same distance before/after the attachment point. This distance should be set to the child branch start radius to ensure a roundness of the child branch after subdivision.

The prepended node should not be generated if it would be positioned too close or prior to the nearest parent node. The nearest parent node then takes the role of the prepended node.

The prepended node is regarded as too close if the distance from the previous node along the branch direction is less than 10 % of the original length of the branch segment. This ensures that child branches that are very close are subjected to the branch connection algorithm appropriately.

4.3 Converting Tree Data to a Polygonal Model

4.3.1 Creating the DAG

Since the addition of intermediate nodes is performed on the tree node data, the process of converting the data into a DAG (Directed Acyclic Graph) is relatively straightforward. It is still convenient to include this stage, since a lot of data can be normalized to a simpler form. At this point, any spatial information is converted to object space, eliminating the hierarchical dependency of the relative transformations.

The remaining alterations to the data structure at this step is adding the end nodes to tree segments.

4.3.2 Preparing for Subdivision

Supporting subdivision is one of the main goals of this project. By generating geometry that is genuinely connected at branching points, one of the criteria is fulfilled.

However, when paired with displacement mapping, subdivision should produce uniformly distributed geometry to avoid extensive variation in displacement density. This is mostly true for highly variant displacement maps, where the end result is highly dependent on the sample frequency, and thus, the geometric density.

To achieve more uniform geometry, DAG nodes may be added wherever the distance between two connected DAG nodes exceeds a certain threshold value. By doing this, we may avoid the low geometric density that may be present at branches with few or no connected child branches.

This could also be achieved by increasing the number of segments per branch (*CurveRes* parameter), but this may lead to a slower generation stage, and excessive cloning whenever splits are generated (*SegSplits* parameter greater than zero).

4.3.3 Geometry

When converting the DAG to the desired geometric output, the following methods should be applied.

Branch Connection

At branching points (DAG node has more than one child), the algorithm for branch connection should be used. In this project, we use the a method based on a proposal by Skjermo and Eidheim [SE05] (see section 2.4.1).

The case of backward connection is disregarded here, since it would not arise when the tree structure follows the natural rules (primary branches would not deviate more than 90 degrees).

An alteration of the process is made where more than one non-primary branch is connected (i.e. a DAG node has more than two child nodes). The primary branch is connected as directed, creating a box between itself and the parent for connection of any remaining branches. Instead of assigning only the secondary branch to one of the sides, all remaining branches are sorted into four lists. Each list represents one side of the box.

These lists are then treated recursively with their related sides as the new parent cross section. Thus, branches may very well be connected to all of the four sides of the original box.

Texture Coordinates

The generation scheme derived from Maierhofer's approach [Mai02] should be implemented as described in section 3.4.

u-coordinates are set to wrap completely in a mirrored fashion on both sides of the trunk. The v-coordinate is increased as we move away from the root. The progressed distance is divided by $0.2 + 0.8r$, where r is the radius at the cross section. This gives more v-space for smaller branches, effectively scaling down the texture in the v-direction as the u-space grows smaller (the cross section circumference). The v-values are finally normalized into the [0,1]-range for the entire mesh.

The flip-coordinate is stored as the third texture coordinate, having the value of zero at seams (front and back) and -1 or 1 at each side.

Chapter 5

Implementation

5.1 Tree Data Representation

The generated tree data consist of a tree structure with nodes representing a branch segment. Each node may have descendants and children.

Descendants are segments that continue the branch. Descendants are therefore positioned at the end of their parent segment. A segment may have multiple descendants if a split has occurred (as defined by Weber and Penn [WP95]), producing clones. Thus, a branch (including clones) are defined by a root segment and a hierarchy of descendants.

Children are segments that are attached at some point on the node's segment, and the root of a new branch. Children are thus on the next level.

At each node, the following data is stored:

Transformation A matrix describing the transformation relative to the parent node's transformation.

Length The length of the segment.

Start and end radius Defines the size of the branch at the start and end points.

Base radius Intermediate parameter used during branch generation.

Segment index The index of the segment in the branch (root segment has index zero).

Parent-relative offset Relative offset on the parent node's segment. Used only for child branches.

Down angle The deviation angle of the child branch. Stored to allow easy post-application of primary branch deviation (see section 5.2.1).

Compensation angle Angle for correcting curvature after splits (see Weber and Penn's article [WP95] for details).

5.2 Tree Data Generation

The generation phase follows the method proposed by Weber and Penn [WP95] closely. Although a complete reimplementation is carried out during this project, a lot is based on the work done in the specialization project. Details regarding this are outlined in section 3.2.

The set of parameters is altered to include only three branch levels (root, primary and secondary), and an additional leaf level. Any level beyond the primary level is considered to be secondary. Typically, the resulting mesh becomes too complex for subdivision purposes after two or three branch levels. More than three levels are therefore rarely generated.

Pruning (envelope) is disregarded to reduce the complexity of the implementation.

5.2.1 Relative Transformations for Post-Application of Natural Rule Constraints

A novelty in this report is the complete inclusion of the natural rules as presented in appendix B. To achieve this without drastically altering the generation process, the generated tree node data is post-processed in two phases.

The first phase adds intermediary nodes at child branch attachment points. Some measures are taken to avoid adding nodes that precede their parent nodes, as well as maintaining reasonable spacing between nodes.

The second phase process the primary branches at attachment points. These are rotated relative to the deviation angles of the non-primary child branches. Since the tree node data now store the transformations relative to parent nodes, instead of global transformations, the rotation will also affect any descendants of the primary branch, which is the desired effect.

To avoid recalculating the deviation angles of the non-primary branches, the deviation angle is added as a property of each tree node. This is also used to identify the attachment points where the primary branch need to be rotated. At these segments, a transformation is applied to the descendant. The transformation is the sum of all rotations resulting from the attachment of child branches.

The rotation caused by each child is the opposite of the child's deviation angle multiplied by the angle ratio (see equation 3.4) for the respective level, where the angle ratio is defined as:

$$\angle_{ratio} = S_s \frac{1 - (1 - S_s^4)Cs^4}{1 + (1 - S_s^4)Cs^4}$$

Where C_s is the constant relation between the parent branch's radius and the secondary branch's radius (see section 3.1).

The angle ratio is calculated once for each level.

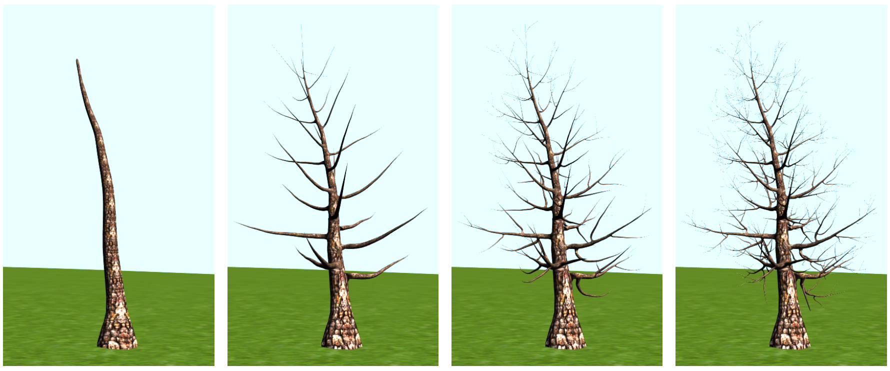

(a) *1 level* **(b)** *2 levels* **(c)** *3 levels* **(d)** *4 levels*

Figure 5.1: *Generating the same tree with different numbers of branch levels.*

5.2.2 Generating the Same Tree with a Different Number of Levels

It may be useful to have multiple versions of the same tree, where the number of generated branch levels varies. One can imagine usage in LOD-schemes, generating leaves at a higher level combined with a lower-level tree mesh, and so on. This is somewhat accounted for by adding a special attribute to vertices describing which branch level they belong to, leaving it up to the user to utilize this information in a useful way.

However, it may still be convenient to have entirely separate meshes of various complexity.

The implemented tree node data generation algorithm in the specialization project followed a straightforward recursive process. This resulted in a depth-first generation where child branches were processed to the end (including any generated descendants, foliage and fronds) before the next sibling branch was created. Since a single pseudo-random number generator is used throughout the generation process, the result of altering the number of generated levels was an entirely different tree structure.

To let users generate the same tree structure with a varying number of branch levels, the implementation of the generation phase is altered. Instead of processing generated descendants as they are generated, they are added to a queue. This results in a breadth-first generation, processing all branches at the same level before processing any descendants. Thus, one can remove or add another level of branches without generating an entirely different tree. See figure 5.1 for an example.

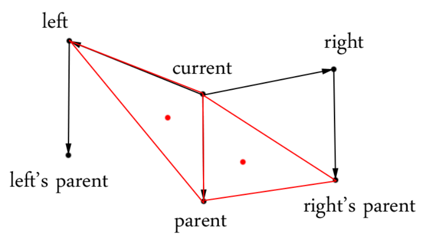
5.3 Polygonal Mesh

5.3.1 Tree Data to DAG

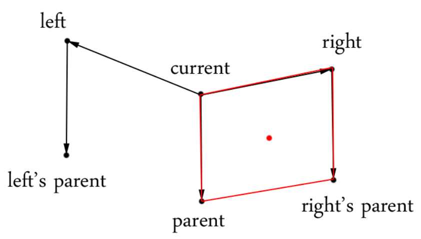
When converting, each tree data node results in a single DAG node. The exception is tree data nodes with no descendants, where a DAG node is added as the end point of the branch segment. Additionally, if a threshold value for maximum node distance is exceeded, additional nodes are added to make the structure more uniformly spaced.

A DAG node holds the following data:

- Position (object space)
- Orientation
- Up vector
- Radius
- Branch level (trunk is level zero)
- Parent node
- List of child nodes



(a) *Triangle faces*



(b) *Quad faces*

Figure 5.2: *Face generation based on mesh point interrelations. Generated faces for the current point are indicated in red.*

Both descendants and child branches in the tree data structure are referenced as child nodes in the DAG structure.

5.3.2 DAG to Polygonal Mesh

The current implementation supports generation of quads and triangles.

The conversion is divided into two phases. The first traverses the DAG, converting each node into four mesh points that will become the vertex positions of the cross section at the node position. The points basically have all the attributes that should be assigned to the vertices, but are also aware of their left, right and parent mesh points. This is where the appropriate texture coordinates are determined.

These relations are used during the second phase, where each mesh point is translated directly to a vertex with related attributes. The relations provide the information needed to generate the index list. This is also the only step where the process depends on which type of output mesh is requested.

Triangles are added between the current mesh point, the left and the parent, as well as the current mesh point, the parent and the right mesh point's parent.

See figure 5.2a.

Quads are added between the current mesh point, the parent, the parent of the right mesh point, and the right mesh point itself. I.e. the quad is added before and to the right of the current mesh point. See figure 5.2b.

Since we have cases where quads or triangles should not be added, adding face indices occur only where relevant mesh points are referenced. Thus, if left, right or parent references are set to null, the appropriate faces are not added. Where child branches are attached, the appropriate mesh point references are set to null.

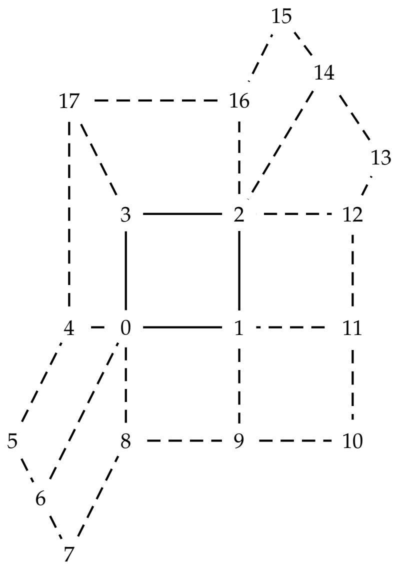
5.3.3 Polygonal Mesh to Patches

To allow for subdivision, the geometry rendered should be a patch structure. We want four-sided patches in our implementation, to allow for bi-cubic approximation of Catmull-Clark subdivision.

Polygonal meshes consisting of quads are processed. Lists of points, edges and faces are obtained, with all relationships stored. This is used to traverse all the faces, finding the 1-ring for the patch. The 1-ring consists of all vertices being part of edges or faces directly connected to the patch, in a counter clockwise order. See figure 5.3.

Up to 28 vertices may be part of the 1-ring, resulting in a maximal patch size of 32 control points. The limitation is caused by the limit of 32 control points for the topology passed to the graphics hardware.

Valence data and index prefixes (offset into 1-ring list for each patch vertex' associated vertices) are calculated and stored in a separate buffer as a way to determine which of the 1-ring control points are connected to which of the four patch corners.



	Valence	Prefix
Point 0	5	9
Point 1	4	12
Point 2	5	17
Point 3	3	18

Figure 5.3: *A patch resulting from the original quad (0–3) and the 14 points (4–17) making up its 1-ring. This is extracted from an existing mesh by defining a start point (0) and traversing edges to find all connected points and their counter-clockwise order.*
Figure based on an ASCII figure in the "SubD" example of the DirectX SDK (June 2010).

5.4 Hardware Tessellation

5.4.1 PN-Triangles

An existing shader implementation for DirectX 11 hardware provided by the DirectX SDK is used to test the PN-triangle approach. The sample is named "PNTriangles11" in the current version of the SDK (June 2010).

The shader requires input of vertex position, normal and tangent for the three vertices in each triangle. This is already present in the generated polygonal mesh. The shader implementation is modified to include some tree model specific operations related to texture coordinates (e.g. texture flip), displacement mapping and shading.

5.4.2 Approximation Using Bicubic Patches

The DirectX SDK provides a sample (named "SubD11" in SDK version of June 2010) including an implementation of Loop and Schaefer's [LS08] approximation method of Catmull-Clark subdivision surfaces using bicubic patches. This is used as a basis for testing a hardware based subdivision method.

The shader requires the input of control points for the patch and its one-ring, as described in section 5.3.3. This is handled by sending a list of 32 indices per patch, where some may not be used. The number of used indices is calculated by the valence and prefix data, which is stored per vertex in a separate buffer.

The shader implementation includes some unneeded functionality (such as bone animation), which is removed to simplify the code. The shader is extended to include texture coordinates throughout the subdivision process. This is done to enable smoothing of texture coordinates, which is essential to reduce distortion of textures mapped to the tree model surface.

Additional modifications include texture operations in the pixel shader for flipping the texture appropriately, smoothing/removing edges, and manually calculating the MIP-level of textures to avoid miscalculations at flip seams. Normal displacements resulting from bump mapping must also be treated based on the flip-coordinate to ensure correct lighting.

5.5 Rendering Methods for Visual Improvements

5.5.1 Displacement Mapping

As the resolution of the rendered geometry increases, displacement mapping is the best and easiest method of adding detail to a mesh. Each generated vertex is displaced along its normal based on a value looked up in a height map, d. A bias, b, may be added, to allow displacements both into and out of the mesh, resulting in the following formula for displacement height:

$$l_{disp} = (d - b)rs_{disp}$$

Where r is the branch radius, and s_{disp} is a user chosen scaling factor. Scaling the displacement relative to radius is done simply to avoid negative volume on very small branches. Also, it looks more natural when the absolute displacement varies in relation to the branch size.

The displaced position of a vertex is then calculated as follows:

$$V_{disp} = V + l_{disp}N$$

5.5.2 Normal Displacement

To increase the apparent amount of detail for low resolution rendering, as well as ensuring correct shading of actual displaced geometry, normals must be displaced.

One technique is to calculate normal displacements per vertex based on the displacement values from a height map. This results in shading that match the geometry, but may cause visible variations when adaptive tessellation is employed.

A better technique is bump mapping. Normal displacements are pre-calculated from a height map, and stored in a normal map. The normal map is then sampled in the pixel shader, adjusting the normals based on the sampled values. The result is exact lighting of all details in the height and normal maps, regardless of mesh resolution.

If the mesh resolution is too low, the lighting will not match the geometry displacement. However, when the mesh resolution is high enough, the bump mapping will result in accurate shading of the displaced geometry. The goal is to have very high resolution models when rendered up close, and should therefore eliminate any noticeable artefacts this approach may introduce.

5.5.3 Faking High Branch Density

By storing the structural data of the tree during generation, this may be utilized for additional purposes.

The data is stored in two variants. The first represent the tree node data, the second represent the DAG node data. These are mostly similar, but the latter may contain more nodes added to produce a more uniform geometric output, as well as the end nodes for segments with no children.

For both variants, the positional, directional and normal (up-direction) vectors are stored for the current node. Radius and level data is also included. Additionally, the position, normal and radius of the parent node are stored as additional parameters.

This allows visualizing the underlying structure of a tree by rendering points and lines based on the node data.

Another usage for this data is more interesting. By rendering view-oriented rectangles between a node and its parent node, properly scaled according to node radii, we have a cheap way of rendering a system of child branches. This is easily implemented as a geometry shader, instancing the rectangles based on node data. Thus, the data may be passed point-wise to the shader with no specification of interrelations, as all nodes know the necessary data for their parent node.

By regenerating the tree data with one or more additional levels, we can use this technique to fake high branch density.

5.5.4 Leaves and Foliage

The method for back-lighting of leaves described by Kharmalov et al. [KCS07] is implemented in the leaf shader.

To reduce the geometry passed to the graphics hardware, only single points are passed per leaf. The leaf geometry are then instanced in the geometry shader, based on the leaf attributes. The leaf attributes include position, direction and orientation vectors, as well as length, width and a special "type" value used to determine which texture coordinate set to use. We allow four quadrants of the leaf texture to contain different leaves.

To increase realism, a simple scheme for self-shadowing between leaves is implemented. It takes each leaf and compares its position relative to the light and the first x leaves in the full list of rendered leaves. If any of the compared leaves is obscuring the light source, the color of the leaf is modulated accordingly. x could be set relative to the leaf count, or simply be a static number.

When converting from tree node data to polygonal data, leaves are picked at random. One leaf is picked at a time from the list of generated leaves, up to the count based on a user controllable leaf density parameter. The resulting list of leaves is therefore randomly sorted. This is an important prerequisite for comparing with only the first x leaves for self-shadowing.

Chapter 6

Results & Discussion

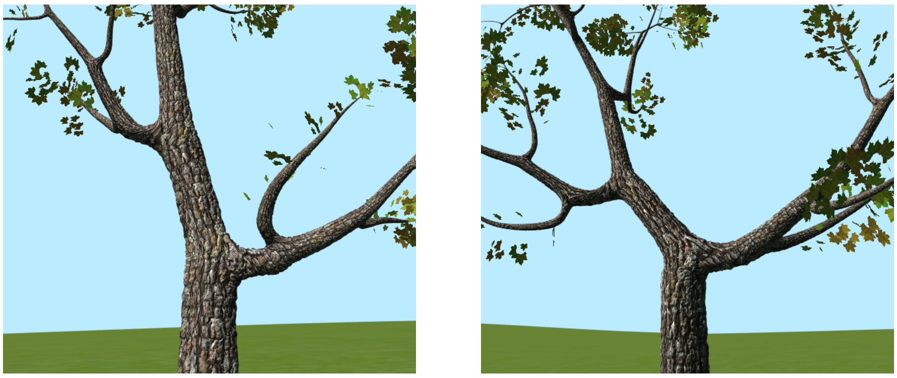
In this chapter, results of various parts of the methodology and implementation choices are presented and discussed.

Some rendered tree models generated from the parametric model are presented at the end of the chapter.

6.1 The Effect of Natural Rules

The natural rules for branch deviations have been fully implemented. The result is that branches are deviated from their original direction when child branches are connected. The deviation is determined by the branch radius ratio and the set tree shape exponent.

Examples of deviation caused by child branch attachments at different branch ratios is seen in figure 6.1.



(a) *Branch ratio set to 0.7* **(b)** *Branch ratio set to 0.9*

Figure 6.1: *The effect of natural rules at different branch ratios.*

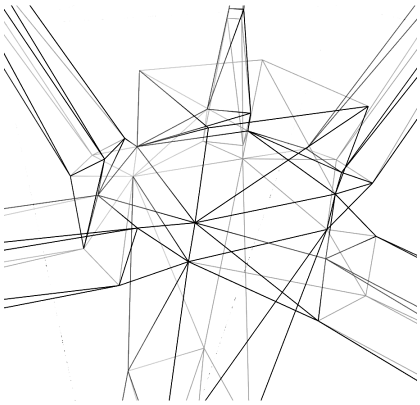
6.2 Single Polygonal Mesh

The main requirement for gaining the full benefits of subdivision was having a properly connected mesh, particularly at branching points, where this is not usually the case in existing approaches.

6.2.1 Branch Attachment

By utilizing the branch connection scheme based on Skjermo and Eidheim's proposal [SE05] the resulting polygonal meshes are fully connected. When subjected to subdivision, the connection points are smoothed properly, resulting in a more natural looking mesh.

The alterations to Skjermo and Eidheim's scheme allowed non-primary child branches to be connected individually to the parent branch's connection box. An example of this and the resulting connected geometry is seen in figure 6.2, together with a subdivided (using Loop subdivision) mesh for comparison.


(a) *Connection geometry at a connection point with five non-primary child branches.*

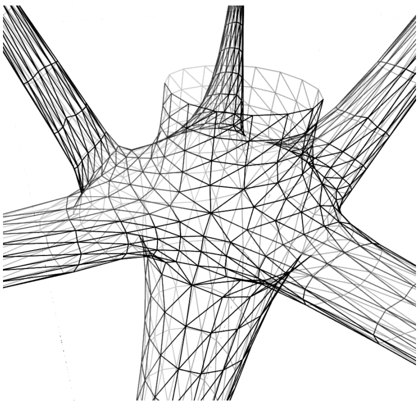
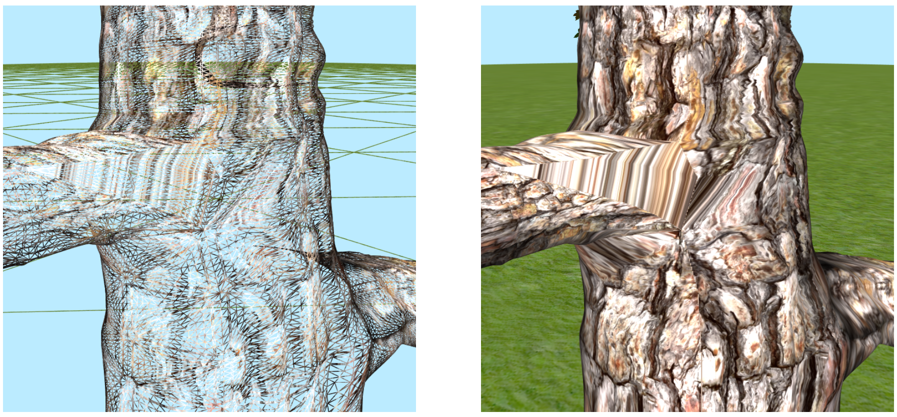

(b) *The same connection geometry after 2 levels of Loop subdivision.*

Figure 6.2: *Altered connection scheme.*

6.2.2 Uniform Geometry Density

By adding cross sections wherever the distance between cross sections exceed a certain threshold we increased the uniformity of the produced meshes. The implementation was designed to let the threshold values be set by the user, to allow customization and disabling of this feature as needed.

Having uniform meshes is regarded as an important factor when applying displacement, as the supported detail size/frequency is directly dependent on the geometric density of the mesh. Uniform meshes should then be less affected by detail size variations due to varying sample frequency of the displacement values.



(a) *Wireframe of high resolution mesh* **(b)** *Texture distortions are not reduced*

Figure 6.3: *A tree model rendered using PN-triangles to increase mesh resolution.*

6.3 Hardware Tessellation

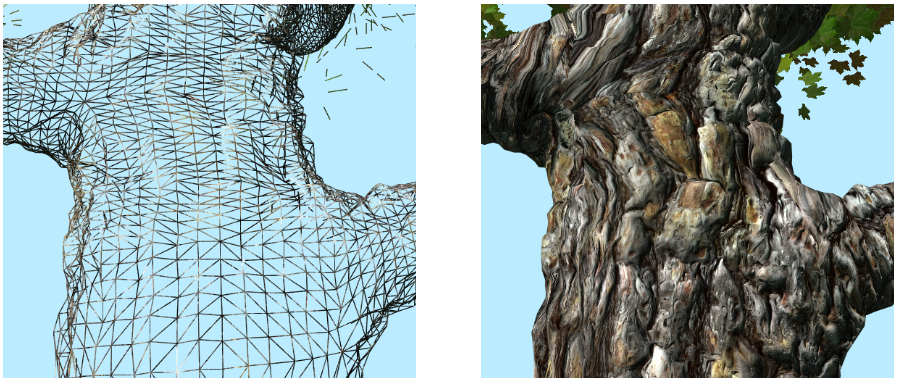
Two methods of hardware tessellation were implemented.

6.3.1 PN-triangles

By implementing the PN-triangles subdivision in graphics hardware, it was possible to render the mesh efficiently at high resolutions. The resulting mesh did, however, have significant texture distortions, as the texture coordinates cannot be smoothed by this method. An example is seen in figure 6.3.

6.3.2 Approximating Catmull-Clark

The modified implementation of Catmull-Clark approximation was able to produce highly tessellated meshes. The modifications included the texture



(a) *Wireframe of high resolution mesh*

(b) *Solid, textured render of high resolution mesh*

Figure 6.4: *A tree model rendered using Catmull-Clark approximation to increase mesh resolution.*

coordinates in the process, removing a lot of the distortion artefacts in branching areas. However, the implementation was not optimized for performance. Thus, meshes with high resolutions prior to subdivision, or applying high tessellation factors would quickly result in unreasonably low frame rates.

An example of a rendered tree mesh with branching points is in figure 6.4.

Texture Space Degeneration

An encountered challenge was that the CPU implementation of Catmull-Clark subdivision had to be modified to avoid texture space degeneration in the u-dimension (see section 3.4.3). The solution was then to leave the u-coordinate out of the process when adjusting existing vertices. This solution is not directly mappable to the graphics pipeline as subdivided coordinates are calculated directly, with no easy way of distinguishing the "existing" vertices from generated vertices.

The solution was a mixed approach. The u-coordinate is treated together with the spatial coordinates during subdivision, let's call this u_{subd}. Another u-coordinate is calculated by linear interpolation of the four core control points, let's call this u_{lerp}. The two u-coordinates are then mixed based on the flip-coordinate, with the following formula:

$$u = |z_{flip}|u_{subd} + (1 - |z_{flip}|)u_{lerp}$$

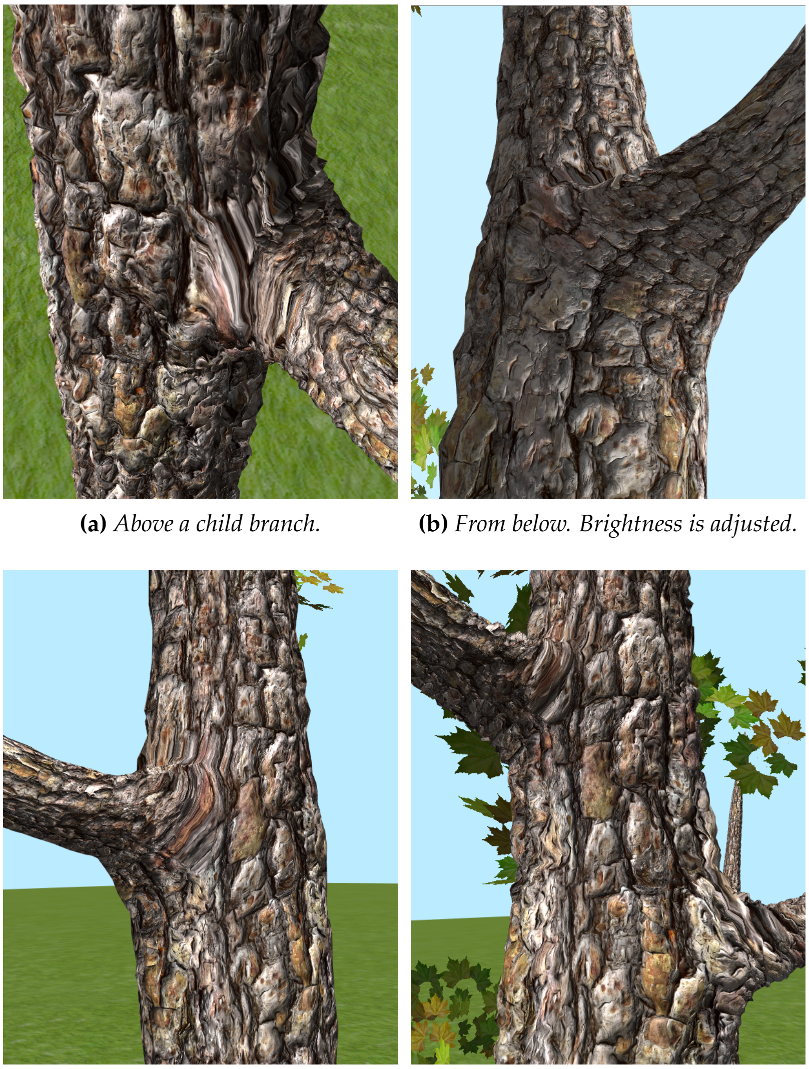
The result is that the interpolated coordinate is used at seams, where degeneration normally occurs. It then fades to the subdivided coordinate as we get further away from the seam, until the subdivided coordinate is used alone. This removes the degeneration artefacts and makes any remaining degeneration unnoticeable.

6.4 Texture Coordinates

The altered texture coordinate generation scheme based on the proposal by Maierhofer [Mai02] was implemented. The resulting meshes have somewhat smooth texture coordinates after subdivision, but stretching artefacts still remain.

The amount of distortion noticeable depends on the chosen texture, and may be reduced by choosing a more uniform texture. This may not always be a valid option.

Examples of varying degrees of distortion at branching points may be seen in figure 6.5. As a result of the texture coordinate generation scheme, the majority of the distortion is located above and at the sides of the connected child branch. In figure 6.5b the distortion is almost unnoticeable when viewed from below the branch.



(a) *Above a child branch.* **(b)** *From below. Brightness is adjusted.*

(c) *Distortion at the side.* **(d)** *Two connected branches.*

Figure 6.5: *Varying degrees of distortion at branch connections of a subdivided tree model.*

6.4.1 Flipping

Mirroring was solved by flipping the texture coordinates based on a special value denoting the side of the branch. This introduced new artefacts.

The first was known beforehand, and lead to a miscalculation of the appropriate MIP-level due to the sudden step of the u-coordinate from -1 to 1 (or equivalent) from one pixel to the next. This was fixed by manually calculating the MIP-level in the pixel shader. The problematic flickering "grey" line at seams disappeared.

The second problem was visible seams at flip edges where the u-coordinate is not an integer value (i.e. at the texture edges). The flip would then cause a sudden jump in texture space. This was solved for the most part by fading in the texture sample at the not flipped texture coordinates. Since these seams occurred only at branching points, the fading was controlled by the fractional level value as well as the absolute flip value, as follows:

$$\begin{aligned} \alpha &= max(1 - 2|z_{flip}|, 0) \\ \beta &= [1 - \alpha frac(n_{level})]^4 \\ color &= \beta sample_{flip} + (1 - \beta) sample_{ord} \end{aligned}$$

Where $frac(n_{level})$ gives the fractional part of the branch level and $sample_{flip}$ and $sample_{ord}$ are the texture samples at the flipped coordinates and ordinary coordinates, respectively.

The result was a significant reduction of the artefacts.

6.5 Visual Enhancements

The visual appearance of a tree depends on more than the trunk geometry. Natural-looking foliage is a requirement for vivid impressions, and may even hide short-comings of other aspects of the tree.

Also, achieving the impression of a dense tree structure without overloading the graphics hardware is also a challenge.

6.5.1 Leaves and Foliage

Leaf generation is part of Weber and Penn's [WP95] model, and produces realistically distributed and positioned geometry. The model is also capable of producing leaf counts reaching hundreds of thousands.

By using the geometry shader to instance one leaf per vertex passed to the pipeline, we reduced the data amount to a fourth (4 vertices per leaf) and the index count to a sixth (two triangles per leaf). This enabled rendering of tens of thousands of leaves without severely affecting the performance.

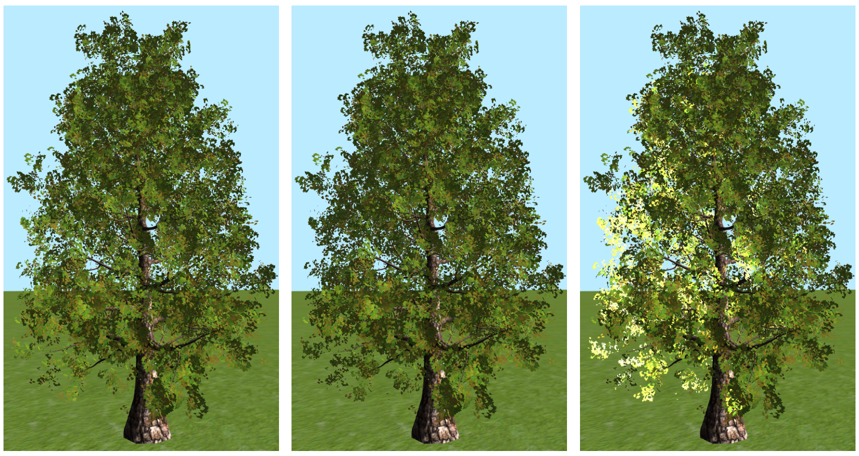
A simple LOD scheme was implemented to reduce the number of leaves rendered based on the distance from the viewer. Further optimization techniques likely exist, and could have been explored further. This was not the focus of this project.

Leaf back-lighting as described in [KCS07] is implemented in the leaf pixel shader. This involves lighting the back of the leaves when the viewer stands directly behind the leaves relative to the light source. This lighting is coloured slightly yellow based on observed natural phenomena.

Kharlamov et al. [KCS07] also describes a technique for leaf self-shadowing when using leaf cluster textures. This is not applicable when rendering single leaves. However, to achieve some degree of self-shadowing the geometry shader was set up to compare inter-leaf vectors with the light direction vector, and adjust the color intensity accordingly.

$$c_{modulate} = saturate(1 - 100[b_{dot} - 0.99])$$

Where $saturate(x)$ clamps x to the range $[0, 1]$ and b_{dot} is the greatest dot pro-



(a) *No self-shadowing* **(b)** *Self-shadowing* **(c)** *Affected leaves highlighted*

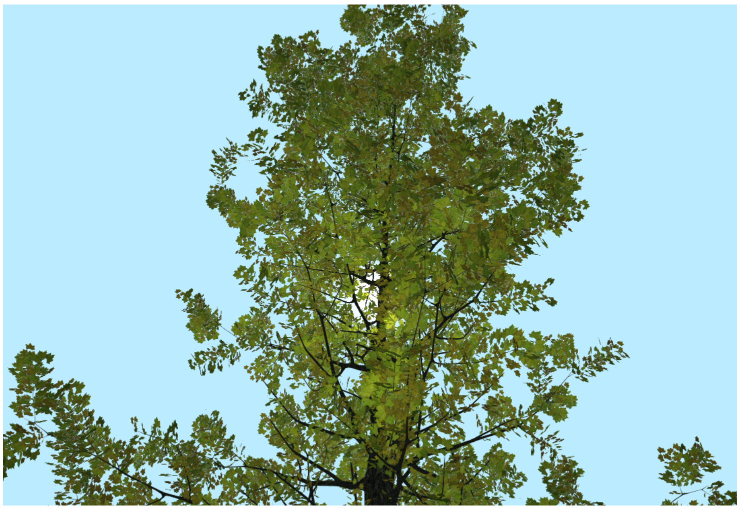
Figure 6.6: *Self-shadowing applied to ~48k leaves comparing with 250 leaves for shadowing. More than 200 FPS was achieved with an ATI Radeon HD 5850 (the trunk was not subdivided).*

duct of the current leaf's light vector and any of the compared leaf directions.

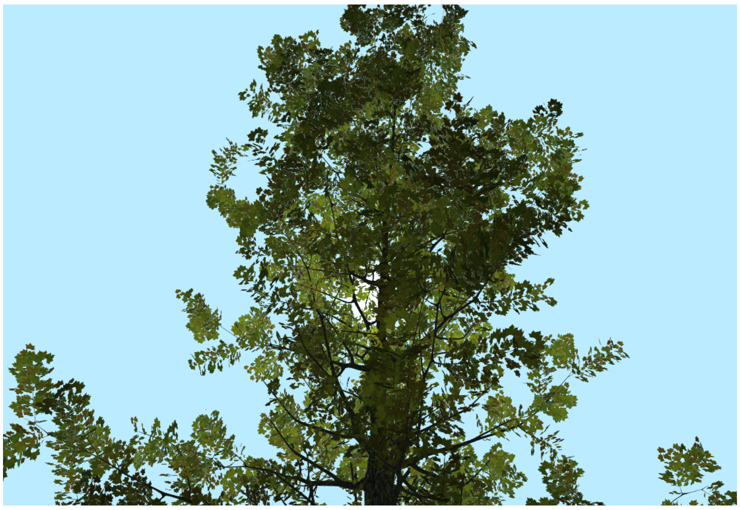
The result was a decent improvement of appearance (see figure 6.6), even when limited to comparing between only a few hundred leaves. The improvement is particularly apparent when the tree is viewed at the shadow side. Since the effect of back-lighting is reduced, the apparent translucency of the whole tree is removed (see figure 6.7).

Again, the technique could certainly have been optimized further. Considering the brute force approach, decent performance was seen even when processing only three leaves at a time while rendering up to tens of thousands of leaves.

By allowing the same tree to be generated with different levels of branches, the leaf data may be generated at a higher level than the trunk mesh. This could increase the leaf count and potential realism (at least when the missing branches are not noticeable) of the rendered tree, without making the tree mesh



(a) *Back-lighting of leaves*



(b) *Back-lighting and self-shadowing*

Figure 6.7: *Self-shadowing improves the leaf lighting as seen opposite to the light source.*

too complex.

6.5.2 Faking High Branch Density

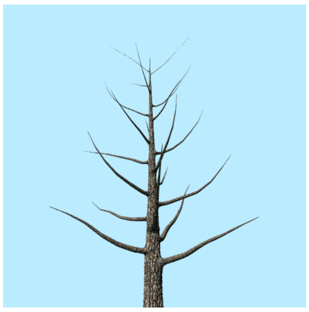
Subdivision takes its toll on the graphics hardware and we would therefore like to limit the amount of geometry this is performed on. The resulting meshes are therefore sparse, which may be fine if enough leaves are covering the void areas, or the viewing distance is sufficiently great.

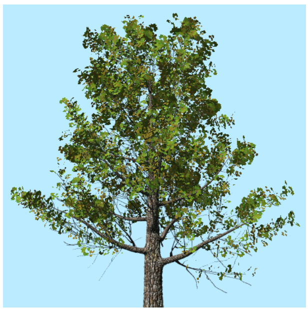
One technique for faking high branch density was explored. The technique takes the spatial data for each tree data node and their parents (i.e. the basic tree structure) and passes it as single points to a shader. The geometry shader then instances view-oriented rectangles along the line between the node and its parent, with appropriate width relative to the branch radius.

This technique is possible due to the alterations to the tree data generation implementation which now allows the same tree to be generated with different levels of child branches without altering the tree structure. Thus, we can generate a basic mesh with for example two levels, and the tree node mesh for faking branch density with four levels.

The result is a relatively cheap way of rendering a high density tree structure, which may even replace the original geometry altogether at great distances. This may also remedy the situation of missing branches when generating leaves at a higher level than the trunk mesh. An example of the technique is seen in figure 6.8.

Since the rectangles are rendered with a solid color, a significant limitation of this technique is that the rendered rectangles do not look realistic at close distances. Its current use is therefore limited, but may be improved by adding shading and/or texture mapping.


(a) *Mesh only*


(b) *Mesh + leaves + fakes*

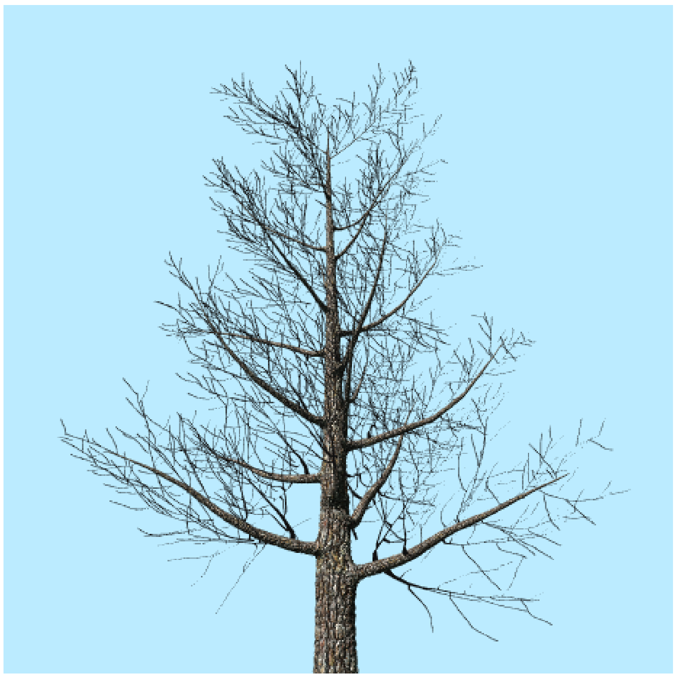

(c) *Mesh and fake branches*

Figure 6.8: *Faking high branch density with child branches rendered as view-oriented rectangles.*

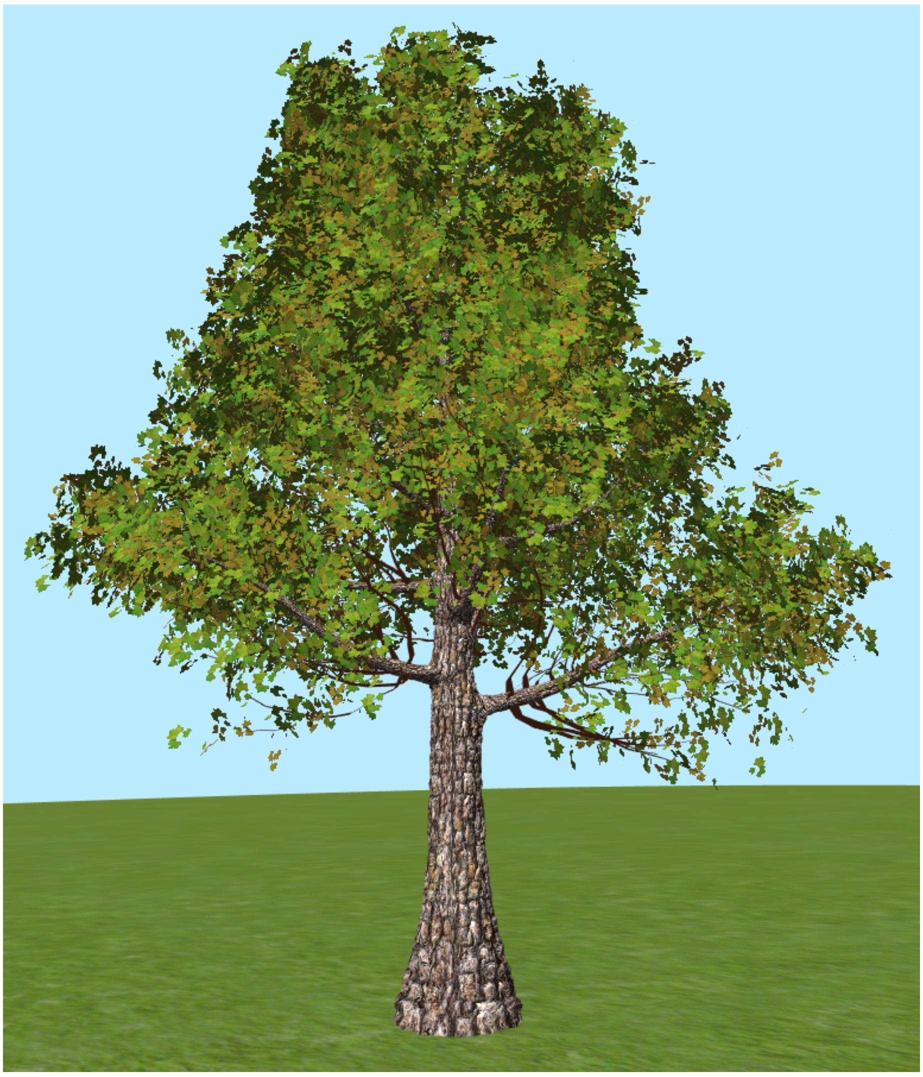


Figure 6.9: *The classical shape of Black Tupelo, based on parameters from [WP95].*

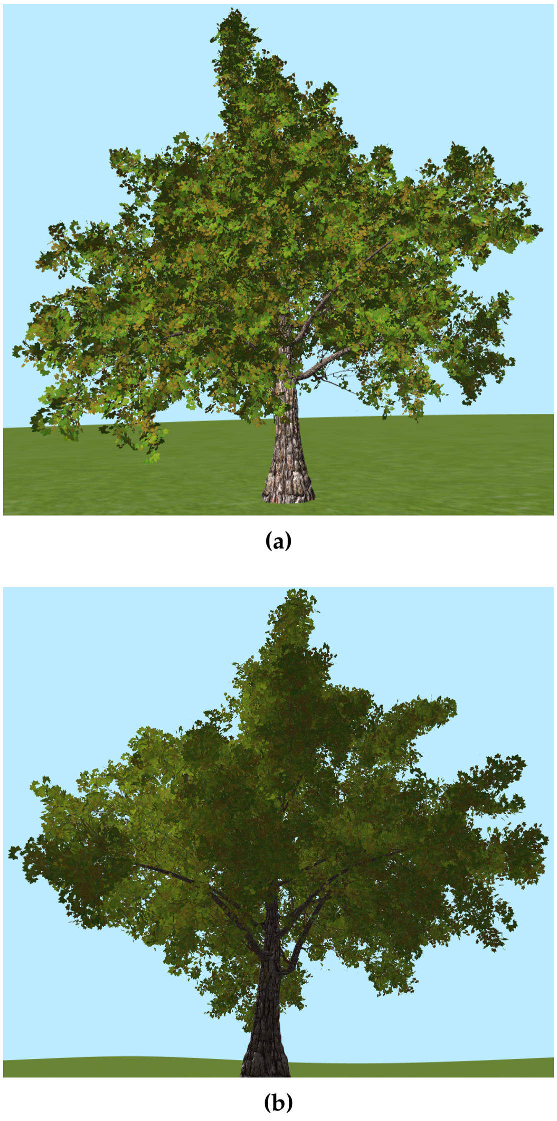


Figure 6.10: *Oak-like tree from front (a) and behind with the light opposing the viewer (b).*

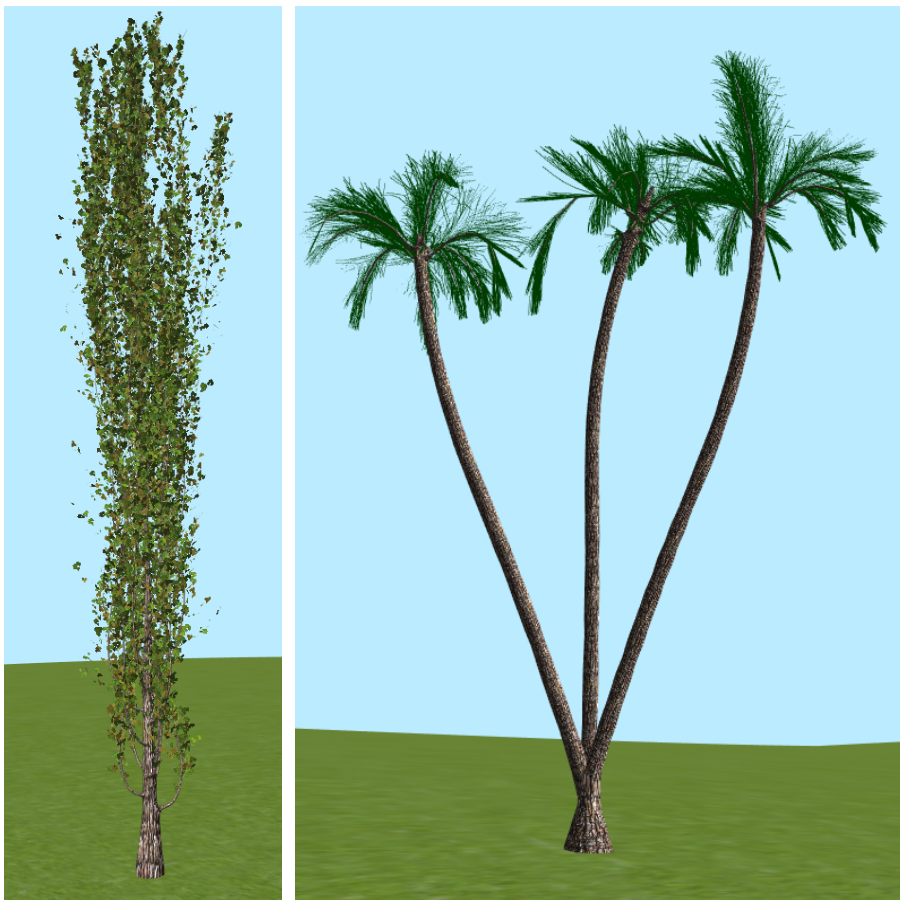


Figure 6.11: *Poplar tree (a) and palm-like tree with fake branches as leaves (b).*

Chapter 7

Conclusion & Future Work

7.1 Natural Rules

The addition of natural rules have been a success. The resulting parametric model is capable of producing natural-looking tree models with great variations. The set of parameters is mostly intuitive and should allow botanically ignorant users to see the relationship between any modifications and the resulting output.

7.2 Single Polygonal Mesh

When following natural rules during generation, a relatively simple scheme for attaching child branches appropriately could be utilized. The result was generated polygonal meshes with proper connections that benefit greatly from subdivision.

7.3 Hardware Tessellation

7.3.1 PN-triangles

PN-triangles is a simple and efficient subdivision method that can be implemented in hardware. It requires little effort to create a very high resolution rendition of an existing polygon model. However, the simplicity of this method also makes its usage limited.

The limitation of this method manifests itself when trying to tessellate the generated tree models. The model itself will result in a smooth and high resolution output, but since the tessellation is based only on the three vertices of a triangle and its data set, the resulting texture coordinates are not affected by neighbouring vertices.

When a CPU-based implementation of Catmull-Clark and Loop subdivision was used previously, the texture coordinates were treated together with the spatial coordinates, thus achieving the same degree of smoothing (although a method customized for the tree models was designed) as the spatial coordinates.

PN-triangles does not smooth any coordinates based on the neighbourhood, and will therefore never achieve the same benefits regarding the texture coordinates. The conclusion is therefore that PN-triangles without custom modifications is not a subdivision method suitable for the generated tree models.

7.3.2 Approximating Catmull-Clark

An implementation of the bicubic patch approximation method of Catmull-Clark subdivision surfaces proposed by Loop and Schaefer [LS08] was modified to handle texture coordinates appropriately, with special considerations for avoiding texture space degeneration due to the mirrored u-coordinate.

The subdivision approach takes the entire 1-ring of a patch as input, and is thus able to let neighbouring vertices affect the subdivided texture coordinates. By this approach we can achieve the benefits of subdivision for a better texture distribution at and near branching points.

The result was a slow, but working hardware-based subdivision scheme that verifies the usefulness of this approach. Texture distortion was improved significantly, but not removed completely. The rendered surface was of high resolution, appropriately smoothed for a more natural appearance.

7.4 Texturing

7.4.1 Texture Distortion

By using the subdivided texture coordinates for most parts of the rendered tree models, the texture distortion at branching points were significantly reduced. They were not completely unnoticeable, however, and numerous adjustments have been attempted to fully remove the distortion artefacts. This includes various projection schemes based on world coordinates, normal orientation or available radius and branch level data. None were particularly successful at eliminating the distortions.

Considerations therefore have to be made as to whether the methods and improvement techniques presented in this thesis are "good enough" for the intended use.

7.5 Visual Improvements

7.5.1 Displacement Mapping

With high tessellation factors, direct vertex displacement based on a height map is a valid approach for achieving realistic and highly detailed surfaces, in this case for tree bark surfaces. It should be coupled with bump mapping to adjust normals appropriately, leading to correct lighting.

7.5.2 Leaves and Foliage

The resulting leaf data from the model is similar to what may be generated from the basic implementation of Weber and Penn's approach [WP95]. It has high quality and may result in hundreds of thousands leaves.

The techniques utilized to improve the appearance of leaves are simple and effective. Including back-lighting of leaves gives an extra dimension to the visual impression. Self-shadowing takes it one step further, improving the apparent realism of the tree.

7.5.3 Faking High Branch Density

The approach for faking a high branch density has proven to be effective. It is only useful when trees are viewed from a certain distance, but may be used as an important part of a LOD scheme when rendering high amounts of trees.

Further explorations in this area may include the addition of shading and texturing techniques of the rendered view-oriented rectangles to increase the realism at closer distances.

7.6 Future Work

7.6.1 Tree Generator Software

The complete set of methods and implementation of the tree generation may be packaged into a single software system. This could be done to allow easy utilization in games, simulation software and 3D content creation software.

Another approach may be to develop plugins for common 3D content creation software, making the tree generator a highly accessible tool for 3D content artists.

Yet another approach may be to develop plugins for game creation tools or 3D rendering engines, allowing the automatic generation of trees on the fly in games and simulation software.

7.6.2 Hardware Tessellation

As graphics hardware of the new generation are refined and gains greater performance, full subdivision of entire forests may become possible. Various LOD schemes will likely be necessary, but subdivision provides a very simple LOD scheme by simply adjusting tessellation factors.

Finding more efficient hardware-based subdivision methods should and will probably be a high priority. Altering these methods to suit tree models and other branching structures with complicated texture mapping could likely be an area of future work.

7.6.3 Texture Coordinates

Since the texture distortion could not be completely remedied, a natural suggestion is further research on this topic to achieve a perfect texture mapping scheme for trees and branching structures.

7.6.4 Customized Texture Generation

One possible solution for the texture distortions could be to automatically generate a custom texture for each tree. The texture could be based on a provided texture. The automatic generation would then take into account the uneven distribution of the texture space, and accommodate for these shortcomings in the generated texture.

7.6.5 Manual Adjustment of Texture Coordinates

An alternative to the fully automatic approach for tree generation may be to allow manual adjustments of the generated meshes. This could be particularly handy when it comes to fixing texture distortion artefacts.

Therefore, a way of exporting texture coordinates in a sensible format and layout appropriate for manual adjustment could be developed.

The automatic generation of varying trees would not be possible with this approach. Instead, the target group would be 3D content artists that require full control over and high quality of the end result.

7.6.6 Visual Enhancements and Photo Realism

There exists many techniques for improving the visual appearance of trees. Most are not explored here, but the topic is likely to remain relevant in the future and seems to be inexhaustible when aiming for photo realistic results.

As hardware continues to become more capable and powerful, using realistic models for lighting of trees and foliage may become available. The list of improvements includes light scattering, translucency, complete self-shadowing and so on. These topics are highly relevant for rendering trees, as lighting and shading accounts for a huge part of the believability of the result.

Bibliography

[Ad81] Harold Abelson and Andrea diSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. MIT Press, Cambridge, 1981.

[BS05] Tamy Boubekeur and Christophe Schlick. Generic mesh refinement on gpu. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 99–104, New York, NY, USA, 2005. ACM.

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

[DKT98] Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 85–94, New York, NY, USA, 1998. ACM.

[dREF$^+$88] Phillippe de Reffye, Claude Edelin, Jean Françon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and development. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1988. ACM.

[FFKW02] Petr Felkel, Anton L. Fuhrmann, Armin Kanitsar, and Rainer Wegenkittl. Surface reconstruction of the branching vessels for augmented reality aided surgery, 2002.

[FKFW02] Petr Felkel, Armin Kanitsar, Anton L. Fuhrmann, and Rainer Wegenkittl. Surface models of tube trees. In *In Computer Graphics International*, pages 70–77, 2002.

[Gee08] Kevin Gee. Nvision 08: Introduction to directx 11, 2008.

[GSSK05] I. Garcia, M. Sbert, and L. Szirmay-Kalos. Tree rendering with billboard clouds. In *Third Hungarian Conference on Computer Graphics and Geometry*, Budapest, Hungary, 2005.

[Hon71] Hisao Honda. 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*, 31(2):331 – 338, 1971.

[KCS07] Alexander Kharlamov, Iain Cantlay, and Yury Stepanenko. *GPU Gems 3 - Next-Generation SpeedTree Rendering*, chapter 4, pages 69–91. Addison-Wesley, 2007.

[KKM07] Adarsh Krishnamurthy, Rahul Khardekar, and Sara McMains. Direct evaluation of nurbs curves and surfaces on the gpu. In *SPM '07: Proceedings of the 2007 ACM symposium on Solid and physical modeling*, pages 329–334, New York, NY, USA, 2007. ACM.

[KP05] Minho Kim and Jörg Peters. Realtime loop subdivision on the gpu. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Posters*, page 123, New York, NY, USA, 2005. ACM.

[Lin68] Aristid Lindenmayer. Mathematical models for cellular interaction in development: Parts i and ii. *Journal of Theoretical Biology*, 18, 1968.

[Loo87] C. Loop. Smooth subdivision surfaces based on triangles. Master's thesis, Department of Mathematics, University of Utah, 1987.

[LS08] Charles Loop and Scott Schaefer. Approximating catmull-clark subdivision surfaces with bicubic patches. *ACM Trans. Graph.*, 27(1):1–11, 2008.

[LVM04] Javier Lluch, Roberto Vivó, and Carlos Monserrat. Modelling tree structures using a single polygonal mesh. *Graphical Models*, 66(2):89–101, 2004.

[Mai02] Stefan Maierhofer. *Rule-Based Mesh Growing and Generalized Subdivision Meshes*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, 2002.

[Mur26] Cecil D. Murray. The physiological principle of minimum work applied to the angle of branching of arteries. *The Journal of General Physiology*, 9(6):835–841, July 1926.

[Mur27] Cecil D. Murray. A relationship between circumference and weigth in trees and its bearing on branching angles. *The Journal of General Physiology*, 10(5):725–729, May 1927.

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

[PEO09] Anjul Patney, Mohamed S. Ebeida, and John D. Owens. Parallel view-dependent tessellation of catmull-clark subdivision surfaces. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 99–108, New York, NY, USA, 2009. ACM.

[PL90] P. Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[Ric70] Jean Paul Richter. *The Notebooks of Leonardo da Vinci*. Dover, New York, NY, USA, 1970.

[SE05] Jo Skjermo and Ole Christian Eidheim. Polygon mesh generation of branching structures. In *SCIA*, pages 750–759, 2005.

Bibliography

[SJP05] Le-Jeng Shiue, Ian Jones, and Jörg Peters. A realtime gpu subdivision kernel. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1010–1015, New York, NY, USA, 2005. ACM.

[TBB09] N. Tatarchuk, J. Barczak, and B. Bilodeau. Programming for real-time tessellation on gpu. Technical report, AMD, Inc., 2009.

[WP95] Jason Weber and Joseph Penn. 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, 1995. ACM.

Appendix A

Parameters

The following lists all parameters included in our specification. Most are taken directly from Weber and Penn's specification [WP95], and the reader is referred to their article for details about the various formulas used to calculate the tree properties.

If significant changes have been made to the interpretation of parameters, more in-depth descriptions are included.

All angles are specified in degrees.

A.1 Tree Parameters

Shape Specifies one of the following eight values introduced by Weber and Penn, controlling the tree shape:

0 Conical

1 Spherical

2 Hemispherical

3 Cylindrical

4 Tapered cylindrical

5 Flame

6 Inverse conical

7 Tend flame

Each shape value refers to a specific function applied to the scaling of radius and length of the branches based on the relative position on the parent branch, resulting in trees resembling the listed shapes.

Weber and Penn also included a ninth value, envelope, which has been left out in our specification.

Levels Number of branch levels to produce, typically 2-4. As opposed to the original interpretation, this does not include the leaf level regardless of the value of **Leaves**.

TreeShapeExponent The shape exponent in da Vinci's equation (see section 3.1)

AttractionUp Upward growth tendency. A value of 1 will approximately result in all branches' end points to point directly upwards.

Scale, ScaleV The base tree scale

A.1.1 Leaf Parameters

Leaves Number of leaves to generate. If non-zero, leaves are generated after the last branch level, based on the leaf level parameters.

LeafScale, LeafScaleX Describes the length and width of each leaf

A.1.2 Trunk Parameters

The following parameters apply to the trunk only.

Ratio Specifies the trunk radius/length ratio. Contrary to Weber and Penn's model, it does not affect successive levels' radii since these are calculated based on new parameters.

BaseSize The relative bottom part of the trunk with no branches.

Flare Specifies the exponential widening of the trunk near the bottom

BaseSplits Number of (additional) clones directly after the first segment of the trunk

A.2 Branch/Leaf Level Parameters

The complete specification includes the following list of parameters for each of the branch levels (excluding the trunk), in addition to the leaf level.

DownAngleV Deviation angle variation

Rotate, RotateV The rotation around the parent branch's axis relative to the previous child branch

BranchRatio The child/parent branch radius ratio as explained in section 3.1

A.3 Stem Parameters

The following parameters are specified per branch level.

Branches Number of branches per parent branch to generate (ignored for the trunk level)

Length, LengthV Controls the length of the branch

Taper Describes the tapering of the branch (cone, cylinder, etc.)

SegSplits Number of splits per segment

SplitAngleV Split angle variation

CurveRes Number of segments per branch

Curve, CurveV Angle distributed between segments

CurveBack If non-zero, Curve specifies the angle distributed along the first half of the branch, while CurveBack specifies the angle distributed along the second half of the branch (allowing S-shaped branches)

Appendix B

Natural Rules

The following text is an excerpt from the report of the specialization project conducted during the autumn semester of 2009, entitled *Parametric Generation of Trees for Rendering on Next Generation GPUs*.

This appendix is included to clarify how certain formulas were derived. See chapter 3 for more information about the specialization project.

B.1 Introducing da Vinci's Rule

Da Vinci noticed [Ric70] that the sum of all branches' cross sectional area at any height of a branching structure, is equal to the cross sectional area of the trunk. This also means that the sum of the child branches' cross sectional area is equal to their parent branch' cross sectional area, or:

$$r_{parent}^2 = \sum_{i=0}^{n} r_i^2 \tag{B.1}$$

Where r_{parent} is the parent branch's radius, and r_i is child branch i's radius.

Murray performed physical measurements of trees presented in [Mur27], and suggested that the correct exponent for the formula is about 2.49. In [Mur26], however, an exponent of 3 is claimed to hold for *small* trees.

In our method, the introduced parameter *tree shape exponent* is this exponent.

Another observation of da Vinci's is that in the case of two sibling branch segments, the one with the largest radius deviates less from the parent segment's direction than the one with the lesser radius. Murray explored this without reference to da Vinci in [Mur26]. Here, artery branching patterns were studied, applying the physiological principle of minimum work. This was referenced in [Mur27] with the assumption that trees grow requiring the minimum amount of wood. The following was suggested (originally with circumferences, but equally valid for radii):

$$cos(x+y) = \frac{r_{parent}^4 - r_a^4 - r_b^4}{2r_a^2 r_b^2} \tag{B.2}$$

Where r_a and r_b are radii of two child branches of a parent branch with radius r_{parent}. $x+y$ is the sum of the deviation angles of the two child branches, as shown in figure B.1.

Following is the exploration of two cases, the equal split case and the child branch case, which will lead us to a general equation that can be used when generating the tree data.

B.1.1 Equal Split Case

If we consider the special case of a split with two child segments with equal radius ($r_a = r_b = r$), the fractional term of equation B.2 reduces to:

$$\frac{r_{parent}^4 - 2r^4}{2r^4} = \frac{r_{parent}^4}{2r^4} - 1$$

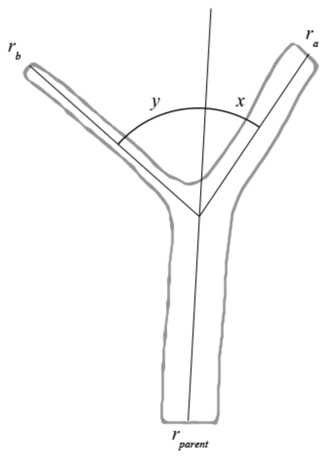


Figure B.1: *Radii and angles at a branching point with two child branches*

If we assume equation B.1 is valid, $r^2_{parent} = 2r^2$, $r_{parent} = \sqrt{2}r$. Thus:

$$\begin{aligned}\frac{r^4_{parent}}{2r^4} - 1 &= \frac{(\sqrt{2}r)^4}{2r^4} - 1 \\ &= \frac{2^2}{2} - 1 \\ &= 1\end{aligned}$$

Since $cos^{-1}(1) = 0°$, the conclusion is that the branches should not deviate at all. However, this is where the observed exponents is of importance. If we replace the exponent in equation B.1 with the measured 2.49. We get the following:

$$\begin{aligned}
\frac{r_{parent}^4}{2r^4} - 1 &= \frac{(\sqrt[2.49]{2}r)^4}{2r^4} - 1 \\
&= \frac{2^{\frac{4}{2.49}}}{2} - 1 \\
&= 1.52 - 1 \\
&= 0.52
\end{aligned}$$

This gives $cos^{-1}(0.52) = 59°$. Thus, large trees following the observation of Murray should have a 59° span where two child branches have the same size. Repeating the process for small trees with exponent 3 gives a span angle of 75°.

B.1.2 Child Branch Case

Let us consider the case of a branching point where the primary child branch has radius r_p and the secondary child branch has a radius of $r_s = S_s r_p$. S_s is the *radius ratio* between the two child segments, which is less than one (or their roles would swap). Starting with the fractional term of equation B.2, we can simplify:

$$\frac{r_{parent}^4 - (1 + S_s^4)r_p^4}{2S_s^2 r_p^4} = \frac{r_{parent}^4}{2S_s^2 r_p^4} - \frac{1 + S_s^4}{2S_s^2}$$

With a general equation, $r_{parent}^E = (1+S_s^E)r_p^E$ and $r_{parent} = \sqrt[E]{1 + S_s^E}r_p$. Thus:

$$\begin{aligned}
\frac{r_{parent}^4}{2S_s^2 r_p^4} - \frac{1 + S_s^4}{2S_s^2} &= \frac{(\sqrt[E]{1 + S_s^E}r_p)^4}{2S_s^2 r_p^4} - \frac{1 + S_s^4}{2S_s^2} \\
&= \frac{(1 + S_s^E)^{\frac{4}{E}}}{2S_s^2} - \frac{1 + S_s^4}{2S_s^2} \\
&= \frac{(1 + S_s^E)^{\frac{4}{E}} - 1 - S_s^4}{2S_s^2}
\end{aligned}$$

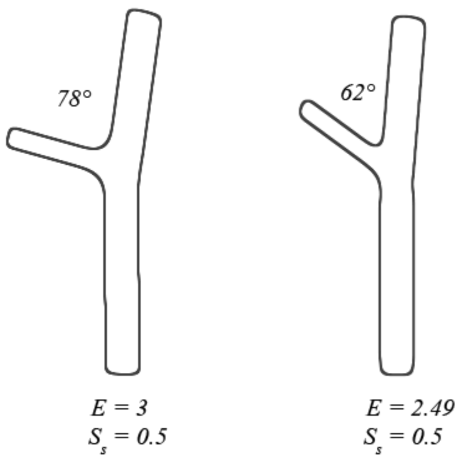


Figure B.2: *Example branch angles with* $E = [3, 2.49]$

Which gives the following general equation for calculating the angle between child segments based only on the *tree shape exponent*, E, and the *radius ratio* S_s:

$$cos(x + y) = \frac{(1 + S_s^E)^{\frac{4}{E}} - 1 - S_s^4}{2S_s^2} \tag{B.3}$$

This can be calculated per constant pair. If we allow different values for the scaling factor per branch level, this typically means 2-4 pairs.

Two examples demonstrating the effect of E on branching angle can be seen in figure B.2.

As we can see from figure B.3, the resulting deviation angles for various values of E range from close to 90° in the case of a *radius ratio* near zero, to the angles found in section B.1.1, which is just the special case of $S_s = 1$.

The angle is distributed between the child segments according to the following ratio (deduced from equation 3 of [Mur27]):

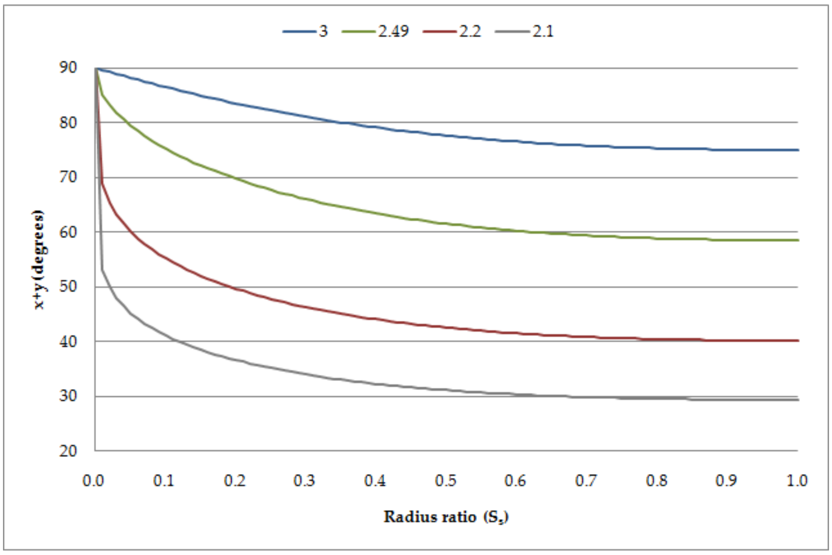


Figure B.3: *Plot of $x + y$ resulting from equation B.3 with $E = [3, 2.49, 2.2, 2.1]$*

$$\angle_p : \angle_s = S_s \frac{r_{parent}^4 - (1 - S_s^4) r_p^4}{r_{parent}^4 + (1 - S_s^4) r_p^4} \tag{B.4}$$

Calculating the radii of the child branches can be done as follows:

$$\begin{aligned} r_{parent}^E &= r_p^E + S_s^E r_p^E \\ r_p^E &= \frac{r_{parent}^E}{1 + S_s^E} \\ r_p &= \frac{r_{parent}}{\sqrt[E]{1 + S_s^E}} \end{aligned}$$

Thus, the radius distribution depends on per-level constants:

$$\begin{aligned} C_p &= \frac{1}{\sqrt[E]{1 + S_s^E}} & C_s &= \frac{S_s}{\sqrt[E]{1 + S_s^E}} \\ r_p &= C_p r_{parent} & r_s &= C_s r_{parent} \end{aligned} \tag{B.5}$$

Replacing into equation B.4, we get:

$$\begin{aligned} \angle_p : \angle_s &= S_s \frac{r_{parent}^4 - (1 - S_s^4) C_p^4 r_{parent}^4}{r_{parent}^4 + (1 - S_s^4) C_p^4 r_{parent}^4} \\ &= S_s \frac{(1 - (1 - S_s^4) C_p^4) r_{parent}^4}{(1 + (1 - S_s^4) C_p^4) r_{parent}^4} \\ &= S_s \frac{1 - (1 - S_s^4) C_p^4}{1 + (1 - S_s^4) C_p^4} \end{aligned} \tag{B.6}$$

Which is also only dependent on E and S_s.

B.2 Splits of Higher Degree

The natural rules are considered only for single splits (either by attaching a child branch or by splitting a branch in two). The model, however, supports splits of higher degree. In these cases we need to determine the radii and split angles of the resulting branches.

To limit the complexity of the model at this stage, the following solution, not based on natural phenomena, is proposed:

When calculating the angle $x + y$, it is normally divided equally between the resulting clones (i.e. $x = y$). For higher degree splits, the angle is divided equally between the resulting n clones, $\angle = \frac{x+y}{n}$.

The radii are treated similarly. By extrapolating the natural rule to higher degrees, the following formula is used for calculating the radius of the resulting n clones:

$$r_c = \frac{r_{parent}}{n^{\frac{1}{E}}} \tag{B.7}$$

Appendix C

Specialization Project Results

The following figures were the output of the specialization project conducted during the autumn semester of 2009, entitled *Parametric Generation of Trees for Rendering on Next Generation GPUs*.

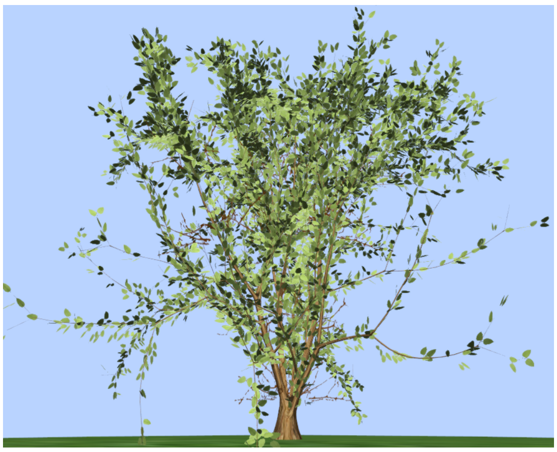


Figure C.1: *Tree model based on parameters for Black Oak as described in [WP95]*

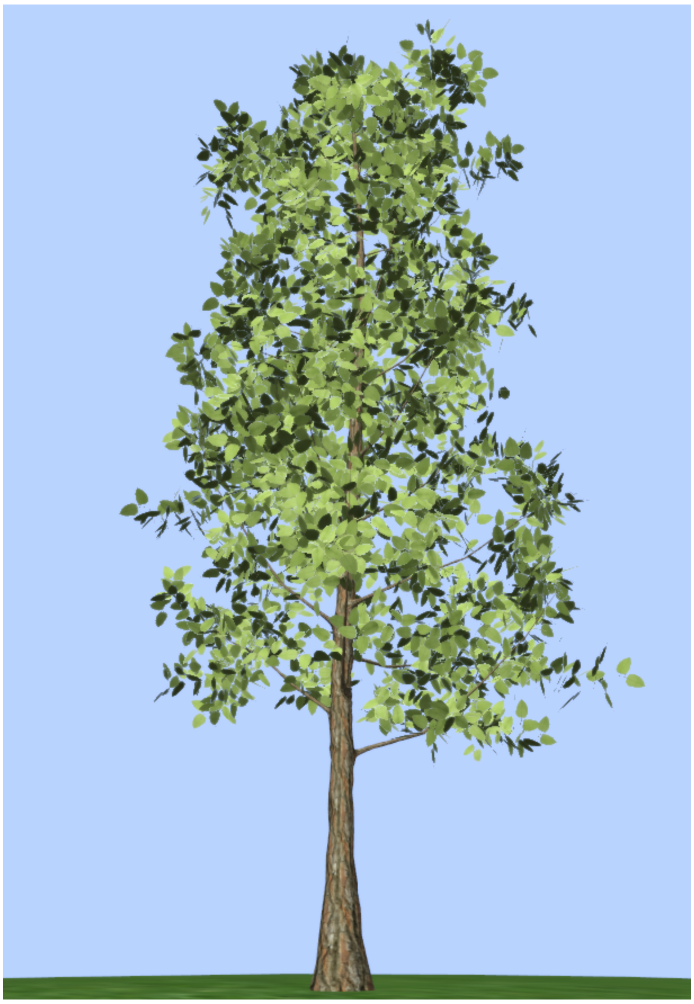


Figure C.2: *Tree model based on parameters for Black Tupelo as described in [WP95]*

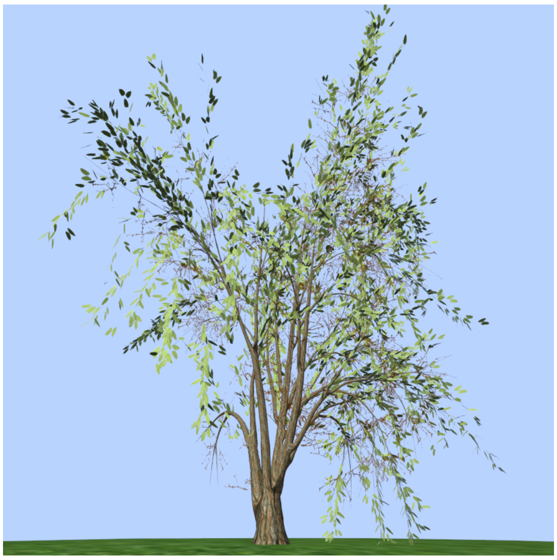


Figure C.3: *Tree model based on parameters for Weeping Willow as described in [WP95]. Lack of pruning leads to some branches piercing the ground.*

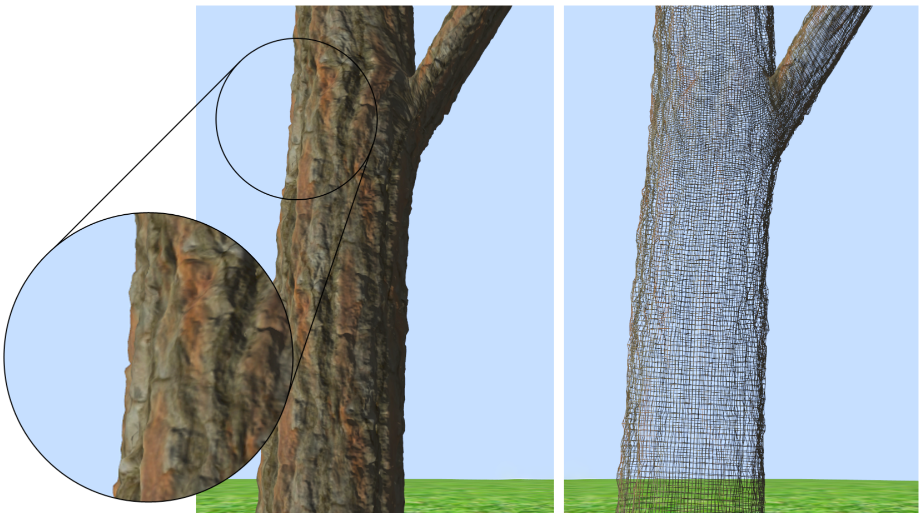


Figure C.4: *Displacement mapping applied to a tree trunk subdivided to level 5 (left) and associated wireframe (right)*

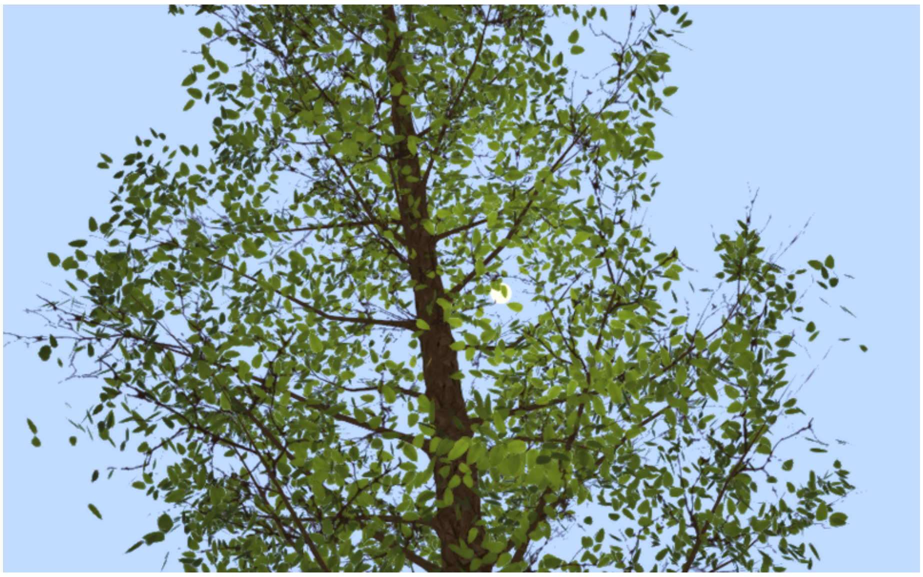


Figure C.5: *Back-lighting of leaves*

Appendix D

Connect Pseudo Code

```
function Connect(Quad, Segments)
  Sort Segments by radius in descending order
  if Quad.normal • Segments[0].direction < 0 then
    Connect backward
  else if Segments[0].radius == Segments[1].radius then
    /* Two equally large segments */
    Create LeftQuad and RightQuad onto Quad
    Sort Segments into LeftSegments and RightSegments
    /* comparing directions with quad normals */
    if LeftSegments.count == 1 then
      Connect LeftQuad and LeftSegments[0] directly
        else
      Connect(LeftQuad, LeftSegments)
    end
    if RightSegments.count == 1 then
      Connect RightQuad and RightSegments[0] directly
        else
      Connect(RightQuad, RightSegments)
    end
  else // Ordinary connection
    Connect Quad and Segments[0], creating SideQuads[0..3]
    Sort Segments[1..N−1] into SideSegments[0..3]
    /* comparing directions with quad normals */
    for i=0..3 do
      if SideSegments[i].count == 0 then
        Fill SideQuads[i] with face(s)
      else if SideSegments[i].count == 1 then
        Connect SideQuads[i] and SideSegments[i][0] directly
      else
        Connect(SideQuads[i], SideSegments[i])
      end
    end
  end
end
```

Listing D.1: *Connect Pseudo Code*