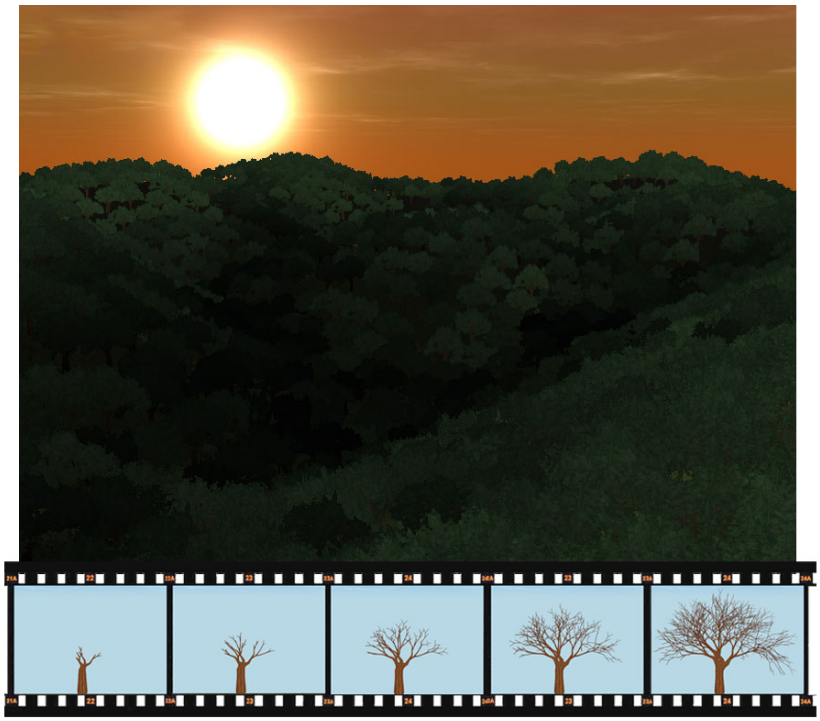
Visualizing procedurally generated trees in real time using multiple levels of detail

René Truelsen Morten Bonding

January, 2008

Department of Computer Science, University of Copenhagen Universitetsparken 1, DK-2100 Copenhagen East, Denmark

[1]rtr@rtr.dk, [2]mobo@diku.dk



Student project, 7.5 ECTS

: Kenny Erleben

Contents

Abstract

In this report we describe a combined method for growing, representing and efficiently rendering procedurally generated trees on commodity hardware. We use Context-free Stochastic Parametric L-Systems for simulating simple tree growth, and use a command-driven skeleton data structure, normally used for character animation, to represent and generate the geometry.

We use the geometry in a dynamic level-of-detail scheme, integrated into a slice-based volume rendering impostor. The data structure of the impostor is represented in a way, that allows for GPU-based transitions and "early exits" in the rendering pipeline to achieve optimal performance. Each slice contains a projection of a given subset of the geometry, enabling us to render a plausible and fast approximation of the original geometry.

Using our innovative optimized Vertex Buffer Object based batch rendering, we can render up to 10.000 trees at a frame rate of 20 to 50 fps, depending on the hardware in use.

This report contributes with the following new techniques and ideas for dynamic growth and LOD rendering[1]:

- Intermediate character-animation based skeleton representation of the L-system data structure, allowing for kinematic animations.
- Command mapping between L-system and internal representation, for enhanced modularity.
- Dynamic slice-based volume impostor representation and creation, allowing for run-time change of the geometric object.
- Advanced indexing scheme for GPU-based LOD transitions and "early exit" rendering.

We have made a home page for our project, http://vegetation.icandy.dk/, which features video demonstrations from our implementation.

[1] to our knowledge

1 Introduction

Rendering an interactive environment in 3D often involves balancing quality and resource usage in order to achieve a sufficiently high frame rate. Particularly when visualizing large quantities of virtual vegetation, which has a high level of detail. Consequently, each plant model must be rendered very efficiently for the program to run at interactive speeds. It is infeasible to render an entire forest consisting of geometrical trees, mainly due to the large amount of polygons involved. The solution to this problem is decreasing the detail level of the model. The concept is called level-of-detail, LOD, and it allows us to render a nicer looking model when it is the most noticeable, and less nice - but significantly faster to render - when it is less noticeable. Precisely how the LOD scheme is implemented varies with the specific needs of the program. It can be a reduced version of the object geometry or merely a picture of the object rendered on a, possibly camera aligned, flat quadratic surface. This is called an impostor.

In most interactive computer graphics software, 3D-models of vegetation such as plants and trees are typically imported from files or (manually) generated off-line prior to rendering. Often such applications has no apparent need for procedurally generated vegetation, the reason being that pre-generated models are both reliable and computationally inexpensive to include at runtime. Also, pre-generated designs allow for greater artistic freedom. However, constructing and editing such static models manually can be a tedious task and the results need not reflect natural plants in any way. In order for a scene to appear natural, especially when rendering a vast landscapes or forests, the graphical designer has to produce a variety of different plant-types and in turn create small variations of these models. Each of these models needs to be imported at runtime and rendered efficiently which often means making due with low quality or low resolution models. These types of models are cumbersome to process or edit procedurally since they often consists merely of a polygon mesh. This makes animation of the models difficult and computationally expensive.

Procedural methods for tree growth and simulation can achieve much greater realism and variation than static models. L-systems is a procedural method which can produce almost any type of structure, organism or fractal but can be very complex depending on the level of realism. Simple L-systems, however, are easily implemented, can be run in real time, and are commonly used and well documented.

1.1 Our goal

The goal of this project is to construct a program for procedurally generate a geometric object from simple skeleton models, based on stochastic L-systems, and efficiently rendering an approximation of the geometry using LOD. We focus on rendering trees and not vegetation in general, and we will focus on finding a representation which makes it possible to create an efficient and nice looking batch rendering of trees, in essence rendering a forest scene.

By utilizing existing programming library, OpenTissue, featuring the needed methods and algorithms, we can prepare for future extensions to our work. Therefore we will base our tree models on

a simple skeleton structure that enables seamless manipulation of the geometry.

1.2 Our solution

The outline of this report, the growth generator and the LOD scheme is as follows:

- Generate L-system string (section 3.1)
- Construct and render skeleton representation (section 3.1.2)
- Create slice textures (section 4.1)
- Render billboards using LOD (section 4.2)
- Optimize for batch rendering (section 5)

1.3 Dividing the work

From the start we had a mutual goal: creating nice looking images of a 3D environment featuring large quantities of trees. Furthermore we wanted to create an interactive system while having the trees look as realistic as possible. We each had favorite subjects, which we will account independently.

1.3.1 Morten Bonding

My individual contribution is focused on the growth of trees and rendering of the skeleton based geometry. My interest is based on a fascination of virtual growth which I first saw convincingly depicted in a video feature of the E-on Software Vue 5. I have partaken in another rendering project where we rendered very simple static trees. We made a very simple and rough 2-level LOD-scheme which blended between geometry and a static single level impostor consisting of two billboards in a cross formation. I became interested in creating better looking virtual trees and methods for growing them in real time. Since René was interested in LOD and rendering multiple trees with complex geometry can be a very demanding task, it was obvious to combine our efforts.

1.3.2 René Truelsen

My individual focus was on the LOD scheme, and finding a proper representation for the impostor, which was to be combined in an non-popping LOD scheme (chapter 4).

My inspiration came from a previous project where the rendering of grass was used in a naive way. It was outside the scope of that project to focus on LOD, but I have been wanting to look into optimizing specifically that part. Combined with Morten's interests in growing vegetation, I revised my focus to larger objects.
During the process of this project, my individual work and learning has been about:

- Using a framebuffer object as a render-to-texture target 4.1

- Different ways to represent geometric object in a real-time environment
- Limitations and challenges concerning LOD
- Discovering a method to achieve non-popping transitions
- Further learning about OpenGL and OpenTissue

1.3.3 Common work

As an extension to our original goals, we worked together on optimizing the initial naive LOD implementation in order to gain better performance in a scene with larger quantities of trees. Our solution includes using static vertex buffer objects, which allowed us to only send the necessary information to the GPU and thereby obtaining higher performance.

2 Previous Work

2.1 Tree growth and geometry generation

Virtual vegetation growth has been implemented and used in many different applications and the majority are based on variations of L-systems or Lindenmayer systems. Most of the previous work utilizing L-systems tend to focus on the growth of vegetation rather than the visualization. However, our focus it to produce nice looking images of trees, and for doing so, some sort of geometry is required. There are numerous ways of generating and representing trees geometrically. [TMW02] describes a combined procedural modeling method for directly constructing plant meshes from Parametric L-systems or "PL-systems". By iteratively applying rules for generalized subdivision and rule based mesh growing, arbitrarily complex and high resolution meshes can be grown. By using models of a series of real plants and a model of the environment [RHK03] simulates physiologic development of plants and their growth. [VHB03] presents a similar approach based on modified L-system for simulating illuminated environments and plant shadowing. By calculating illumination through photon flow simulation, avoiding other objects and detecting potential inter-branch collisions, a highly detailed rendering is achieved. The work of [OHKK03] focuses on interactive plant growth-modeling, describing an optimized L-system, which allows users to interactively generate, manipulate and edit trees based on L-System growth simulation.

A fractal model of branching objects are presented in [Opp86]. They use stochastic modeling for generating a variety of different natural phenomena such as plants and snowflakes. They describe their parameters as analogous to DNA. Their implementation which was made in 1986 allowed real time geometry manipulation. Unlike the work of any of the articles we have obtained, we will be using a skeleton representation for an intermediate data structure. How to create a skeleton and geometry using this approach will be one of the main focuses in this report.

2.2 Levels of detail

There are many different ways of representing the lower detailed vegetation, and each representation has its advantages and disadvantages depending on which problem it should solve. Amongst these methods are impostor billboards and aperiodic tiling. Which method should be used, depends on the scene and how the camera views the vegetation.

Aperiodic tiling using volumetric slicing is described by [DN04] to represent a large amount of trees as seen from the air, e.g. in a flight simulator. This idea is to slice the geometric tree along a horizontal plane. The slices are then tiled on top of the landscape, which, combined with a proper lighting scheme, can result in rendering a forest consisting of thousands of tree at interactive frame rates since a single tree is represented by a few tiles.

However, since most games require the vegetation viewed from the ground, the representation of impostor billboards is still the most common way to represent a high amount of trees. Although this method is old in the world of 3D real-time rendering [Bly98], its continuing popularity lies in its

simplicity and hence ability to be rendered fast, and it has the ability to be batch rendered as described by [Pel04], which makes it even faster and optimized for todays 3D graphics cards.

[Ash06] describes how to implement dynamic 2D impostors in a scene using singular billboards. Though this article does not refer directly to trees, it describes some of issues that has to be faced when implementing dynamical impostors. The problem is often that certain objects in a game can be considered secondary or tertiary to the gamer, and by rendering these object to a billboard texture, this texture can be used over several frames without having influence to the experience; for example if the camera vector to the object is unchanged or very small since the last rendering, or if the lighting changes significantly.

[Gue06] wrote an article on how to include a LOD scheme that goes from geometry, to multiple billboard impostors and finally to single billboard impostors. He especially focuses on the lighting problems that occurs when using multiple billboards. The solution was programmed on the OGRE engine which has a build-in octree representation of the landscape and objects in it. This helped improve the speed of the solution and resulted in some impressive results rendering 75000 trees at 35-100 frames per second. It also has some build-in blending options when changing between levels, but the article does not describe how to eliminate the ghosting effect that often occurs during this blend. The problem with this method is the increased amount of draw calls, as the distance to each tree has to be calculated and drawn individually.

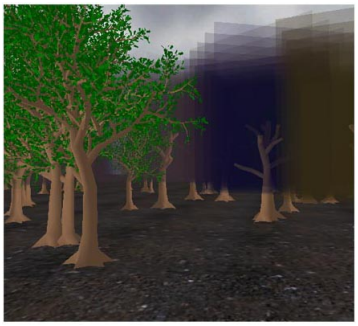


Figure 1: This image is from [Jak00] which illustrates how slices are used to represent the foliage of the. The left side is rendered with textured slices, and the right side of the image the slices are rendered with full color for easier view ability.

A more advanced way of using multiple billboards is called *billboard clouds*, and some methods of use to this method are described by [Ism05], [DDSD03] and [BCF+05]. The main concept is to use sliced 3D-textures to render the trees, but unlike the static axis aligned slicing scheme used by [DN04], the position and orientation of the slices are calculated statically or dynamically using some clustering algorithm (e.g. K-means algorithm). By calculating the slices dynamically it allows the geometric trees to be defined procedurally, and therefor gives the option to dynamically grow the trees at some interval. For [BCF+05] this method renders impostor trees very similar to the geometric versions, only using as low as 50-60 billboards (depending on the tree structure). As the camera is

distanced from the tree, this amount is decreased and thereby resulting in a LOD differentiation that allowed them to render 6.600 trees, 12.000-50.000 polygons, on hardware similar to[Gue06] at 9-21 fps.

[Jak00] introduced a simplified billboard based rendering approach that instead of dynamically calculated slices, uses multiple asix-aligned slicing to capture the foliage into a texture, and combines these with a geometric base. His method results in nice results by only using 3-7 slices, which equals 6-14 billboards (figure 1).

3 Growth simulation and rendering

This section describes the choices we make regarding the type of L-System we will use, the method used for constructing a skeleton representation and finally the process of creating and rendering a geometric model. Similar to [TMW02], we create a geometry from the L-system, however, we will use a skeleton representation as an intermediate data structure.

3.1 L-System

An L-system, also known as an Lindenmayer system, is a formal grammar used to model the growth processes of plant development. This basic system for rewriting strings were created by Aristid Lindenmayer as a formal description of the development of simple multicellular organisms such as Algae. The system, now commonly know as a Parametric L-system, were extended to incorporate more complex vegetation types and branching structures.

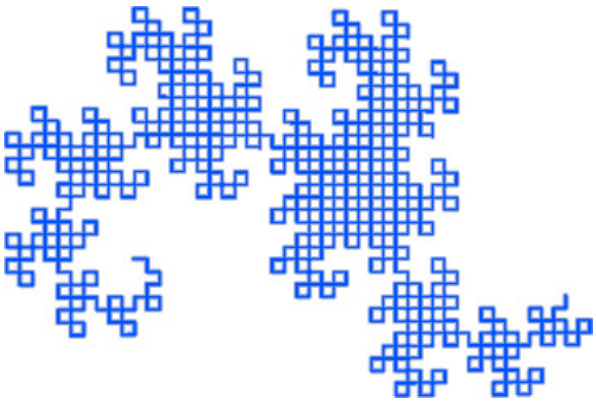


Figure 2: Dragon curve generated with our implementation.

The self-similarity property of simple L-system rule sets can be used to generate approximations of fractal curves. This family of curves are called Dragon curves and an example of such a curve is shown in figure 2.
An L-system is defined by a grammar:

$G = \{V, S, \omega, P\}$,

where

V is the alphabet - the set of symbols, or variables, that can be replaced

S is the constants - the set of non changing symbols

ω is the start or initiator, consisting of a sequence of symbols from V defining the initial state of the system. Also called `axiom`.

P is a set of productions. These are rules defining the way variables are replaced during an iteration. Two elements define a rule: the predecessor and the successor:

$predecessor- > successor$.

A string maintaining a system state, i.e. the series of symbols in the system at time t, we call an L-string.

This is an example of a (Context-free) L-System - Lindenmayer's original L-system for modelling the growth of algae:

Variables:	A B
Constants:	none
Initiator:	A
Productions:	A → AB, B → A

Doing 7 iterations produces the following output:

n = 0:	A
n = 1:	AB
n = 2:	ABA
n = 3:	ABAAB
n = 4:	ABAABABA
n = 5:	ABAABABAABAAB
n = 6:	ABAABABAABAABABAABABA
n = 7:	ABAABABAABAABABAABABAABAABABAABAAB

Production rules of an L-system determines how the system evolves. There are different ways of applying and restricting these rules:

- An L-system is said to be context-free if the individual production rule predecessors only refers to a single symbol and not its neighbors.
- If a rule depends on both the predecessor symbol and its neighbors, it is called a context-sensitive L-system.
- When a single predecessor symbol has more than one rule and the rule used for replacing is selected with a certain likelihood, the system is called a Stochastic L-system.
- By incorporating partial differential equations into productions, dL-systems can model growth in high detail

Since we have limited time for this project and also want to create a skeleton based implementation, we will use a Context-free L-system. By choosing the context-free version, we can simplify and optimize our string replacement implementation. Instead of a trial-and-error approach by simply trying to replace every possible rule predecessor by its successor, we will iterate the characters of the L-string and select the proper rule. This means that each symbol (a single character) can be used directly as a

means of indexing a pre-generated rule array. This is easily implemented and has constant time complexity when doing a lookup. The time complexity of the development of the L-system is exponential in L-string size for the grammars we will use, requiring fast rule lookup when replacing strings. A context-sensitive version would require some sort of hashing for selecting the proper rules, since it can consist of arbitrarily long strings.

The individual symbols in the grammar should be independent of the internal representation, which is why we choose to create a set of internal commands enabling us to a map a given symbol to an arbitrary internal action. This allows for much greater freedom than a static mapping, where e.g. F always result in "draw forward". Instead F could be defined as a production describing $F \rightarrow AB$, meaning "draw 0.1 forward and rotate 34 degrees" and $E \rightarrow AC$ meaning "draw 0.1 forward and then draw 0.5 forward". Execution of the internal commands will result in manipulation of the skeleton structure. The command-set we will be using is accounted for in section 3.1.2

3.1.1 Stochastic L-systems

For rendering nice and plausible looking images of trees, we want to apply variations to the fractal structures, which can be provided by a Stochastic L-system. We can incorporate this by allowing multiple rules per symbol and defining a likelihood for each rule. A Stochastic L-System created from the above example reads:

Variables:	A B C
Constants:	none
Initiator:	A
Production:	A → AB
Production:	B → A, likelihood = 0.1
Production:	B → C, likelihood = 0.9

Where $B \rightarrow A$ has only a 10% and $B \rightarrow C$ a 90% probability of being applied. Choosing a random value and weighting by the likelihood gives us the needed variation.

Doing 7 iterations produces the following output:

n = 0:	A
n = 1:	AB
n = 2:	ABC
n = 3:	ABCC
n = 4:	ABCCC
n = 5:	ABACCC
n = 6:	ABCABCCC
n = 7:	ABCCABCCCC

Clearly the obtained string length is much smaller than the non-stochastic version, giving a more controlled growth of the system.

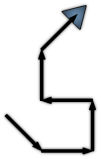


Figure 3: Example of turtle graphics. Only position and orientation is known at any time, no information is maintained about previous operations.

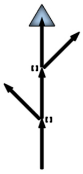


Figure 4: Simple L-system tree. The current state is saved([) and restored(]) for each branch adding

Furthermore, we can add noise to the rotation and length of the individual branches to avoid the artificial, structured visual look and thereby enhance the illusion of an natural environment.

3.1.2 Generating a skeleton representation

Rendering fractals are typically done using so-called turtle graphics, which supports only two basic operations: "draw forward" and "rotate". The approach is illustrated in figure 3.

When drawing tree-like structures, a "turtle state" is maintained. When starting a new branch we save the current position and orientation and then draw the branch. When we are done drawing, we restore the state and are ready for the next operation. This process can be thought of as a stack of states using push and pop operations as the saving and restoring of states. This method is simple and easy to implement when creating 2D line drawings. However, when working with 3D graphics, and especially when using a skeleton to store the intermediate structure, there are some considerations to keep in mind. The bone-skeleton representation uses a per bone relative coordinate system structure. This requires creation of new bones to incorporate the bone vector orientation for applying correct orientation of subsequent bones. Also, calculating the absolute coordinate system must be done explicitly

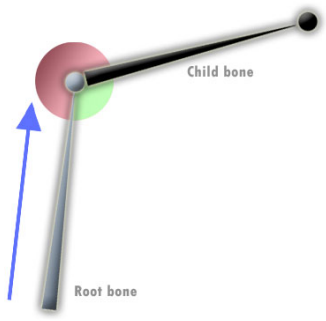


Figure 5: Translation(blue) and orientation(red and green) of the root bone. Notice that the orientation of the root bone determines the orientation of the child bone.

after bone creation- or orientation procedures. In the creation of bones, we need two informations: a vector describing the translational position of the bone and a rotation specifying the orientation of

the relative coordinate system. If a bone has no parent it becomes the root of the skeleton otherwise it becomes a child of the current root. The orientation of a child bone is determined by its parents orientation. An example of the skeleton-bone structure is shown in figure 5.

Each command an L-string corresponds to some action to be performed on our skeleton structure. This is the set of possible internal commands for manipulating the skeleton:

F	Increase the current bone length
R	Rotate the current bone ()
[	Push the current bone on to a stack of root bones
]	Pop bone from the root stack and make it the current bone

The root stack is used for saving the current state when branching to a sub-tree. Without this stack we would have no way of knowing which root to return to.

Given an L-string,

$FF[-F][+F]F + F$

we can make the following command mapping:

$$
\begin{array}{lcl}
F & \rightarrow & F \\
+ & \rightarrow & R(30^\circ) \\
- & \rightarrow & R(\text{-}30^\circ) \\
[& \rightarrow & [\\
] & \rightarrow &]
\end{array}
$$

The process of creating a skeleton representation of the L-string, $FF[-F][+F]F + F$, using the above command-set is depicted in figure 6.

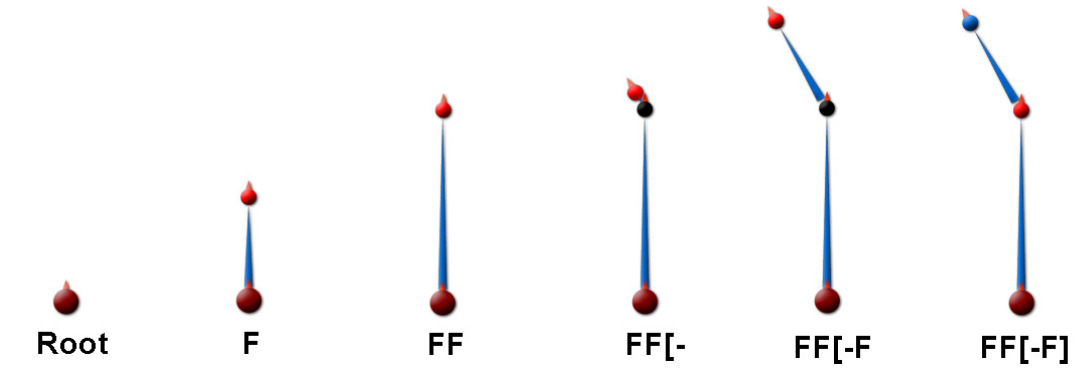
The orientation of a parent bone effects all children of that particular bone, illustrated in figure 7, therefore we rotate the bone translation instead of modifying the parent bone orientation. This is similar to applying forward kinematics to an existing skeleton: For each bone we specify its orientation thereby affecting all subsequent bones. We will use this property for applying randomness to our trees adding a little noise to these rotations.

3.1.3 Skeleton growth

Since we have a skeleton representation, we can easily modify the resulting geometry by manipulating the skeleton directly, without adding iterations to the L-system. By increasing only the bone lengths orientations remain the same and no bones are added or removed. See an example in figure 9.

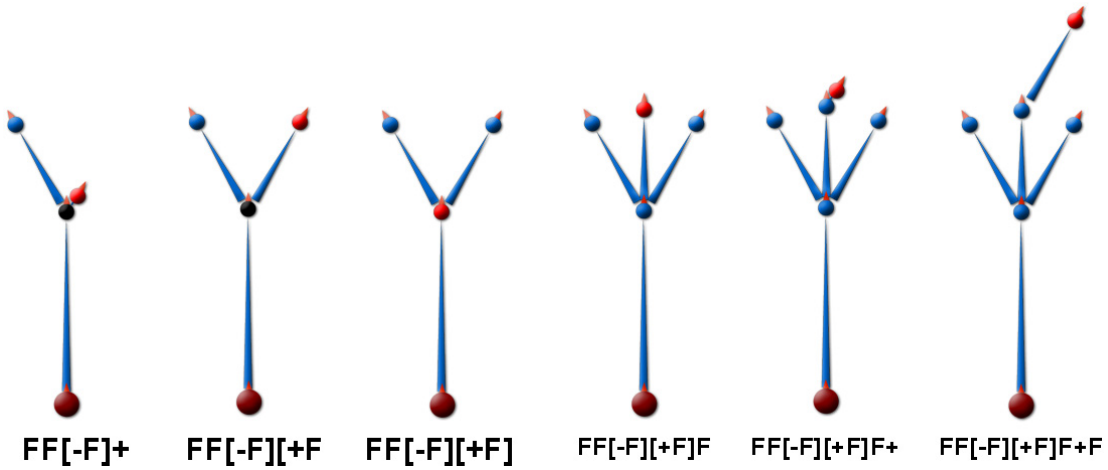
3.2 Geometry generation

Existing implementations(e.g. [TMW02], [OHKK03]) tend to use a mesh representation of the generated tree, allowing easy and nice rendering. This was also our initial idea because of the obvious advantages of doing so. Since we base our implementation on a skeleton representation, we have direct access to fast tools for rendering a mesh. Specifically we could create a character skinning mesh



(a) Creation of the first child node and applying forward growth. Notice the creation of an oriented node with zero length.

(b) A new bone is created when branching, [, or applying a rotation to a node which has already been grown forward.



(c) Like in the previous figure, the current node is pushed and popped from the root stack.

(d) When drawing forward after branching a new bone is added and becomes the new root. As long as there are no branching, the root stack is left unchanged

Figure 6: Process of constructing a skeleton representation from the L-string $FF[-F][+F]F + F$ using the turtle graphics approach. The blue sticks represent translation and circles represents orientations. The bone colored red denotes current root, black means that the bone has been pushed to the root stack.

and render it using the optimized OpenTissue skin-renderer. However, since we will be rendering multiple static billboards, using dynamically generated 2D impostor texturing, we find that we need a representation that enabled us to freely select an arbitrary subtree for rendering. For impostors to adequately represent the geometry with regards to visual quality, the amount of clipping should be reduced to a minimum. A single connected mesh skin will create clipping errors when spatially sub-

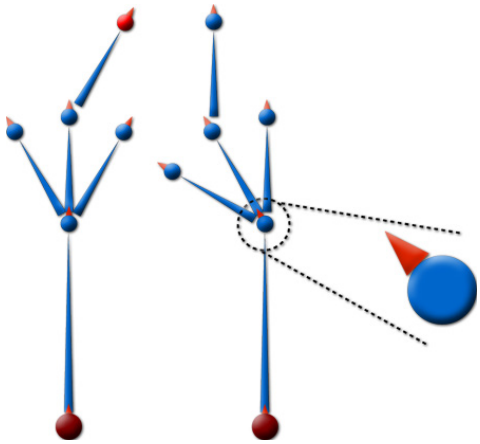


Figure 7: Illustrated how the orientation of a parent bone effects all children.

divided and rendered to slices. Therefore we decide to create a per-bone geometry allowing slicing and minimizing clipping.

For each bone we create a simple geometric object that can be rendered independently of the rest of the skeleton. Evidently a cone will be the best choice since each end can have a radius different from the other and thus allows variable thickness of bones. The thickness of each bone is dependent on which level it resides in the tree. This means that when a bone branches, each branch is a bit thinner. A single cone or cylinder will create the same problem as mentioned previously, namely the clipping artifacts. By rendering a half sphere at each end of the cone we obtain an apparently closed mesh. An example of tree growth and rendering is depicted in figure 8.

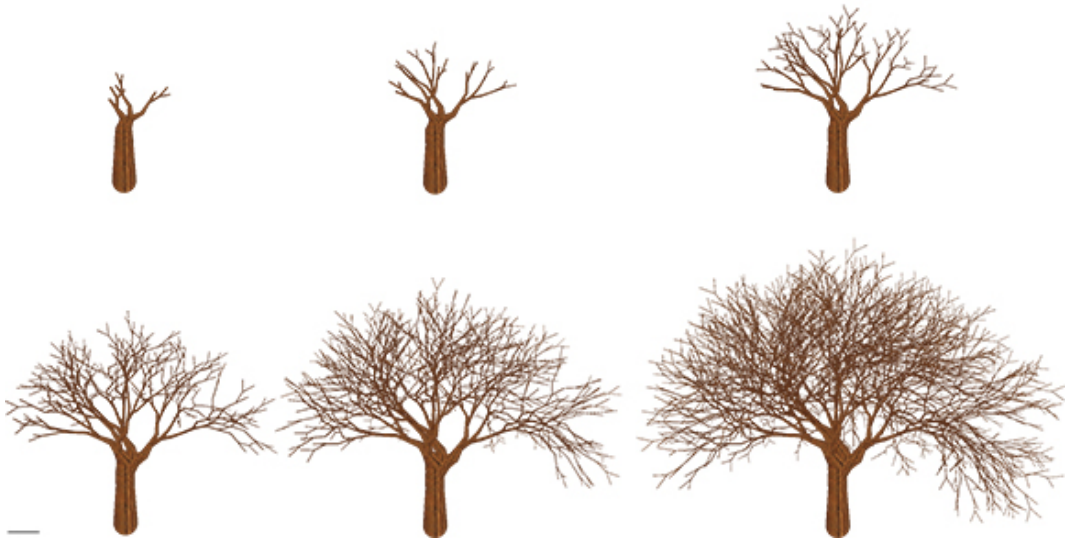


Figure 8: Example of the "Fractal plant" growth from age 2 to 7

Figure 9: Example of growing the skeleton independently of the L-system. Newly created skeleton representation of the L-system to the left, and the skeleton grown 10 times to the right. Note: no bones are added and orientations remain the same, only the bone lengths increase

Figure 10: Rendering tree trunk geometry using simple cylinders (left) and textured cones and half-spheres(right). Notice the softer transitions between the bones in the rightmost image.

Choices regarding the implementation of the geometry renderer is discussed in section 6.1.2.

3.3 XML import

For runtime access to L-system grammars and for easing testing procedures, we have decided to make an XML file import mechanism. For maximum usability, this importer should include as many of the user-definable variables as possible. In the XML a command should be a variable or constant which resembles an internal structuring command and a list of arguments for that command. Additionally, for debugging and easy editing, a semantic description of their function should be added to each element in the XML. The specifics of our implementation of the XML importer and the actual format, is accounted for in section 6.1.1.

We will use two example L-systems and create XML and command sets for them. One is a modified version of the so-called "Fractal Plant" depicted on WikiPedia.org. The other is our own empirically designed L-system.

4 Impostors and LOD

The LOD scheme and the impostor rendering are closely related, and this chapter will treat these two subjects.

An LOD scheme can be based on different layers. The scheme described in [Gue06] uses spatial-subdivision using octrees, and the detail level is given by which octree a specific item is located in. Or even if an octree is at all visible. Theirs is a general solution, which can be applied to any object in the scene. We will instead look more locally on the objects (trees) we wish to include in our scheme.

We looked at three ways of changing the level of detail for our procedurally generated tree:

Geometric reduction - As the distance to a geometric object increases, the amount of rendered geometry on the tree is reduced. In most cases this solution will lead to the best visual result, but with a huge penalty in performance.

Impostors - Reducing the geometric object to an impostor representation. This could be as billboard clouds as done by [Ism05] or [DDSD03], and reducing the amount of billboards rendered as the distance from the camera to the object increases. Depending on which impostor representation is used, this can result in very different visual result.

Combination - The combination of geometry and impostors is an often used technique, where the geometric object is rendered when close to the camera, and the faded into an impostor billboard as the distance increases. The impostor billboards can be either a single dynamic billboard impostor ([Ash06]), or some static billboard representation as shown by [Gue06]. For this scheme to work, it requires a proper transition between levels or obvious popping effects will occur when an object changes level. If the transition is done properly and the geometric object is relatively fast to render, this combination can lead to a solid combination of visual quality and speed.

Initially our problem was that we had no idea on how the L-system would perform, and therefore we were less inclined to focus on a scheme which included geometric rendering; especially a solution that would include the rendering of a full L-system tree. Instead we would focus on some impostor representation that could be used in a LOD scheme.

We were inspired by the slice-based foliage representation described by [Jak00]. This would have some possibilities in reducing the amount of billboards as the distance increased and less details were necessary. Again, since we at the decision time did not know exactly what our L-system tree would produce for us, we decided to replace the geometric trunk with textured billboards. The representation would allow us to dynamically generate and update the impostor, since the textures can be generated based on the L-system, and updated as the L-system ages.

We ended up with a method we call *dynamically generated slice-based volume rendered impostor*. With the selection of the impostor representation being based on billboard slices, our LOD variable will be the amount of slices that are rendered in our tree. Figure 11 illustrates how our LOD works

based on the distance to the object.

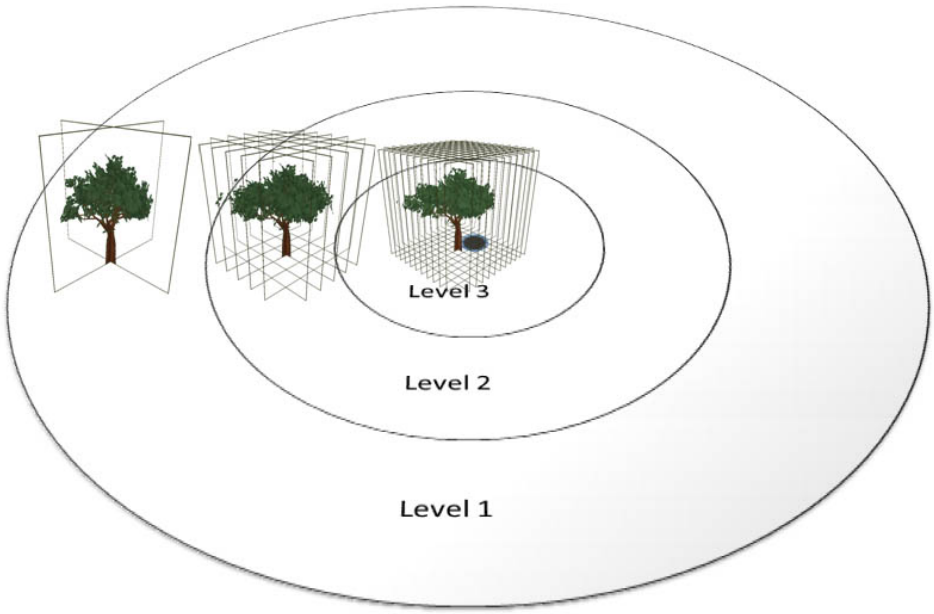


Figure 11: The different layers of detail. Level 3 is rendered with a higher amount of gridded billboards than level 2 or level 1, which is simply two crossing billboards. The border surroundings the billboards are added here for better visibility.

4.1 Slice-based Volume Rendering

This section will discuss the steps behind the slice-based volume rendering of the impostors, which includes how the slice textures are dynamically created at run-time, and how they are rendered on billboards. The main goal of the impostor rendering is to achieve the best possible visual result.

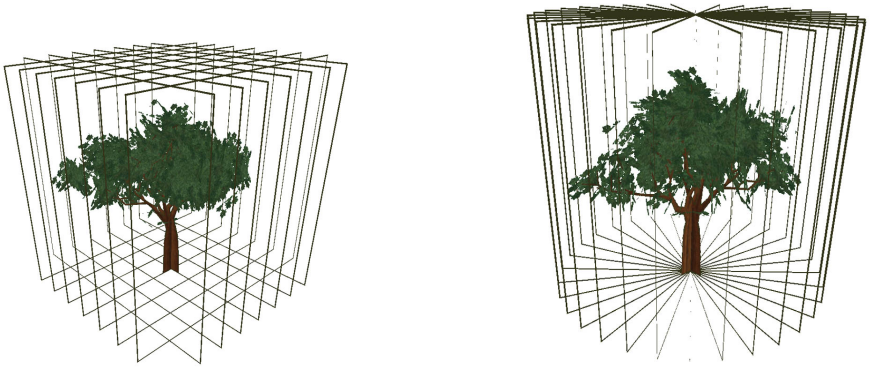
4.1.1 Basic steps

The outline for creating the slice-based volumes contains of two main steps: *create slice textures* and *render billboards using textures*. For more information on what billboards are and how they work, see appendix A.

4.1.2 Creating slice textures

We build the volume by rendering separate slices of texture and combining them in a structure. We tried both combining them as a grid (fig. 12(a)) or as a star (fig. 12(b)), but the problem is the flatness

of the billboards, which becomes very apparent when rotating around the star structured impostor. For this reason we decided to focus solely on the grid representation. Our LOD scheme requires our grid to consist of two billboards crossing at the center, one in each direction. We call them the base billboards. The scheme requires the remaining billboards are placed equidistant on each side of the center.



(a) Slicing volumes based on grid structure

(b) Slicing volumes based on star structure

Figure 12: Slicing based on grid structure vs. star structure. The viewed closer, the star structure as well as the flatness becomes apparent

Creating the slices are based on four actions. Each of these actions are performed on each texture.

1. Bind to a framebuffer for RTT (Render-to-texture)
2. Set up view frustum and clip planes
3. Render L-system
4. Unbind the framebuffer

Render to texture - The two classic ways of getting the geometric tree to the impostor textures are either by rendering the tree to the blank viewport and then copying the content to a texture (copy to texture, CTT), or by rendering to an alternative framebuffer that is later accessed as a texture (render to texture, RTT).

CTT is normally a good solution for impostors rendered to a lower resolution, for example for impostors only viewed at a distance, but our impostor is likely to be viewed up close, which requires our impostor texture to be rendered at at least the same resolution as the rest of the scene. In our case we used the resolution 1024x768, which requires our texture to be at least 1024x1024 to support mipmapping and copying this amount of data ($2 * 1024 * 1024 * 4 = 8MB$) might take its tow in the GPU.

Considering that we still want to be able to do this procedure while the program is running, for example to illustrate the growing of a tree, we decided to introduce ourselves to framebuffer objects,

FBOs. By utilizing the FBO as a render target, it would allow us to render the tree directly to a render buffer on the graphics card, later accessible as a texture without copying any data around on the GPU. Although changing the render target is a somewhat costly procedure, it gives us the freedom to alter the texture resolution without considering performance problems due to data copying. RTT is slightly more advanced to implement, since it requires the setup of bindings and memory allocation, and some older hardware might not have support for additional framebuffers.

As a side note we later realized that the bottleneck to our program turned out to be the texture lookup in the fragment program, which limited us to a 1024x1024 resolution. This limitation is most likely due to the cache handling on the GPU, where smaller textures requires allows for higher cache hit-rate. We did some testing for different texture sizes, and these results can be seen in chapter 7.3.3 on page 39.

View frustum and clip planes - Setting up the frustum and clip are key steps to a proper visual result. Figure 13 illustrates this step of the texture generation.

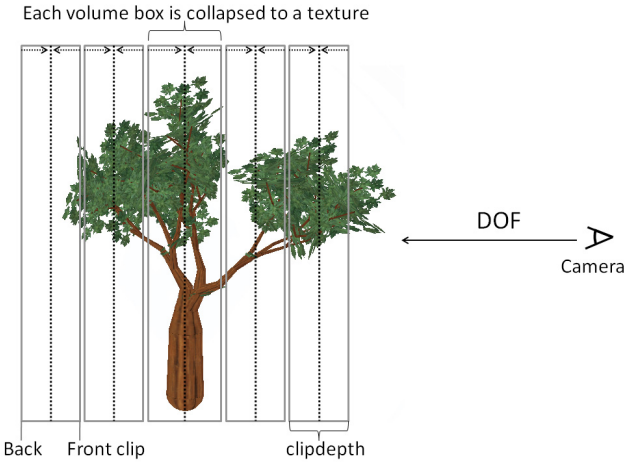


Figure 13: This illustrates how the geometric object is divided into volumes that are collapsed to a texture. Notice that the these volume boxes are seen from the side. The sides of the boxes would be considered the back and front of the box with respect to the camera.

The slice-textures are created by subdividing the geometric tree using clip planes. Respectively used to cut of the back and the front of the tree at some distance, depending on which texture is being processed. Since the two clip planes are parallel to the view frustum and to each other, the clipping can easier overcome by simply defining the far and near option in the view frustum settings. The distance between these volume boxes should be the same as the distance between the rendered billboards, to ensure we cover the entire tree. And the distance between the billboards should be determined by the size of the tree and the amount of billboards in each direction. Since our view port is quadratic, and we have to capture the tree from two perpendicular angles, we have to ensure that the full rotation of

the tree can fit within the viewport. The equation for determining the clipdepth is given by eq. 1.

$$\text{clipdepth} = \frac{\text{max\_tree\_dimension\_xy}}{\text{billboards}} \tag{1}$$

For the volume box to have parallel sides, the projection view for our FBO needs to be set up as parallel projection (non-perspective). Otherwise it will result in a distorted texture which we do not want.

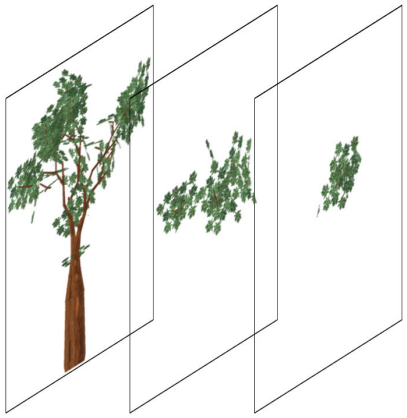


Figure 14: An example of three textures, which in this case would combined make out one fourth of a tree

The LOD require that the two center billboards will be the only billboards containing the trunk, and their texture will cover more depth than the remaining billboards. By empirical approach, this depth gave the best results for a depth equal a quarter of the trees radius.

Render L-system - The structure of the impostor is done in such a way that the base billboards, are the only billboards rendered with the trunk and the branches. The subsequent billboards are only rendered with foliage. This is because the gap between the billboards will become more apparent if the branches and trunk are shown. However, our L-system has been designed in such a way, that each branch is covered by leafs, and therefore this lag of branches is not noticed. The amount of foliage simply covers this fact.

The L-system is rendered according to chapter 3.1.2.

Unbind and clean up - As we unbind the FBO, we let the hardware create mipmap levels, so the texture is nicer to look at, but it also optimizes the texture lookups for billboards further from the camera. This generation of mipmap textures is a very important feature in the optimization process, and it is because of this step that we need to use a quadratic and a power-of-two resolution for our FBO.

The mipmap process is also a reason why 3D textures are not usable for storing the billboards, which we would have preferred for technical reasons (see section 4.1.3. Mipmapping in a 3D texture

requires a cubic texture, and it is not possible to separately mipmap the single texture layers within. This means that our 1024x1024 viewport requires the entire texture to be 1024x1024x1024, which is too high as we only need 10-20 textures for storing. We did some experiments with 3D textures, since they could ease the administration of the individual textures and it would make it possible to render the entire impostor using the same texture. Unfortunately, we could only conclude that the above statement is indeed valid.

After the mipmap levels have been generated, the texture is closed for writing and we can immediately use the texture in our rendering pipeline.

4.1.3 Render impostor

With the textures ready, the billboard coordinates are found as offsets from the base billboards. The billboards are rendered as quads, with regular texture coordinates in the interval $[0; 1]$. It is obvious that more billboards lead to a better visual result. Our impostor is based on 7+7 billboards, which means that it requires 28 textures for rendering. Because we are not able to use 3D textures, and being as the amount of billboards might vary depending on how many slices the impostor contains, the billboards in the naive version are rendered with separate render calls. We have to do this, as our fragment program cannot take a variable amount of textures, and since each billboards is rendered with a different texture. An optimized rendering process is described in section 5.

The render process will be revised in section 4.2 by the LOD scheme.

4.2 LOD

This section will discuss how we apply our LOD scheme to the impostor representation. Our scheme is applied in a way which allows for early culling of LOD billboard in the rendering process, while still being able to optimize further to batch rendering as described in section 5.

The main goal of the LOD scheme is to achieve better performance, but a close second is to preserve a good visual experience for user. This includes proper transitions and comfortable low detail quality. We believe our scheme fulfills both of these goals, as well as high performance.

We decided to isolate our LOD problem to only focus on how to represent our trees and how we can get them to look nice during the LOD transitions. We will not look into optimizing the representation of the landscape, as done by the OGRE engine in the article by [Gue06]. This section will go through how we setup the impostor for the LOD scheme, and we will show the naive implementation of the scheme. In section 5 we will go a step further, and take a look at an optimized implementation which to a higher degree utilizes the GPU.

4.2.1 Our method

Our LOD schemes is based on the representation of the billboards in the impostor. The billboards are indexed according to level, such that the lowest level billboards, which should always be rendered as shown by figure 11, are indexed lowest. We have illustrated this indexing in figure 15. The figure

shows how the billboards are distributed according to level, and it shows their corresponding index value. The billboards of higher level should only be rendered if we are within a certain distance. *By using this indexing order, we are able to loop through our billboards at render time, and when we come upon a billboard that is outside the distance of that specific level, we can exclude subsequent billboards (higher level billboards), since these will also be outside the distance.*

Notice that the base billboards will always be rendered as long as the impostor is considered visible. This is why these billboards should contain the trunk and also have a deeper clipping depth that the rest. The larger clipping area creates a visual difference between the impostor and the geometric, but this is an acceptable error.

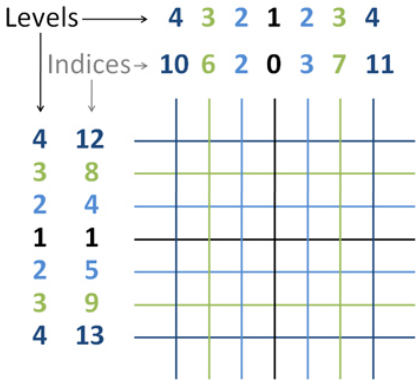


Figure 15: This image illustrates our slices seen from the top. The slices are formed in the grid structure which forms out our impostor. The image illustrates how each slice is linked to a level, and how it is indexed in our impostor representation. As level 1 is rendered, billboards 1 and 2 are rendered, for level 2 billboards 3-6 are rendered and so forth. In the image we have added color codes to each level of billboards for easier visibility.

The following LOD procedure is performed for each impostor in our naive implementation:

1. Determine distance to imposter
2. Loop through billboards
3. If the billboards is within its specific distance then render
4. If not then break loop and render next impostor if any

The *maximum distances* for the levels are passed on to the LOD scheme, and transformed to an array so every billboard has its own index to the maximum distance. For illustrations on how the rendering of each level corresponds to the billboards, see section 7.4.1.

4.2.2 Transitions

A general problem to LOD is the transition between levels, as some LOD schemes require changing from one impostor representation to another, e.g. from geometric to billboard. Such a transition is

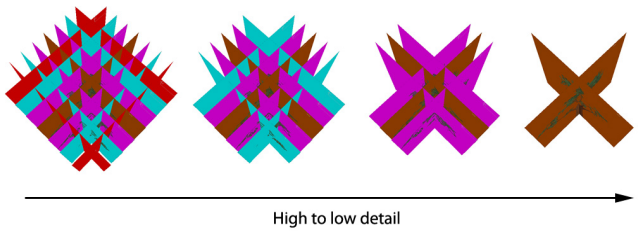


Figure 16: How the impostor levels from high to low detail, by varying the amount of billboard. Can be seen in effect in figure 27 on page 40.

likely to create a popping effect. This is sometimes solved by setting up transition intervals where the impostors are blended over a period of distance. The problem with using a transition interval is the lost of ability to batch render the object, since each object has to be handled individually due to blending process and so forth [GW07].

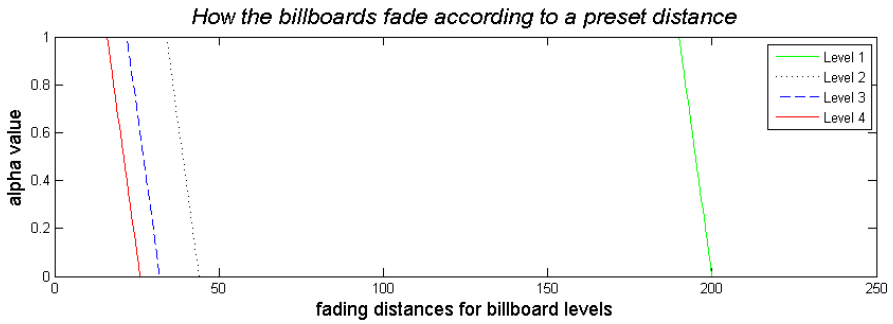


Figure 17: How the billboards at certain levels fade depending on the distance between the camera and the impostor. At the distance "maxleveldist", the billboard has been faded to alpha = 0. This fade is handled by the GPU.

In our scheme we have moved the transition to the GPU, which is significantly faster. This is possible since our impostor representation does not change, it only simplifies. We applied a linear fade to the alpha value according to formula 2, using the *maximum distances* for each billboards that we have already prepared in an array as mentioned in the previous section.

$$\text{alpha} = \beta * (\text{maxleveldist} - \text{distance}) \tag{2}$$

Figure 17 shows the graphs of the previous formula with fading values.: 200, 44, 32, 26 as the max level distances, and a gradient, β, of 0.1, which means the billboards have a transition fade over a distance of 10 units. This was empirical valued to give a nice fade.

This way of implementing the transition level is highly optimized for speed, and it also minimizes or eliminates the popping effect, depending on the implementation.

5 Batch Rendering

So far we have explained the basics of our LOD scheme based on a naive and brute-force rendering process. One of our goals were to be able to render large quantities of trees, and even though this is possible using the brute-force method, the performance is not impressive. See chapter 7 for performance tests. One of the bottlenecks is the amount of data we have to constantly send to the GPU as each impostor is rendered separately. This chapter will treat how we optimized this pipeline by reusing the vertex data already stored on the GPU, and only uploading the necessary data.

5.1 Vertex buffer objects

A vertex buffer object, VBO, is a powerful feature that allows us to store certain data in high-performance memory on the graphics cards. The idea is that a part of the memory on the graphics card can be allocated and later accessed through a handle. This is especially useful for static objects, where the vertex information is mainly unchanged, as it is the case for our trees. A VBO consists of two buffers: a vertex buffer and an index buffer. The vertex buffer contains information for each vertex, e.g. position, normal, texture coordinates and other information. The index buffer contains pointers to entries in the vertex buffer.

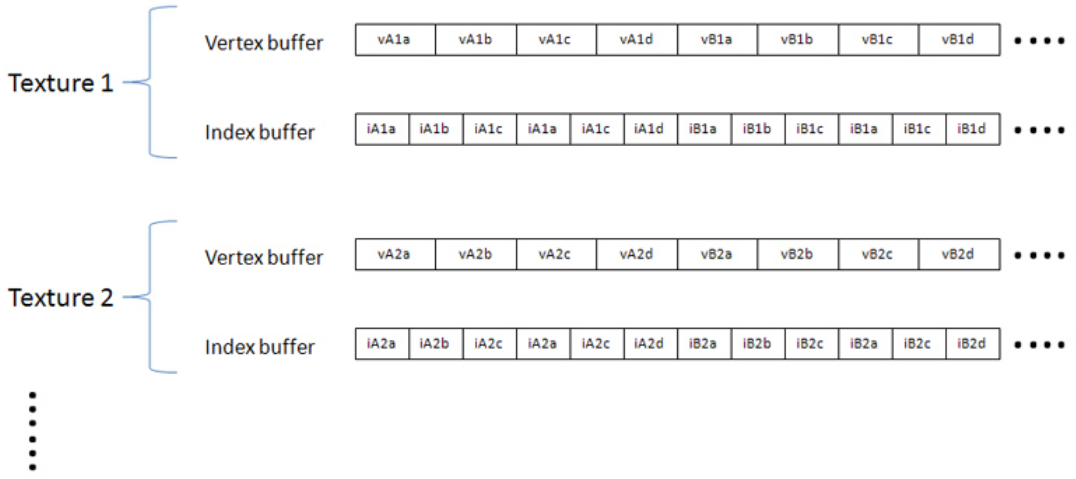


Figure 18: An illustration of how each texture is connected to a VBO. Each VBO consists of a vertex buffer and an index buffer. In this example vA1a-vA1d are the vertices to impostor A, billboard 1. Notice how the index buffer in this example renders triangles even though the billboards are defined by 4 vertices.

5.2 Our method

We are still limited in our fragment program by only one texture per billboards, which is why the naive implementation basically loops through the billboards, while each time binding a new texture to the fragment shader.

Our method to exploit the VBOs, is to exploit that billboards across the impostors all contain the same textures. We have therefore grouped the billboards in separate VBOs, one per texture. Each VBO contains one billboard from each impostor, and all billboards with the same specific texture are batch rendered. This is illustrated in figure 18.

The vertex buffer can contain a lot of extra information for each vertex. In our case we decided to apply a simple light model to our scene, using the normal from the landscape as the normal for the tree. The means that the tree will be shadowed as the landscape, which gives it a more varied visual result (see the picture on front page).

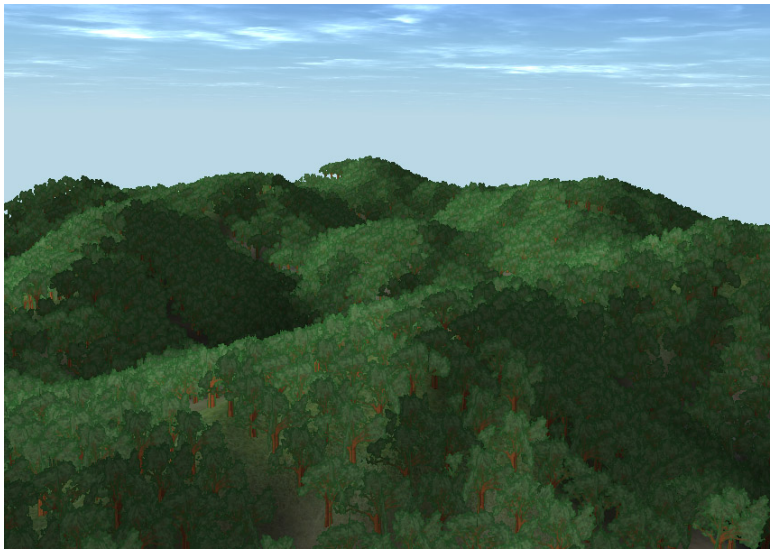


Figure 19: Landscape containing 2.385 trees, running at 30 fps using the VBOs.

5.3 Uploading index buffer

Before each render call we optimize the render process by looping through the impostors, and we check if the impostor is within the view frustum and what level it is at, so we can exclude its billboards from rendering. From these informations we organize the index buffer for each VBO, which is then sent to the graphics card before rendering.

During this process, we optimized the part of OpenTissue which handles the VBOs for us, by allowing the amount of uploaded data to be dynamic for the index buffer. This means that during these steps, we optimized the data stream to only contain the necessary data.

5.4 Compare to brute-force

In an example of 4 levels, each impostor will have 14 textures. Consequently we will have 14 VBOs, each containing the information of exactly one billboard from every impostor. In this case we have to do only 14 render calls, one per VBO. With 2000 impostors this means that we have saved $2.000 * 14 - 14 = 27.986$ render calls. The corresponding analysis of the data-transfer complexity in the case where all billboards are rendered at high level:
$\alpha = number\_of\_impostors$
$\beta = number\_of\_billboards\_per\_impostor$
$\gamma = number\_of\_vertices$
$\theta = datasize\_per\_vertex$

The datasize per vertex:

$$sizeof(normal + coords + tex.coords + dista.to.cam + maxfade) = \quad (3)$$
$$(3 + 3 + 2 + 1 + 1) * 32bit = 320bit \quad (4)$$

For brute-force:

$$\alpha * \beta * \gamma * \theta = \quad (5)$$
$$2000 * 14 * 4 * 320bit = 35840000bit \quad (6)$$
$$= 34.18Mbit \quad (7)$$

For VBO optimized: $\alpha = number\_of\_impostors$
$\beta = number\_of\_billboards\_per\_impostor$
$\gamma = number\_of\_vertices$
$\theta = index\_datasize$

$$\alpha * \beta * \gamma * \theta = \quad (8)$$
$$2000 * 14 * 4 * 32bit = 3584000 \quad (9)$$
$$= 3.42Mbit \quad (10)$$

The factor 10 difference is due to the fact that our trees are static positioned, and we only upload these coordinates ones.

6 Implementation

In this section we will discuss and describe some of the specific choices we took during the process of implementation. Our implementation is programmed on computers running Microsoft Windows and Visual Studio 2005 SP1. We used the Game Animation 2007 framework and incorporated the classes, types and data structures from OpenTissue whenever possible. The program was tested and run on a variety of computer hardware (see section 7.1).

First we will describe a skeleton based implementation of stochastic L-Systems and dynamic creation of geometric representations. Then we will discuss the details of implementing an efficient LOD-scheme using sliced rendering and static billboards.

6.1 L-systems and geometry rendering

The growth process can be divided into several minor steps:

- Import grammar and commands from XML into internal data structure
- Iteratively grow the L-system by doing string replacing
- Create a skeleton from the resulting l-system
- Create a geometric representation of the skeleton
- Render the geometry on demand

The implementation of these steps has been divided into the following C++ template classes:

LSystem Imports XML, has internal L-System grammar and grows the L-system.

LSystemSkeleton Inherits from LSystem. Creates and maintains the skeleton tree representation of the L-system

SkeletonTreeRenderer Renders a LSystemSkeleton as a geometrical model

In addition to these classes, we have made a convenience class working as an extension of OpenTissue. The `TreeBone` class inherits from the `ConstrainedBone` class and extends its functionality with type tags such as "leaf" or "trunk" used when generating the geometry. Also, it adds accessors for the data structures in the super class of `ConstrainedBone`, the OpenTissue `Bone` class for character animation.

6.1.1 Class LSystem

The `LSystem` class imports XML and by pure string replacing it manipulates the L-string iteratively. We have generalized the implementation to make it completely independent of our specific usage. The `LSystem` is meant for inheritance, allows easy extending and features its own XML importer.

Based on the requirements outlined in section 3.3, the set of items in the XML format is:

Variable single symbol

Constant single symbol

Initiator sequence of symbols

Production a single symbol predecessor and a multiple symbol successor

Command single symbol to single symbol mapping as well as an argument list

For reading XML formatted text files, we use the `TinyXML` library included with OpenTissue. This allows for simple access to variables, constants, productions and commands. Since we found it to be the easiest way of obtaining data, the XML-format is flat and uses element attributes for storing data in stead of using a tree structure.

An example of the resulting XML format is shown in appendix C.

6.1.2 Class LSystemSkeleton

`LSystemSkeleton` inherits from LSystem extending it by mapping the characters of the L-string to a series of turtle-graphics bone manipulation commands, as described in section 3.1.2. `LSystemSkeleton` uses the extended OpenTissue kinematics Skeleton template class with our own `TreeBone` bonetype. This extended bone type is used for storing additional information about the bone: thickness at each end and its internal type, i.e. "trunk", "branch" or "leaf" used by the geometry renderer. Also it maintains an AABB [2] for the resulting tree, used by the Volume Renderer to set up the view port.

6.1.3 Class LSystemSkeletonRenderer

For generating and rendering the geometry we used OpenTissue utilities for drawing simple geometric object. We created a highly modified version of the Capsule drawing utility, which renders a cone with a half sphere in each end. The implementation features texture mapping using vertex and pixel shaders. An example of an L-system rendering is shown in figure 20.

Figure 20: Example of a rendering of the L-system "Fractal plant" (appendix C)

[2] Axis Aligned Bounding Box

6.2 Impostors

The basic steps of the impostors were described in section 4.1, and we will in the following section go into some specifics on how we implemented the decided procedures.

6.2.1 Creating the texture

We implemented a class for containing the functionality of our FBO. This included a constructor, a destructor, a `bind` and an `unbind` method. We did not use the OpenTissue FBO class, since we needed to be able to manipulate the class, for example to be able to handle 3D texture.

The FBO is initiated with a depth- and color buffer, along with the allocation of a dynamic array of textures. We initially followed the classic procedure of binding each texture to a color buffer, as described by [Per07]. However, we experienced that OpenGL is limited to only 16 color buffers per FBO, but even more restrictive was our hardware that only allowed 4. Instead we decided to completely rebind color buffer 0 using `glFramebufferTexture2DEXT` to separate textures at every `bind` to the FBO-class. This gave us an unlimited amount of color buffers textures, though with some extra overhead. We did not use stencil buffers in our FBO, since it was to be used for texturing and therefore deemed unnecessary.

6.2.2 Rendering the impostors

To represent our impostors we created a class with the somewhat bewildering name `Billboards`. Initially the idea was to generate every billboards as an object, but it quickly became apparent that in order to keep a good performance, this was not the way to go. Instead we expanded our class to contain more billboards, ans so the correct name for the class would have been `Impostor`. The primary responsibility for this class is to keep track of:

- Create the billboards for the impostor
- Number of billboards
- Brute-force rendering the impostors

This is a list of the primary methods from `Billboards`:

`set_dimensions`	Sets the dimension of our billboards (width,height)
`set_static_radial`	Sets the impostor to be have a radial structure. Parameters define angles between billboards.
`set_static_grid`	Sets the impostor to be grid structured. Parameters set the amount and the distance between billboards.
`set_static_angle`	Sets the impostor as a single billboards at some angle to the XZ-plane.
`set_non_static`	Sets the impostor as a single cylindrical billboard.
`set_texture`	Points the texture to either an FBO or loads a texture from file (was used for testing).
`transform`	Translate, rotate or scales the impostor. This was used by our LOD scheme.
`render`	Renders the impostor.

The impostors were rendered with alpha testing, `glEnable(GL_ALPHA_TEST)`, and an alpha check value of 0.5, `glAlphaFunc(GL_GEQUAL,.5f)`. This means that our transitions are not as smooth as we would have liked. Alpha blending was not a possibility since our billboards are crossed, which does not work well with blending. Furthermore blending requires back to front rendering. We also tried to enable the GL-extension `GL_ALPHA_TO_COVERAGE`, but this requires multisampling which is done in the fragment pipeline, and our bottleneck was concluded to lie in the fragment shader, see section 7.3. Further strain on the fragment pipeline would only lead to worse performance, and we were also only able to enable it properly on one of our computers. Our guess is, because of some limitations in the driver.

Considering that blending requires both billboard subdividing and sorting, along with the fact that blending is a more costly operation, we decided to use alpha testing.

All impostors are rendered using our own shaders. See appendix B.

6.3 LOD

To handle our LOD scheme, we created the class `LOD_scheme`. The primary responsibility for this class is to keep track of:

- The number of levels in our scheme
- Number of impostors in our scheme
- Transformation details of each impostor (position, orientation and scale)

The reason why the LOD class is responsible for the transformations of the impostors is that, it only instantiates one impostor with center in `(0,0,0)`, and by each render call it translates this object into the correct positions. Thereby saving memory.

This following is a list of the primary methods from `LOD_scheme`:

`make_texture`	This method binds the FBO, and creates the textures depending on whether radial- or grid structure
`render_billboards`	Calls the impostor prerender, which loads and enables shader programs along with variables and textures. Loops through the impostor positions, and then transforms and renders the impostor.
`create_impostor_vbo`	Creates the VBO, and organizes and uploads the positions, normal and orientations to the graphics card. The index buffer is used for later.
`render_imposters_vbo`	Calls the impostor prerender, but instead of calling the impostor render method, it updates index buffer according to impostor visibility and uploads to the graphics card.

The brute-force implementation was the initial version of our LOD scheme, which is based on loop of rendering every impostor individually.

A billboard was rendered as `GL_QUADS`, and its coordinates and normal were passed on the shaders, along with the texture coordinates, the distance to the camera and maximum level distance which were passed on in a 4-vector multitexture.

7 Results and evaluation

This section describes the evaluative tests and benchmarks we have performed on our implementation. It is divided into our 3 subcategories, and within each of these we evaluate with regards to *correctness* and *performance*.

7.1 Hardware

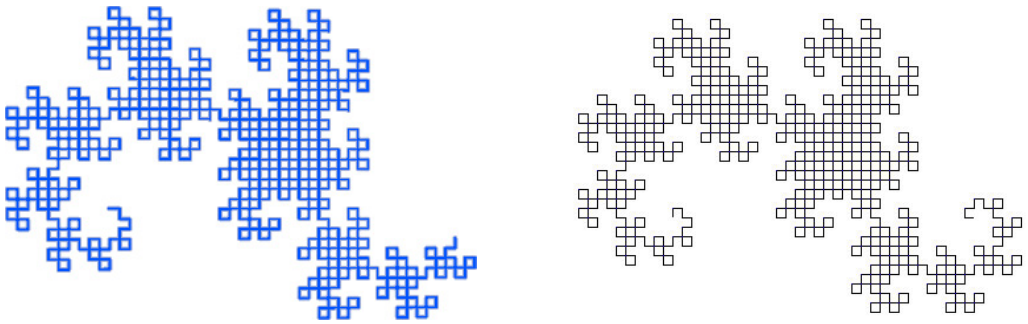
We will test our implementation on a computer with the following hardware configurations: AMD Athlon 64 3400+, 1GB RAM, Radeon X800 256MB The machine runs Windows XP Pro SP2+ and Visual Studio 2005 SP1. We use the "`mobo`" `OpenTissue` branch, revision 3954 (publicly available at `http://opentissue.org`).

All the performance benchmarks were run at a viewport resolution of 1024x768, with a frame-buffer object resolution of 1024x1024, if nothing else is mentioned.

7.2 L-System and bone rendering

7.2.1 Correctness

We have compared the rendering of a Dragon curve fractal using our own implementation with a reference image from WikiPedia.org. See 21.



(a) Dragon curve fractal generated using our implementation

(b) Dragon curve from Wikipedia.org

Figure 21: Example of the Dragon curve fractal generated using our implementation(left) and the refererence image from WikiPedia.org

7.2.2 Visual Evaluation

The visual result for rendering the bones according to the L-System are very satisfactory. Figure 22 shows the real tree compared to one of our L-System based trees.

As L-Systems are basically fractals, then it is possible to achieve as high a detail levels as needed, by just increasing the amount of iterations. The cost for this is of course performance penalty. Figure

(a) Real tree (b) Synthetic tree

Figure 22: Comparison between a real tree and a bone-rendered L-System tree. Notice the fractal realism in the synthetic tree, which comes from adding deterministic Gaussian random to the bone rendering.

23 illustrates how the limitations of the fractal resolution visually affects the synthetic tree when zooming in. At a distance the two trees looks reasonably alike.

(a) Real tree. (b) Synthetic tree

Figure 23: Comparison between the fractal resolutions of the real and the synthetic tree. Notice how the branches of the real tree have a naturally high fractal resolution, which gives it smooth curves, compared to the synthetic tree which has rough curves. An increase in fractal resolution will improve this, but at the cost of performance.

As illustrated by figure 24, the branches have very similar endings. This is a limitation caused by our L-System language. It can be avoided by additional tweaking, but we decided not to go into this. The visual exposure of this limitation can also be reduced by increasing the fractal resolution.

7.2.3 Performance

Table 1 shows the frame rate for the geometric object at different ages and different amount of bones. This table emphasizes the importance of creating an efficient geometry representation. The frame rates also shows the need for some batch optimization of the rendering procedure for the skeleton bones, which was implemented as a brute-force rendering procedure.

(a) Real tree (b) Synthetic tree

Figure 24: The trees viewed at a glance to illustrate how the branches in the synthetic tree have very similar endings compared to the real tree. This is because of a limitation in our L-System language, but our limited fractal resolution makes it more noticeably. Also notice how simple appliance of a bark-texture can add some realism to the tree.

Age	Bones	FPS
0	2	275
1	5	260
2	17	220
3	53	78
4	161	27
5	486	9
6	1460	3

Table 1: The frame rate for the geometric object at different ages and different amount of bones. More bones results in a significant frame rate reduction.

7.3 Impostors

7.3.1 Correctness

Figures 25 and 26 show the geometric tree compared against the impostor from two different angles. The first angle (fig. 25(a) and 25(b)) is from a side that is parallel to the grids, where the correspondence between the two is highest. This is because the grid is aligned with the view frustum when the texture were generated. The second angle (fig. 25(a) and 25(b)) is at an angle which is not aligned to the grid, and here the error is more noticeable. It is obvious in figure 25(a) that the density of the foliage is higher, which is due to the increased clip depth of the texture on the base billboards. Also there is an obvious branch that is cut off. An issue which is a problem for our impostor representation is the base. It is very obvious that this is made up by two billboards. This is a known limit to our method.

(a) Geometric tree (b) Impostor tree

Figure 25: Notice how well aligned the two images are. This is because the viewport is parallel to the grids. This is when the error is less. The impostor is composed by 7+7 billboards.

(a) Geometric tree (b) Impostor tree

Figure 26: Notice how the difference is higher when viewed from an angle. There is a branch that is cut short, and the foliage is more dense because of our increased clip depth of the base textures. The impostor is composed by 7+7 billboards.

7.3.2 Visual evaluation

The visual problem with out method is that the grid structure becomes apparent as the impostor is seen at an angle where the billboards are parallel to the eye vector. This is to some degree the same problem as mentioned with the star structured impostor. [Jak00] solved it by blending out these billboards. We did not like the ghosting effect that appears during as the billboards blend in and out, so to solve this problem, an alternative slicing scheme and billboard structure could be explored. This could be changing the current static equally distributed grid structure to a more adaptive one.

	Distance from camera		
Representation	**3.9**	**5.6**	**19.0**
Geometry (161 bones)	28 fps	28 fps	28 fps
Impostor (14 billboards)	54 fps	84 fps	185 fps

Table 2: Showing how the frame rate for the geometric tree and the impostor at different distances.

	Enabled		Disabled	
Position	Billboards	FPS	Billboards	Disabled
1	1830	11	10890	9
2	1860	23	10002	20
3	2938	42	4018	38

Table 3: Performance (in frames per second) with frustum culling enabled or disabled at three different camera positions, with different amount of rendered billboards.

7.3.3 Performance

A requirement for performance is that the impostor is rendered faster than the geometry. Table 2 is comparison between a geometric tree and an impostor and different distances from the camera. It is noticeable how the frame rate for the impostor increases as it takes up less of the viewport. The reason is that at shorter distances the impostor takes up more fragments of the viewport, and this requires more texture lookups in the fragment program. The texture lookup is our bottleneck, since it is done 14 times on a 1024x1024 texture. This bottleneck is supported by 3, which shows the frame rates for different camera

The frame rate for the geometric tree is stable, since its bottleneck is the brute-force rendering of the skeleton bones.

As we have previously stated, our bottleneck is the texture lookup in the fragment program. More specifically because our textures go up to 1024x1024 when close to the impostor. Further away the textures are mipmapped to lower resolutions. Table 4 shows the performance for different FBO resolutions. For the static scene we can see that one of the bottlenecks is the texture lookup in the larger texture. The dynamic scene is a camera "fly over" of a forest containing 3000 trees. Notice that the frame rate changes significantly when changing to the lowest FBO resolutions, which states a much lower visual quality. Since we had a moving camera in the dynamic scene, the quality of the texture was not noticeable, and so these numbers stretches the importance of tuning the FBO size to how the scene is observed.

	FPS	
FBO resolution	**Static Scene**	**Dynamic Scene**
1024x1024	46.8	13.1
512x512	56.2	13.6
256x256	56.9	16.2
128x128	57.2	19.8
64x64	62.1	22.7

Table 4: Showing how the frame rate for different texture sizes (FBO resolutions). The static scene is a single tree which is rendered at an angle where it covers most of the viewport. The dynamic scene is a scene containing 3000 trees, with the camera moving around and the trees are covering the entire viewport.

7.4 LOD

7.4.1 Correctness

Figure 27 shows the correctness of our LOD scheme. The snapshot is taken from a secondary camera, which shows how the detail levels are rendered, based on the distance to the primary camera. The impostors furthest away are rendered with two billboards, and by every level another 4 billboards are added to the impostor. The impostor representation in this figure is the same as illustrated in figure 16 on page 26.

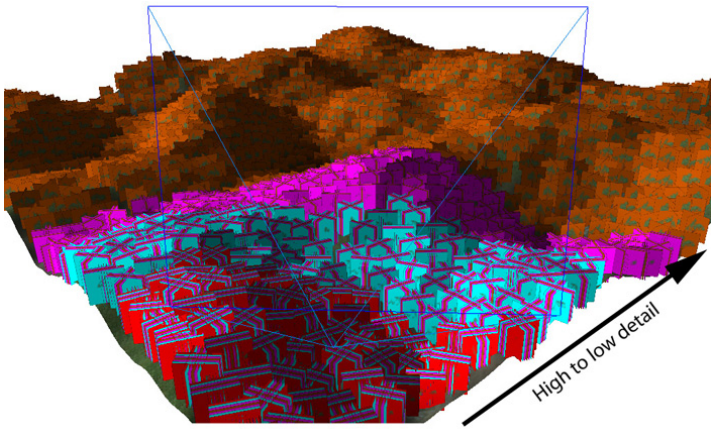


Figure 27: A snapshot that illustrates the different levels of detail. The camera and its view frustum is illustrated as well. Notice how the colors change as a result of lower detailed impostor.

Figure 28: The same scene with the impostor trees.

7.4.2 Visual evaluation

Figure 29 illustrates the difference in the low level representation of the impostor compared to the high level. All though there is an obvious difference, this difference becomes much more apparent when motion is applied. This is due to the increased parallax which is caused by the added layers of billboards. Unfortunately this paper is limited to only show 2D images, and it is not possible to show this parallax effect.

(a) Low level impostor with 2 billboards (b) High level impostor with 14 billboards

Figure 29: Comparison between a low level impostor vs. a high level impostor. These images illustrate that there is a difference between the two levels, but they do not illustrate how big this difference is in motion. The reason for this difference is caused by the increased amount of parallax for the high level impostor.

Some tests with dissolving impostors, by applying alpha noise to the texture in the fragment program, showed some visual improvements to the visual quality of the LOD. A proper implementation of this feature would be to apply the alpha noise to the texture at creation time. Time restrictions kept us from doing this, but when applied properly, the alpha noise will be a part of the texture on the GPU, and this feature should leave no performance penalty to the scheme.

7.4.3 Performance

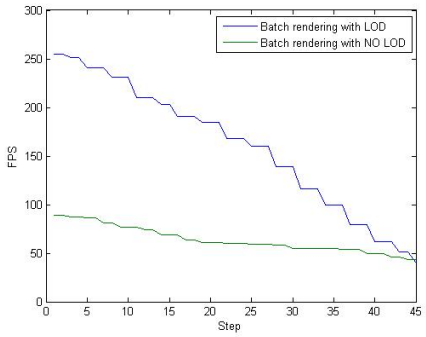


Figure 30: Camera tour with/without LOD activated.

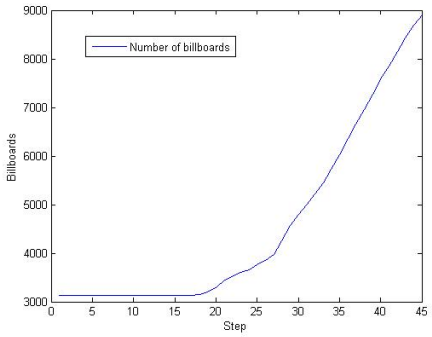


Figure 31: Number of billboards activated during the camera tour. We did not plot the non-LOD tour, as this is constant 21966 billboards.

Figure 30 shows the camera tour run twice with and without LOD active. Clearly LOD gives an advantage as long as the camera is far away and the number of billboards present is limited (see figure 31). When positioned close to the trees the performance is nearly identical. Again, this indicates that the fragment program is the bottleneck.

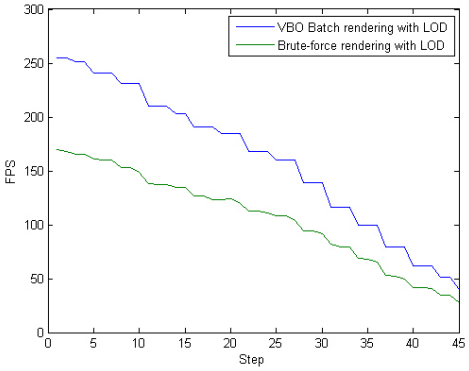


Figure 32: Running the camera tour with VBO rendering and brute force approaches respectively

7.5 Batch rendering

Since the batch rendering is merely a generalization of a single tree, the quality is unchanged. However, when rendering the same impostor numerous times in the same scene, the uniformity becomes visible.

7.5.1 Performance

Figure 32 shows the obvious benefit of our VBO based batch rendering. Again, the results emphasize the speed limitation of the texture lookup in the fragment program as the camera is moved close to the trees.

8 Conclusion

We have met the goals we set for ourselves, and our results are quite satisfactory regarding both performance and visual quality. On some points have our results surpassed our expectations.

We went into this project with little knowledge of the outcome, if it would work at all. To our knowledge some of the methods we have implemented and parts of the basic concepts are innovative and we have not seen any previous work documenting these methods. We think we have produced results with novelty value for the community.

We have described and successfully implemented a combined method utilizing the following techniques:

- Context-free Stochastic Parametric L-Systems - For simulating simple tree growth
- Skeleton based L-System representation - Constructed using the turtle graphics paradigm
- Skeleton geometry renderer - Allowing individual rendering of Trunk, branches and leaves
- An dynamic impostor representation - Allowing run-time changes applied to the object.
- An LOD scheme with smooth transitions - For optimized performance and visual quality
- A VBO-based batch rendering procedure - For optimized data stream to graphics card.

9 Future work

There are several optimizations and many ways of enhancing our implementation. Here is a compiled list of the ideas we would have focused on, if we have had the time.

9.1 Growth and rendering

- Performance of the geometry rendering is currently not adequate. For making growth and impostor updates run interactively, optimization of the geometry rendering is crucial
- Seamless transitions between individual levels of age enabling continuous growth
- Amortization of the skeleton creation process
- Context-sensitive L-systems for greater variability
- Time-of-year leaf rendering

9.2 Impostors and LOD

- The LOD scheme has proved to be bound by the fragment part of the rendering pipeline. This means that for improved performance this should be addressed.
- Self shadowing
- Amortized updates of growing L-system on texture slices
- Noise on the alpha channel for impostor textures. This can help eliminate popping during transitions, as billboards will dissolve during alpha test.
- Sorted front-to-back rendering of the impostors. This can help improve the fragment pipeline.
- A more adaptive slicing scheme in the impostor, for improving the billboard distribution to fit the geometric object.
- Include the camera speed to the LOD scheme. High speed requires lower details.

References

[Ash06] Ashley Davis. Dynamic 2D Imposters: A Simple, Efficient DirectX 9 Implementation. http://www.gamasutra.com/features/20060105/davis_01.shtml, January 2006. Accessed Dec, 8th 2007. 9, 19

[BCF+05] Stephan Behrendt, Carsten Colditz, Oliver Franzke, Johannes Kopf, and Oliver Deussen. Realistic real-time rendering of landscapes using billboard clouds. *Comput. Graph. Forum*, 24(3):507–516, 2005. 9

[Bly98] David Blythe. Advanced Graphics Programming Techniques Using OpenGL. In *SIGGRAPH 98 Course*, pages 51–53, April 1998. 8

[DDSD03] Xavier Décoret, Frédo Durand, François X. Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. *ACM Trans. Graph.*, 22(3):689–696, 2003. 9, 19

[DN04] Philippe Decaudin and Fabrice Neyret. Rendering forest scenes in real-time. In *Rendering Techniques '04 (Eurographics Symposium on Rendering)*, pages 93–102, June 2004. 8, 9, 49

[Gue06] Paul Guerrero. Rendering of Forest Scenes. TECHNISCHE UNIVERSITÄT WIEN, Institut für Computergraphik und Algorithmen, September 2006. 9, 10, 19, 24

[GW07] Markus Giegl and Michael Wimmer. Unpopping: Solving the Image-Space Blend Problem for Smooth Discrete LOD Transitions. *Computer Graphics Forum*, 26(1):46–49, March 2007. 26

[Ism05] Ismaer Garcia, Mateu Sbert and László Szirmay-Kalos. Tree Rendering with Billboard Clouds. University of Girona and Budapest University of Technology, 2005. 9, 19

[Jak00] Aleks Jakulin. Interactive vegetation rendering with slicing and blending. In A. de Sousa and J.C. Torres, editors, *Proc. Eurographics 2000 (Short Presentations)*. Eurographics, August 2000. 9, 10, 19, 38

[OHKK03] Katsuhiko Onishi, Shoichi Hasuike, Yoshifumi Kitamura, and Fumio Kishino. Interactive modeling of trees by using growth simulation. In *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 66–72, New York, NY, USA, 2003. ACM. 8, 15

[Oli03] Olivier Renouard. Lighting flat objects - (an application of normal maps and rayDiffuse). http://www.opengl.org/documentation/red_book/, January 2003. Accessed Nov, 25th 2007. 48

[Opp86] Peter E. Oppenheimer. Real time design and animation of fractal plants and trees. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 55–64, New York, NY, USA, 1986. ACM. 8

[Pel04] Kurt Pelzer. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, chapter Rendering Countless Blades of Waving Grass, pages 107–121. Addison Wesley, 2004. 9

[Per07] Emil Persson. Framebuffer objects. ATI Technologies, Inc., May 2007. 32

[RHK03] Michael Renton, Jim Hanan, and Pekka Kaitaniemi. The inside story: including physiology in structural plant models. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 95–ff, New York, NY, USA, 2003. ACM. 8

[TMW02] Robert F. Tobler, Stefan Maierhofer, and Alexander Wilkie. A multiresolution mesh generation approach for procedural definition of complex geometry. In *Shape Modeling International 2002*, pages 8, 11, 15

[VHB03] William Van Haevre and Philippe Bekaert. A simple but effective algorithm to model the competition of virtual plants for light and space. 11(3):464–471, february 2003. 8

[Wha05] David Whatley. *GPU Gems2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Toward Photorealism in Virtual Botany, pages 7–25. Addison Wesley, 2005. 48

A Impostor billboards

We will go through some of the ideas and problems that occurs when using billboards..

A.1 Single billboards

Impostors represented as single billboards are commonly rendered as camera-oriented billboards. This means that they are always rotated toward the camera and are usually called spherical billboards. However, considering we are rendering trees, this will look very unrealistic as the tree will not be fixed to the ground. To avoid this the rotation of larger billboards are often combined with a static up-axis relative to the ground, which keeps them upright relative to the ground (fig. 33). These are generally called cylindrical billboards.

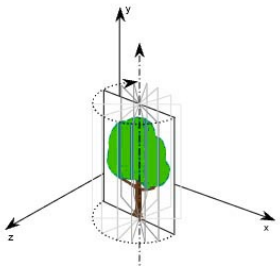


Figure 33: Cylindrical billboard: the billboard will orient toward the camera around a static axis. In this case the y-axis.

There are several drawbacks to cylindrical billboards, and they are especially obvious with larger object close to the camera:

Parallax - Because the billboard will rotate toward the camera as this is moved, there will be no parallax distortion, which for larger object will look obviously unreal.

Lighting - The flatness of larger objects becomes very visible if it is not implemented with a proper lighting scheme applied to the quad, for example by using the normal of the quad. A proper scheme is proposed by [Oli03], who suggests using a customized normal-map of the tree during the lighting process.

Popping - This may occur with tree placed close together and as the camera moves around the two trees, suddenly one tree pops in front of the other due to their rotation.

Blending - For a proper blend of the objects they are required to be rendered back to front, which requires a sorting of the tree. [Wha05] proposes that the trees are rendered with pure aliasing instead and then manually alpha aliased in the fragment shader using a noise texture. This method will leave large objects like trees with visible rigid edges.

A.2 Multiple billboards

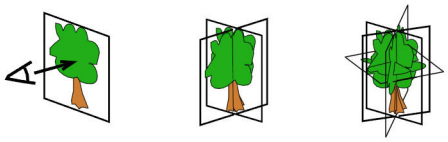


Figure 34: The different ways of using and combining billboards. This picture is from [DN04]

The parallax problem is overcome by combining multiple static billboards in different formations (fig.34), and thereby adding some depth to the impostor. However multiple billboards still experience problems with the obvious flatness of the impostor, and thus this leaves requirements to apply a proper lighting model, or a different scheme.

B Shader Programs

B.1 Vertex program

```
struct vertex_input
{
  float3 pos       : POSITION;
  float3 normal       : NORMAL; // .xy = uv koords, z = z-offset, w = fadedist
  float4 data      : TEXCOORD0; // .xy = uv koords, z = z-offset, w = fadedist
};

struct vertex_output
{
  float4 pos       : POSITION;
  float4 uv        : TEXCOORD0;
  float4 data      : TEXCOORD1;
  float3 tint      : TEXCOORD2;
};

vertex_output main(
     vertex_input     IN
   , uniform float4x4   ModelViewProj
   , uniform float1     grad
   , uniform float3     sun
   , uniform float3     tint
   , uniform float1     foglevel
   )
{
  vertex_output OUT;
  OUT.pos       = mul(ModelViewProj,float4(IN.pos, 1.0));

  // Light intensity
  float intensity = clamp(dot(normalize(sun),IN.normal),0.3,0.8);

  // Trees higher in the terrain gets more light
  float heightbonus = clamp((IN.pos.z)/10, 0.0, 0.5);

  // Fog
  float fog = clamp(length(OUT.pos)/200, 0.0,0.9);

  OUT.uv     = IN.data;
  OUT.uv.zw     = float2(intensity + heightbonus*tint.b, foglevel*fog*tint.b);

  // LOD Fade
  OUT.data.x    = grad*(IN.data.w - length(OUT.pos));

  // Tint = Tint * fog
  OUT.tint.rgb  = (0.3*tint + float3(0.3)) * OUT.uv.w ;



  return OUT;
} // end of main
```

B.2 Fragment program

```
struct vertex_output
{
  float4 pos       : POSITION;
  float4 uv        : TEXCOORD0;
  float4 data      : TEXCOORD1;
  float3 tint      : TEXCOORD2;
};


struct fragment_output
{
   float4 color : COLOR;
};

fragment_output main( vertex_output IN, uniform sampler2D texture, uniform sampler2D noise)
{
   fragment_output OUT;
   float4 tex     = tex2D(texture,IN.uv.xy);

   // Adds light, tint and fog
   OUT.color.rgb = tex.rgb * float3(IN.uv.z) + IN.tint.rgb ;
   OUT.color.a = tex.a * IN.data.x;

   float noisetex = tex2D(noise,IN.uv.xy);

   // With only one tree (no tint and shading)
   //OUT.color.rgba = tex.rgba;

   // When using noise dissolve (optimally this would be applied to texture at tex-gen- time)
   OUT.color.a = tex.a * IN.data.x * noisetex;

   // For enabling 3rd person view of the scene
   // OUT.color.a    = tex.a ;

   //if(IN.uv.x >= 0.995 || IN.uv.x <= 0.005 || IN.uv.y >= 0.995 || IN.uv.y <= 0.005) {
   // OUT.color  = float4(0.2,0.2,0.1,1);
   //}

   return OUT;
}
```

C XML Files

```
<?xml version="1.0" ?>
<LSYSTEM DESC="Fractal Plant" INITAGE="4">

  <VARIABLE CHAR="F" DESC="Draw forward"></VARIABLE>
  <VARIABLE CHAR="X" DESC="Structure"></VARIABLE>
  <VARIABLE CHAR="Y" DESC="Structure"></VARIABLE>
  <VARIABLE CHAR="Z" DESC="Structure"></VARIABLE>

  <CONSTANT CHAR="+" DESC="turn right n degrees"></CONSTANT>
  <CONSTANT CHAR="-" DESC="turn left n degrees"></CONSTANT>
  <CONSTANT CHAR="L" DESC="Mark as leaf"></CONSTANT>

  <INITIATOR START="X"></INITIATOR>

  <PRODUCTION P="X" S="F-[[X]+YF]+F[+FX]-FX" ></PRODUCTION>
  <PRODUCTION P="Y" S="Z" LIKELIHOOD="0.1"></PRODUCTION>
  <PRODUCTION P="Z" S="X" LIKELIHOOD="0.1"></PRODUCTION>
  <PRODUCTION P="Y" S="F" LIKELIHOOD="0.9"></PRODUCTION>

  <PRODUCTION P="p" S="FLFL"></PRODUCTION>

  <COMMAND SYMBOL="F" CMD="F" ARG="0.1"></COMMAND>
  <COMMAND SYMBOL="X" CMD="F" ARG="0.1"></COMMAND>
  <COMMAND SYMBOL="[" CMD="[" ARG=""></COMMAND>
  <COMMAND SYMBOL="]" CMD="]" ARG=""></COMMAND>
  <!--Spherical coordinates in DEGREES-->
  <COMMAND SYMBOL="+" CMD="R" ARG="40.0,50.0,-60.0"></COMMAND>
  <COMMAND SYMBOL="-" CMD="R" ARG="-40.0,-50.0,60.0"></COMMAND>
  <COMMAND SYMBOL="L" CMD="L" ARG="1.0"></COMMAND>

</LSYSTEM>
```

```
<?xml version="1.0" ?>
<!--  -->
<LSYSTEM DESC="Simple tree">
  <VARIABLE CHAR="F" DESC="Draw forward"></VARIABLE>
  <VARIABLE CHAR="B" DESC="Branch"></VARIABLE>

  <CONSTANT CHAR="a" DESC="Spherical rotation"></CONSTANT>
  <CONSTANT CHAR="b" DESC="Spherical rotation"></CONSTANT>
  <CONSTANT CHAR="c" DESC="Spherical rotation"></CONSTANT>
  <CONSTANT CHAR="+" DESC="Spherical rotation"></CONSTANT>
  <CONSTANT CHAR="[" DESC="Start scope"></CONSTANT>
  <CONSTANT CHAR="]" DESC="End scope"></CONSTANT>

  <INITIATOR START="FB"></INITIATOR>
  <PRODUCTION P="B" S="Fa[B]b[B]bFc[B]aF"  DESC="" ></PRODUCTION>
  <PRODUCTION P="a" S="aF"  DESC="" ></PRODUCTION>
  <PRODUCTION P="b" S="bF"  DESC="" ></PRODUCTION>
  <PRODUCTION P="c" S="cF"  DESC="" ></PRODUCTION>


  <COMMAND SYMBOL="F" CMD="F" ARG="0.1"></COMMAND>
```

```
  <COMMAND SYMBOL="[" CMD="[" ARG=""></COMMAND>
  <COMMAND SYMBOL="]" CMD="]" ARG=""></COMMAND>
  <!--Spherical coordinates in DEGREES-->
  <COMMAND SYMBOL="a" CMD="R" ARG="35.0,-30.0,120.0"></COMMAND>
  <COMMAND SYMBOL="b" CMD="R" ARG="-35.0,20.0,240.0"></COMMAND>
  <COMMAND SYMBOL="c" CMD="R" ARG="35.0,45.0,360.0"></COMMAND>

</LSYSTEM>
```

```
<?xml version="1.0" ?>
<!-- Lindenmayer's Algae system) -->
<LSYSTEM DESC="AB stochastic">
  <VARIABLE CHAR="A" DESC=""></VARIABLE>
  <VARIABLE CHAR="B" DESC=""></VARIABLE>

  <INITIATOR START="A"></INITIATOR>
  <PRODUCTION P="A" S="AB" DESC="" LIKELIHOOD=""></PRODUCTION>
  <PRODUCTION P="B" S="A" DESC="" ></PRODUCTION>
</LSYSTEM>
```

```
<?xml version="1.0" ?>
<!-- Stochastic AB (modified algae system) -->
<LSYSTEM DESC="AB stochastic" INITAGE="2">
  <VARIABLE CHAR="A" DESC=""></VARIABLE>
  <VARIABLE CHAR="B" DESC=""></VARIABLE>
  <VARIABLE CHAR="C" DESC=""></VARIABLE>

  <INITIATOR START="A"></INITIATOR>
  <PRODUCTION P="A" S="AB" DESC=""></PRODUCTION>
  <PRODUCTION P="B" S="A" DESC="" LIKELIHOOD="0.1"></PRODUCTION>
  <PRODUCTION P="B" S="C" DESC="" LIKELIHOOD="0.9"></PRODUCTION>
</LSYSTEM>
```