

Charles University in Prague
Faculty of Mathematics and Physics

Diploma thesis



Pavel Celba

Dynamická úroveň detailu

Dynamic Level of Detail

Department of Software Engineering

Supervisor: Bedřich Beneš, Ph.D

Study program: Computer science, Software systems

Acknowledgments

Many thanks to Bedřich Beneš, Ph. D, to Greenworks organic–software for gracefully providing tree creation software and tree library, to my family and everyone else providing me with some support or advice in relation to this work.

I declare that I wrote my diploma thesis independently and exclusively with the use of the cited sources. I agree with lending the thesis.

In Prague 16. 7. 2007

Pavel Celba

Contents

1	Introduction	1
2	Previous work	2
2.1	Level of detail	2
2.1.1	Categories of level of detail	3
2.1.2	Level of detail selection criteria	4
2.1.3	Geometry reduction techniques	5
2.1.4	Non-geometric level of detail	8
2.2	Ecosystem rendering	10
2.2.1	Non real-time ecosystem rendering	10
2.2.2	Real-time ecosystem rendering	12
2.2.3	Real-time grass rendering	23
2.3	Ecosystem motion	25
2.3.1	Tree animation methods	26
3	Problem definition	33
4	Our approach	35
4.1	Introduction and overview	35
4.2	Obtaining data	37
4.2.1	Basic tree representation	37
4.2.2	Modeling and exporting data	38
4.2.3	Exporting utility overview and usage	40
4.2.4	Exporting utility	41
4.2.5	Possible improvements and other usage	43
4.3	Simple tree representation	45
4.3.1	Simple tree representation format	45
4.3.2	Simple tree conversion utility overview and usage	46
4.3.3	Simple tree conversion utility	47
4.3.4	Possible improvements	47
4.4	LOD tree representation	47
4.4.1	LOD tree representation format	47
4.4.2	LOD tree conversion utility overview and usage	49
4.4.3	LOD tree conversion utility	51
4.4.4	Possible improvements	59
4.5	Creating forests	60
4.5.1	Forest representation structure	60
4.5.2	Forest generation utility overview and usage	62

4.5.3	Forest generation utility	64
4.5.4	Possible improvements	65
4.6	Rendering engine	67
4.6.1	Viewer overview and usage	67
4.6.2	Viewer	71
4.6.3	Rendering engine with impostor system	72
4.6.4	Possible improvements	79
4.7	Forest motion	81
5	Results	83
5.1	Measurement conditions	83
5.2	Test suite overview	84
5.3	Results	87
5.4	Interpreting results	111
6	Conclusion and future work	114
6.1	Conclusion	114
6.2	Subjective opinion	115
6.3	Future work	117
7	Image gallery	119
A	Installation	133
B	Catalogue of trees	135
C	Catalogue of forests	153

Abstract

Title: Dynamic Level of Detail

Author: Pavel Celba

Department: Department of Software Engineering

Supervisor: Bedřich Beneš, Ph. D

Supervisor's e-mail address: bbenes[at]purdue.edu

Abstract: The work aims to render real-time forest with lighting. Detailed tree models created by professional application are used. A point-based method for level of detail is used for tree model simplification and rendering. The rendering of forest is based on hierarchical view-dependent pseudo-continuous level of detail. The frustum culling is used for scene visibility testing. Novelty hierarchical impostor system is presented to speed up rendering. Two forest lighting models are used. First is directional diffuse lighting to simulate sun light and the second is ambient only lighting model. The work includes implementation of rendering system using shader programs running on graphic card. Utilities are provided for obtaining data from XFrog tree modeling utility, for converting tree to format for viewing, for creating level of detail representation of trees and placing trees into the forest.

Keywords: Real-time, Forest, Render, Level of detail, Tree

Problem assignment

Implement dynamic level of detail for rendering of lit moving ecosystems. The task is to render animated forest with changing lighting in real-time. The work requires to program pixel and vertex shaders.

- In first phase generate 3D models of trees. The best will be to use Maya.
- From models create dynamic level of detail. All will be working as pre-processing and result will be validated and time-balanced levels of detail. Different levels of detail will depend on concrete graphic card on which the preprocessing will run. Use and discuss existing methods of billboarding, directional billboards, point and line-based rendering, study new methods and implement them.
- Arrange generated tree levels of detail into ecosystems and properly measure and discuss how the rendering speed is changing. Apply 2D BSP for scene partitioning and frustum culling of invisible objects.
- On partially visible trees will be applied hierarchical level of detail on the level of branches. Invisible parts of trees will not be rendered.
- Apply global level of detail techniques. Impostors, and billboards and discuss how they are affecting quality and rendering speed of scene. All (for example the position of impostor) will be again dependent on concrete graphic card.
- Animate individual trees. The wind will be generated by Gauss fractal noise generator. Individual trees will react on physically based model of motion. Do not take into account collision detection.
- Consider and try to implement ecosystem lighting. Simplified shadows, precomputed billboards and intensity of light falling onto leaves and branches.

Chapter 1

Introduction

The real-time forest rendering is interesting and challenging field of research. The structure of forest is complex one consisting of hundred thousands of plants or more when counting smaller plants. Only a single tree can have ten thousand of leaves and several thousand of branches in case of adult tree. This presents a difficult area to master with only limited resources available on nowadays computers to mimic forest or to create virtual forest in real-time.

It's recently that outdoor scenes appeared in games such as FarCry or The elder scrolls IV: Oblivion in decent quality. Many people devoted years of research to invent methods and algorithms for forest and tree simplification in order to bring commercially available forest rendering solution to the developers and therefore to the final users.

The area of research is large. One can study various tree types structures and growing habits, tree simplification methods, forest creation algorithms, special rendering engines often requiring sophisticated spatial data structures to speed up rendering, forest lighting, forest motion by wind or user action, and finally the most difficult task is to put everything together. As some research papers may look promising, it is often showed that implementing them for forest rendering solution is unbearable because of various reasons and that a lot of important implementation details must be often invented to provide decent visual quality of rendered forest in most cases.

By this work we hope to provide deep insight into the topic and introduce some new ideas in real-time forest rendering to be considered and added to the state of the art knowledge.

The outline of this work is now overviewed. We present overview of previous work done in this area in chapter 2. We start with overview of level of detail techniques continuing through image-based and point-based methods for forest rendering and we finally discuss grass rendering and forest motion. Then exact problem to solve is defined in chapter 3 based on previous work done. We present our approach to forest rendering in chapter 4 followed by results achieved in next chapter 5. Then our conclusion is stated in chapter 6. Finally an image gallery presents visual achievements of our forest rendering solution in chapter 7. Our work is complemented by three appendixes. The first one with installation notes for accompanied DVD disc, the second is catalogue of trees and the third is catalogue of forests.

Chapter 2

Previous work

A lot of work has been done in computer graphics generally and in ecosystem rendering specifically. In this chapter an overview of work done is reported. Although it's impossible to discuss everything which was done even in as small part of computer graphic as is real-time ecosystem rendering, an effort to provide nice overview of currently used techniques is the aim of this chapter. First a sketch of optimization techniques used in rendering is given in section 2.1 to familiarize a reader with basics. Then ecosystem rendering is discussed in section 2.2 including off-line and interactive rendering in subsection 2.2.1, real-time rendering in subsection 2.2.2 and grass rendering in subsection 2.2.3. Finally animation of ecosystems is reviewed in section 2.3.

2.1 Level of detail

Nowadays graphic cards performance and increasing need for real-time rendering of more and more complex scenes requires sophisticated optimization techniques to be applied to the scene before it's send to the graphic card and finally rendered. One of these optimization techniques is called level of detail. The idea of level of detail is based on simple assumption that objects, which are further from the viewer of the scene¹, will not be seen by the viewer in such detail as objects which are near the viewer of the scene. Therefore complexity, detail of objects far from the viewer can be significantly reduced before they are rendered by graphic card.

Level of detail is meant in relation to computer graphics as reduction of geometry details of objects depending on the reduction criteria usually distance of object from the viewer of the scene. Although there are other usages of level of detail, for example level of detail for textures. There are several ways how one can achieve reduction of object's geometry and how one can choose geometrically reduced object to render. There are three main categories or frameworks of level of detail techniques as stated in *Level of Detail for 3D graphics book*[15] and with one additional category stated in *A Review on Level of Detail*[12]. These categories are:

Discrete level of detail Several reduced geometry models are created for object. Then the model of object for rendering is selected according to

¹Viewer of the scene is the one who is looking at the scene

particular selection criteria at run-time.

Continuous level of detail Geometry reduction is continuous for object according to reduction criteria.

View-dependent level of detail Reduction of object's geometry depends not only on reduction criteria, but also on viewing parameters such as direction from which is object viewed.

Hierarchical level of detail Like View-dependent level of detail, but additionally objects can be merged together to achieve further reduction.

We will now discuss these categories in more detail. Then we will follow by discussion on criteria or metrics by which is particular model selected to be rendered at run-time and techniques used for geometry reduction. Finally we will discuss non-geometric level of detail.

2.1.1 Categories of level of detail

Discrete level of detail Discrete level of detail is the simplest of level of detail techniques used for achieving geometry reduction of the scene. Several geometry reduced models of objects are created in preprocess either manually or automatically. At run-time geometry model to be rendered is chosen according to particular selection criteria. This selection criteria is usually for real-time graphic applications distance of the object or approximation of the size of the object. Discrete level of detail is also the fastest and easiest to implement of level of detail techniques. Because of that it's most widely used technique nowadays and popular among game developers. The disadvantage is that object introduces ugly popping effect when geometry model changes. In practice some hysteresis is therefore added to selection criteria in order not to introduce popping effect too often when being near the edge of selection between two geometry models. Another approach to reduce popping effect is to alpha-blend models when change of geometry happens. This is done so that the transparency of current geometry model changes from maximal to minimal value while the transparency of final model increases from minimal to maximal value over some short period of time. But this alpha-blending level of detail method requires to render both models as it's obvious from explanation above.

Continuous level of detail Continuous level of detail gets rid of discrete geometry model changes popping just introducing smooth reduction of model geometry depending on reduction criteria for example approximated size of object. It also renders only geometry which is really needed, opposing discrete level of detail where only several discrete levels were selected. Disadvantage of this technique is that models don't always look good and object may appear to constantly change which may be distracting. It's also slightly less efficient to use Continuous level of detail than Discrete level of detail on current graphic hardware.

View-dependent level of detail Specially, when an object is large, it will be very good to reduce detail on the far side of the object, but leave more detail in the near side of the object to further reduce scene complexity. This

introduces View-dependent level of detail technique taking into account from which direction is object actually viewed. This allows to concentrate more geometry complexity on parts of object which need it and further reduce parts of objects far away or on which attention isn't focused. Also silhouette of object can be rendered with more detail and inside of object with less detail since our eyes are more sensitive on borders. Since it's more advanced technique, it takes much more time and memory resources than previous techniques.

Hierarchical level of detail Hierarchical level of detail introduces further concept by allowing multiple objects to be united to one object. It's very good for scenes with a lot of small objects because uniting this objects can reduce scene complexity greatly. This is by far most complex technique of level of detail and thus takes most computer resources and it is its disadvantage. The technique's advantage is in great flexibility it provides.

2.1.2 Level of detail selection criteria

It's important for level of detail to know how much to reduce objects in the scene. For that purpose various criteria, metrics are introduced which allows the control of object reduction amount. We will discuss these metrics one by one:

Distance Distance is simplest and easiest metric to choose as criteria. Usually for discrete level of level a set of distances is chosen where change of object's geometry occurs. For continuous level of detail an amount of object's geometry reduction can be computed based on object distance and projection parameters of scene's camera. However distance metric doesn't take into account the size of the object on the screen. This leads to situations that same level of detail is used for both big and small objects at some fixed distance. Thus either oversampling small object because big object takes larger area on the screen and must be shown at better level of detail, or undersampling big object because not enough detail is shown on it.

Size Object size metric is used to overcome limitations of distance metric. Object size metric computes or more often approximates object's pixel size on the screen and according to this pixel size information computes desired level of detail for object. In practice object's size is approximated by some type of object's bounding envelope most commonly bounding box or bounding sphere.

Advanced metrics More sophisticated metrics were invented according to human eye system and behaviour research. It can be taken into account that human eye sees best objects on which is focused. It allows to use worse level of detail ² on objects on which isn't human eye focused. Simply center of the screen can be supposed as an area on which human will mostly focus. This allows to reduce detail on the borders of the screen. More complex evaluation of human attention can be also taken into account. There are several classes of objects which attracts attention. First type is moving object. For very fast

²Worse level of detail means less geometry used on the object. This is for clarity of discussion.

moving objects level of detail can be reduced significantly because it's hard for human to see it's real detailed shape especially when the object is small. But for slow or moderate speed moving objects it's little bit more complicated because such object usually attracts a lot of attention. So it's good to set better level of detail or when speed of such object is sufficient to motion blur object allowing reduction of detail. It has also been shown that object with significantly different color than rest of the objects attract a lot of attention. For example red flower blossoms on the green grass field. It's better to leave better level of detail for such objects. In practice evaluation of this requires special importance information to be associated with the object or to apply some image color analysis technique. Environmental conditions such as haze, fog or dim lighting can help in further detail reduction of objects in the scene.

2.1.3 Geometry reduction techniques

Geometry reduction techniques are divided into two types:

Geometry simplification techniques Geometry simplification techniques reduce number of primitives, but leaves important object's topological properties intact.

Topology simplification techniques Topology simplification techniques reduce number of primitives, which can incorporate change of object's topological properties such as number of holes.

It's also important to somehow measure error created by object's geometry reduction. It introduces the need to provide metric to measure such geometry reduction error. This metric can serve as criteria for selecting which part or primitive of the object will be reduced. Usual practice is to compute error for all object's reduction options and then use some kind of greedy algorithm to reduce geometry for reduction options with minimal error. We will now discuss various reduction options or operators and then discussion about metrics will follow.

Geometry simplification techniques There are several options for geometry simplification by which geometry can be reduced (simplified). Attention must be made for sharp edges of object because removing them (or part of them) can introduce big change in object's geometry. On the other hand removing or merging nearly coplanar primitives generally introduces small object's geometry changes and thus this parts of object are candidates to be removed. Simplification options or operators are these:

Edge collapse When edge collapses two edge vertices are united to one vertex. When the united vertex is on the position of one of collapsed edge's vertices, the operation is called half-edge collapse. Edge collapse involves typically two decisions. First what edge on object to collapse and second where to place new united vertex. It's very popular geometry simplification technique because it doesn't involve filling hole in geometry. But it can introduce undesired fold over in object's geometry.

Vertex removal Vertex removal removes vertex from object's geometry just creating hole which must be filled then. Usually geometry is represented

by set of triangles in real-time computer graphics so in that case hole is triangulated. One of such triangulation is similar to half-edge collapse so vertex removal can be threatened as generalization of half-edge collapse. Caution must be taken to avoid folding. Edge swap operation can help to prevent folding.

Triangle collapse Triangle collapse chooses triangle and collapses it to a single vertex. But it has little practical usage.

Primitive clustering Object's primitives are clustered together to one big primitive or to lesser number of primitives simplifying geometry. This is very good for nearly coplanar primitives in the object.

Topology simplification techniques

Vertex clustering Vertex clustering technique divides object by 3D uniform grid and then merges all vertices in single grid cell to one. So all triangles or primitives within single cell are destroyed and single vertex is connected to neighbouring cells to which previous cell geometry was connected. Technique can be improved by using non-uniform grid.

Vertex pair contraction Vertex pair contraction is an extension, more general version of edge collapse, and allows to pick two different vertices in the object regardless of vertices being neighbouring, or not and contract them to one vertex. Because of generality of vertex pair contraction second vertex is chosen locally only from some fixed diameter around first vertex. This is very good for closing holes and gaps in the object.

Geometry reduction metrics There are various approaches to measure error of geometry reduced object against original object. There are not only geometry-based metrics which measure geometry error usually by computing distance between vertexes, planes, or surfaces, but also image-based metrics measuring pixel differences and moreover attribute-based metrics which measures such things like vertex normal change or change of object's texture coordinates.

Geometry-based metrics **Vertex to vertex distance** The easiest thing is to simply compute vertex to vertex distance. This metric can be especially used with edge or half-edge collapse.

Vertex to plane distance Minimal distance from vertex to plane is computed.

Vertex to surface distance Either minimal distance from vertex to surface is computed or surface is sampled by points and minimal distance from vertex to these sampled points on the surface is taken.

Surface to surface distance This is the most general case. There is little bit problem with saying what is surface to surface distance. Typical approach to compute all distances between all points and choose the minimum is not optimal because it doesn't account for local surface deviations. Much more optimal is to use Hausdorff distance which takes maximum of minimal point distances. Hausdorff

distance from first surface, call it A, to second surface B is computed exactly so that for each sampling point of surface A minimal distance to sampling points of surface B is found and then maximum of these minimal distances is taken as it's shown on equation 2.1.

$$h(A, B) = \max_{a \in A} \min_{b \in B} \|a - b\| \quad (2.1)$$

But because Hausdorff distance from surface A to surface B can differ from Hausdorff distance from surface B to surface A, maximum of these two values is taken. This is called symmetrical Hausdorff distance which is much more optimal to use. Symmetrical Hausdorff distance is shown on equation 2.2

$$H(A, B) = \max(h(a, b), h(b, a)) \quad (2.2)$$

It's also possible to compute maximal distance of all points which takes excellent account of local surface deviations, but ignores global surface shape.

Image-based metrics Image-based metrics function in a way that some set of fixed cameras, scene viewers, is chosen. Usually from 20 to 30 cameras is enough. Then simplification step is done on the object. Both original and simplified object is rendered from every camera and finally for each pair of images rendered some image difference metric is used. According to [12] most well-known metric for comparing images is the Lp pixel-wise norm and d2 root mean square error.

Attribute-based metrics Object's geometry representation often includes attributes such as vertex normals, vertex colors, texture coordinates or animation indices. These attributes are in nowadays real-time computer graphics stored in vertices. By reducing geometry reduction of attribute information happens. Metrics should therefore include evaluation of attribute changes.

Corners Special attention while reducing geometry must be paid for the corners of object because their reduction often changes object's geometry too much. It's good idea to carry corner attribute within object's geometry representation and penalize or disallow removal of such parts of geometry.

Colors Difference between colors in RGB space can be computed by equation 2.3.

$$D(C_1, C_2) = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2} \quad (2.3)$$

The problem with RGB color difference metric is that RGB space isn't perceptually linear. For better results it's better to convert color information into some more perceptually linear space like CIE-Luv³.

³It's three component model, where L component defines luminance and u and v components defines chrominancy.

Normals The distance between two 3D normals can be computed by angular distance metric shown on equation 2.4.

$$D(N_1, N_2) = \arccos((x_1, y_1, z_1) \cdot (x_2, y_2, z_2)) \quad (2.4)$$

Normals can be used for detecting fold-overs. It's also possible to optimize geometry reduction by minimizing normal metric error.

Texture coordinates Representation of texture coordinates is usually (u, v) , where u, v are from range $[0, 1]$. By reducing geometry fold-over in texture space may occur. This should be penalized, disallowed or completely avoided if possible.

2.1.4 Non-geometric level of detail

The concept of level of detail isn't bounded only by geometry reduction. There is possibility to use various imaging tricks to improve performance. In the todays world of computer real-time graphic final scene is highly dependent on rendering hardware – graphic card. Attempts to optimize rendering performance of graphic card can be made with level of detail techniques. We will first introduce image space level of detail like impostors and point–sprite rendering. And then we will focus on graphic card performance optimization using level of detail techniques.

Image space level of detail Image space level of detail generally replaces geometrically complex objects with their image representation.

Impostors Impostor is preprocessed or off–line created image of object rendered in scene. Often object is modeled in some 3D modeling editor such as 3D Studio MAX or Maya and then off–line rendered to a picture. Alternatively a picture of some object can be directly taken from for example photography and altered for application needs. In application run–time such picture, impostor is rendered as textured quad⁴. When the impostor's quad is always facing toward scene viewer then it's called billboard. Billboards are very common technique used in computer games for such effects like trees in distant, light coronas and so on. An object can be rendered from multiple sides or angles, typically around vertical axis, to achieve more 3D realism. In such case appropriate image from rendered object's image collection is chosen at run–time according to camera viewing direction and object's position. Smooth transition between various images can be done by texture blending⁵. Especially when an object has complex geometry such as tree leaves or fences, impostors can greatly increase performance.

Point sprites Point sprite is 3D point of selected size and texture. Because rendering of sets of point sprites is hardware supported, it's very fast rendering method. Sets of Point sprites are typically used to simulate environmental effects like show or rain, flame and smoke effects or falling

⁴Quad is simply rectangle in 3D space consisting of two triangle rendering primitives.

⁵Texture blending means combining two textures together. In most cases using alpha–blending.

leaves for example. Anything, which is small and there is a lot of quantity of it, can be simulated by point sprites. Sometimes it may be more efficient to render whole object as set of point sprites than rendering its geometrical representation. It occurs for example when object rendered takes only few pixels⁶ on the final rendered image.

Render to texture support Nowadays graphic cards offer render to texture capability. This introduces option to create impostors at application run-time from object's geometry.

Hardware dependent level of detail For nowadays graphic cards pixel throughput and vertex transformation pipeline performance is most important along with number of pixels written per second into final image. Various hardware dependent optimizations can be made to minimize graphic card workload. Some of these optimizations related to level of detail are now discussed.

Texture level of detail For distant objects having only few pixels on screen after rendering, it's wasteful to texture them using big textures. Even more alerting is the fact that big texture area is used for only one pixel that means to average whole texture area to obtain correct pixel texture color. For performance reasons averaging isn't done properly and this introduces distracting popping of such pixel. Because of this texture level of detail was added to hardware support. This is called mip-mapping. For every texture, that is mip-mapped, a chain of smaller textures is created. These smaller textures are then used for texturing when rendering object in distance.

Shader level of detail Graphic cards used in real-time graphics are often programmable. Program for graphic card is called shader. By applying level of detail on complexity of shaders one can save a lot of rendering time. It's also unpractical to use complex shaders for objects in distance.

Lighting level of detail To light scene properly is one of the most performance drawing tasks. Less computational and visually correct lighting model can be used to compute distant object's lighting. Also reducing number of light sources per object helps greatly. Typically maximal number of lights per object is determined by programmer of lighting system and only nearest light sources is taken into account with lighting model up to maximum number of light sources allowed. More complex approach can be used just by determining light sources importance for given object and prioritizing light sources with bigger importance for such object. Moreover light sources can be clustered together and replaced by single light source to further reduce rendering cost.

Shadow level of detail Rendering believable shadows for scene is still difficult task to cope with current graphic hardware. Again for distant objects no shadows can be rendered while for near objects shadows of better quality can be rendered. There are also methods which intrinsically supports level of detail for shadows just introducing continuous level of detail to this problem area.

⁶Pixel is basic unit of image. Image is consisting of pixels.

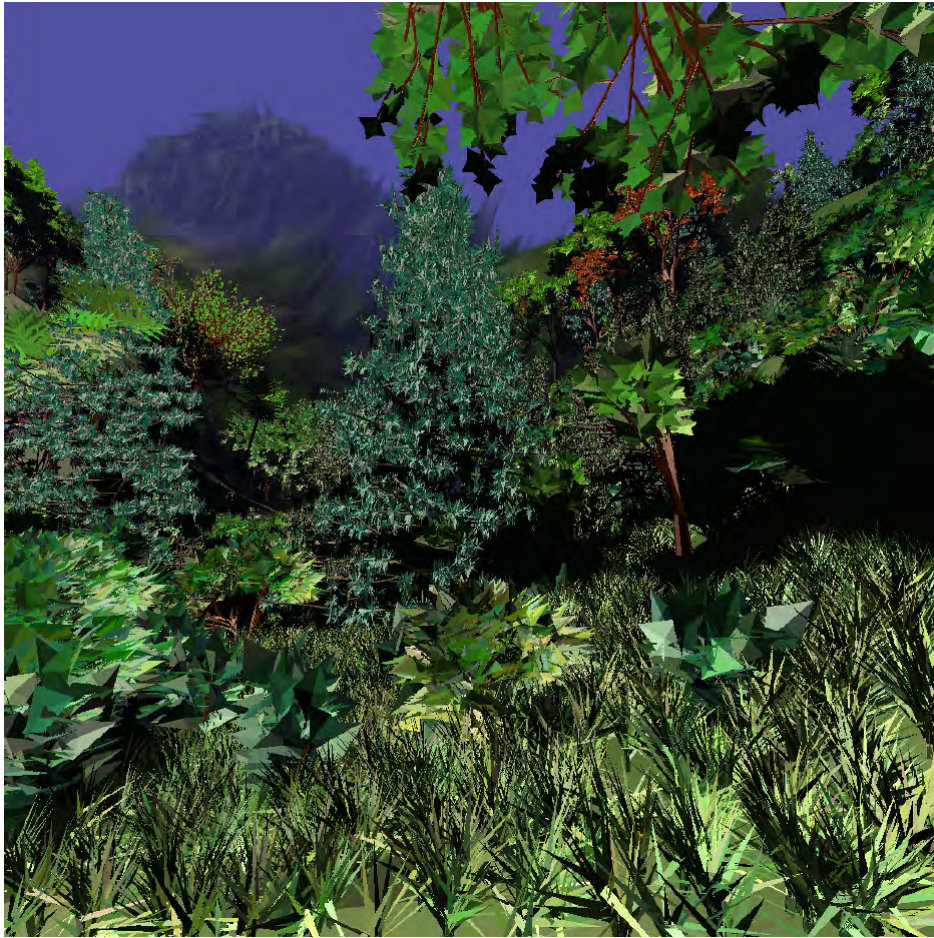


Figure 2.1: An image showing result of rendering complex botanical scene from *Multiresolution Rendering of Complex Botanical scenes* [16]

7

2.2 Ecosystem rendering

2.2.1 Non real-time ecosystem rendering

Although this work focuses on real-time ecosystem rendering, it's not completely worthless to briefly look at few slower methods of rendering ecosystems. They can introduce some concepts which possibly can be adapted by real-time ecosystem rendering solution. We will discuss two of these works: *Multiresolution Rendering of Complex Botanical scenes* [16] and *Realistic and Interactive Visualisation of High-Density Plant Ecosystems* [8].

In *Multiresolution Rendering of Complex Botanical scenes* [16] a method is

⁷Material as reference to write level of detail section is Level of detail for 3D graphics book [15], article Review on level of detail [12] and Real-time rendering second edition book [2].



Figure 2.2: An overview of ray traced scene with 365 000 plant instances from *Realistic and Interactive Visualisation of High-Density Plant Ecosystems* [8]

introduced to render off-line botanical description of plants of as large scenes as 100 million primitives. For plant models only procedural information is stored which describes each part of plant sufficiently to create it's polygonal representation when needed. Tetrahedral subdivision is used for scene partitioning. This tetrahedral subdivision can be traversed in similar way to BSP tree⁸. Initially viewing pyramid is subdivided by two tetrahedrons and then refined according to resulting image resolution and scene complexity. Space outside viewing pyramid is also included into tetrahedral subdivision mainly for lighting and shadowing reasons. Scene refinement is done in loop of three parts. First visibility determination is done by low resolution image of tetrahedron rendered front to back in order to get rough estimate of total number of pixels each tetrahedron's geometry will occupy in final scene. Then tetrahedron refinement step divides tetrahedrons bigger than some amount. Finally update step distributes contents of previously subdivided tetrahedrons to its children.

After final update step rendering is done front to back with previously collected shadowing information from light sources. Although method is not fast – 21 minutes of refinement and modeling and 6 minutes for rendering of view from 78 million polygon scene, method eliminates 100% hidden polygons keeping memory demands only as big as is size of polygons of currently rendered scene from certain view.

Realistic and Interactive Visualisation of High-Density Plant Ecosystems [8] uses ray-tracing rendering on clusters of PCs in order to allow interactive walk-through in high-density plant ecosystems. L-system generated models of 68 plants by Xfrog application are used. Then functional specification of 365

⁸BSP tree is binary space partitioning tree which divides space by using arbitrary planes.



Figure 2.3: A close look on ray traced scene from *Realistic and Interactive Visualisation of High-Density Plant Ecosystems* [8]

000 plant instances is created by special paint program. This specification is converted to tree positions only when it's needed because of high amount of plants. Scene is divided into several layers of geometry depending on the type of plant for example trees or shrub. OpenRT ray-tracing engine is used to provide interactive ray-tracing rendering support. Two-level spatial kd-tree⁹ is used to subdivide scene. One level is used for subdividing plants and other level is used for subdividing scene using plant bounding boxes. Plant leaves are represented by coarse triangle meshes with associated texture with alpha channel. Because of high scene transparency ray can hit up to 30 transparent surfaces before non-transparent surface is hit, recursion depth isn't used as ray termination criteria, but instead of it distance of ray from viewer is used. Realistic lighting is simulated by adding directional lights to the scene. For ray only small number of directional lights is used to light the scene which is sufficient to approximate realistic lighting. Progressive refinement is applied so that, when viewer is standing at place, image quality is improved. According to article interactive performance around 1 FPS can be achieved with clusters of PCs while at single PC an image is rendered in about 30 seconds.

2.2.2 Real-time ecosystem rendering

To achieve real-time ecosystem rendering large scale of methods have been researched ranging from geometric based methods, image based methods to point sprites rendering. We will now introduce various recent methods and practices

⁹Kd-tree is specific type of BSP-tree where division planes are parallel to coordinate system axes.

to achieve real-time performance as stated in selection of articles contributing to this topic.

Article *Procedural Multiresolution for Plant and Tree rendering* [14] presents method for procedural multiresolution level of detail based on parametric L-system representing the tree structure. A set of axioms and derivation rules is given representing general tree structure. Axioms and rules are defined for example for branch length, thickness, position and orientation. Leaves are not addressed in this paper. Concrete tree is then constructed by giving initial value of axiom and then applying derivation rules to some chosen extent. The result is output chain representing concrete tree. For set of axioms and derivation rules listed on 2.5

$$\begin{aligned} \text{Axiom} &: A(\text{length}) \\ \text{Rule1} &: A(l) : \text{itNum} < \text{maxIt} \rightarrow B(l)[A(l/2), A(l/2)] \\ \text{Rule2} &: A(l) : \text{itNum} == \text{maxIt} \rightarrow B(l) \end{aligned} \quad (2.5)$$

can be produced for example this output chain shown on 2.6.

$$B(1)[B(.5)[B(.25)B(.25)]B(.5)] \quad (2.6)$$

Generated output chain is interpreted on the basis of turtle metaphor. Turtle is having its position, rotation and associated stack for storing turtle's position and rotation attributes to allow branching of the tree. An idea is to create multiresolution output chain based on the turtle interpretation from tree's output chain which preserves visually important parts of the tree for lower quality levels of detail. Branching length is used as part of importance length because it reflects density of descendants of a branch. Branching length can be computed as seen on equation 2.7 where $l(n)$ is length of the branch n and $d(n)$ is set of descendants of branch n .

$$L(n) = l(n) + \sum_{m \in d(n)} l(m) \quad (2.7)$$

Weighted tree is built to contain metric information and to resemble branch tree structure. Multiresolution output chain is created then by extracting path from weighted tree. Multiresolution output chain is consisting of branch information in turtle like style, SAVE and RESTORE instructions in order to be able to recover the turtle state of previously visited branching. First lowest level of detail is created by traversing tree from root to one of its leaves. At each step node with the largest branching is chosen and branch information is output to the chain. The next level of detail is generated so that next largest branching node, call it A, is chosen (not previously output to multiresolution output chain), RESTORE(A) instruction is appended to multiresolution output chain and similarly path from this node A to leaves is added to multiresolution output chain. For nodes, which are restored, SAVE instruction must be added to multiresolution output chain. Individual levels of detail can then be interpreted so that nth level of detail is multiresolution output chain from beginning to the nth RESTORE instruction. The first level of detail is the lowest. For example for tree on 2.6 with branch naming 2.8 following multiresolution output chain is produced, see 2.9.

$$A[B[CD]E] \quad (2.8)$$

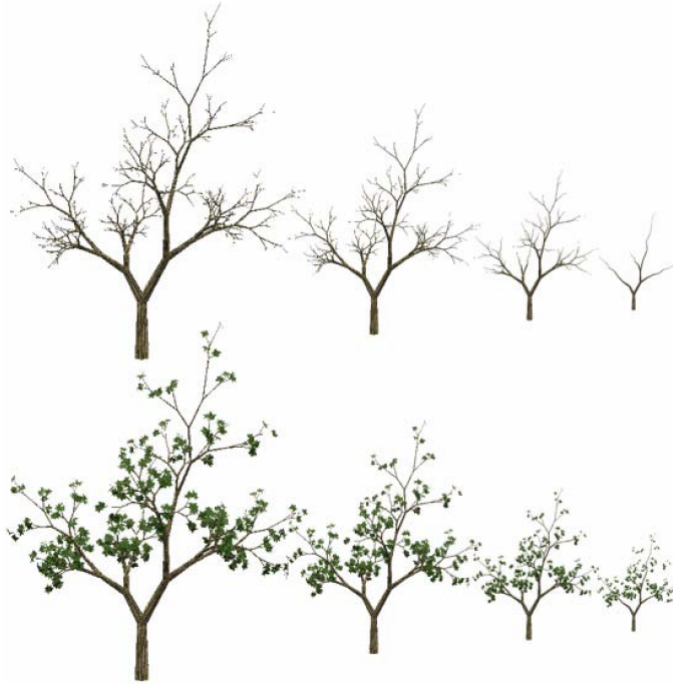


Figure 2.4: An example of branch reduction based on importance metric from *Procedural Multiresolution for Plant and Tree rendering* [14]

$$A \text{ SAVE}(A) \ B \ \text{SAVE}(B) \ C \ \text{RESTORE}(A) \ E \ \text{RESTORE}(B) \ D \quad (2.9)$$

The resulting multiresolution output chain represents levels of detail for tree in way that preserves perceptually important parts of tree and allows fast extraction of individual level of detail for further rendering.

An image-based approach is chosen in *An Image-Based Multiresolution Model for Interactive Foliage Rendering* [13]. An algorithm for precomputing textures for tree leaves based on the bounding boxes is presented. From given tree L-system definition data structure with appropriate bounding boxes is constructed using turtle metaphor. For every new node of tree push is used to save turtle state and pop is used to retrieve turtle state from stack. Every branches and leaves of tree have some geometrical representation called form. Turtle can move forward and rotate. Tree representation is given as parametrized string consisting of pop, push, form, forward and rotate. Bounding boxes are generated as follow:

While processing parts of parametrized string defining tree structure in case module is:

push Create a new box
 Give initial values to min, max
 Mark the box as last and open
 Increment level

pop Close the last box



Figure 2.5: An orchard rendered by image method explained in *An Image-Based Multiresolution Model for Interactive Foliage Rendering* [13]

Save the box

form Update min and max of all open boxes

forward Modify the turtle position

rotate Modify the turtle orientation

To generate textures from bounding box an orthographic projection is used from each side of box where camera is viewing from center of that box side to the center of the bounding box. All leaves which are inside the bounding box are then rendered to texture and stored for further use. Texture size is chosen for all textures to be 128x128 pixels. Number of levels of detail is reduced by eliminating bounding boxes which have ratio of current bounding box to its parent bounding box smaller than some chosen threshold. Rendering of leaves is then done by selecting level of detail according to bounding box projected area size. Six mapped textures are rendered on three quads instead of complex leaves geometry. The lowest level of detail are textures for tree root bounding box. Distances for finer levels of detail are precomputed in such way that projected area of such level of detail bounding boxes will be roughly 128x128 pixels. To improve look of result six diagonally aligned textures can be rendered instead of bounding boxes side textures. Results indicate that significant rendering performance is achieved allowing to render hundredths of trees in real-time without losing realism when looking close at tree.

Real-time Hardware Accelerated Rendering of Forests at Human Scale [19] uses another image-based method based on 2.5 impostors. An arbitrary group of leaves arranged randomly around the center is rendered to texture with its depth information stored in texture's alpha channel. Tree leaves are then rendered

using vertex shader¹⁰ and pixel shader¹¹ program. Vertex shader program is used to calculate correct depth offset for rendered texture. Pixel shader program is used to write correct depth value to the depth buffer. Using this technique improves realism of scene because leaves are correctly intermerging with branches of the tree. Even when using single 2.5D impostor for whole tree image quality and realism is increased dramatically in comparison with standard billboards or impostors. Rendering of whole forest showed that some sort of visibility test is needed. For that reason inclosing spheres have been implemented for frustum culling. It turned out that writing scene depth in pixel shader program is bottleneck because whole pixel shader program often run for nothing hence computed pixel color is then discarded by depth test. Therefore an idea to use alpha-blending instead of depth write for 2.5D impostors on far trees were invented. Although this alpha-blending impostors showed some artificial popping, it showed in practice that this is hardly noticed in resulting scene. For close-up views high detailed geometry of tree branches and trunk is used while for distant trees small branches are simply omitted from scene. Final forest rendering takes four steps. First geometry is sorted because alpha-blending impostors requires back-to-front rendering. In second step impostors are generated into single large texture¹² capable of holding 16x16 impostors. In third phase all "other" geometry is rendered. Finally 2.5D impostors are rendered in front-to-back order and alpha-blending impostors are rendered in back-to-front order. Test results showed that, after first 15 to 20 trees rendered in high quality, rest of trees can be rendered in low quality using alpha-blending impostors. It has been shown that moderate sized forest consisting of 1024 trees can be rendered in real-time performance on ATI Radeon 9700Pro graphic card. The technique is capable of rendering high quality forest scenes without visible popping for first person simulation and low altitude flights.

While previous image-based methods mainly focuses on rendering forests for first person simulation or low altitude flight, *Rendering Forest Scenes in Real-Time* [6] introduces rendering technique capable of rendering forests in much larger scale of ten thousands of trees for both low and high altitude flights. This approach is based on volumetric textures. Because of problem with volumetric filtering on graphic hardware, slices are stored in 2D textures instead of 3D volumetric textures. Volumetric data are created with off-line renderer from standard polygonal representation of part of forest. Camera is orthographic projection looking down on the part of the forest. Slices are used parallel to terrain. Slices are stored in alpha premultiplied textures. Alpha premultiplication is necessary in order to be able to use slices level of detail. For standard textures (not alpha premultiplied) color slightly changes when number of slices changes. For each slice texture associated MIP-map chain is created. Slices level of detail is created so that finest level of detail contains $n = 2^N$ slices. Each lower level of detail then contains half of slices then previous finer level of detail. Each lower level of detail slice is created by blending two slices of finer level of detail into one slice. Slicing quality can be tuned during rendering according to distance from the viewer. Terrain elevation, viewing angle and height of the forest part must be accounted because it changes distance between individual slices and so

¹⁰Vertex shader program is geometry transformation program running on graphic card.

¹¹Pixel shader program is per pixel (or fragment) computation program running on graphic card.

¹²Commonly called texture atlas in computer graphics.



Figure 2.6: A forest rendered using 2.5 impostors technique from *Real-time Hardware Accelerated Rendering of Forests at Human Scale* [19]

opacity must be multiplied with opacity multiplication factor at each vertex to correct it. Silhouette texcells¹³ are used for situations where viewing angle of scene is so that there is a possibility of viewing through slices. Because slices are rendered from top and used in way parallel to terrain, this happens when camera is viewing part of forest in way near parallel to terrain. The idea is to transform polygons previously stored on GPU in order to avoid creating new geometry on CPU and to minimize data transfer. Every triangle is tilted so that approximately faces toward the viewer. Additionally filtering for all slices is chosen in way that MIP-map filtering level is as close to the voxel size as possible. Aperiodic tiling is used to avoid repeating when mapping parts of forest to the terrain. Three directions triangular tiling is used with two different boundary conditions in each direction. It gives 16 possibilities – 8 for triangles pointing north and 8 for triangles pointing south. Only 8 triangles were chosen to save memory. An aperiodic reference texture is created from triangles with boundaries in a way that triangles can share edge only when edge have same boundary type. Reference texture is used to arrange parts of the forest (represented by triangles on texture) in aperiodic manner. Types of forest parts must be created according to triangles with boundaries and boundary condition must be leaved intact so that pixels are same on the edge of forest parts with same boundary type. This is achieved by replicating trees which cross boundary of the part of the forest on all parts of the forest with same boundary type. It's also possible to create more than one part of forest per "tiling triangle" as long as boundary conditions are left intact. Results show that not repeating forest of size as large as 37000 trees can be rendered on GeForceFX 5800 graphic card with real-time performance achieved and no popping even on limited implementation authors of article implemented.

Similar method as in *Rendering Forest Scenes in Real-Time* [6] is used in *Realistic real-time rendering of landscapes using billboard clouds* [5] for rendering

¹³Authors of article consider using word texcell rather than using word texel because of possible confusion with texture pixel - also called texel.



Figure 2.7: A screenshot from fly above forest rendered using texcells and aperiodic tiling using technique from *Rendering Forest Scenes in Real-Time* [6]

distant trees enhanced by rendering near trees with billboard clouds. Clusters are generated using hierarchical information from tree model. Branches and leaves are distinguished by texture and ratio of transparent and opaque texels analysis. User can specify branching level so that all sub-branches bigger than that level are included into one cluster. Minimal volume oriented bounding box is calculated for each cluster using approximation algorithm that rotates given initial bounding box by one degree steps in order to find better one. After the bounding box is found special representation is taken. If the bounding box has two long sides and one short, it's represented by single billboard. If the bounding box has only one long side, it's represented by two crossed billboards. If all the sides are roughly same, the bounding box is represented by three crossed billboards. If $\sqrt{a \cdot b} > f \cdot c$ then billboard is created parallel to a and b. Authors found that $f = 0.5$ produces good results. Rendering can be done in several ways. The simplest way is to precompute lighting into billboard textures. When also storing normal information in additional normal map texture, lighting can be computed at run-time. Third variant is to approximate reflexion function using spherical harmonics basis functions. Fade between simple two crossed billboards and complex billboard cloud representation is used in relation to distance of tree from scene viewer. In order to avoid back-to-front rendering the alpha channel of billboard texture is premultiplied with gauss curve. During run-time pixel shader implements simple test that discards pixel with alpha value smaller than some threshold. The threshold is computed as linear function related to distance from the viewer. When test is passed alpha value is simply set to full opaque. A technique similar to one presented in *Rendering Forest Scenes in Real-Time* [6] is used to further accelerate rendering of distant forest. Shadows for scene were implemented using shadow mapping. Results show that forest of

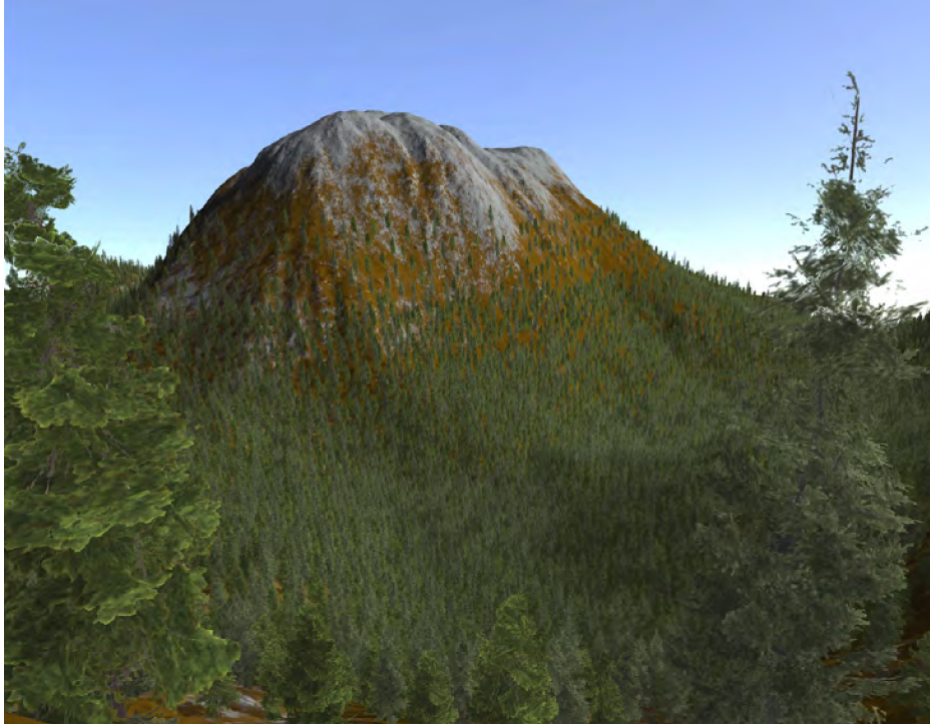


Figure 2.8: Forest rendered using billboard clouds at close distance and shell textures at far distance from *Rendering Forest Scenes in Real-Time* [6]

size 21 300 trees were rendered using NVidia FX6800 graphic card in 4 to 12 FPS.

Point-based rendering is another approach to render ecosystem researched more recently than geometric or image-based approaches. We will discuss article *Point-based rendering of trees* [11] presenting typical point-based rendering approach for tree rendering. Representation of scene consist of blocks of vegetation prepared in some standard modeling software. An "unorganized soup" of polygons describing block of vegetation serves as an input for preprocessing step. Polygons are converted to triangles and then to points. Each block is then subdivided by regular grid ranging from 4 x 4 x 4 for a single tree to 4 x 16 x 16 for group of trees. For each cell hierarchical clustering is defined. The finest level of detail forms triangular representation of cell's geometry itself. Lower levels of detail are based on point representation. Cell is subdivided by computing eigenvectors of covariance matrix 2.10 of cluster point set call it P_i using binary space partitioning.

$$C = \begin{bmatrix} P_{i_1} - \bar{P} \\ \dots \\ P_{i_j} - \bar{P} \\ \dots \\ P_{i_n} - \bar{P} \end{bmatrix}^T \cdot \begin{bmatrix} P_{i_1} - \bar{P} \\ \dots \\ P_{i_j} - \bar{P} \\ \dots \\ P_{i_n} - \bar{P} \end{bmatrix}, i_j \in P \text{ and } \bar{P} \text{ is centroid} \quad (2.10)$$

This always splits point cloud along the direction of greatest variation. Subdi-

vision process is iterated until threshold of 5 points is reached. Coarser point representation for each non-leaf node of subdivision is then computed by averaging position of it's left and right children nodes. Radius of new point R is computed as average radius taking into account radius of all children points R_l and R_r , their relative orientation $\alpha = |N_l N_r|$ and normalization factor $\frac{2}{\pi}$ averaging all the possible view directions giving equation 2.11.

$$R = \frac{2}{\pi}(R_m + \alpha(R_M - R_m)), \text{ where} \quad (2.11)$$

$$R_M = R_l + R_r \text{ and } R_m = \frac{1}{2}(R_l + R_r)$$

Because hardware doesn't allow to render hierarchical tree structures efficiently, all points are then sorted according to their size to single array. The same process is done with cell's triangle geometry. This allows to select point and triangle size according to pixel size they will take on resulting image. Rendering of cells depends on choosing two thresholds. One threshold is for determining whether to render points and when to render triangles. When projected size of primitive point is smaller than threshold S_M its rendered, otherwise triangular representation is rendered. Second threshold S_m is chosen for defining minimum size of point projected area to render. When point projected area is smaller than S_m , bigger sized point must be rendered. Precision of the cell rendering is further adjusted according to viewing position and block position to take advantage of masking. Distance d_1 between camera position and center of block is computed. Then distance d_2 between camera position and center of cell is computed. Z_i ranging $[-1, 1]$ is computed by equation 2.12.

$$Z_i = \frac{(d_2 - d_1)}{R}, \text{ where } R \text{ is block size} \quad (2.12)$$

Thresholds S_M and S_m can then be adjusted by following equations 2.13.

$$\begin{aligned} S_m^i &= S_m + k \cdot Z_i \\ S_M^i &= S_M + k \cdot Z_i \end{aligned} \quad (2.13)$$

Where k is user adjustable constant. Shading is done by hardware for both points and triangles using stored normal information. Results show that for block with 300 000 points storage of approximately 26 MB is needed. A landscape with 200 000 trees were rendered using this method on NVidia GeForceFX 5800 with framerate varying from 3 to 10 FPS. Authors suggest that for forests with less than 100 000 trees real-time performance is achieved. This methods allows very good continuous fine tuning of level of detail.

Even more hybrid method is used by *Interactive Visualisation of Complex Plant Ecosystems* [7] using both point and line-based technique to achieve real-time performance. Plants are generated from surface oriented plant descriptions produced by plant modelers. In pre-processing step models are converted to point and line representation which is stored along with polygonal representation in data file for each plant. Eco system files are used to describe positioning of the plants in the scene. Eco system files are hierarchically organized so that one eco system file can reference several other eco system files. For each eco system file a sparse point representation is computed which is used to render geometry at far distances. Bounding boxes are used for visibility culling. Point



Figure 2.9: Hills with trees rendered using point-based approach from *Point-based rendering of trees* [11]

representation is typically used for leaves. For an object with n triangles $2n$ points are generated and distributed randomly in such way that equally sized surfaces receive equal number of points. Different point generation technique is used when an object has all triangles of about the same area. In this case 2 points are created for each triangle. Such list of points is then randomly reordered, but triangle-points correspondence is sustained in order to be able to merge triangles and points rendering. For long things like stems of straw is generated line representation. Line representation is generated during modelling phase where all informations about objects are known. Generated line set is also randomly reordered and stored along with polygonal representation into file. Important parts of geometry are marked during plant modeling on which geometry is reduced more slowly during scene rendering than rest of the scene. It helps to preserve better image quality and hide reduction of non-important parts. Object which color is clearly distinguishable from majority of other plant's color is usually marked important. For example blossoms of daisies are important. In run-time rendering of points and lines are handled separately. The number of points that is needed to render plant faithfully depends on the surface area A_p and its average distance r from camera. The approximate projected area A'_p is computed by equation 2.14.

$$A'_p = \frac{1}{2} \frac{A_p}{r^2} \quad (2.14)$$

Factor $\frac{1}{2}$ is accounted for geometry orientation and $\frac{1}{r^2}$ corresponds to perspective shortening. The number of points required to render plant is determined by equation 2.15,

$$p = c_p \frac{A'_p}{A'_{sp}} n_p \quad (2.15)$$



Figure 2.10: Complex plant ecosystem rendered using point-based approach from *Interactive Visualisation of Complex Plant Ecosystems* [7]

where n_p is number of generated points for plant, c_p is point scaling factor used for closing holes in rendered point geometry and A'_{sp} is point splatting area which can be user controlled by defining average distance between two neighbouring point samples d' , $A'_{sp} = d'^2$. Polygonal representation is rendered if $p \geq n_p$, otherwise the prefix of p generated points for plant is rendered. For line rendering projected area A'_l representing corresponding triangle associated with lines is computed by equation 2.16.

$$A'_l = \frac{1}{2} \frac{A_l}{r} \quad (2.16)$$

The image plane area covered by lines with line width d' is computed by equation 2.17

$$A'_{sl} = \frac{ld'}{2r}, \text{ where } l \text{ is length of all lines in world space} \quad (2.17)$$

Ratio is $q_l = \frac{A'_l}{A'_{sl}}$. If $q_l > 1$ then triangular representation is drawn otherwise $q_l \cdot n_l$ first lines from lines list is drawn. Triangle and point representation is blended in such way that for k triangles and p points following rendering method is used. For $p < 2k$ first p points is rendered. For $p > 3k$ only triangle representation is rendered. And for $2k \leq p \leq 3k$ first $p - 2k$ triangles are rendered and then points corresponding for remaining triangles are rendered.

While small plants are grouped together using eco system files, large plants are subdivided by an octree of small depth. Shadow computation is done either by obtaining standard shadow maps in preprocessing step or using advanced shadowing technique called perspective shadow mapping. Results show that on NVidia GeForce3 graphic card scene with 13 000 sunflowers and large tree with 70 million triangles can maintain framerate 3 – 4 FPS. For scene representing small palms mainly by lines and having 12 million polygons performance ranges from 5 to 10 FPS.

2.2.3 Real-time grass rendering

Rendering grass introduces similar problems as forest or shrub ecosystem rendering. However typical grass is much smaller than a tree or a shrub. This allows to use different optimization techniques.

In *Real Time Animated Grass* [3] texture shells approach is used along with vertex position displacement to animate grass. Shell textures contains grass color and height of the grass encoded in alpha channel. When alpha channel value is zero, ground color is stored in texture instead of grass color. Several layers of grass is rendered displacing vertex positions of terrain along vertex normals. Alpha testing is used to render only appropriately high grass blades. For every vertex wind vector is projected along vertex normal in order to get wind vector perpendicular to vertex normal. This is precomputed in preprocessing step using equation 2.18,

$$\vec{V}_w = \vec{W} - (\vec{W} \cdot \vec{N}) \cdot \vec{N} \quad (2.18)$$

where \vec{V}_w is wind vector perpendicular to vertex normal, W is wind vector and N is vertex normal. Every vertex is moved along vertex normal and perpendicular wind vector. The amount by which is vertex moved along normal is determined by equation 2.19,

$$\sum_{i=0}^N \cos\left(\frac{i \cdot I \cdot \pi \cdot S}{2 \cdot N}\right) \quad (2.19)$$

where i is the current shell or layer number, I is wind intensity ranging $[-1, 1]$, S is inter-shell distance and N is total number of shells or layers. The amount by which is vertex moved along perpendicular wind vector is given by equation 2.20.

$$\sum_{i=0}^N \sin\left(\frac{i \cdot I \cdot \pi \cdot S}{2 \cdot N}\right) \quad (2.20)$$

An animation frame is used to get grass blades closer to viewer effect of sudden spike in wind intensity before distant blades. This allows more realistic wind simulation across land. Simulating winding "waves" or attenuation of wind is possible by using animation frame. Results show that with NVidia GeForce3 graphic card real-time performance of 23 FPS can be achieved using 8 texture layers of grass with 1024 x 768 resolution. The method has advantage that the number of grass blades used is independent on performance.

Another solution is used in *Animating Prairies in Real-Time* [18]. Three levels of detail are used for grass. 3D grass blades geometry is used close to viewer. Then 2.5D level of detail is used consisting of layers of vertical polygon strips covered with semi-transparent texture. There are two perpendicular



Figure 2.11: An example animated grass using texture shell approach from *Real Time Animated Grass* [3]

orientations for each polygon strip. One that isn't parallel to view direction is chosen. A 2D texture is used as lowest quality level of detail. Grass on terrain is divided into square elements to support various levels of detail. Level of detail for given square element is selected according to number of square elements between camera and given square element. Range of distances is chosen for each level of detail. Whole scene is rendered back to front because transparent textures are used, but no sorting takes place because it's enough to render square elements in right order. A concept of receivers is introduced to allow grass to be animated and thus convincingly simulate the wind blowing over the grass. At each frame receivers associated with grass receive information from wind primitives. For 3D geometry receivers are associated to 3D blades of grass and for 2.5D texture receivers are associated to the vertical edges. For 2D textures receiver isn't associated because it isn't animated. The receiver contains two information for every wind primitive affecting given receiver – direction in which grass is bent and amount of how much is bent ranging from -1 to 1. The largest bend is of 90 degrees. In practice each grass primitive has precomputed bent postures for fixed number of bend amounts. When rendering nearest precomputed bent posture is selected according to bend amount. To provide fine control over various wind primitives on grass field a 2D mask is provided that represents extent of each wind primitive action on grass field. An action procedure is defined for each wind primitive which sends information to appropriate receivers, using 2D mask, according to given time. This enables animation of grass field. Authors are saying that they successfully implemented gust of wind, whirlwind, blast of air and gentle breeze wind primitives. They also suggest to add some random oscillation of bent direction and bend amount around prescribed value to achieve more realism. For transition between 3D and 2.5D level of detail morphing is used. For every grass blade on 2.5D patch of grass a relative texture coordinates of several blade positions are stored. Then, when morphing, 3D blade geometry is forced to move to stored positions. By this way the transition between 2.5D and 3D representation is smooth. For

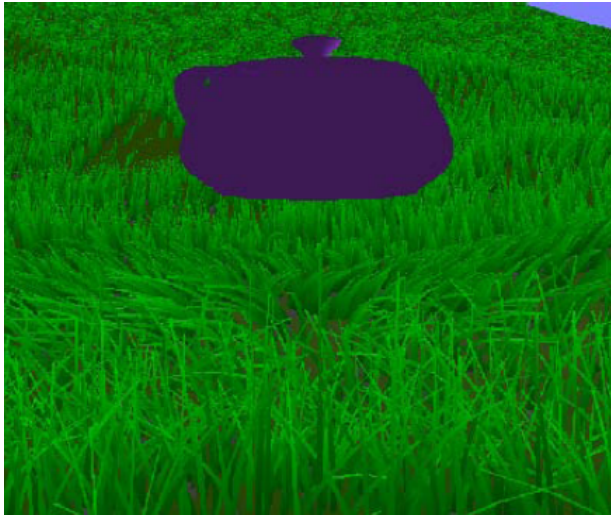


Figure 2.12: Blast of air wind effect on grass from *Animating Prairies in Real-Time* [18]

making transition between 2.5D and 2D level of detail two tricks are applied. Rather than texturing the terrain an offset of terrain located as average distance between tip and top of the grass is textured. Transition is then made by making 2.5D grass patches progressively grow or shrink to match 2D textured offsetted terrain. Results show that terrain of 100 x 100 meters with approximately 1 000 000 blades per image can be rendered on ONYX 2 infinite reality computer with 8 FPS. This is slow computer compared to nowadays processing power and so speed should be more than enough for real-time rendering on current graphic hardware.

2.3 Ecosystem motion

A motion is very important factor which adds a lot of realism to the whole scene. However it's not easy to achieve real-time ecosystem motion on current hardware due to complexity of scene. Even worse is the fact that not all ecosystem rendering methods described above in section 2.2 allows ecosystem motion to be implemented. Especially image-based rendering methods generally prohibits to implement motion of tree leaves and also greatly limits motion of tree branches. Geometrically-based rendering methods are best suited to ecosystem motion, but are often to slow to render forests with thousands of trees in real-time. This leaves point-based rendering methods and hybrid approaches as best candidates for both real-time ecosystem rendering and motion.

But in this section we will look rather at methods researched to bring trees in motion then coping with problem how to bring motion and rendering together. We will discuss one work looking at this problem at the end of this section.

2.3.1 Tree animation methods

A hybrid method of procedural and physically-based animation of tree branches is described by *An interactive forest* [10]. The tree model used consists of nodes defining topology and meshes defining geometry. Nodes defining geometry defines length and angle of branches (h, θ, ϕ) in hierarchical way. A wind primitive is defined by area of influence ($disk(C, r)$), force vector F and pulsation ω . For procedural animation each branch of tree in area of influence undergoes the force f defined by equation 2.21,

$$f = \frac{r-d}{r} \sin(\omega t) F \quad (2.21)$$

where d is 2D distance to the branch from center of influence area of wind primitive. A wind force creates a torque at base of the branch approximated by equation 2.22,

$$\tau(t) = Lz \times f(t) \quad (2.22)$$

where L is length of the branch and z its axis. Torque is then projected to rotation axis to obtain a new rotations θ and ϕ given by equation 2.23,

$$\begin{aligned} \theta &= \theta_0 + \frac{1}{m} \tau(t) z_{parent} \\ \phi &= \phi_0 + \frac{1}{m} \tau(t) y \end{aligned} \quad (2.23)$$

where z_{parent} and y are axes of rotation of the branch with respect to its parent and m is approximation of the inertia of the branch. For more wind primitives torque is simply summed just giving equations 2.24

$$\begin{aligned} \theta &= \theta_0 + \frac{1}{m} z_{parent} \sum_i \tau_i(t) \\ \phi &= \phi_0 + \frac{1}{m} y \sum_i \tau_i(t) \end{aligned} \quad (2.24)$$

For physically-based animation wind action is modeled as real force using equation 2.25.

$$\tau_{physical} = -\omega^2 \frac{r-d}{r} \sin(\omega t) Lz \times F \quad (2.25)$$

The torque is modeled by linear damped angular springs. The stiffness k is estimated as $\frac{d^2}{t}$ and damping v is proportional to stiffness. The torque generated by joints is then computed as in 2.26,

$$(k\theta + v\dot{\theta})z_p + (k\phi + v\dot{\phi})y \quad (2.26)$$

where $\dot{\theta}$ and $\dot{\phi}$ denotes derivation of θ and ϕ respectively. The torque applied to given branch is then computed as sum of wind primitive actions and joint forces by following equation 2.27.

$$\begin{aligned} \tau &= - \sum_{i \in winds} \omega_i^2 \tau_i \\ &\quad - (k\theta + v\dot{\theta})z_p - (k\phi + v\dot{\phi})y \\ &\quad + \sum_{j \in children} (k_j \theta_j + v_j \dot{\theta}_j)z - (k_j \phi_j + v_j \dot{\phi}_j)y_j \end{aligned} \quad (2.27)$$



Figure 2.13: Example of wind primitive affecting a tree from *An interactive forest* [10]

Angular joints accelerations are computed by equations 2.28,

$$\begin{aligned}\dot{\Omega}_{rel} &= \frac{1}{m}\tau - \dot{\Omega}_p \\ \ddot{\theta} &= z_p \dot{\Omega}_{rel} \\ \ddot{\phi} &= y \dot{\Omega}_{rel}\end{aligned}\tag{2.28}$$

where m is proportional to ld^2 and $\dot{\Omega}_{rel}$ is derivation of parent's angular velocity. Time integration is then performed using standard Euler method 2.29.

$$\begin{aligned}\dot{\theta}(t + dt) &= \dot{\theta}(t) + \ddot{\theta}(t)dt \\ \dot{\phi}(t + dt) &= \dot{\phi}(t) + \ddot{\phi}(t)dt\end{aligned}\tag{2.29}$$

A hybrid animation is produced so that bottom branches of the tree are animated physically, intermediate branches of the tree are animated procedurally and top branches of the tree are fixed (no animation). For making transition between procedural and physically-based animation both representations are computed and linear interpolation between representation takes place over some time. Results show that animating 256 trees forest is possible on Pentium III 800 MHz with 512 MB RAM and NVidia GeForce256 graphic card in real-time.

A method for animating both leaves and branches using $1/f^\beta$ noise is discussed in *A hybrid method for real-time animation of trees swaying in wind fields* [17]. The $1/f^\beta$ noise is defined in article [17] as follows:

$1/f^\beta$ noise The spectral density $S(f)$ of a function $X(t)$ is the mean square of the Fourier transform of X where t means time and f means frequency.

$1/f^\beta$ noise is a noise of which the spectral density is proportional to $1/f^\beta$. The value β determines the correlation between noise values varying along time axis t . As β decreases, the fluctuation of the noise increases. On the other hand, as β increases, the fluctuation decreases.

$1/f^\beta$ noise is generated by Fourier filtering. Discrete $1/f^\beta$ noise is generated and then values are interpolated to obtain smooth $1/f^\beta$ noise. Such generated $1/f^\beta$ noise is periodic. The discrete noise values are normalized and then stored in one-dimensional array. A leaf motion consist of horizontal, vertical and rotational motion around leaf's petiole. The leaf has some initial state rotation and position in world coordinates and then horizontal and vertical motion angles are added to it's position. The horizontal and vertical motion is determined by equation 2.30,

$$\begin{aligned}\theta_x(t) &= W_x N_x(t) \\ \theta_y(t) &= W_y N_y(t)\end{aligned}\tag{2.30}$$

where W_x and W_y are maximum motion angles, N_x and N_y are $1/f^\beta$ noise functions. Various effects of leaves motion can be achieved by changing β value, period and phase in noise functions N_x and N_y . For computing rotation angle around petiole's axis similar equation is used, see 2.31.

$$\theta_{Rf}(t) = W_r N_r(t)\tag{2.31}$$

In order to achieve more realistic motion a helix angle is added according to following equation 2.32,

$$\theta_{RX} = a\theta_x(t)\tag{2.32}$$

where a is user definable. The total rotation angle is therefore given as 2.33

$$\theta_R(t) = \theta_{Rf}(t) + \theta_{RX}(t)\tag{2.33}$$

Only one $1/f^\beta$ noise is used for all leaves. Variation of period and phase is used instead to attain realism. For hierarchical branch model a simulation based on spring model is used. A branch is approximated as squared timber and the deflection range is determined from load given to the branch. The loads are determined by following equations 2.34,

$$\begin{aligned}P_X(t) &= F_X(t) + P_B N_X(t) \\ P_Y(t) &= F_Y(t) + P_B N_Y(t)\end{aligned}\tag{2.34}$$

where F_X and F_Y are directional loads usually from wind or user action, P_B is the maximum non-directional load, $N_X(t)$ and $N_Y(t)$ are $1/f^\beta$ noise functions. The deflection ranges $\delta_X(t)$ and $\delta_Y(t)$ are determined by following equation 2.35,

$$\begin{aligned}\delta_X(t) &= \frac{P_X(t)}{k} \\ \delta_Y(t) &= \frac{P_Y(t)}{k}\end{aligned}\tag{2.35}$$

where k is the spring constant of the timber computed by equation 2.36,

$$k = \frac{Ebt^3}{4l^3}\tag{2.36}$$

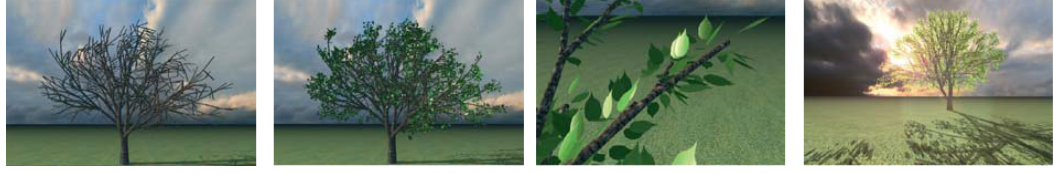


Figure 2.14: Serie of pictures showing results from *A hybrid method for real-time animation of trees swaying in wind fields* [17]

where E is elastic modulus specific to each tree specie, b is width, t is thickness of branch and l is span length of timber. Motion angles are then obtained by following equation 2.37,

$$\begin{aligned}\theta_X(t) &= \arcsin\left(\frac{\delta_X(t)}{L}\right) \\ \theta_Y(t) &= \arcsin\left(\frac{\delta_Y(t)}{L}\right)\end{aligned}\quad (2.37)$$

where L is length of branch. The motion angles of individual branches are accumulated from root toward children. In order to properly accentuate overall motion of the tree a noise modulation of W_x , W_y , W_r and P_B is used. The noise modulation is defined by two equations 2.38,

$$\begin{aligned}W'(t) &= (1 - m)W + mWN(t) \\ P'_B(t) &= (1 - m)P_B + mP_BN(t)\end{aligned}\quad (2.38)$$

where m is modulation factor ranging from 0 to 1. Results show that realistic animation of a tree can be achieved by tuning few parameters manually.

More recent method is described in *Real-time visualisation of animated trees* [20]. Trees are built from stems and leaves. Stems are branches and tree trunk. Creation of tree starts from tree trunk and then children branches and leaves follow in hierarchical way. Random variations of branches and leaves properties are allowed to support wider variety of tree instances. A health property is introduced to support creating of less healthy trees. Rendering of trees are done using line and point primitives where possible. For close view branches are rendered geometrically while leaves are rendered by single quad to reduce rendering overhead. Animation incorporates swaying of branches and fluttering of leaves. Swaying is implemented as periodical rotational motion around the origin of the trunk. Fluttering done by quickly rotating around the spine of the leaf. The animated position of vertex swaying is found by following equation 2.39,

$$pos_{vertex} = (x, r \cos \alpha, r \sin \alpha)^t \quad (2.39)$$

where r is polar distance to vertex from base of the trunk and α is computed by equation 2.40

$$\alpha = \theta + r \cdot sway\ amount \cdot \sin(frequency \cdot time) \quad (2.40)$$

and θ being polar angle from base of the trunk to vertex. Authors further state, that it was empirically shown, that swaying frequency can be computed from tree height using equation 2.41,

$$frequency = A \cdot height^B \quad (2.41)$$



Figure 2.15: Benchmarking scene of 1210 trees from *Real-time visualisation of animated trees* [20]

where $A = 2.55$ and $B = -0.59$. For leaves fluttering final vertex position is found as in equation 2.42

$$pos_{vertex} = pos_{leaf} + local_x \cdot \cos \beta + local_y \sin \beta \quad (2.42)$$

and $\beta = max\ flutter \cdot \sin(global\ offset + local\ offset)$. Six different classes of vertices is defined for use with static, swaying and fluttering mode – textured and untextured to get number of six. This can be packed in vertex data with some effort and optimization in order to compute animation solely by graphic hardware. For effective rendering and level of detail automation, geometry is arranged into two vertex buffers, one for stems and other for leaves, in advance while connection between leaves and branches is left intact. By rendering only prefix of these vertex buffers one can achieve lower levels of detail. Stems are textured by photographs taken from real trees and lit using precomputed lighting information stored in vertex buffers. Self-shadowing is approximated by reducing lighting intensity of vertices near the center of the tree. A cheap trick is used to simulate reduction of lighting in densely populated areas of forest by rendering barely visible black polygon across bound of each tree. Results show that 139 trees forest can be rendered at 32 FPS at resolution 1600 x 1200 on ATI Radeon 9800 XT graphic card. Result also show that adding more trees further back into the scene doesn't influence rendering rate greatly.

A complete motion framework is outlined in *Simulation Levels of Detail for Plant Motion* [4]. Plants are described using L-system grammar with stochastic turtle interpretation where several parameters can be randomly scaled. The basic information described by the grammar is length of branches, orientation of child branches, strength of joints and whether a branch has a leaf attached to the end and the area of that leaf. Plant motion is done through applying external force. The model used is angular spring model. At each joint two springs are defined, first oriented in *parent direction* \times *child direction* and second oriented in *parent direction*. External force is applied to the leaves and

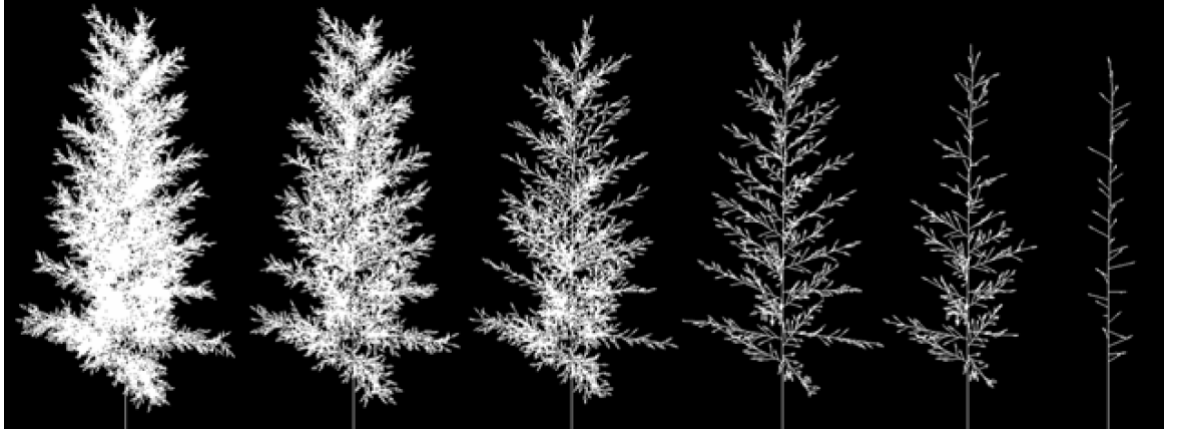


Figure 2.16: Simulation level of detail of a tree from *Simulation Levels of Detail for Plant Motion* [4]

propagated down to the branches and trunk. Euler integration scheme is used to update positions at each step. Level of detail called simulation level of detail is generated to simplify computation of motion animation. Two operations are used for computation simplification. First operator simplifies parent with one terminal child by combining them to single simplified branch. Second operator simplifies parent with more than one terminal children by combining all children into one child branch. All simplifications are done so that new structure approximate non-simplified version within some error bound. A propagation factor for each simplified branch is stored in order to account propagated bend strength correctly according to simplification when computing animation. A lookup table indexed by leaf area and bend strength is generated in order to be able to look for similar amplitude and frequency characteristics for given pair of branches. This lookup table is used during preprocessing for creating simulation level of detail. Maximum strength of wind force is applied during simplification calculations. An error metric chosen is one involving world space distance between tips of pair of branches and their maximal amplitudes. This metric has an advantage of being directly related to screen space error. Parent-child error in amplitude is straightforwardly computed from lookup table. Child-child amplitude error is computed by equation 2.43,

$$Children_{error} = 2 \cdot MO - MS \quad (2.43)$$

where MO is maximum amplitude of children branches and MS is maximum amplitude of simplified branch. Screen space error is specified as maximum error in pixels ahead of time – few steps of computation. An error calculated is stored as the distance to camera, in which it is acceptable, and used as run-time error bounds. In order to secure smooth transition between simulation levels of detail an interpolation must take place when transiting from higher level of detail to lower level of detail. In case of parent-child simplification linear interpolation between key properties is sufficient, but in case of child-child simplification linear interpolation must occur between both computation models, simplified one and normal one. Simulation level of detail can be to high degree independent on

geometric level of detail. Mapping can be easily established between simulation level of detail and geometric level of detail with cost of additional data storage. Results show that animation cost can be significantly reduced using simulation level of detail while specified error bound is kept.

Chapter 3

Problem definition

The image-based techniques do not provide enough detail for close look-ups. It's widely known that billboarding doesn't provide enough visual quality for small viewing distances. From our review the *Realistic real-time rendering of landscapes using billboard clouds* [5] uses most advanced billboarding technique of billboard clouds for near trees, but realistically looking leaves are not achieved for close look-ups. With 2.5 impostors used in *Real-time Hardware Accelerated Rendering of Forests at Human Scale* [19] it's little bit better because of depth information stored in alpha channel, but still quality for very close look-up is miserable.

Moreover a lot of attention was devoted to research image-based techniques for forest rendering and the knowledge in this area is large. The more mysterious, unexplored area of forest rendering lies within point-based techniques. There are still many questions unanswered and only few implementations exist. We therefore focus our attention on point-based techniques as a candidate for future use in real-time forest rendering.

Our aim is to implement hybrid forest rendering solution usable for both forest walkthroughs and flights above the forest in real-time or at least interactive rendering speed! We will focus on mixed triangle and point rendering technique similar as in *Point-based rendering of trees* [11] which we view as promising approach. We believe that such approach will lead to detailed close look-ups as well as for good tree detail in distance. Moreover by having similar implementation as in [11] we can potentially discuss and compare our method with authors of [11] to pinpoint most problematic parts and to confirm or deny their results independently.

Our primary aim is to render middle-sized forest of few thousand trees, but we will also try more demanding big forest rendering of ten thousands or even hundredth thousands of trees. It is difficult to say what to name a "forest"? One must ask how dense and how big must be the group of trees to name it forest. The question of forest size is from our viewpoint easier. We think that trees on area of 1 square kilometer is more than enough to name it forest although we are aware of fact that some Russian living on taiga will disagree with us completely saying that it's only very very small forest. The question of forest density is more demanding especially when one realizes that it's very difficult to achieve real-time rendering of trees with density of real forest. We consider that 20 to 30 trees per 10000 square meters is enough to at least resemble forest.

The resulting look will depend on tree size and look. If trees are small, much bigger density is needed to provide realistically looking forest.

We will first make forest rendering solution and then, if there is enough time left, we'll possibly try to add some procedural animation by wind to our forest rendering solution.

By this our problem is sufficiently defined and task to work on is set forth.

Chapter 4

Our approach

4.1 Introduction and overview

Our approach aims to render middle-sized forest of few thousand trees in real-time or at least in interactive framerates. We have chosen mixed triangle and point based method in order to extend further research in this area of forest rendering and to evaluate usefulness of this rendering method. While point based method is somewhat slower than nowadays leading approaches for forest rendering using sprite-based methods such as SpeedTree, we believe, that for future use, point based method will be candidate to choose as soon as graphics hardware becomes fast enough.

The usage of mixed triangle and point representation is promising in sense of quality it can provide for closer look on tree's leaves and branches while also providing decent detail of trees further in the scene. Moreover level of detail can be adjusted by quality settings either for high detail slow rendering or lower quality fast rendering, which introduces nice flexibility of whole approach.

By providing an implementation of our forest rendering method we are also able to outline pros and cons of our approach and to pinpoint most problematic parts of it. In many cases we are able to outline the direction of potential improvements to make our approach better and more ready to use for potential practical applications.

Nevertheless we are putting here some words on an approach of how an implementation was developed because this puts some light on structure of whole approach and further improves understanding of why some choices has been made so and not the other way. We have partly applied practices of extreme programming in making of our implementation. This implies that the structure of whole application resembles an iterative process of improving application and its functionality rather than having one big analysis of whole project done at the beginning and implementing project along with the analysis provided. This approach has several advantages. First it decreases the risk of whole project failure because in iterative process of implementation individual steps are solved separately and for each step a basic functionality is assured before next step in development is taken. This also leaves open choice whether to proceed with next step and so improve functionality of whole solution or to work more on perfection of current step. This is ideal for research development where there

is often necessary to make choices to improve certain part, or even postpone selection of next step in development with respect to performance and other issues which may arise. We are also counting with fact that it's often extremely difficult to divide code well into classes or provide useful set of classes methods before some implementation has been made. On the contrary it's mostly not so difficult to refactor or recode parts of code which call for improved functionality.

In the sense of implementation scheme selected we first created a way to obtain tree data to our basic tree representation used as source tree representation for whole implementation. Then we developed utility to convert this tree data to simple viewing representation. And by creating simple viewer application we were able to see that data are correctly converted to our representation and also we have a viewer of our tree data. Only after then we proceeded with creation of an utility for tree level of detail and improved our simple viewer to provide viewing capability of single tree with level of detail. We followed by creating an utility for positioning trees into a forest and again updated our viewer application to provide viewing capability for forests. Then we further improved our viewer application and our utilities.

On the contrary implementation using previously made analysis might look differently. For example steps of creating independent basic tree file representation and simple viewer representation may be completely omitted. Although these steps doesn't relate to diploma thesis problem assignment in any way, we are sure that they provide added value and much more flexibility to the whole solution. For these reasons we have chosen the iterative approach for our implementation.

In this chapter we'll describe our approach in detail. First we'll describe data structure of our basic tree representation. Basic tree file representation is representation of tree with all information necessary to create further tree data representations used for rendering and viewing. Then we'll explain a way of how we obtained a tree data along with textures and other needed things. This includes process of creation and preparation of data followed by process of converting data into our basic tree representation. This is described in section 4.2.

In next section 4.3 we'll describe the structure of our simple tree representation data structure which is just simple representation of tree used for fast rendering on current graphics hardware. No level of detail is applied to simple tree representation. This simple tree representation can then be viewed by viewer application explained in section 4.6. Then we'll introduce an utility for creation of simple tree representation from our basic tree representation data structure and explain how it works.

In section 4.4 we'll explain our single tree data structure format with level of detail. We'll call it LOD tree representation. We then follow by describing an utility for creation of LOD tree representation from basic tree representation. This LOD tree representation can be viewed by viewer application.

The section 4.5 introduces our forest representation along with utility for creating forests from LOD tree representation, forest definition text files and tree definition text files. This forest representation can be viewed by viewer application.

Next section 4.6 explains functionality of viewer application capable of viewing simple tree representation, LOD tree representation and forest representation. Then more in-depth explanation of various parts of viewing application

takes place.

In sections 4.3, 4.4, 4.5 and 4.6 a subsection for possible improvements is included where various future improvements are discussed.

Last section 4.7 of this chapter discusses possible future improvement of adding forest motion system into implementation. Forest motion system wasn't implemented due to time limitations and thus is only discussed.

4.2 Obtaining data

This section is outlined as follows. Subsection 4.2.1 explains basic tree representation, subsection 4.2.2 gives insight of how are data created and exported into basic tree representation. Exporting utility functionality is described in subsection 4.2.3 while algorithmical details are discussed in next subsection 4.2.4. Last subsection 4.2.5 exploits various possible improvements and other usage.

4.2.1 Basic tree representation

Basic tree representation is source representation for single tree. This is not a representation used for viewing or rendering, but basic tree representation containing data necessary for creation of such representation. This means that structure of basic tree representation is not optimized for today's hardware rendering, but provides good structure for working on given data such as computation of normals from triangle faces.

Basic tree representation structure The structure of basic tree representation is:

Array of branches Array containing all tree branches data including tree trunk.

Array of leaves Array containing all tree leaves data.

Array of materials Array of materials used for tree parts texturing.

Hierarchy Tree hierarchy containing connection logic of tree's branches and leaves.

Branch representation structure The branch representation structure is:

Flag Determines which information is present for branch - can be any combination of Coordinates, Normal, Texcoord, Faces, Material, Level, BoundingBox flag.

Array of coordinates Array of vertex positions for given branch. Coordinates flag indicates that this information is present.

Array of normals Array of vertex normals for given branch. Normal flag indicates that this information is present.

Array of texture coordinates Array of texture coordinates for given branch. Texcoord flag indicates that this information is present.

Array of triangle faces Array of triangle faces for given branch. Faces flag indicates that this information is present.

Material ID Identification of material for given branch. This ID is number pointing into **Array of materials** in basic tree representation. Material flag indicates that this information is present.

Level Level of branch in branching hierarchy. The trunk branches have level 0. Branches growing from trunk branches have level 1. Branches growing from level 1 branches have level 2 and so on. Level flag indicates that this information is present.

Bounding box Minimal axis-aligned bounding box containing all branches vertexes. BoundingBox flag indicates that this information is present.

Leaves representation structure Leaves representation structure is same as branch representation structure with exception that **Level** data entry is not present.

Tree hierarchy structure Tree hierarchy structure is:

Root Root to tree hierarchical structure. Root is of type tree hierarchy part structure.

Tree hierarchy part structure Tree hierarchy part structure is:

Object type Type of object. Can be leave, branch or root. In case of root the hierarchy part is actually not pointing to any branch or leave data representation, but is virtual root having level 0 branches (trunks) as its children.

Object ID Unique object identification in hierarchy.

Array of children Contains children of current hierarchy part.

The basic tree representation is in binary form and in such form can be saved into and loaded from hard drive. The basic tree representation is usually saved as *.FullTree file. Note that implementation saves basic tree representation into a file with 4 byte magic number followed by 4 byte file version which identifies whether the file is basic tree representation or not. Basic tree representation files and any other representations saved in our implementation onto hard drive uses this mechanism in order to recognize file regardless of its filename.

4.2.2 Modeling and exporting data

It isn't easy to model realistically looking trees. In professional 3D modeling programs like Maya or 3D Studio MAX this task requires professional skilled modeler and many hours to work before a single tree is made. That's why special plants and tree modeling tools have been made available for plants modeling purposes.

The other approach is to generate tree based on grammatic language automatically. A set of rules for tree structure, branches and leaves size, position, rotation and other important things is defined using grammatic. This is done with some degree of freedom so that, when final automatic generation process takes place, there can be some variation of trees of same type generated. However it requires detail knowledge of particular tree type to achieve realistically looking results. And also set of attributes to model such a tree is relatively high

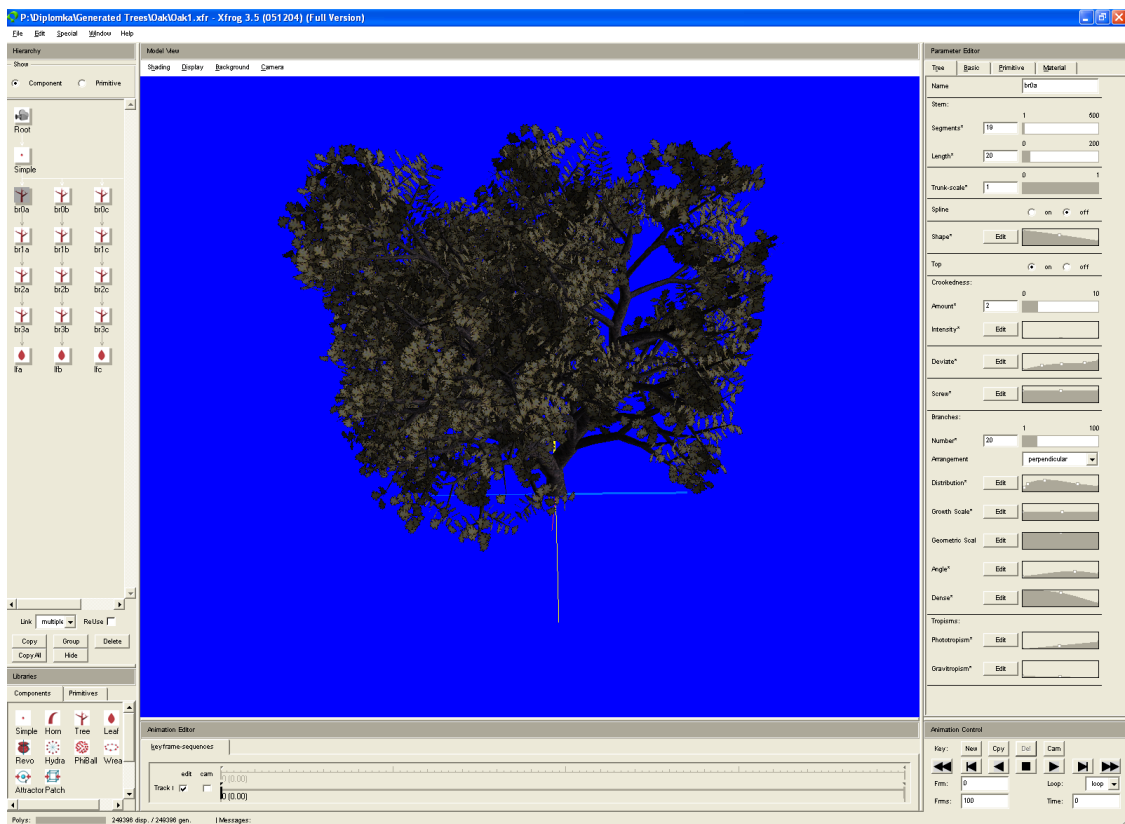


Figure 4.1: XFrog 3.5 tool environment showing custom modeled oak tree

incorporating attributes like branches twist, photo-tropism and gravitropism. That's why we decided to use professional tool for generation of trees called XFrog. The XFrog 3.5 tool was gracefully donated to us by Greenworks organic software company in exchange for our research results.

The tool environment consist of tree structure part in the left, tree viewing window in the middle and properties in the right. This is shown on the figure 4.1 along with custom made oak tree. Tree creation process is simple enough. The modeler starts with creating tree structure in tree structure part in the left from various primitives such as tree primitive used for trunk and branches or leaf primitive used for tree leaves. The resulting tree can be immediately viewed in the middle viewing window. If the modeler is unsatisfied with the result, primitive's properties can be adjusted in the properties part in the right.

Using XFrog modeling tool has another big advantage and it's so that it comes with tree library consisting of twenty trees each modeled in tree ages. This gives us enough testing data without the need to model trees ourselves although we tried to model our own custom oak which is also showed on the figure 4.1.

The resulting tree models are then exported from XFrog into *.3DS file format. And after that we use our exporting utility which converts *.3DS tree file (3D studio file format) into our basic tree representation This is discussed

in next subsection.

4.2.3 Exporting utility overview and usage

Because we didn't get any XFrog software development kit, but only modeling tool, we weren't able to convert our tree data from XFrog file format directly into our basic tree representation. That's why we examined exported tree data in *.3DS file format by dumping whole 3DS file into textual representation. We then decided that's fair enough for us to use this easy understood dumped textual representation for conversion into our basic tree representation. We know that this is not an optimal solution, but we only needed light-weight tool for obtaining data into our basic tree representation and for this purpose such solution was easiest we found.

For this whole process we made conversion utility called FullTreeConvert. With this utility we have not only been able to convert data, but also to compute missing normals, split branches and leaves groups into individual branches and leaves and even successfully reconstruct tree hierarchy by our own invented reconstructing algorithm. And this all can be done with minimal manual effort of naming primitives parts by specified naming convention in XFrog tree modeling tool.

We will now describe a usage of our FullTreeConvert utility and leave algorithmical details into next subsection where we'll also further explain meanings of usage parameters.

Fulltree utility is a command-line utility with usage:

FullTreeConvert [-m(1 or 2)] [-s] [-n] [-h] [-f Tolerance] [-o OutputFileName] InputFileName

FullTree utility parameter meaning:

- m1** Coordinates with minimal texture coordinate in y axis are averaged and average is used as significant point for leaf. This relates to tree structure reconstruction algorithm and defines method for finding leaves connection with branches.
- m2** All leaves vertex positions are averaged and average is used as significant point for leaf. This relates to tree structure reconstruction algorithm and defines method for finding leaves connection with branches.
- s** Do not split loaded data into parts. Branches and leaves are exported by XFrog in groups. By specifying this option groups are not split into individual branches or leaves.
- n** Do not compute missing normals. If vertexes normals are not present in the data, normals will not be computed.
- h** Do not try to reconstruct tree hierarchy. Tree hierarchy is not reconstructed by FullTreeConvert conversion utility.
- f Tolerance** Distance by which to grow bounding boxes when reconstructing tree hierarchy. This relates to tree structure reconstructing algorithm and defines maximal distance for tree part bounding box in which connection with other tree part can be found.
- o OutputFileName** Specifies custom output filename. Otherwise input filename is used having *.FullTree extension.

InputFileName Specifies input filename which must be 3D Studio file usually with extension *.3DS. This is only mandatory parameter. Special naming convention must be used in 3DS file for object names. Leaves names must start with "lf" and branches names must start with "br" followed by hierarchy number. Hierarchy number is "0" for trunk, "1" for branches growing from trunk and so on. For example "br0MainTrunk" is correct branch name conformable with our naming convention.

4.2.4 Exporting utility

Our FullTreeConvert utility works in following succeeding steps:

1. Converts 3DS file format into textual dump file DumpFile.Dump.
2. Loads textual dump file and parses it in order to read all tree data information.
3. Splits branches groups into individual branches.
4. Splits leaves groups into individual leaves.
5. Computes normals for branches.
6. Computes normals for leaves.
7. Computes axis-aligned bounding boxes for branches.
8. Computes axis-aligned bounding boxes for leaves.
9. Computes so called significant points for tree hierarchy reconstruction for branches.
10. Computes so called significant points for tree hierarchy reconstruction for leaves.
11. Reconstructs tree hierarchy.
12. Saves result into basic tree representation file.

3DS file is required to have special names of data groups, otherwise data are skipped and not loaded into basic tree representation. This special name convention is used to recognize what are leaves and what are branches. And for branches their hierarchy level is also determined with this special naming convention. This is only extra manual step required to convert data into basic tree representation. Leaves must start their names with "lf" and branches with "br" followed by hierarchy number. Hierarchy numbering start from 0 for trunk branches and continues increasingly. So for branches growing from trunk it's 1 and so on. An example of naming convention can also be seen on figure 4.1 with right naming convention for our oak tree.

The branches and leaves groups are split by optimized equivalence class finding algorithm. The equivalence class is in this particular case specified so that all vertices indexed by triangle face lies in the same equivalence class and when two triangle faces index same vertex then all vertices of this two faces lies also in the same equivalence class. The algorithm 1 shows a pseudo-code of equivalence class finding algorithm.

Algorithm 1 Equivalence class finding algorithm

```

Vertexes – An array of all vertexes
Faces – An array of all triangle faces
Groups – Resulting equivalence classes aka split parts of tree
procedure FINDEQUIVALENCE(Vertexes, Faces, Groups)
    Assign each vertex its unique group number
    for all Faces do
        Find group of first vertex in triangle face. Call it A.
        Find group of second vertex in triangle face. Call it B.
        Find group of third vertex in triangle face. Call it C.
        if A is different from B then
            Renumber all vertexes having group B to group A
        end if
        if A is different from C then
            Renumber all vertexes having group C to group A
        end if
    end for
    Return found equivalence groups as Groups
end procedure

```

The trick in implementing this fast is to optimize renumbering steps by also remembering mapping from group numbers to vertexes. This can be done by some sort of multimap container.

Normal for vertex is computed by first computing adjacency information for each vertex. Adjacency information for vertex contains all triangle faces the vertex is lying on. Then normal for each triangle face is computed as plane normal aligned with triangle face vertices. Finally normal for each vertex is averaged as average of triangle face normals of faces vertex is lying on. For this last step adjacency information is used in the obvious way.

Computation of axis-aligned bounding boxes is straightforward. Minimum and maximum in each axis is taken as bounding box minimum and maximum by which axis-aligned bounding box is defined.

Perhaps most interesting part of FullTreeConvert utility is tree hierarchy reconstruction algorithm. We use branches hierarchy level information and whether a tree part is branch or leaf information acquired in previous processing steps. First we compute point we call significant point for each tree part. The significant point of some tree part can be understood as point which is possibly nearest point from lower level tree part to which is currently examined tree part connected or more subtly from which is current tree part growing.

With all significant points computed we start reconstructing our tree hierarchy from trunk. First all branches of level 0 are automatically considered as trunks. Then for each branch of level 1 its significant point is tested against grown axis-aligned bounding box of all branches of level 0. When level 1 branch significant point lies inside grown axis-aligned bounding box of some level 0 branch, this level 0 branch is taken as candidate for connection. On the contrary when significant point falls outside axis-aligned bounding box, the level 0 branch is discarded as not connected with this level 1 branch. From all level 0 branches considered as candidates for connection minimal distance of all

branch vertexes from significant point is found. The level 0 branch with nearest vertex from significant point is taken as hierarchy parent for examined level 1 branch and connection is established. When all level 1 branches connection is established, the method proceeds with level 2 branches and so on until all branches levels are connected and so branches hierarchy reconstructed. Then in the same manner leaves are connected with highest level branches only. This reconstructs whole tree hierarchy.

It's very important to choose right significant point for tree parts. By examining data structure exported from XFrog we found that for branches it's good to choose significant point as vertex which appears on the lowest number of faces. For leaves we are using significant point as an average of vertexes positions with minimal texture coordinates on axis y. This is very good assumption when leaf texture has leaf petiole in the bottom. We are providing an option to alternatively choose significant point as an average of all leaf vertexes positions.

The degree of how much branches or leaves are discarded from tree hierarchy reconstruction can be influenced by setting an amount of axis-aligned bounding box growth. By setting large enough value one can assure that all branches and leaves are connected, but with higher potentiality of bad connection. This is however not so big issue to have few leaves are small branches connected incorrectly.

The algorithm 2 shows tree hierarchy reconstruction in pseudocode.

The whole FullTreeConvert utility speed depends on size of the input data. On our AMD 3.8+ Ghz the utility takes tens of seconds to few minutes to convert a single tree from 3DS file representation into basic tree representation with all things computed and tree hierarchy reconstructed. We just assume that utility is fast enough and doesn't provide any detailed utility speed measurements. Note that in our collection of trees we have tree with as much as one million of triangle faces which also completes its conversion within specified time range. Memory requirements are proportional to the size of the input data.

On the contrary of approach used for conversion, we have successfully converted more than 30 trees from 3DS file format into our basic tree representation with no problem. Many of trees are from XFrog basic plants library and so these trees haven't been modeled by us. This shows that our conversion utility isn't dependent on any single tree modeling style used within XFrog modeling tool.

We note that current implementation of FullTreeConvert handles case of leaf or branch with no vertexes incorrectly when reconstructing hierarchy and displays warning that such leaf or tree cannot be added to tree hierarchy. This is however minor issue.

4.2.5 Possible improvements and other usage

There can be done various improvements. First 3DS file format can be read directly from its binary form and not its textual dump file. We however believe that this improvement will mainly bring utility speed enhancement and introduce possible option to read normals directly from 3DS file if they are present. This is reasonable believe because 3DS file format is quite simple and doesn't provide many other options.

Potential extension of our basic tree representation can be made to provide support for more types of tree parts like tree roots, fruits and blossoms. Even

Algorithm 2 Tree hierarchy reconstruction algorithm

Branches – Array of all tree branches
 Leaves – Array of all tree leaves
 TreeHierarchy - Reconstructed tree hierarchy returned from algorithm
procedure TREEHIERARCHYRECONSTRUCTION(Branches, Leaves, Tree-
 Hierarchy)
 Compute significant points for branches
 Compute significant points for leaves
 Let L be branches level and set it to 0
 while Some branch has level L **do**
 for all Branches with level L **do**
 Let A be branch of level L
 if Level is 0 **then**
 Add branch to hierarchy
 else
 for all Branches of Level (L - 1) **do**
 Let B be branch of level (L - 1)
 if Significant point of A is in growed
 axis-aligned bounding box of B **then**
 Compute minimal distance of branch A
 significant point from branch B vertexes and store it
 end if
 end for
 Connect branch A to branch B with lowest minimal
 distance. If there is no such minimal distance do nothing.
 end if
 end for
 Increase level L by 1
 end while
 for all Leaves **do**
 Let A be leaf
 for all Branches of Level (L - 1) **do**
 Let B be branch of level (L - 1)
 if Significant point of A is in growed
 axis-aligned bounding box of B **then**
 Compute minimal distance of leaf A
 significant point from branch B vertexes and store it
 end if
 Connect leaf A to branch B with lowest minimal
 distance. If there is no such minimal distance do nothing.
 end for
 end for
end procedure

larger extension can be made to provide support not only for trees, but also for grass, flowers and shrub types of plants.

We are sure that used solution is far from any general conversion tool into our basic tree representation. But we see that for our purposes our data conversion solution is good enough. To write any more general tool would probably require to solve much more than loading from various data file formats. Various modeling tools exporting habits should be discovered and taken into account for such general solution and this is far beyond our need and aim of our diploma thesis.

We presented tree hierarchy reconstruction algorithm which we believe can be adapted for many other reconstruction tasks. We note that we aren't using reconstructed tree hierarchy in the current implementation in any place. We have planned its use for level of detail tree representation generation, but never actually get so far in implementing level of detail tree representation conversion utility to use it. Other tasks were more important. But still we leave this reconstructing feature in for possible further use by anyone who wants to extend our work or simply use our basic tree representation for his own purposes.

4.3 Simple tree representation

In this sections we'll discuss our simple tree representation used for basic viewing of single tree. In subsection 4.3.1 we'll present data structures used by simple tree representation. Then we'll overview our simple tree conversion utility called `ConvertToSimple` in subsection 4.3.2. We'll proceed with explaining `ConvertToSimple` utility in more depth in subsection 4.3.3. Finally possible improvements are discussed in subsection 4.3.4.

4.3.1 Simple tree representation format

Simple tree representation is representation used for basic viewing of single tree. Simple tree representation is optimized for viewing by sorting triangle faces and vertices by material and also designed with respect to the fact that all data are finally loaded into OpenGL vertex buffer object and stored in graphic card's memory for rendering. There are no level of detail techniques applied for trees stored in this representation. This representation can be viewed by our viewer application.

Simple tree representation structure Simple tree representation structure is as follows:

Array of branches sorted by material Array of branches data arranged in a way that every array entry has unique material and all branches having such material are included as data.

Array of leaves sorted by material Array of leaves data arranged in a way that every entry has unique material and all leaves having such material are included as data.

For every array entry all vertices data are stored in single continuous block of memory and also all indices (triangle faces) data are stored in another single continuous block of memory. The difference from basic tree representation is

that in basic tree representation vertices data are stored in multiple blocks of memory for each data component. For example vertex normals are stored in one memory block while vertex coordinates are stored in another memory block. In simple tree representation data are interleaved so that all vertex components are stored in defined order before next vertex is stored. This is optimal for today's graphic hardware because it uses vertex shader unit's memory cache on graphic card in the best way. There is no information of tree hierarchy or axis-aligned bounding boxes stored in simple tree representation structure.

Material with associated data Material with associated data structure is arranged as follows:

Material Material definition.

Vertexes data Single block of continuous memory containing all vertexes having material defined by **Material**. Vertex format is introduced below.

Indices data Single block of continuous memory containing all indices having material defined by **Material**. Each indice is stored as triple index into **Vertexes data**. This is arrangement used for triangle list OpenGL draw call.

Interleaved vertex structure Interleaved vertex structure looks as follows:

Position Vertex position x, y and z.

Normal Vertex normal nx, ny and nz.

Texture coordinates Single set of texture coordinates u and v.

Note that this arrangement is also optimal for vertex shader unit's memory cache, because its size is 32 bytes and most of graphic hardware memory caches are arranged in multiplies of 32 bytes.

Simple tree representation can be saved onto and loaded from hard drive in binary representation.

4.3.2 Simple tree conversion utility overview and usage

Our simple tree conversion utility is used to convert data from basic tree representation into simple tree representation. This utility is called `ConvertToSimple` and it is the simplest utility we created.

`ConvertToSimple` utility is command-line utility with usage:

```
ConvertToSimple [-o OutputFileName] InputFileName
```

Usage parameter meaning:

-o OutputFileName Output filename. The `*.SimpleTree` is usually used as filename extension.

InputFileName Input filename. Basic tree representation file must be used as input filename. If no output filename is specified, input filename is used with changed file extension to `*.SimpleTree`.

4.3.3 Simple tree conversion utility

ConverToSimple utility works in three steps:

1. Loads basic tree representation.
2. Converts basic tree representation into simple tree representation.
3. Saves resulting simple tree representation into file.

From this only second step is interesting. Conversion is done by first sorting branches and leaves arrays in basic tree representation according to their material ID. Then for every material vertex interleaved structure is assembled from coordinate, normal and texture coordinate arrays and filled into single continuous block of memory. Triangle indices are reindexed accordingly and filled into another single continuous block of memory. Material definition is also added. This process is done separately for branches and leaves.

Memory requirements are proportional to the size of input file. Because of its relative simplicity the ConvertToSimple utility is running fast in terms of seconds or tens of seconds on our computer with AMD 3.8+ Ghz processor. We therefore see no need in exact measurements of utility running time.

4.3.4 Possible improvements

Possibly even better performance could be achieved by constructing triangle strips for branches. It's not good to arrange data in a way that for each constructed triangle strip there is a separate draw call by OpenGL. This would lead in too much draw calls and predictably in loose of performance rather than gain. We believe it's possible to connect constructed triangle strips by insertion of fake triangle which is never rendered and discarded in triangle culling step by graphic card hardware. For leaves no such trick is good because leaves are rendered double-sided and single leaf is often represented as quad consisting of two triangles which is very small number for construction of triangle strip.

Some memory can be saved by making face indexes only two bytes long instead of four bytes long in exchange of some increase in draw calls.

4.4 LOD tree representation

This section explains our level of detail approach for single tree. First we introduce our LOD tree representation format in subsection 4.4.1. In next subsection 4.4.2 we overview LOD tree conversion utility which is followed by detail explanation in subsection 4.4.3. Finally we discuss possible improvements in last subsection 4.4.4.

4.4.1 LOD tree representation format

The main purpose of LOD tree representation is to be used for single tree level of detail. Its structure is on the contrary to previous formats more general and can be easily extended. For our needs we implemented it in a way to support hierarchical pseudo-continuous view-dependent level of detail with storage capability for mixed point and triangle tree data. We also made an extension of

some parts of LOD tree representation in order to support entire forest. This extension is discussed in section 4.5. LOD tree representation can be viewed by our viewing application.

LOD tree representation structure LOD tree representation structure is defined as follows:

Root node Pointer to root node of whole hierarchy.

LOD tree representation node structure Node structure for LOD tree is:

Data type Type of data stored in **Data pointer**. This can be:

None No data.

Points Data contains only point representation.

Geometry and points Data contains mixed triangles and points representation.

Instance info Instance information data used to instantiate tree into forest.

Data pointer Stores data representation specified by **Data type**.

Bounding type Type of bounding object stored in **Bounding pointer**. This can be:

None No bounding object.

Axis-aligned bounding box Axis-aligned bounding box is used as bounding object.

Bounding pointer Stores bounding object specified by **Bounding type**.

Distance type Type of object used to determine distance from node stored in **Distance pointer**. This can be:

None No distance object.

Point Single point.

Points Multiple points.

Distance pointer Stored distance object specified by **DistanceType**.

Additional data with its size This can be any user added data.

Children Any number of children nodes of this hierarchy node. In this way tree hierarchy is defined.

As it can be seen, LOD tree representation creates tree structure and just is hierarchical. In our implementation it offers either to carry solely points data or mixed points and triangles data. **Instance info** is special data node used for instantiation of the tree into the forest. And will be described in section 4.5. Either none or axis-aligned bounding box can be carried which is mostly used for frustum culling. Distance object is used for fast determination of distance from the node object. Any additional user data can be carried in **Additional data**.

Point data structure We'll now describe point data structure:

Points Single continuous block with points data.

Point area Sorted points area corresponding with points by one to one mapping.

Single Point data entry is organised in following way:

Position Point position x, y and z.

Normal Point normal nx, ny and nz.

Size Diameter of point.

Color Point color in RGB.

Mixed point and triangle data structure Mixed triangle and point structure is defined as follows:

Geometry Array of branches or leaves geometry with associated material per array entry. Material with associated data is used as defined in simple tree representation in subsection 4.3.1.

Points Single continuous block with points data.

Geometry area Array of arrays of sorted triangle areas with one to one mapping into **Geometry**.

Point area Sorted points area corresponding with points by one to one mapping.

Is branch Array of booleans defining whether geometry are branches or leaves. Corresponds with **Geometry** by one to one mapping.

It can be seen that point data structure is similar to mixed triangle and point structure. While for points data a single continuous block is used, because points don't have associated material information with it, for triangles data storage scheme is similar to simple tree representation having vertexes and indices stored with associated material information. Sorted point and triangle area are used in viewer to determine current node level of detail. This is explained in detail later.

LOD Tree representation can be saved into and loaded from hard drive in binary form. The usually used extension is *.LODTree.

4.4.2 LOD tree conversion utility overview and usage

For the purpose to get our level of detail tree representation, we have developed utility which converts basic tree representation into LOD tree representation. This utility is called ConvertToLODTree.

Our LOD tree generation method is slightly changed one used in *Point-based rendering of trees* [11]. In short first for each triangle exactly one point is generated. Then these generated points are subdivided by regular grid into cells. For each cell a tree of points is constructed where subdivided cell points are merged together to create higher and bigger tree level points used for lower level of detail. For each cell a LOD tree representation node is created. Mixed triangle and point representation is used. All cell's points are gathered from created tree and sorted by its area. In such sorted way they are stored in mixed triangle and point data structure. Corresponding triangles to points in cell are also sorted by material and then by area and stored in mixed triangle and point data structure. Finally solely point representation is created for whole tree by gathering some user specified percentage amount of biggest points from all nodes with mixed triangle and point data structure. Point binary tree is constructed

having these gathered points in lowest nodes of tree. In the same manner as previously higher level points are created by merging these points into bigger ones. Points are then gathered from binary tree, sorted by their area and stored in point data structure. All nodes with mixed triangle and point data structure are then added as children into node with point data structure. For all nodes axis-aligned bounding boxes are computed and in runtime used for frustum culling.

We now describe usage of ConvertToLODTree conversion utility. Algorithmical detail and in-depth study follows in next section 4.4.3. ConvertToLODTree command-line utility usage is:

ConvertToLODTree [-c ConfigurationFileName] [-o OutputFileName] InputFileName

ConvertToLODTree parameters meaning:

-c ConfigurationFileName Configuration filename with all options. If not specified config.ini is used as configuration filename.

-o OutputFileName Output filename in LOD tree representation. If not specified input filename is used with extension *.LODTree.

InputFileName Input filename. This must be basic tree representation file.

Our configuration file has similar structure to a *.ini file structure. Textual representation is used where two things may appear on the line of configuration file:

; commentary Line starting with apostrophe serving as commentary.

Key=Value ; commentary Pair of key and its assigned value. This can be optionally followed by commentary.

Configuration keys for ConvertToLODTree are:

GridSizeX Size of regular grid by which is tree subdivided in X axis. Can be positive integer smaller than 11.

GridSizeY Size of regular grid by which is tree subdivided in Y axis. Can be positive integer smaller than 11.

GridSizeZ Size of regular grid by which is tree subdivided in Z axis. Can be positive integer smaller than 11.

PercentageOfUpliftPoints Percentage amount of biggest points used for creation of node for whole tree. Can be floating point number within range [0, 1]. This also relates to change distance from mixed triangle and point nodes rendering to solely point rendering of tree with only whole tree node.

PointScaleFactor A factor by which to scale each point size and area, and triangle area. Can be floating point positive number.

NumberOfColorApproxPoints Number of color samples used for point color approximation while creating point from triangle. Can be positive integer.

NumberOfColorApproxRetries Number of retries for acquiring color sample used for point color approximation in case that acquiring of color sample fails by low alpha value of texel accessed. Can be positive integer or zero.

MinAlphaValue Minimal alpha value to accept color sample for point color approximation. Can be floating point number within range [0, 1].

MaxPointSize Maximal point group size. This is used for generation of bigger points for cells. This number tells maximal number of points which can be united to generate first bigger point from points created from triangles. This can be positive integer.

SixPointsTreeLevelDistance Enables to use six points for distance determination for tree level node instead of one point. This can be 0 or 1.

CircumscribedCircleArea Enables to use circumscribed circle triangle area as point area instead of triangle area. This can prevent holes in tree while rendering with points. This can be 0 or 1.

DesiredDistance Desired distance in meters where change from mixed triangle and point nodes rendering to solely point rendering of tree with only whole tree node occurs. Serves only for display of statistics where amount of **PercentageOfUpliftPoints** is displayed in order to achieve desired change distance. Note that **TriangleCutOffToleration** may also alter change distance. This is positive floating point number.

TriangleCutOffToleration Relates to change from mixed triangle and point nodes rendering to solely point rendering of tree with only whole tree node. This occurs in certain distance, but distance must not be too small to discard rendering of some percentage of biggest triangles. This floating point number means maximal percentage amount of biggest triangles which can be discarded from rendering. Refer to next subsection 4.4.3 for more information.

Metric options can also be specified in order to evaluate desired change distance setting and statistics:

ScreenWidth Width of rendering screen in pixels. Can be positive integer number.

ScreenHeight Height of rendering screen in pixels. Can be positive integer number.

MinPointArea Minimal point area on screen in pixels to render point. This can be positive floating point number.

MaxPointArea Maximal point area on screen in pixels to render point. This can be positive floating point number.

FOV Field of view in y axis. Can be positive floating point number.

NearPlaneDistance Distance to near plane from eye in meters. Can be positive floating point number.

Many configurations parameters closely relates to used level of detail generation technique. To gain better insight into parameter usage, we recommend to read next subsection 4.4.3 where whole `ConvertToLODTree` utility is described in much more detail and depth.

4.4.3 LOD tree conversion utility

`ConvertToLODTree` utility works in listed succeeding steps:

1. Loads basic tree representation from file.
2. Loads textures belonging to loaded basic tree representation from files.

3. Creates point for each branch triangle face.
4. Creates point for each leaf triangle face.
5. Divides points by regular grid.
6. For each cell
 - (a) Bigger points are generated by merging cell's points.
 - (b) Creates mixed triangle and point data structure.
 - (c) Computes axis-aligned bounding box and distance point.
 - (d) Creates node data structure to carry mixed triangle and point data structure along with axis-aligned bounding box and distance point.
7. Creates point data structure for whole tree node data structure, computes axis-bounding box and distance point(s) for whole tree.
8. Creates whole tree node data structure carrying point data structure.
9. Prints statistics regarding to change distance where mixed point and triangle rendering is changed only to point rendering of whole tree node.
10. Saves resulting LOD tree representation into file.

Loading of basic tree representation from file is self-explaining process. Appropriate textures are loaded using DevIL, a cross-platform image library, and texture's data are then converted into same RGBA format. This is done to simplify task of point color approximation. It's much more easier to read color from texture's data only in one specified color format then from many color formats.

Branches and leaves are first sorted by material in basic tree representation and then points are generated from triangle faces in the manner that for one face exactly one point is created. This one to one mapping is also stored to be able to identify which point belongs to which triangle face. For each generated point its position, normal, color and diameter is computed. We have implemented computation of point components as callback functions in order to be easily interchanged by anyone trying to program its own point generation behaviour.

We tried to compute point position as an average of triangle face vertices positions as it's suggested in *Point-based rendering of trees* [11], but we discovered that it has a drawback of having generated point positions in circle shapes for branches and thus branches covering by points is not good, holes may appear in points representation during rendering and one can see through branches. This has to do with a method of generating branches by segments with same or almost same triangles around branch. We therefore abandoned method suggested by [11] and computed points positions as random position on triangle. For this we used triangle interpolation method from *Linear interpolation* [1]. Having triangle with vertices positions A, B, C we compute vectors $U = B - A$ and $V = C - A$ and then random numbers α and β where $0 \leq \alpha + \beta \leq 1$. The random point position is finally computed as $P_{random} = A + \alpha \cdot U + \beta \cdot V$.

We compute point normal as triangle face normal. Point color computation is algorithmically most difficult to compute from all point components. Having three texture coordinates, texture size and texture data the task is to get color

for point. Our first approach was to compute all texels of three lines on the texture determined by triangle texture coordinates and save them as bounds. These points bound all texels on texture which are used as color for triangle while rendering. So we just computed an average of all such texel colors being bounded by bounding points on texture. Process of computing point color is shown on algorithm 3. Only texels with alpha value greater or equal to specified amount in configuration as **MinAlphaValue** are used to average point color. A coverage amount is also returned and computed point diameter is multiplied by it to account texture coverage for point size. If texture is transparent in a way that point color cannot be computed, the point for such triangle face is not generated and triangle face is discarded from further processing.

Algorithm 3 Precise point color

TexCoordA, TexCoordB, TexCoordC – Texture coordinates of triangle face vertices
 TextureSize – Texture width and height in pixels
 TextureData – Texture color data
procedure PRECISEPOINTCOLOR(TexCoordA, TexCoordB, TexCoordC, TextureSize, TextureData)
 Compute line points determined by TexCoordA and TexCoordB on texture
 Compute line points determined by TexCoordA and TexCoordC on texture
 Compute line points determined by TexCoordB and TexCoordC on texture
 for all texels bounded by line points **do**
 Compute average color of texture’s texels color with alpha value \geq minimum alpha value
 end for
 Return average color of texture’s texels color and coverage amount
end procedure

This approach however proved to be slow. It takes a lot of time to compute precise point color and so we decided to only approximate point color. Approximation of point color is done in similar way as precise point color computation. First points of bound lines are computed and then only color from specified number of texels determined by config option **NumberOfColorApproxPoints** is used for point color approximation. If texel with too much transparent color value is encountered, an another random texel is taken instead. For too transparent texels this process of new texel selection is repeated until configuration option **NumberOfColorApproxRetries** reaches zero. If **NumberOfColorApproxRetries** reaches zero, a texel is skipped and color approximation is taken from one less point. The algorithm 4 describes point color approximation in pseudo-code for clarity.

The diameter for point is computed such that $D_{result} = Scale \cdot D$ where D is a point diameter computed in a way that point area $Area_{point} = D^2$ equals to triangle area. $Scale$ is specified by configuration parameter **PointScaleFactor**. We decided to use square area instead of rectangular area, because points render as squares by default in OpenGL. This behavior can be changed

Algorithm 4 Point color approximation

TexCoordA, TexCoordB, TexCoordC – Texture coordinates of triangle face vertices
 TextureSize – Texture width and height in pixels
 TextureData – Texture color data
procedure APPROXIMATEDPOINTCOLOR(TexCoordA, TexCoordB, TexCoordC, TextureSize, TextureData)
 Compute line points determined by TexCoordA and TexCoordB on texture
 Compute line points determined by TexCoordA and TexCoordC on texture
 Compute line points determined by TexCoordB and TexCoordC on texture
 for Specified number of texels **do**
 for Specified number of retries **do**
 Let T be random texel from bounded area determined by line points
 if Minimum alpha value of T \geq minimum alpha value **then**
 Account color of T in point color average
 Break
 end if
 end for
 end for
 Return average color of texture's texels color and coverage amount
end procedure

by enabling point smoothing, but when we do so, it drastically reduces point rendering performance by an order of magnitude. When configuration option **CircumscribedCircleArea** is enabled, a point area $Area_{point}$ must instead equal to an area of circumscribed circle to given triangle. An area of triangle's circumscribed circle can be easily computed as $Area_{circumcircle} = \pi \cdot r^2$ where $r = \frac{a \cdot b \cdot c}{4 \cdot Area_{triangle}}$ with a, b, c being sizes of triangle sides. Usage of circumscribed circle area prevents point representation from showing holes between points while rendering.

After points are generated for triangle faces with one to one mapping, points are divided by regular grid of size specified as configuration parameters **GridSizeX**, **GridSizeY** and **GridSizeZ**. First minimal axis-aligned bounding box is computed from all generated points to determine entire grid size. Then points are divided into cells separately for branches and leaves points. This separate division is because we need to remember which points belong to branches and which to leaves. Points division is simply done by computing to which cell given point comes.

Number of steps is then performed for each cell separately. If some cell is empty with no points, processing for such cell is skipped and cell is discarded. Very interesting is generation of bigger points for given cell by merging cell's points. But before we proceed with explanation, we need to clarify terminology. We are using general, in meaning that nodes can have multiple children, tree data structure in next utility step and for this reason we'll use a word "tree" as data structure and not a plant with branches and leaves. We'll also refer to

"tree node", "leaf nodes" and "tree root" as parts of this data structure.

For the purpose of bigger points generation we need to divide cell points into small groups with property that points in the group have close distance to each other. We are using axis-aligned bounding box for points division. First axis-aligned bounding box is computed for all cell points. Points are then divided into two groups by plane coming through and perpendicular to longest side of axis-aligned bounding box. The plane just divides axis-aligned bounding box into two parts and creates two point groups. For each group the same process is then repeated. The points division stop when number of points in group is equal or smaller than configuration parameter **MaxPointGroupSize**. This points division process just creates a tree structure with cell points stored in the leaf nodes of the tree. The tree has a property that only parent nodes of leaf nodes can have more than two children. On the upper levels, meaning that lowest level are tree leaf nodes, the tree is binary. Created tree is then used for bigger points creation in a way that for each non-leaf tree node exactly one point is created from its children node points. Algorithm 5 shows cell points division along with creation of point tree in pseudo-code.

Algorithm 5 Point tree creation

```

Points - Cell points
PointTreeNode - A node of tree, first called with tree root
procedure CREATEPOINTTREE(Points, PointTreeNode)
  if Number of Points  $\leq$  maximal number of points in group then
    Add Points to PointTreeNode
    Return
  end if
  Compute axis-aligned bounding box B for points
  Divide point to groups G and H by plane
  coming through longest side of B
  Create two children nodes C and D for PointTreeNode
  Call CreatePointTree(G, C)
  Call CreatePointTree(H, D)
end procedure

```

The bigger points creation is done by traversing previously constructed tree nodes from bottom to top by recursive algorithm. For each non-leaf node one bigger point is created with position, normal, color and diameter. Bigger point for given node is computed from points contained in its children nodes. The creation of bigger point can be also viewed as merge of smaller points into one bigger point. We have implemented bigger point components computation by callback functions in order to allow anyone to easily write its own bigger point components computation. In our implementation we compute bigger point position as an average of all point positions. The normal as a weighted average of all point normals. The weight used is point diameter. We compute bigger point color as a weighted average of all points color with point diameter used as weight too. These computations are similar as used in *Point-based rendering of trees* [11]. We however were unsatisfied with heuristic used for point size computation in [11]. We have discovered that much more predictable and good behaving is to determine point diameter by computing area of squares drawn on the plane.

We treat all points as squares with side size of point's diameter and project them on XY coordinate system axis-aligned plane. Then we compute area on the plane covered by union of projected squares. This is called Klee's problem and can be solved by a sort of sweepline algorithm. Bigger point diameter is simply a side of square with area equal to area of projected squares union. In our opinion this solution returns exactly the information we need to know. For the case when point squares don't overlap, we get union square area equal to sum of all point squares. For the case with a lot of point squares overlapping, we get union square area which doesn't overgrow region determined by point squares.

We'll now describe our implementation of union square area computation. We have implemented a hybrid algorithm. For one point degenerate case we simply return point's square area. For two points we determine square union area by subtraction of square intersection area from sum of two square areas. For more points we use a sort of sweepline algorithm to compute union square area. For each point square its starting and ending position on axis X is added into event queue. Event queue is sorted in ascending order. Then events are processed one by one. For square starting position event minimal and maximal square position on axis Y is added into height list. For square ending position event appropriate minimal and maximal square position on axis Y is removed from height list. For each event a rectangle area is added to whole area sum. This rectangle area is determined by $(X - X_{prev}) \cdot (MaxY - MinY)$ where X is position X of current event, X_{prev} is position X of previous event and $MaxY, MinY$ is maximal and minimal Y value currently in height list. One can imagine whole algorithm as summing area of rectangles going from left to right with maximal height. This is in fact re-representation of overlapping squares to rectangles which don't overlap. Algorithm 6 recapitulates whole computation in pseudo-code.

After whole hierarchy of bigger points is created, all points are gathered from tree and mixed point and triangle data structure is constructed from gathered points and appropriate triangles. First point to triangle face mapping is used to determine which triangle faces and vertices belong to given grid cell. All vertices are then searched in basic tree representation, converted into interleaved vertex component format and stored in single continuous block of the memory in mixed triangle and point data structure. Because in basic tree representation branches and leaves were sorted by material previously in `ConvertToLODTree` utility, its enough to sort point to triangle mapping in ascending order and then incrementally walkthrough all branches and collect vertices indexed in mapped triangle faces. Vertices are stored in single entry of **Geometry** part of mixed triangle and point data structure as long as they belong to same material. When branch with new material ID appears, vertices are stored to next **Geometry** entry. Faces are stored in same manner, but must be reindexed to correctly index vertices stored in new arrangement. Exactly same process of storing faces and vertices is done then for leaves. An info whether branch or leaf is stored in **Geometry** array, is written into **Is branch** array of mixed point and triangle data structure.

After vertices and faces storage is complete, triangle areas are computed with use of callback function for computation of point diameter from triangle. This guarantees that generated point from given triangle and the triangle have same areas saved in **Geometry area** and **Point area** data entries of mixed triangle

Algorithm 6 Union square area computation

Squares - Array of squares for which to compute union square area
 ResultingArea - Resulting union square area

procedure UNIONSQUAREAREA(Squares, ResultingArea)
 Let ResultingArea = 0
 Let QUEUE be event queue
 Let H be height list
 Add square's starting X positions to QUEUE as starting events
 Add square's ending X positions to QUEUE as ending events
 Sort QUEUE by X in ascending order
for all Events E in QUEUE **do**
 if E is starting event **then**
 Add square's minimal and maximal position on axis Y into H
else if E is ending event **then**
 Remove square's minimal and maximal position on axis Y from H
end if
 Find MAXY – maximal Y in H
 Find MINY – minimal Y in H
 Add $(MAXY - MINY) \cdot (X - PREVX)$ to ResultingArea
end for
end procedure

and point data structure. This area doesn't have to be real triangle area because it is only used by metric to determine what will be rendered. It is better to have triangle and point area same because, when triangle falls off from rendering, immediate replacement by appropriate point occurs. Triangle faces and triangle areas are then sorted in ascending order. Point areas are computed, stored and points along with point areas also sorted in ascending order. Basically this whole step of creating mixed triangle and point representation is about data format conversion, but, as it can be seen, even data conversion can be non-trivial task.

New LOD tree representation node is created. Mixed triangle and point data structure is stored in it. Axis-aligned bounding box is computed from vertex positions stored in **Geometry** array of mixed triangle and point data. Distance point is computed as points position average. Both axis-aligned bounding box and distance point are stored in LOD tree representation node.

When all LOD tree representation nodes are created for grid cells, a process of creating node for whole tree follows. This node contains only point rendering data and is used to render more distant trees. First a percentage amount of biggest points is extracted (by copying) from nodes with mixed triangle and point data structure. This percentage amount is specified by **PercentageOfUpliftPoints** configuration parameter. From extracted points a point data structure is created. This includes bigger points generation with only one point allowed in point tree data structure leave nodes, point areas computation, points and point areas sort to ascending order. Only one point is allowed in point tree data structure leaf nodes because bigger points are meant to be continuation from previously generated bigger points for grid cells.

New LOD tree representation node is created. Point data structure is stored in it. Nodes with mixed triangle and point data structure are connected to

whole tree node as children. Axis-aligned bounding box is computed from children axis-aligned bounding boxes and stored. Distance point is computed either from points of point data structure when **SixPointsTreeLevelDistance** configuration option is switched off or as six points of axis-aligned bounding box side centers when **SixPointsTreeLevelDistance** is switched on.

The smallest point of points in the whole tree node determines safe change distance when rendering of tree can be done by rendering only points in the whole tree node. However this change distance can be too small so that some big triangles can be still rendered. In such case by changing rendering from mixed triangle and point representation to only point representation, it's obvious that all such triangles disappear from resulting rendered tree. This can cause sudden change in tree's visual quality and even introduce holes in the tree. To prevent this behavior we introduce **TriangleCutOffToleration** configuration parameter which determines percentage amount of biggest triangles that can be discarded when changing rendering representation. Therefore the area of first "non-discardable" triangle is saved in whole tree node's additional data. In runtime smallest point of whole tree node and saved triangle area is used to determine real safe change distance dependent on metric settings. The metric and this computation is described in section 4.6 in more detail.

This rendering representation change is vital and necessary for entire forest rendering. Having for example only 1000 trees in the forest and tree divided by 4x4x4 grid into 64 cells. One can easily count that, when viewing all 1000 trees, 64000 nodes must be walked by computer processor. This is not affordable even for interactive performance on nowadays processors and burdens graphic hardware with overwhelming number of rendering calls. We found by experience that representation change should occur between 50 and 100 meters distance of camera from tree.

To allow fine tuning of change distance `ConvertToLODTree` utility prints statistics according to specified target metric settings. This information helps user to set **PercentageOfUpliftPoints** and **TriangleCutOffToleration** configuration parameters to optimal values for given tree. Note that really used change distance is computed at runtime for every type of tree as we stated before. This is only hint. But still very useful hint. One might set target metrics setting little higher and set configuration options accordingly in order to be sure that rendering representation change occurs in appropriate distance not to ruin viewer application rendering performance totally!

The last step is straightforward. Generated whole tree node is set as root in LOD tree representation and entire LOD tree representation is saved into file.

The memory usage is proportional to the size of input data. We provide a table 4.1 of time measurements for various tree conversions. Conversion speed is measured on AMD Athlon 3.8+ Ghz with 2 GB of RAM. The parameter settings for measurement is 4x4x4 division grid with 100 color approximation points and 10 retries when point color is not acquired. As it can be seen small tree models are converted almost instantly while for larger ones it takes some time. The most time consuming part is point creation from leaves triangle faces. We note that false acacia model is the biggest one we use in our tree dataset. We find `ConvertToLODTree` utility speed fast enough considering that it's preprocessing utility.

Tree name	Branch faces	Leaf faces	Running time
Spruce	9 K	11 K	2 s
Oak	220 K	29 K	16 s
Plum	41 K	164 K	23 s
Chestnut	300 K	112 K	30 s
Silver maple	452 K	393 K	91 s
Red Oak	164 K	643 K	101 s
False Acacia	35 K	1100 K	160 s

Table 4.1: Conversion times for ConvertToLODTree utility.

4.4.4 Possible improvements

There is a drawback of randomizing position for points created from triangle faces. Leaf point positions can be chosen on place where texture is transparent. These points then causes visual artefact of solely flying in the air. This is disturbing. An improvement can be made to choose point random position only when appropriate texel on texture has enough opacity.

We aren't completely satisfied with one point from triangle face creation. For branches triangles are long in branch growing direction while in the direction of branch thickness short. This is problem because generated points are squares and square of the same area as long triangle cannot cover such triangle sufficiently just introducing holes in branches. It showed that this coverage problem is especially bad for small branches where branch segments are created only from few triangles. Resulting visual quality of such small branch is often really bad. The small branch look is scattered.

We tried to fix this issue with use of circumscribed circle triangle area. But we are aware that it is far from good solution to this problem and we have reasons to prove it. First circumscribed circle triangle area is for long triangles multiple times bigger than triangle area itself. This postpone points usage to farther view of tree. This is bad because high-detail rendering of tree using triangles is very costly and thus need arises to switch into point rendering as early as it only goes! Also secondary problem with circumscribed circle triangle area emerged unexpectedly. The color for bigger points creation is weighted by point sizes. Having a lot of small branches with triangle area much bigger than it really is introduces too much branch color in bigger points. This affects tree rendering in two ways. Usually too much branches color points are rendered at certain viewing distance making tree look like having much less leaves. Then at even bigger distance tree look goes back to leave color, but this color is slightly different than it is for close-up view. The tree changing its appearance isn't certainly what we wanted.

We therefore propose to use another points from triangles generation scheme. We believe that using scheme as in *Interactive Visualisation of Complex Plant Ecosystems* [7] will yield much better quality results. Points are assigned to triangles based on triangle area. We advise to use equal number of points as there is number of triangles on the tree. Then separately for branches and leaves distribute points based on triangle areas in a way that equally sized triangles receive equal number of points. We expect that effect of using this distribution scheme will be that bigger branches and trunk receive more points while small

branches will not receive a point. Distributing points in such way will hide holes on bigger branches and trunk while small branches will not be rendered at all from some distance. We propose that for leaf triangles area is computed as triangle area multiplied with texture coverage percentage to account for leaves texture transparency. This is more important than it may seem. In our tree dataset we have one tree where leaf texture is positioned on quad consisting of two triangles so that leaf petiole takes one triangle and the rest of leaf the other triangle. Unfortunately leaf petiole has yellow color and the rest of the leaf is dark green. So a yellow point is generated for triangle with leaf petiole. These yellow points are very disturbing during tree viewing. Accounting for leaves texture transparency with introduced point distribution scheme will eradicate these yellow points completely because petiole has much smaller area than the rest of the leaf and so triangle with petiole will never receive a point.

A configuration parameter specifying minimum branch triangle area for which a point is generated can be also added to complement proposed points from triangles generation scheme or to be used alone. We note that implementation of this small improvement may not be as easy as it seems because our implementation is heavily one to one triangle and point mapping driven. And just a code may require some amount of refactoring.

Another division scheme than division by regular grid can be used. With our reconstructed tree hierarchy a scheme more compliant with tree hierarchy can be used. We expect that this will have a primary effect on allowing easier and more realistic animation of trees by wind.

For bigger points creation a point color average can also be computed in more perceptually linear space such as CIE LUV space. Maybe even adding a little color randomization can help to compensate lack of complex lighting.

The size of bigger point can be also computed by enhancing union square area computation algorithm 6 to compute union of cube volume instead and thus taking into account a view of tree from more sides. A tree with big differences of side lengths of its axis-aligned bounding box will benefit from this improvement most. Algorithm 7 shows the modified version of algorithm 6 in pseudo-code.

4.5 Creating forests

This section discusses our forest solution. Forest representation structure is described in subsection 4.5.1. Forest generation utility is then overviewed in subsection 4.5.2 followed by detailed explanation in subsection 4.5.3. Finally possible improvements are considered in subsection 4.5.4.

4.5.1 Forest representation structure

Our forest representation is used to represent a whole forest of trees. Forest representation heavily rely on previously described LOD tree representation node structure. In fact forest representation can be viewed as extension of LOD tree representation with added instantiation support for various types of trees stored in LOD tree representation. Whole forest stored in forest representation can be viewed by our viewer application. We'll now describe forest representation structure along with its instantiation support provided by **instance info** data structure.

Algorithm 7 Union cube volume computation

Cubes - Array of cubes for which to compute union cube volume

ResultingVolume - Resulting union cube volume

procedure UNIONCUBEAREA(Cubes, ResultingVolume)

Let ResultingVolume = 0

Let QUEUE be event queue

Let H be height list

Let D be depth list

Add cube's starting X positions to QUEUE as starting events

Add cube's ending X positions to QUEUE as ending events

Sort QUEUE by X in ascending order

for all Events E in QUEUE **do** **if** E is starting event **then**

Add cube's minimal and maximal position on axis Y into H

Add cube's minimal and maximal position on axis Z into D

else if E is ending event **then**

Remove cube's minimal and maximal position on axis Y from H

Remove cube's minimal and maximal position on axis Z from D

end if

Find MAXY – maximal Y in H

Find MINY – minimal Y in H

Find MAXZ – maximal Z in D

Find MINZ – minimal Z in D

 Add $(MAXZ - MINZ) \cdot (MAXY - MINY) \cdot (X - PREVX)$
 to ResultingVolume **end for****end procedure**

Forest representation structure Forest representation structure is defined as follows:

Root node Pointer to root node of whole hierarchy.

Instance table Array with filenames of trees saved as LOD tree representation files

LOD tree representation node structure is used for nodes in forest representation. Nodes in current implementation don't carry any data for rendering nor distance data, but only axis-aligned bounding boxes used mainly for frustum culling. Special **instance info** is carried in data for tree instantiation into forest.

Instance info data structure Instance info data structure is defined:

Instance type Type of instantiated object. Can be FOREST for forest representation or LODTREE for LOD tree representation, but only LODTREE is fully supported now.

Instance type number Entry identification for **Instance table**. This describes what to be instantiated.

World matrix World matrix storing instantiation transformation of instantiated object.

Forest representation can be saved into and loaded from hard drive as binary file. Usually used filename extension is *.Forest.

4.5.2 Forest generation utility overview and usage

We have developed forest generation utility called CreateForest in order to automatize tree placement into forest. The CreateForest utility takes as input forest definition, tree definitions, and LOD tree representations and returns forest representation as an output. The method used for tree positions generation is variant of Poisson sampling where trees cannot be overmuch overlapped.

We now present usage of CreateForest utility. CreateForest utility is command-line utility with following usage:

CreateForest [-c ConfigFilename] [-o OuputFilename] ForestDefinitionFilename

The parameter meaning is:

-c ConfigFilename Name of configuration file for application. If not entered config.ini is used as default configuration file.

-o OutputFilename Output filename of forest tree representation. Usually with extension *.Forest. If not entered forest definition filename is used as output filename with extension *.Forest.

ForestDefinitionFilename Forest definition filename. Forest definition file is in form of configuration textual file and is explained below. Usually decorated with *.ForestDef extension.

Parameters meaning for configuration file of CreateForest utility is:

Method Method for tree placement into forest. Currently only supported method is Poisson sampling with allowed amount of tree axis-aligned bounding boxes intersection. Must be set to 1.

AllowIntersection Allowed amount of intersection between axis-aligned bounding boxes of two trees. If the percentage amount of intersection volume compared to candidate tree axis-aligned bounding box volume is smaller or equal than **AllowIntersection**, tree position is accepted and tree is placed on generated position. Otherwise the tree placement is rejected and process of generating tree position is repeated. This can be floating point number from interval $[0, 1]$. Zero means that no amount of overlapping is acceptable between trees. One means that any amount of overlapping is acceptable between placed trees.

NumberOfRetries Number of retries for tree position generation when tree placement is rejected because of unallowed intersection amount with already placed trees into the forest. This can be zero or positive integer number.

The forest definition file specifies parameters for forest generation such as forest size and tree types used. The forest definition file is textual configuration file with same structure as utility configuration file. Allowed keys, values and their meaning will be now described.

Forest definition file Forest definition file is described:

Width Width of the forest in meters. Can be positive integer number.

Depth Depth of the forest in meters. Can be positive integer number.

NodesPerNode Maximum number of tree instances in one node or maximal number of nodes in higher level node. This is used as limit for subdivision of hierarchy nodes. Can be positive integer number.

TreesPath Basic path to tree data folder structure. This can be string and should contain valid path to folder present on hard drive.

Tree0 – **TreeN** Filenames of tree definitions containing definition of types of trees used for forest creation. This starts with **Tree0** key and continues incrementally with keys **Tree1**, **Tree2** and so on until all tree definition filenames are specified. For example forest definition with four tree definitions will have **Tree0**, **Tree1**, **Tree2** and **Tree3** keys used. String values are associated with keys.

The tree definition file contains definition of tree type for forest creation. This includes LOD tree representation used for tree type, density of tree type in forest and basic transformations. The tree definition file is textual configuration file with same structure as utility configuration file.

Tree definition file Tree definition file is described:

TreeFile Filename and path to LOD tree representation used for this tree type. **TreesPath** configuration value is used as basic path to tree data.

TreeFile can contain path relative to basic tree path. This can be string value.

Density Density of trees in number of trees per 10000 square meters. Note that requested number of trees may not fit into forest and so less trees can be placed into forest. This can be positive floating point number.

SizeDeviationXMin Minimal scale of tree along axis X. Tree scale along axis X is randomly chosen between **SizeDeviationXMin** and **SizeDeviationXMax**. This can be positive floating point number.

- SizeDeviationXMax** Maximal scale of tree along axis X. This can be positive floating point number.
- SizeDeviationYMin** Minimal scale of tree along axis Y. This can be positive floating point number.
- SizeDeviationYMax** Maximal scale of tree along axis Y. This can be positive floating point number.
- SizeDeviationZMin** Minimal scale of tree along axis Z. This can be positive floating point number.
- SizeDeviationZMax** Maximal scale of tree along axis Z. This can be positive floating point number.
- BaseRotationX** Basic tree rotation along axis X in degrees. This can be floating point number.
- BaseRotationY** Basic tree rotation along axis Y in degrees. This can be floating point number.
- BaseRotationZ** Basic tree rotation along axis Z in degrees. This can be floating point number.
- BaseTranslationY** Basic tree translation along axis Y. Note that basic tree translation along axis X and Z doesn't have sense, because tree positions in forest are randomly generated. This can be floating point number.

4.5.3 Forest generation utility

The CreateForest forest creation utility works in following steps:

1. Loads configuration file, forest definition file and appropriate tree definition files.
2. Loads all LOD tree representations specified in tree definition files.
3. Computes basic transformation matrix for every tree type.
4. Computes axis-aligned bounding box of every tree type transformed by basic transform matrix.
5. Generates valid tree sizes and positions in forest.
6. Creates instance info data structure for each placed tree.
7. Creates LOD tree representation node's hierarchy carrying all trees instantiation information.
8. Creates and saves forest representation into file.

First two steps are apparent. Third step computes basic transformation matrix for each tree type from **BaseRotationX**, **BaseRotationY**, **BaseRotationZ** and **BaseTranslationY** values of tree definition file. This allows user to reorient each tree type data. Transformations are applied in this exact order:

1. Rotate along axis X.
2. Rotate along axis Y.
3. Rotate along axis Z.

4. Translate along axis Y.

An axis-aligned bounding box is then to be recomputed for trees transformed by basic transformation matrix. These bounding boxes are used in next step of tree positions generation for tree overlapping amount computation.

Tree positions and scales generation starts with computation of expected number of trees to be placed into forest. This is computed from forest sizes determined by **Width** and **Depth** configuration parameters and each tree type density determined by **Density** configuration parameter. Then for each tree type, starting with tree type zero, an attempt is made to place expected number of trees into forest. A random position of tree is computed along with random scale in given maximal and minimal bounds denoted by **SizeDeviation** configuration parameters of tree definition. Appropriate tree type axis-aligned bounding box is adjusted to generated tree's scale. Adjusted tree's axis-aligned bounding box is then searched using quad tree data structure for overlapping with other axis-aligned bounding boxes. If there is overlapping with some bounding box, an amount of overlapping is computed as ration of overlapped volume to adjusted axis-aligned bounding box volume. If an amount of overlapping is higher than **AllowIntersection** utility configuration parameter, then process of position and scale generation is repeated number of times specified in **NumberOfRetries** configuration parameter. When there is no or acceptable amount of overlapping, tree position and scale is stored and adjusted axis-aligned bounding box is added into quad tree to be used for overlapping test of trees. Process of tree position generation is showed in algorithm 8.

Because of trees placement generation method we advise user to specify tree types with low density first in order to assure that all low density trees will be placed. For mixed trees with approximately same tree type densities it's left on the user to adjust densities accordingly to get expected resulting forest trees distribution.

Then for each tree position LOD tree representation node with instance info is created. Nodes are then subdivided by quad tree into groups with maximal number of nodes determined by **NodesPerNode** configuration parameter. For each group higher level node is created carrying nodes with instance info as its children. Axis-aligned bounding box is computed for all nodes to be used mainly for frustum culling. Higher level nodes are then also subdivided by quad tree into groups and parent nodes created for them. This process is repeated until only one node is created as top of hierarchy.

Forest representation is created having top hierarchy node in its root. Instance table is filled with LOD tree representation filenames and relative paths for each tree type. Finally forest representation is stored into file.

The CreateForest utility memory requirements are proportional to size and density of forest and size of LOD tree representations used. The utility running time is low and even forests with 200 000 trees can be generated in matter of tens of seconds.

4.5.4 Possible improvements

We don't support any random rotation of tree in some interval. This can be added to make forest look even more natural. Also support for tree elevation by terrain high isn't currently implemented. This is because the issue with

Algorithm 8 Trees positions and scale generation

TreeTypes - Array of tree types containing density and scale bounds

Boxes - Array of axis-aligned bounding boxes of tree types

PositionsAndSizes - Resulting generated tree positions and sizes

procedure PLACETREESINTOFOREST(TreeTypes, Boxes, PositionsAnd-
Sizes) **for all** TreeTypes **do**

Compute expected number of tree types

end for

Let Q be quad tree of axis-aligned bounding boxes

for all TreeTypes and expected number of tree types **do**

Generate random tree position and scale

Adjust axis-aligned bounding box according to scale, name it A

Search Q for overlapping with A

Let B be axis-aligned bounding boxes from Q overlapping with A

for all Axis-aligned bounding boxes C in B **do**

Compute intersection volume VI of C with A

Compute volume V of A

if $\frac{VI}{V} >$ Allowed amount of overlapping **then**

Repeat position and scale generation specified number of times

end if **end for**

Store tree position and scale into PositionsAndSizes

Insert A into Q

end for**end procedure**

terrain rendering on forest sizes as large as 100 square kilometers isn't easy one. While large area terrain rendering is covered quite good in literature, it's not an easy implementation task to do and our estimation shows that it can take two weeks or more to implement terrain rendering solution into diploma thesis. It was never worth the time. Because of that it was pointless to add support for terrain elevation into CreateForest utility.

Tree positions generation method can be improved by adding an option to specify places where trees don't grow such as on forest glades or dusty roads. Different method can be implemented to support plantation or park trees placement based on the fact that trees are often planted in rows or alleys. Support for tree positions input list can be also added to provide a way to specify directly look of the park or garden.

4.6 Rendering engine

This section 4.6 introduces our forest rendering solution. The Viewer application is overviewed in 4.6.1 and explained in detail in next subsection 4.6.2. Rendering engine with impostor system is then separately described in subsection 4.6.3. Finally possible improvements are discussed in subsection 4.6.4.

4.6.1 Viewer overview and usage

We have implemented our rendering solution as an application called Viewer. The Viewer application provides hardware accelerated rendering of our representations. For simple tree representation the Viewer offers pure viewing capability of tree with no extra options. For LOD tree representation viewing capability of tree with hierarchical view-dependent pseudo-continuous level of detail is offered with additional directional lighting option. For forest representation the Viewer application supports forest viewing capability with level of detail, directional forest lighting, frustum culling and impostor system rendering acceleration.

From implementation view-point the Viewer application is based on OpenGL for rendering using GLUT (OpenGL Utility Toolkit written by Mark Kilgard). The GLUT allows an easy usage and initialization of OpenGL with a degree of platform independence. While GLUT consists of purely functions, we have wrapped its necessary functionality to object-oriented model or classes, which can be denoted as the core of the Viewer application. Other important parts are resource management system, rendering engine, impostor system, animation and statistics system. Our own resource management system handles loading of data and preparatory steps before rendering. The rendering engine renders trees and forest with the help of shader programs running on graphic card. The NVIDIA CG Toolkit is used as framework for shader programs. The impostor system speeds up rendering by rendering distant parts of the forest into images and displaying them in the scene as billboards. Animation and statistics system is used for scene fly-through and statistical data gathering.

The user control for the Viewer application is simple. Holding left mouse button moves camera position forward along view direction. Holding right mouse button moves camera position backward along view direction. Moving mouse to left or right rotates view direction along its Y axis. And moving mouse upward

Action	Result
Left mouse button	Moves camera forward
Right mouse button	Moves camera backward
Move mouse left or right	Rotates camera along its Y axis
Move mouse up or down	Rotates camera along its X axis
ESC key	Exits application

Table 4.2: Application controls.

or downward rotates view direction along its X axis. The escape key exists application. The table 4.2 summarizes application controls.

We'll now describe usage of Viewer application. The Viewer application is command-line application with graphical window interface with usage:

Viewer [-c ConfigFileName] ViewFileName

With parameter meaning:

-c ConfigFileName Configuration filename containing application configuration. If not specified config.ini is used as default configuration filename.

ViewFileName Filename to be viewed using Viewer application. This file can be simple tree, LOD tree or forest representation.

The configuration file is the textual file containing application configuration in the same way as for ConvertToLODTree and CreateForest utilities. Because this configuration file is rather long, we divide its keys and values description in several paragraphs related to concrete parts of Viewer application. We believe this will lead to better orientation in waste number of configuration options.

Configuration options related to Viewer application core:

ScreenWidth Screen width in pixels. This can be positive integer number.

ScreenHeight Screen height in pixels. This can be positive integer number.

MovementSpeed Speed of movement in meters per second. This can be positive floating point number.

FullScreen Whether to run application in fullscreen or not. This can be one for fullscreen mode and zero for windowed mode.

ShowMouseCursor Whether to show mouse cursor. This can be one for shown mouse cursor and zero for hidden mouse cursor.

Configuration options related to resource management system:

TreePath Basic tree path. Specifies directory base for tree data loading.

Configuration options related to rendering engine:

MinPointSize Minimal rendered point size on screen in pixels. This relates to metric for rendering. This can be non-negative floating point value.

MaxPointSize Maximal rendered point size on screen in pixels. This relates to metric for rendering. This can be non-negative floating point value.

MinGeometrySize Minimal rendered geometry size on screen in pixels. This relates to metric for rendering. This can be non-negative floating point value.

MinTreeLevelPointSize Minimal rendered point size on screen in pixels for whole tree node rendering. This relates to metric for rendering. This can be non-negative floating point value.

- MaxTreeLevelPointSize** Maximal rendered point size on screen in pixels for whole tree node rendering. This relates to metric for rendering. This can be non-negative floating point value.
- FarPlaneDistance** Distance to far plane in meters. This is maximal viewing distance. This can be positive floating point number.
- LightingMethod** Lighting method used for lighting of LOD tree representation and forest rendering. This can be one for ambient lighting only and two for diffuse directional lighting with ambient light.
- AmbientR** Amount of ambient light for red component. This can be floating point value in range [0, 1].
- AmbientG** Amount of ambient light for green component. This can be floating point value in range [0, 1].
- AmbientB** Amount of ambient light for blue component. This can be floating point value in range [0, 1].
- UsePerTreeAmbient** Whether to use per tree ambient randomization. When enabled this causes to have slightly randomized ambient value for each tree. The amount of randomization is specified by **AmbientRandomAmount** configuration key. This can be one for enabled and zero for disabled.
- AmbientRandomAmount** The amount of per tree ambient value randomization. This specifies interval with center in ambient light color value. This can be non-negative floating point number.
- DiffuseR** Amount of diffuse light for red component. This can be floating point value in range [0, 1].
- DiffuseG** Amount of diffuse light for green component. This can be floating point value in range [0, 1].
- DiffuseB** Amount of diffuse light for blue component. This can be floating point value in range [0, 1].
- LightDirectionX** The X component of light direction. This can be floating point value.
- LightDirectionY** The Y component of light direction. This can be floating point value.
- LightDirectionZ** The Z component of light direction. This can be floating point value.
- BackgroundR** The red component of background color. This can be floating point value in range [0, 1].
- BackgroundG** The green component of background color. This can be floating point value in range [0, 1].
- BackgroundB** The blue component of background color. This can be floating point value in range [0, 1].
- UseGround** Whether to render ground. This can be one for rendering with ground or zero for rendering without ground.
- GroundTexture** The name of filename with ground texture. This is string.
- GroundColorR** The red component of ground color. This can be floating point number in range [0, 1].
- GroundColorG** The green component of ground color. This can be floating point number in range [0, 1].
- GroundColorB** The blue component of ground color. This can be floating point number in range [0, 1].

Configuration options related to impostor system:

- UseImpostorSystem** Whether to use impostor system or not. One for enabled and zero for disabled.
- ImpostorTextureWidth** Width of impostor texture atlas in pixels. This can be positive integer number.
- ImpostorTextureHeight** Height of impostor texture atlas in pixels. This can be positive integer number.
- ImpostorWidth** Width of single impostor in pixels. This can be positive integer number.
- ImpostorHeight** Height of single impostor in pixels. This can be positive integer number.
- ImpostorMinimalDistance** Minimal distance from node's axis-aligned bounding box to allow impostoring of node. This is in meters and can be positive floating point number.
- ImpostorRefreshNumber** A number of impostors to refresh. This can be positive integer number.
- ImpostorRefreshInterval** Interval between impostor refresh steps in milliseconds. In each impostor refresh step a number of impostors specified in **ImpostorRefreshNumber** is refreshed. This can be non-negative integer number.
- ImpostorUpdateInterval** Interval between impostor update steps in milliseconds. During impostor update step impostors are created and destroyed. This can be non-negative integer number.
- ImpostorHierarchyMaxDepth** The maximal depth from the top of the node's hierarchy to allow for impostoring. This limits node candidates for impostoring. The zero means that only top node can be impostored, one means that top node and its children can be impostored and so on. This can be non-negative integer value.

Configuration options related to statistics and animation system:

- UseAnimationSystem** Whether to use animation system for automatic scene fly-through. This can be one for enabled or zero for disabled.
- AnimationSystemFileName** Filename with fly path definition for animation system. This can be string value.
- LoopAnimation** Whether to loop animation. When this is enabled, the animation is played from start after its ending. This can be one for enabled or zero for disabled.
- TerminateAppAtEndOfAnimation** When enabled application is exited at the end of animation. This can be one for enabled or zero for disabled.
- UseStatisticalSystem** Whether to use statistical system for statistics logging. This can be one for enabled or zero for disabled.
- StatSystemSavePath** Path to directory where statistical results should be saved.

The fly path definition for animation system is line oriented text file with two allowed line formats:

- X Y Z Azimuth Ascendent Roll Time** Defines camera position, rotation and time. The X, Y and Z determines camera position. The Azimuth, Ascendent and Roll in degrees determines camera rotation where north for azimuth is in Z+ axis. Roll is about direction determined by azimuth and ascendent. Time is animation time in seconds and must be entered in non-decreasing order for line entries. The X, Y, Z, Azimuth, Roll and Time can be floating point values.

; **Commentary** Line starting with apostrophe is commentary.

4.6.2 Viewer

The Viewer application work can be summarized as follows:

1. Loads configuration and initializes.
2. Loads viewing data by resource management system.
3. Prepare viewing data for rendering.
4. Enters render loop and renders scene using rendering engine with or without impostor system.
5. Deallocates data and exits.

First configuration parameters are read. Then window or fullscreen mode is initialized along with OpenGL, DevIL library for texture loading and NVIDIA CG framework for shader support. Vertex and pixel shader programs for graphic card are loaded from files and compiled for use. If impostor system, animation system or statistics system is in use, they are also initialized. Then data are loaded using resource management system.

Our resource management system handles all data loading and deallocation. First data types are registered as callback functions for data loading and deallocation. Then resource management system can be called in order to load specific data of registered data type regardless the data are yet loaded or not, because resource management alone handles data multiplicity. This prevents from having same data multiple times in memory and just saves memory. The main advantage is that once types are registered and loading and deallocation functions written, there is no need to care about data in any way. The resource management system cares for loaded data instead.

The data are loaded into memory from file and then converted to format used for rendering. This includes allocating OpenGL textures in graphic card's memory, storing textures there, allocating Vertex buffer objects for vertices, and indices data and storing geometry data there. The vertex buffer object is OpenGL extension for geometry data storage in the memory of graphic card. For simple tree representation loading process is straightforward by only loading data from file and then converting them into rendering representation storing textures and geometry data in the graphic card memory. For LOD tree representation its same with the addition of change distance computation determining distance where to switch from mixed triangle and point data rendering into whole tree node point rendering. This computation is done according to metric and is explained in rendering engine subsection 4.6.3.

For forest representation not only forest hierarchy but also all trees from LOD tree representation must be loaded. Then using **Instance table** data from forest representation and instance info information from forest nodes all trees are connected as children of appropriate nodes with instance info. This is done in a way that a node with instance info can have only one children which is connected LOD tree representation hierarchy. By these connections the scene acyclic graph is formed.

When all data are loaded and application initialization is successful, a rendering loop is entered. When animation system is used, the camera automatically flies through the scene and user controls are suppressed except for application exit. When animation system isn't used, the user can view rendered scene and exit the Viewer application at any time. When statistics system is enabled, statistics with average rendered frames per second, lowest framerate per second, average number of points rendered, average number of triangles rendered and average number of nodes walked are written into uniquely named statistical text file along with application configuration at application exit.

4.6.3 Rendering engine with impostor system

Our rendering engine handles several steps to render resulting scene. First node's hierarchy is traversed in order to determine what nodes will be rendered. This includes hierarchical frustum culling and level of detail computation for nodes with geometry data. All nodes with some triangle or point geometry selected for rendering into scene fill in rendering request during node's hierarchy walkthrough. After all rendering requests have been gathered, they are sorted by certain criteria to speed up rendering. Then rendering itself occurs either with ambient only rendering path or diffuse directional light with ambient rendering path. We can summarize rendering process in steps as:

1. Walkthrough node's hierarchy. Determine what to render and fill in rendering request.
2. Sort rendering requests.
3. Render scene using sorted rendering requests.

This is view of rendering process without the impostor system. With impostor system the work of updating and refreshing impostored parts of the scene is added as well as impostor rendering. Impostoring is only available for forest rendering. With added impostor system rendering process looks like:

1. Update and refresh impostors.
2. Walkthrough node's hierarchy. Determine what to render and fill in rendering request.
3. Sort rendering requests.
4. Render scene using sorted rendering requests.
5. Render impostors.

The simple tree representation rendering is an exception, because it's rendered directly without the need to walkthrough hierarchy or sort rendering requests and just stays as separate part from the rest of the rendering engine. For simple tree representation a simple ambient shader program is used for rendering branches and leaves. Rendering follows standard steps of specifying shader program parameters, vertex format, texture and vertex buffer objects and then rendering using geometry draw calls. No extra steps are done.

For LOD tree representation and forest rendering the situation is different. As stated above first node's hierarchy is walked through. The node's bounding

boxes are tested for visibility against camera view frustum consisting of six planes. With every camera movement or change of its looking direction its view frustum planes are extracted along with eye position and stored for future use. They are then used for frustum culling in a way that for every frustum plane an effective radius of bounding box is computed and intersection with sphere with size of effective radius is performed against frustum plane. When node is not visible, its traversal is ended. When node is visible there can be several cases. When node is purely used for frustum culling such as node from forest hierarchy, node's traversal continues with its children. When node carries instantiation data, the transformation is accounted and its one child is traversed. Lastly when node carries data for rendering, a distance from eye position to the node is approximated using distance information, level of detail is calculated using metric and rendering request is filled.

Metric Our used metric calculates approximate screen space taken by triangle or point in the pixels on the screen. This is dependent on distance from the eye position, width and height of the screen and projection parameters. In fact from screen space area, triangle or point area and distance we can compute any one of these parameters having two other parameters as input.

We now present our metric in detail. Let $FOVY$ be vertical viewing angle, $WIDTH$, $HEIGHT$ width and height of the screen in pixels and N the distance of perspective projection center from near plane, we can compute perspective projection rectangle width W and height H using equation 4.1.

$$\begin{aligned} H &= N \cdot \tan \frac{FOVY}{2} \\ W &= \frac{WIDTH}{HEIGHT} \cdot H \end{aligned} \quad (4.1)$$

Then our metric constant C can be computed as in equation 4.2 where scaling from perspective rectangle to screen size is accounted along with perspective projection.

$$C = \frac{WIDTH \cdot HEIGHT \cdot N^2}{W \cdot H} \quad (4.2)$$

Let A be area of triangle or point, A_s area on screen in number of pixels and let D be directional distance from eye position to triangle or point position along view direction. Then we can compute A_s from known A and D as in equation 4.3, A from known A_s and D as in equation 4.4 and D from known A_s and A as in equation 4.5.

$$A_s = \frac{C \cdot A}{D^2} \quad (4.3)$$

$$A = \frac{D^2 \cdot A_s}{C} \quad (4.4)$$

$$D = \sqrt{\frac{C \cdot A}{A_s}} \quad (4.5)$$

For point data the level of detail is computed in the following way. The **point area** data are present in mixed triangle and point data structure or point data structure. The minimal and maximal screen area bounds are the Viewer application configuration parameters **MinPointSize**, **MaxPointSize** for mixed

triangle and point data and **MinTreeLevelPointSize**, **MaxTreeLevelPointSize** for point data. The points with screen area between minimal and maximal screen area bounds are displayed. For node this involves computation of point maximal and minimal point area from node's directional distance and maximal and minimal screen area. The points with point area between computed range are then searched using binary range search algorithm in sorted **point area** array.

For triangles the situation is similar. All triangles with screen area bigger than **MinGeometrySize** application configuration parameter are displayed. For node this involves computation of minimal triangle area from directional distance to node and **MinGeometrySize**. Then all triangles bigger or equal to minimal triangle area are searched using binary range search algorithm in sorted **geometry area** array. When some geometry falls into the interval for displaying, a render request is filled and stored for upcoming scene rendering.

Change distance issue There is one last issue with node's hierarchy traversal we have talked about in LOD tree representation section 4.4. When node's distance is bigger than change distance, only whole tree node point data are rendered instead of its children node's data and no children nodes of whole tree node are traversed. This is because traversing too many nodes makes application processor bounded and just slows down rendering speed and also because whole tree node contains bigger points which can be used for distant rendering. The change distance is computed as maximum of two values. First value is computed as distance of smallest point area in whole tree node to take screen space of **MinTreeLevelPointSize** configuration parameter. Second value is computed from additional node's information containing area of largest triangle which can be omitted from rendering after it's switched only to point rendering. This second value is distance computed using metric from additional information triangle area and **MaxTreeLevelPointSize** configuration parameter. This is very important thing in whole node's hierarchy traversal because it reduces traversal time substantially. The bigger number of nodes is used for tree, the bigger is the traversal time reduction.

Rendering After all rendering requests are filled and stored, they are sorted by following criteria:

1. Triangles before points.
2. Leaves before branches.
3. Different vertex buffer objects.
4. Different textures.
5. Distance in ascending order.

It's difficult question what is the best order because some criteria are contradictory to other. We have however selected this order of criteria for smallest number of vertex buffer object and texture change calls and then for reduced redraw of scene pixels by partially sorting geometry in front to back order.

Sorted rendering requests are then processed one by one. This is done by one of two rendering paths. First rendering path uses only ambient lighting. Second

uses directional diffuse lighting with ambient component just approximating sun lighting. For ambient only lighting path two things are interesting. Point size must be attenuated by distance in shader program in order to get right point size on the screen. This is done by computing attenuation constant from node's distance, perspective projection parameters and screen size. Because of bad visual appearance of distant trees which have all same color and so there was no detail apparent in distance, we have used a cheap trick to resolve this issue. We have assigned each tree a slightly different ambient lighting color. It leads to differentiation of trees in the distant parts of forest and just a viewer can see tree silhouettes. The **AmbientRandomAmount** configuration parameter specifies how much are ambient values randomized for trees in the way that the ambient lighting color is in the center of interval specified by **AmbientRandomAmount** configuration parameter.

For directional diffuse lighting with ambient path there is only one thing worth mentioning. We have used only one-sided leaves triangles representation and because of that we have only one normal for one leaf side. For the other side of leaf the normal must be flipped in order to correctly compute directional lighting for that side. This flipping is done in shader program by computing signed distance of eye position to the plane defined by triangle's normal. This is done in view space where eye position is in the center of coordinate system and just computation is simplified a little bit. We note that testing in view space whether triangle's normal has positive or negative Z component isn't a right way to determine whether to flip normal or not because of perspective projection used for viewing of scene.

Impostor system The impostor system consists of several parts. At application initialization top hierarchy nodes are taken as impostor candidates. During rendering loop impostors must be updated, refreshed and rendered. The update step consists of planning what nodes to impostor from impostor candidates and what impostors to delete. This is followed by impostor removal and creation. The refresh step renew impostor positions and images and rendering step renders impostors to the scene.

Our impostor system approach is novelty because our impostor system is hierarchical. We therefore build impostor candidate tree at application initialization. The depth of tree is controlled by **ImpostorHierarchyMaxDepth** configuration parameter specifying maximal depth of nodes from the top of the node's hierarchy to be taken as impostor candidates. We treat top hierarchy node as level zero, its children as level one and so on. This means that value of one in **ImpostorHierarchyMaxDepth** configuration parameter takes top hierarchy node and its children as impostor candidates.

The impostor candidate tree is built in depth linearized fashion. This means that impostor candidates are organized in a way to allow easy walkthrough through impostor candidates with same depth level. In fact impostors with same depth level are stored in one array. This however doesn't mean that tree fashion has gone. The impostor candidates from arrays of different depth levels are interlinked in parent-children manner. We have chosen this data organization because it's best for our hierarchical impostor planning algorithm.

The planning is most difficult task for our impostor system in algorithmical sense. The task is to use impostor atlas texture which is holding impostor

images in an efficient way. We define that we want to plan impostors in a way that, if node is impostored, no sibling of this node have impostor. This is an invariant we want to hold in all cases. And the meaningful one because we don't waste an impostor's texture atlas space with impostor images which will not be in fact used. The impostor candidate can be impostored if the metric allows it. We also want to impostor higher level nodes of node's hierarchy if it is allowed by metric. To achieve this we prioritize such nodes over lower level nodes, but this isn't an invariant. The planning process therefore selects higher level nodes for impostoring first and then lower level nodes if there is enough space left in impostor atlas texture. We must also remove all impostors which are disallowed by metric usually because they are too close to camera. We therefore summarize our planning efforts in following way:

- Prioritize higher level nodes to be impostored before lower level nodes.
- Don't allow siblings of impostored node to be impostored. (Invariant)
- Remove impostors which are disallowed by impostor metric.

We now describe our planning algorithm. The impostor candidates are walked one by one starting from highest level to the lowest level. If the impostor candidate isn't impostored and metric allows its impostoring, then the impostor candidate is added into possible creation list and its siblings must be walked in order to remove impostored ones. If an impostored sibling is found, it's added into removal list with mark that its removal is dependent on creating specific higher level impostor. If the impostor candidate is impostored, metric is checked for that candidate. If metric disallows this candidate to be impostored, it's added into removal list, otherwise nothing is done.

By this way removal list is assembled. The creation list is then assembled from possible creation list. All impostor candidates which causes to remove some impostored sibling are automatically added because they paid for itself. For the other candidates on the possible creation list the higher level ones are taken before the lower level ones until there is free place for them on impostor texture atlas. The algorithm 9 shows impostor planning algorithm in pseudo-code.

After removal and creation steps are planned, the removal step takes place. This is straightforward. All impostors from removal list are removed and it's space on the texture atlas is marked as free. Then comes the creation step where for all impostor candidates from creation list impostors are created and space for them is allocated on the texture atlas. All created impostors are then immediately refreshed to obtain their image and billboard shape in the forest scene.

The metric used for impostor system is two part. First it's tested that distance of camera from impostor candidate node is bigger than minimal distance specified by **ImpostorMinimalDistance** configuration parameter. Then the area of rectangle is computed from node's axis-aligned bounding box corner points projection onto plane going through center of axis-aligned bounding box and being perpendicular to direction going from eye position to axis-aligned bounding box center. Using same metric as for level of detail computation the rectangle area is projected onto screen and compared to impostor size given by **ImpostorWidth** and **ImpostorHeight** configuration parameters. If the rectangle screen area is smaller than impostor area size, the impostor candidate is allowed to be impostored, otherwise it's not allowed.

Algorithm 9 Impostor planning algorithm

ImpostorCandidates - Impostor candidate nodes for impostoring

RemovalList - List of impostors to be removed

CreationList - List of impostors to be created

procedure PLANIMPOSTORS(ImpostorCandidates, RemovalList, CreationList)

Let C be possible creation list

for all ImpostorCandidates walked in depth level
 order from highest to lowest **do**

Let I be impostor candidate

if I is not impostored **then** **if** Metric allow impostoring of I **then**

Add I into C. Search siblings of I for impostors.

Add them into RemovalList.

Mark them as dependent removal on I.

end if **else if** I is impostored and metric doesn't allow impostoring **then**

Add I into Removal list.

end if **end for** **for all** Impostor candidates I in C with marked dependent removal in RemovalList **do**

Add I into CreationList

end for **while** There is free impostor space **do**

Add highest level impostor candidate I from C into CreationList

Remove I from C

end while**end procedure**

The impostor refresh assures that visual appearance of impostored nodes is correct while the camera is moving. We decided to refresh specified amount of impostors by round robin scheme at the end of some time interval. The **ImpostorRefreshNumber** specifies the number of impostors and **ImpostorRefreshInterval** specifies the time interval. We found that it's better to refresh small number of impostors more often than large number of impostors less often. This prevents from application stall during refresh step. We also found that update step doesn't need to be done as often as refresh step and so we differentiated update and refresh step intervals. Our implementation however do refresh with every update step which was easier to code. This means that there can occur separate refresh step or update with refresh step in our implementation. The **ImpostorUpdateInterval** configuration parameter specifies the time interval between two impostor updates.

There are two things which must be done in refresh step for each impostor refreshed. The billboard position and rotation in the scene must be computed and actual image of forest from current eye position must be taken. The billboard position and rotation is computed from node's axis-aligned bounding box. The corner points of axis-aligned bounding box are transformed into projection space, the axis-aligned bounding rectangle is computed from transformed corner points and transformed back to world space where it's used as billboard position. Then metric is adjusted according to impostor size and image is rendered into the impostor texture atlas by rendering impostor's node and its siblings by rendering engine. We don't move or rotate billboard position elsewhere. We assume that refresh step is taken often enough for billboards to have approximately correct position and rotation. The impostor rendering is straightforward and just displays impostor billboards.

Implementation and rendering method errors We'll now discuss our implementation and method visual appearance errors because nothing is errorless. Sometimes one or more big points appear on the rendered tree model. We claim that is our implementation error. This can be corrected by recreating LOD tree representation few times and observing resulting rendered tree for occurrence of big points. We haven't yet discovered what is causing this error and because of it no more information can be given to this error.

Sometimes holes may appear in points rendered tree as we discussed in section 4.4. This is because points may badly approximate triangle shape and it happens mainly with small branches. This is more error of used method than implementation issue, but can be solved with points having a size of triangle's circumscribed circle area. In our implementation this is good solution for some trees while for the other it's not optimal. This is also discussed in section 4.4 and depends on the way the LOD tree representation is created.

The trunk from tree in the distance may also disappear. This is caused by LOD tree creation method. The reason why it's happening is in fact that trunk has limited number of points in comparison with the rest of the tree. By point averaging scheme the bottom of the trunk is shifted upward for bigger points and just there are no bigger points for the bottom of the trunk.

There can be points flying separately from the tree. This is caused by random point position generation which can be generated on the places where tree leaf texture is transparent. This is discussed in section 4.4 and it's our imple-

mentation issue.

Tree visual appearance in far distance is insufficient. This is caused by small number of bigger points for far away tree rendering. Depending on level of detail metric setting and tree size the point representation is sufficient to distance from 2000 to 3500 meters. Exceeding this distance will cause nothing to be rendered. We e-mailed with authors of *Point-based rendering of trees* [11] on behalf of this issue and they also admitted that even using point representation for couple of trees produces bad visual quality of far trees. We therefore claim that it's point-based rendering technical issue which wasn't addressed adequately yet.

The point representation is noisy during movement. This is visible on colorful trees. We suspect that this is caused by the way the bigger points are created. The authors of *Point-based rendering of trees* [11] don't mention this, but maybe they don't test their rendering method enough. We however have done testing on numerous tree types and because of it we are of opinion that this is also used method drawback.

White points may appear on the tree. This is caused by approximating transparent leaf texture where transparent points are usually in white color with full transparency. If the border of texture is correctly edited, white point rarely appears. But when resizing of big leaf texture is done to smaller one, the left border is damaged and white points may appear on the tree. This can be always fixed by choice of right texture size and manually editing border of the leaf on the texture.

These are the errors we are aware of, but we don't claim that other errors may not occur. We are however confident that these errors don't lessen or somehow disturb the resulting scene in a way that is crucial for gaining look and feel of the used method and its speed measurements. Every method usually has its own unique look and feel. For our method the look and feel is little bit like in fairy tale images.

4.6.4 Possible improvements

The problem with movement from fixed function pipeline rendering to shader programming is in increased difficulty to make fast and reliable rendering engine. One might think that using such library as NVIDIA CG for shader programs handling is enough, but it's naive approach to rely only on basic functionality NVIDIA CG is offering. The more shader programs are used the harder it becomes to manage them. One must load every shader program separately, fetch all of its constant parameters and set them separately before rendering. This becomes quickly annoying and it's the waste of time. In fact this practically means to program separate rendering path for every pair of vertex and pixel shader to provide effective means of rendering. This is the reason why we have two rendering paths in our Viewer application.

Because of that it's actually a very good idea to build a shader handling system upon NVIDIA CG. The simple rule of having same meaning for uniquely named constant parameters in all shader programs allows to build an effective shader handling system where every constant parameter can be set by using same handle for it to all shader programs using it. Or even better the constant parameters can be set only once to shader handling system and shader handling system can set them automatically to currently used shader program. Using such shader handling system makes shader programs much more easier to han-

dle. Let say we would like to add some shadows into our forest. With current state we are forced to write shader loading, constant parameter fetching, new rendering path and so on. With shader handling system shader program part is almost free and we can focus on implementing shadow solution solely.

The same it's with geometry data vertex format which is now fixed. It'll be great improvement to have more flexible way to set vertex format either from shader program used or from geometry data structures. Having more flexible data structures will be even better. This will simplify task of computing ambient occlusion term and adding it to vertex and point data. Setting aside that computing ambient occlusion term requires to program ray-casting system, the secondary most difficult task is to add computed ambient term value to data. With our fixed data structures it's simply not an easy task. Having more flexible data structures and rendering engine will help much. It's especially important when one realizes that lighting using ambient occlusion is probably one of the best ways to add cheap and realistically looking lighting to the forest and therefore the way to go to improve forest lighting.

The level of detail computation can be speeded up by precomputing tables of point and triangle intervals to render based on distance. We note that this lowers number of level of details for tree and may introduce more popping to the scene.

Our node's hierarchy walkthrough can be easily parallelized and just utilize multiple processor architecture which is becoming common on desktop computers. The rendering requests are independent to each other and because of it different threads can be assigned to walkthrough different parts of node's hierarchy. The good candidates are children of top level node. They can be assigned to threads by round robin scheme. Each thread can also sort its filled rendering requests to further put off computation burden from rendering. The sorted rendering requests from threads can be then merged together using merge part of merge sort algorithm. Because of the way we create our forest level node hierarchy, we have the tree which is balanced or almost balanced. We believe that this is almost ideal for parallelization and will speed up processor computations in optimal way. Because walking through node's hierarchy is the most computational expensive part of the Viewer application, we expect huge performance gain on processor side and we just get rid of processor bound limitation for rendering of the forest scene totally.

Questionable is whether to implement some sort of occlusion culling. It's true that recent graphic hardware supports occlusion queries natively, but it's not without cost. The result of occlusion query isn't available immediately. One must wait for the result some time and it's not clear what to do in that time on processor side. The naive approach of rendering near trees and then ask for occlusion query for each distant tree, wait for occlusion query result and based on the result decide whether to render tree or not, is simply not possible. We believe that it slows application in most cases, if not in all cases! Obviously the more dense is the forest the bigger is the chance to save rendering resources by using occlusion queries. Unfortunately this is only true for forest walkthroughs and not for flying above the forest where almost every top of the tree can be seen. For coniferous forest its also very questionable how far can one see through the dense forest. If the forest is old, it's not a rare case to see the bottom of tree trunks half or one kilometer far away. For such forests it would require to divide each tree approximately in the level where leaves start to appear to even

begin thinking about occlusion queries! Because of these reasons we strongly advise to implement occlusion queries only with some sort of mechanism which can detect whether occlusion queries bring performance advantage or not.

Another interesting problem is putting forest on the terrain. The algorithms for large scale terrain rendering exist, but they aren't cheap in terms of rendering power they take. The *Geometry clipmaps* [9] is probably best method to choose. We actually believe that this approach can solve problem with poor quality of distant trees. The idea is to render whole forest from top into one extremely big texture of for example 30000 x 30000 pixels and then generate chain of lower resolution textures from it. Then higher resolution textures are discarded and lower resolution textures starting for example from resolution 4000x4000 pixels are used for terrain texturing. This fits perfectly into geometry clipmaps terrain rendering algorithm where terrain is rendered as number of regular grid rings each assigned different resolution of terrain heightfield texture. As we need to start using forest textures in distance of 1.5 to 2 kilometers, we can omit high resolution textures and start using relatively low resolution forest textures for distant rings as terrain texture combined with ground texture. All textures can be generated in preprocessing step by some utility. To provide even better resolution, we can slice up the forest and render several slicing textures from the top in the same way as previous texture and then we can render far terrain several times lifted up by some amount according to slice height in which the slicing texture was taken. We note that by using such scheme the need for aperiodic tiling of forest parts is surpassed. We predict that the need for aperiodic tiling will be one of the reasons why developers will not use methods incorporating it in practice. This is often simply too limiting.

4.7 Forest motion

We'll discuss forest animation in order to get brief overview of potential extension of our implementation by animation system. We haven't implemented animation system because of the lack of time. It's relatively easy to extend our implementation by some simple tree animation system, but it becomes more challenging as more realistic looking animation or leaves animation is required.

The simplest way to animate a tree is to swing whole tree according to wind direction and intensity. This can be simulated with added whole tree transformation according to some noise function defining intensity and direction of the wind over time for whole forest. Only trees to specified maximal distance from the camera should be animated in order to cut down overall animation computation time and not to have problem with bad looking animation because of used impostor system. The impostor system is in fact not an animation compatible part. We expect that animating impostored parts of the forest will look unrealistically and disturbingly, the trees will suddenly change their positions when impostor refreshes. Also circle wind primitive can be added to simulate local strong wind conditions.

To achieve more realistically looking animation, there need to be some more tree structure information present. Unfortunately such information is not present in our current implementation. The only format having such information is our basic tree representation where reconstructed tree hierarchy is held. Probably the easiest way is to incorporate information about branch level

into vertex and point geometry. Then for every tree the transformation matrix for each branch level is computed by the Viewer application and set into vertex shader program. The vertex shader program determines which branch level has given vertex or point and applies right transformation matrix to vertex or point position. We note that matrixes must be created in a way that for higher level branches all lower level branches transformations must be also accounted, otherwise the tree will fall into parts during animation.

Another way, how to enable wind animation, is to make tree node hierarchy structure according to tree structure. This however means for our implementation to rewrite LOD tree representation utility `ConverToLODTree` in rather radical way. The advantage of this method is practically no additional geometry size cost and ability to handle wind animation solely by the Viewer application without the need to use shader programs. Also animation level of detail can be controlled with this method easily. As tree gets further not all tree node's hierarchy is walked and just lower animation level of detail is computed.

The hardest part is to achieve animation of the leaves. The paper *A hybrid method for real-time animation of trees swaying in wind fields* [17] achieves animation of the leaves for single tree. We therefore expect that it will be possible to animate leaves only for few closest trees to the camera. We prefer to use single vertex format representation for both static and animated leaves because otherwise memory requirements will be unacceptable. We expect that assigning each leaf a unique identification number and position of the leaf petiole should be enough to achieve some leaves animation, but acknowledgement of this expectation should be done by an effort to implement leaves animation.

Chapter 5

Results

This chapter discuss results of our approach. First measuring conditions are introduced in section 5.1. Then our test suite developed for measurement purposes is overviewed in section 5.2. Results are presented in a form of graphs in section 5.3 and finally discussed in section 5.4.

5.1 Measurement conditions

Before any measurement is taken, there is a need to discuss measurement conditions. We have decided to test on our computer with powerful graphic card. We believe that this will give more proper view on speed of our method for future use. The table 5.1 shows parameters of our testing machine.

We have included 3DMark 2005 score as overall information of our testing machine speed. The 3DMark 2005 is industry benchmarking professional solution and thus this score captures graphical power of testing machine in the most appropriate way. We hope that it becomes more usual to include such information in measurement conditions.

Our diploma thesis aims to achieve interactive to real-time graphical speed of our forest rendering solution. The real-time rendering speed is considered 30 frames per second (FPS) and above. However in most cases everything above 15 FPS is just enough to view resulting scene in relatively smooth way. We therefore threat range from 15 to 30 FPS as near real-time speed. We threat range from 8 to 15 FPS as interactive speed.

We have measured all our test forests in fullscreen mode and with framerate synchronization turned off in order not to be limited by monitor refresh rate. The framerate synchronization was turned off in driver settings. As it's stated

Processor:	AMD Athlon 64 3800+
Graphic card:	NVIDIA GeForce 8800 GTX with 768 MB RAM
RAM:	2 GB
Motherboard:	ASUS A8N-SLI DELUXE
Operating system:	Microsoft Windows XP Service Pack 2
3DMark 2005 score:	10834

Table 5.1: Parameters of computer for results measurement.

in subsection 4.6.2, we have incorporated animation and statistical system into our Viewer application. This ensures that results are correctly measured under same condition for every Viewer application setting. We have however done measurement only once mainly because it takes several hours to complete our test set. We still believe that taking measurements only ones doesn't alter resulting data in any relevant way since graphical rendering power of computer is stable.

For FPS measurements we count every frame rendered and running time. Then we compute average FPS as division of counted frames by running time. In this way the exact average FPS is computed. The minimal FPS is taken as the lowest FPS counted by our application. There can be some variation however because our application waits until time counter exceeds one second and then computes FPS by dividing frame rate count by elapsed time which can be slightly bigger than exact one second. We also count average triangles and points rendered. This is also done by counting total number of triangles or points rendered during application run-time and then dividing it by application running time. By triangles or points rendered we mean number of triangles or points sent to graphic card for drawing by OpenGL draw calls. The last measured parameter is average number of hierarchy nodes walked. This includes every walked node of scene acyclic graph including forest nodes used for frustum culling, nodes with instantiation information and all tree nodes. When impostor system is turned on, the nodes walked, in order to create impostor images, are also counted.

5.2 Test suite overview

We have developed test suite in order to carry on with application rendering speed testing. The test suite is two part. First part is designed to test single tree LOD representation against simple tree representation. Second part tests various aspects of forest rendering.

We decided not to show results from first part because it isn't interesting enough. The rendering of single tree is just fast enough and for both representations the performance is in hundredths of frames per second in every case. The only interesting fact is that our LOD tree representation outperforms single tree representation in every case and even for close look-ups. We expected that from some distance LOD tree representation will be faster, but we didn't expect that it will be so for close look-ups. But it showed that frustum culling had done its job for close look-ups well.

The second part consists of various tests for forest rendering speed. We'll now discuss second part in more detail. First series of tests is designed to test forest rendering speed according to five metric settings and three screen resolutions. Also performance of lighting paths and impostor system is evaluated. The table 5.2 shows metric settings for each detail level chosen. The testing screen resolutions are 800x600, 1024x768 and 1600x1200. We note that using for example high detail setting of metric for 800x600 screen resolution is not the same as using it for 1024x768. The quality of using high detail setting of metric is in fact higher for 1024x768 screen resolution. This is because of chosen metric which is dependent on screen resolution. We however do not recompute metric settings based on screen resolution. We just left quality to be higher for bigger

Detail level	Min point size	Max point size	Min triangle size	Min tree level point size	Max tree level point size
Extra high	1	5	5	1	5
High	1	5	5	4	10
Normal	5	10	10	7	15
Low	7	15	15	10	20
Extra low	10	20	20	15	30

Table 5.2: Metric settings for test suite.

resolutions in order to put more stress to forest rendering. According to our opinion for screen resolutions 800x600 and 1024x768 the normal detail metric setting has enough quality while for 1600x1200 low detail metric settings can be used without problem for scene viewing in practice.

We test for two scene animation paths: the walkthrough through the forest as someone walks or runs in the forest and flight above the forest. The animations are 50 and 55 seconds long. We have adjusted the animations to resemble real scenario of how will probably the user look at the forest. But because the animations are relatively short we tried to avoid not to look into forest even for short period of time. Our previous tests showed that even short look into the ground while walking in the forest can alter average FPS to become 20 or 30 frames higher. For forest flight above the minimum and average FPS is usually very different. This is because the animation starts with low altitude flight upon the forest while at the end of the animation the flight upon the forest is in much higher altitude. We therefore strongly advise the reader to take a look at animation fly paths for itself in order to make an idea about relevancy of our results for various viewing conditions.

Getting back to first series of tests we have chosen a poplar forest with density of 20 trees per 10000 square meters. This choice ensures enough visibility to the distance while maintaining relatively good speed in all of our tests. We test five detail metric settings for three screen resolutions and two fly paths resulting in 30 tests. This is done three times. Once for directional diffuse lighting path with impostor system turned on, once for directional diffuse lighting path with impostor system turned off and lastly for ambient lighting path with impostor system turned on. This yields total of 90 test runs for first series of testing. The purpose of this first test series is to test forest rendering performance in various detail and screen resolution conditions. Moreover the impostor system performance and the effect of different lighting paths is measured.

The second series of tests are designed to test forest rendering performance under various forest types from our vast tree library. Nine forests are tested for two fly paths at normal metric settings in 1024x768 screen resolution. The forests are created with density of 30 trees per 10000 square meters with three different ages of each tree type.

The next series aim to measure impact of memory consumption on forest rendering. Our most memory intensive tree model of falseacacia taking 82 MB of hard disk space is used six times in one to six copies for forests. The most memory intensive forest just takes approximately 492 MB in graphic card memory.

The fourth series of tests are designed to test forest rendering performance

depending on the forest size. We do this on four forest sizes of 4x4, 5x5, 10x10 and 20x20 km of forest and with four far plane distance settings starting at 2000 meters and ending at 3500 meters. We use special high altitude and super fast fly path to test this. We note that our forest rendering ends at far plane distance, but still added high level hierarchy to handle such big forest may also have its impact on forest rendering speed. We note that 20x20 km forest contains almost 400 000 trees.

The fifth series of tests measure forest rendering performance according to various forest densities. Spruce, poplar and falseacacia forests are measured with tree densities of 5, 10, 20, 40 and for poplar forest also 80 trees per 10000 square meters. Although the forest density is one of the most important factors for forest rendering, we hardly find such information in any tech paper we read about forest rendering. This we consider as serious flaw and mistake of the authors of these tech papers. With our results we can show, that by altering forest density, practically any speed of rendering results can be achieved.

The last but one series of tests are designed to test performance impact of various LOD tree representation grid subdivision schemes. The spruce and Colorado spruce forests are used to test five grid subdivision schemes of 1x1x1, 2x2x2, 3x3x3, 4x4x4 and 5x5x5 nodes. This test is undertaken in 800x600 screen resolution with normal detail in order to test application processor bound and because of that the graphic card is relieved from any extra rendering burden.

Final series of tests measure impact of forest subdivision scheme. This is done for 16, 32, 64, 128 and 256 maximal nodes per node setting with impostor system turned on and off. The forest used is of 2x2 km size because smaller forest of 1x1 km size is not enough for bigger maximal nodes per node value to manifest its properties.

If it isn't said otherwise all tests are done with forests of size 1x1 km with trees subdivided by 4x4x4 grid and with 64 maximal nodes per node forest subdivision scheme. If the reader is interested in any other parameter used and its setting, there is always a possibility to find it in test suite rendering settings, tree creation settings or forest creation settings. The appendixes also contain much more information about settings used for various trees and forests.

5.3 Results

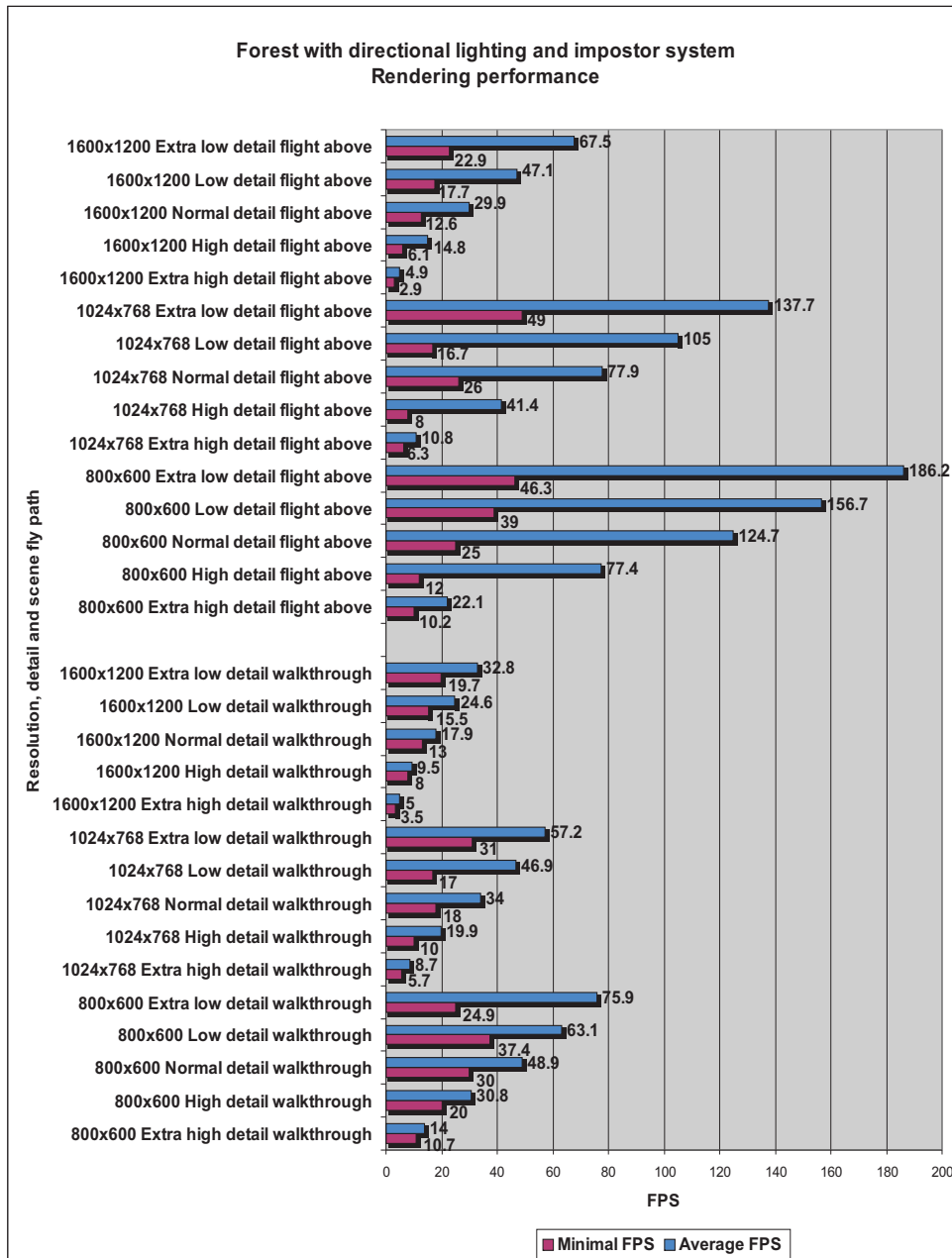


Figure 5.1: First series of tests. Rendering performance results of forest with directional lighting and impostor system.

The rendering performance shows that most of resolution and detail settings run in real-time or in near real-time. The slowest is 1600x1200 extra high detail setting which don't run even in interactive speed. The speed up between extra

high and high detail setting is caused by using different detail settings for tree level rendering by points only.

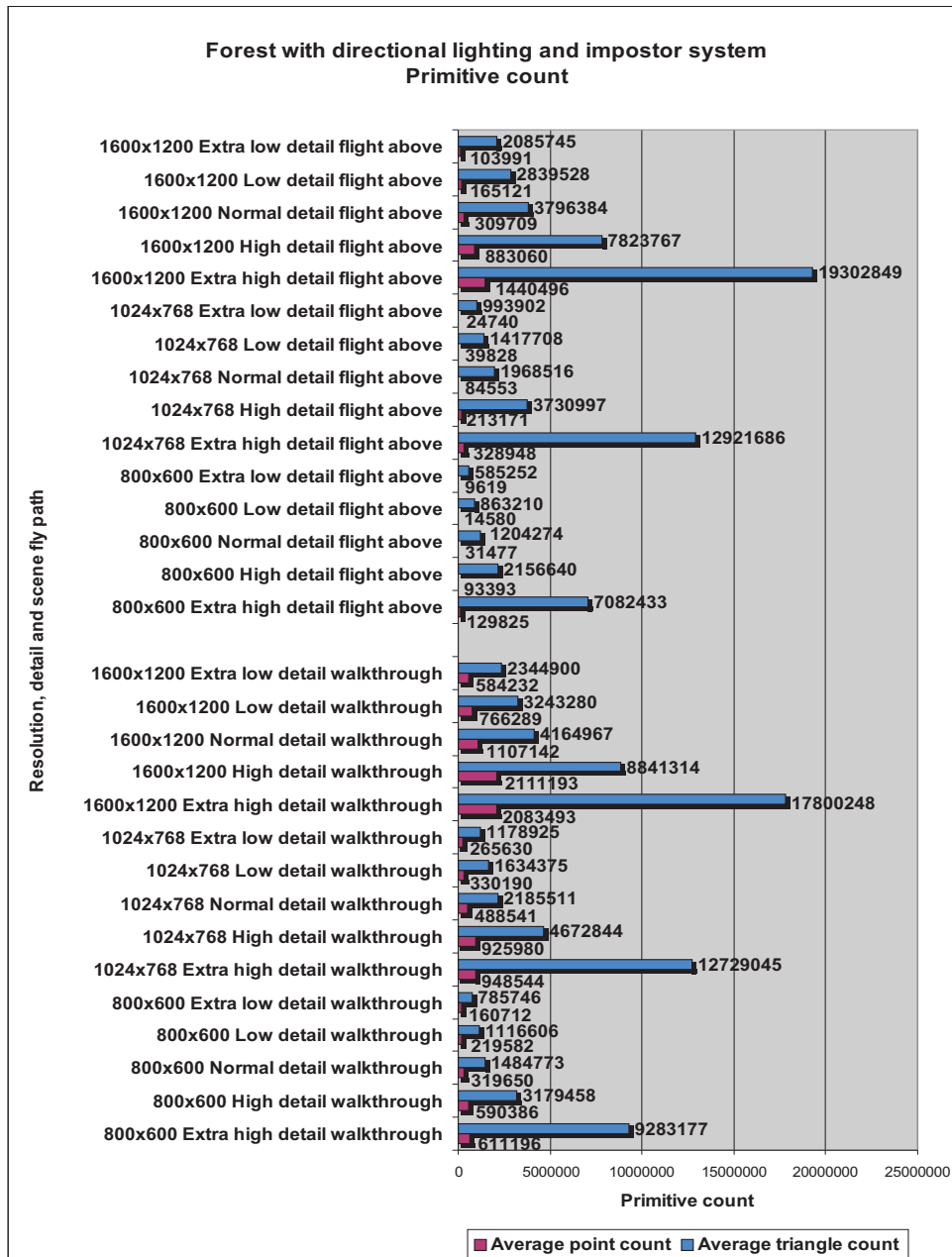


Figure 5.2: First series of tests. Primitive count results of forest with directional lighting and impostor system.

The primitive count results show that point rendering is most costly for our method since there is only small amount of triangles rendered. The extra high

detail setting uses extreme amount of points to be rendered.

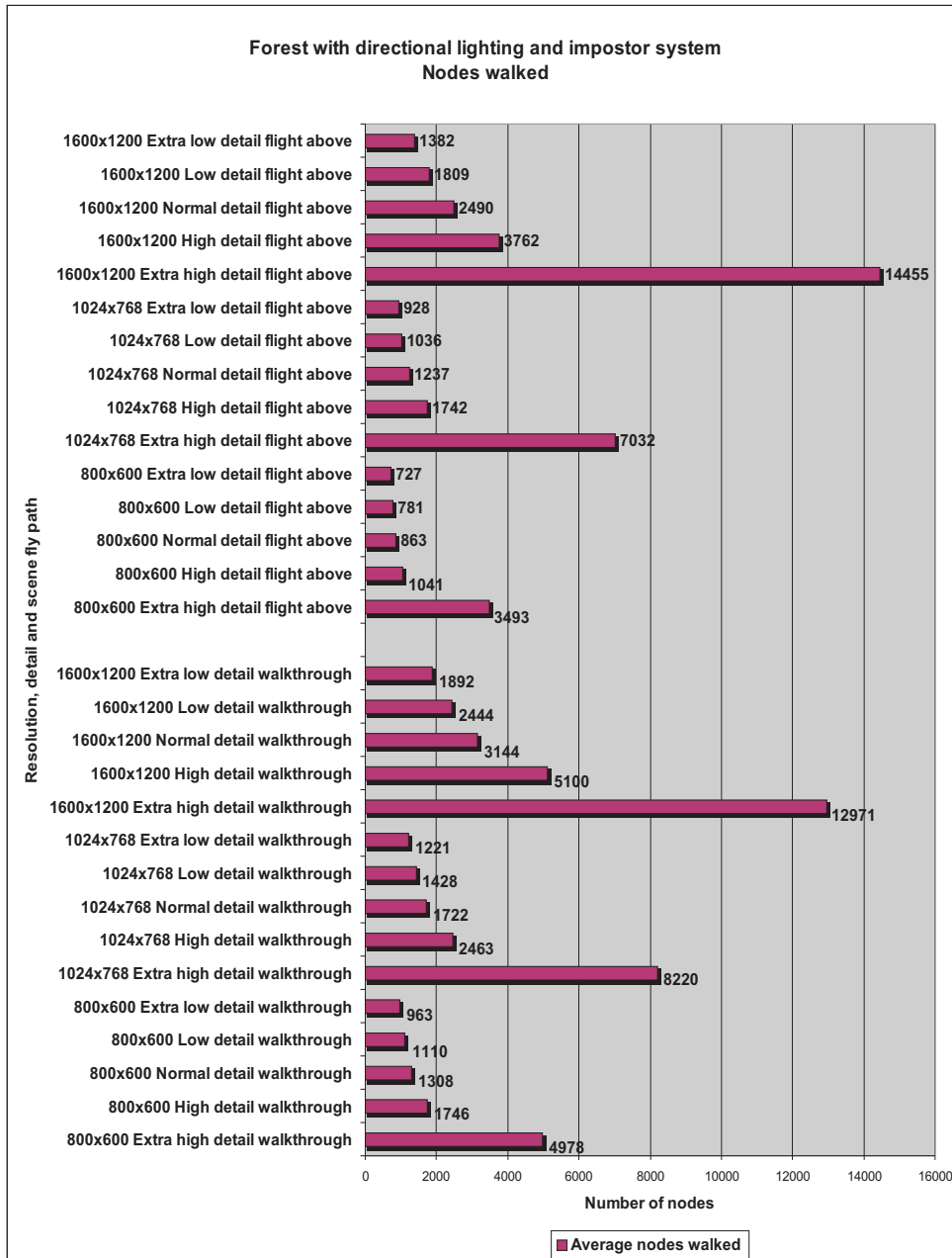


Figure 5.3: First series of tests. Nodes walked results of forest with directional lighting and impostor system.

Notice how number of nodes walked falls off between extra high and high detail setting. Setting tree level detail settings for lower detail allows this. The number above 8000 average nodes walked definitively causes the application to

be processor bounded.

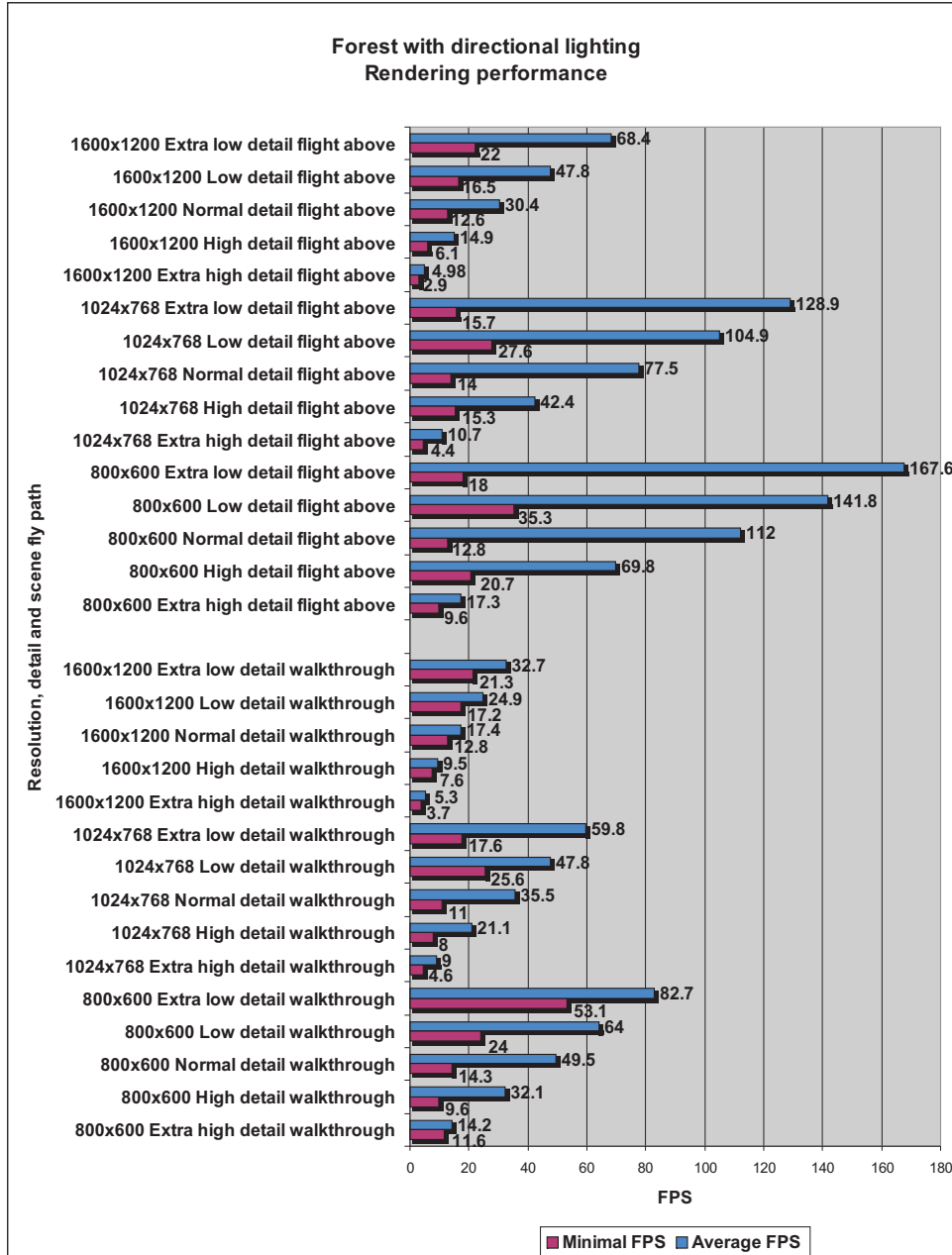


Figure 5.4: First series of tests. Rendering performance results of forest with directional lighting.

Unfortunately the impostor system doesn't yield any speed performance improvement, but also doesn't slow down application. We'll discuss this issue in next section 5.4.

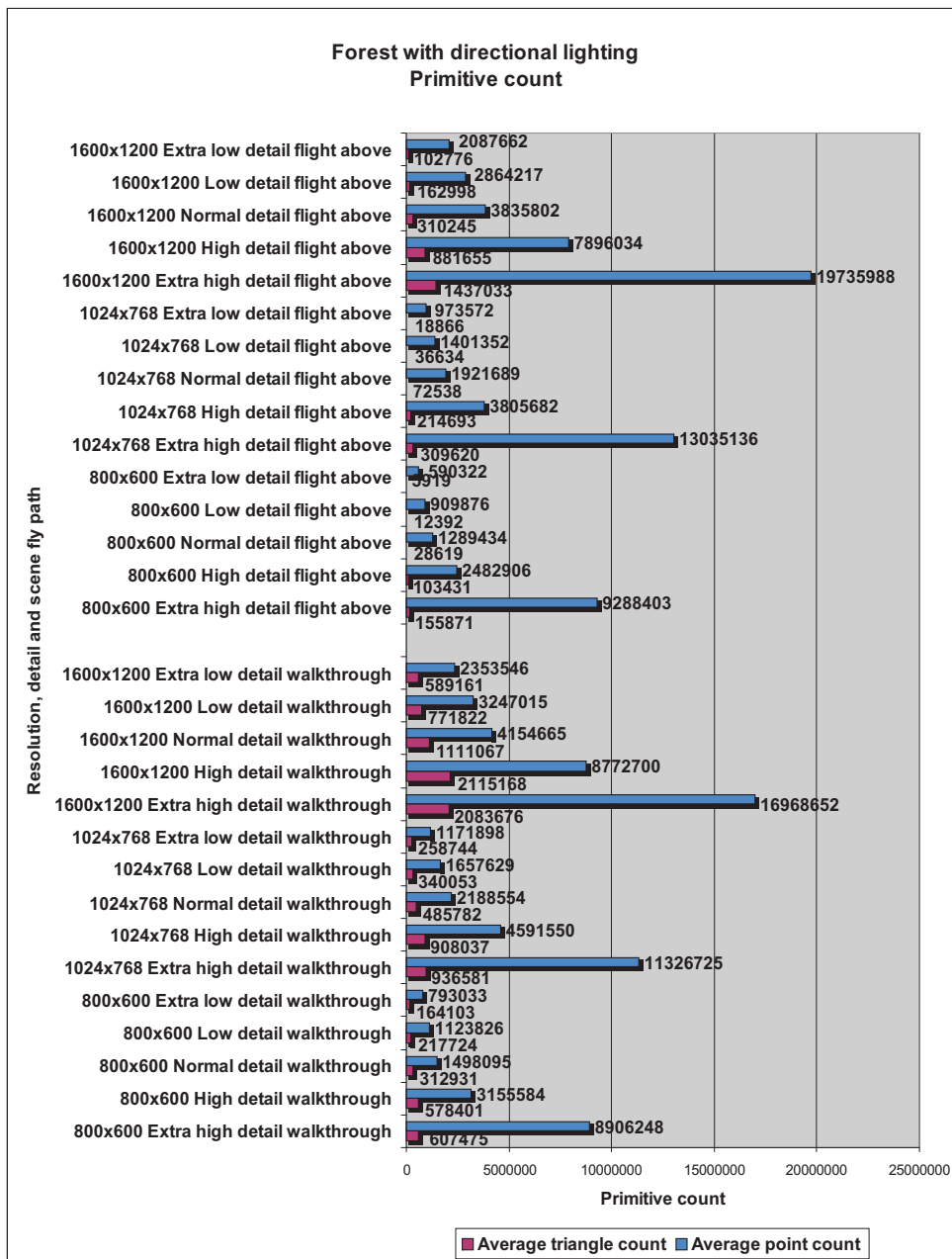


Figure 5.5: First series of tests. Primitive count results of forest with directional lighting.

The primitive count for scene without impostor system is much the same as with impostor system used.

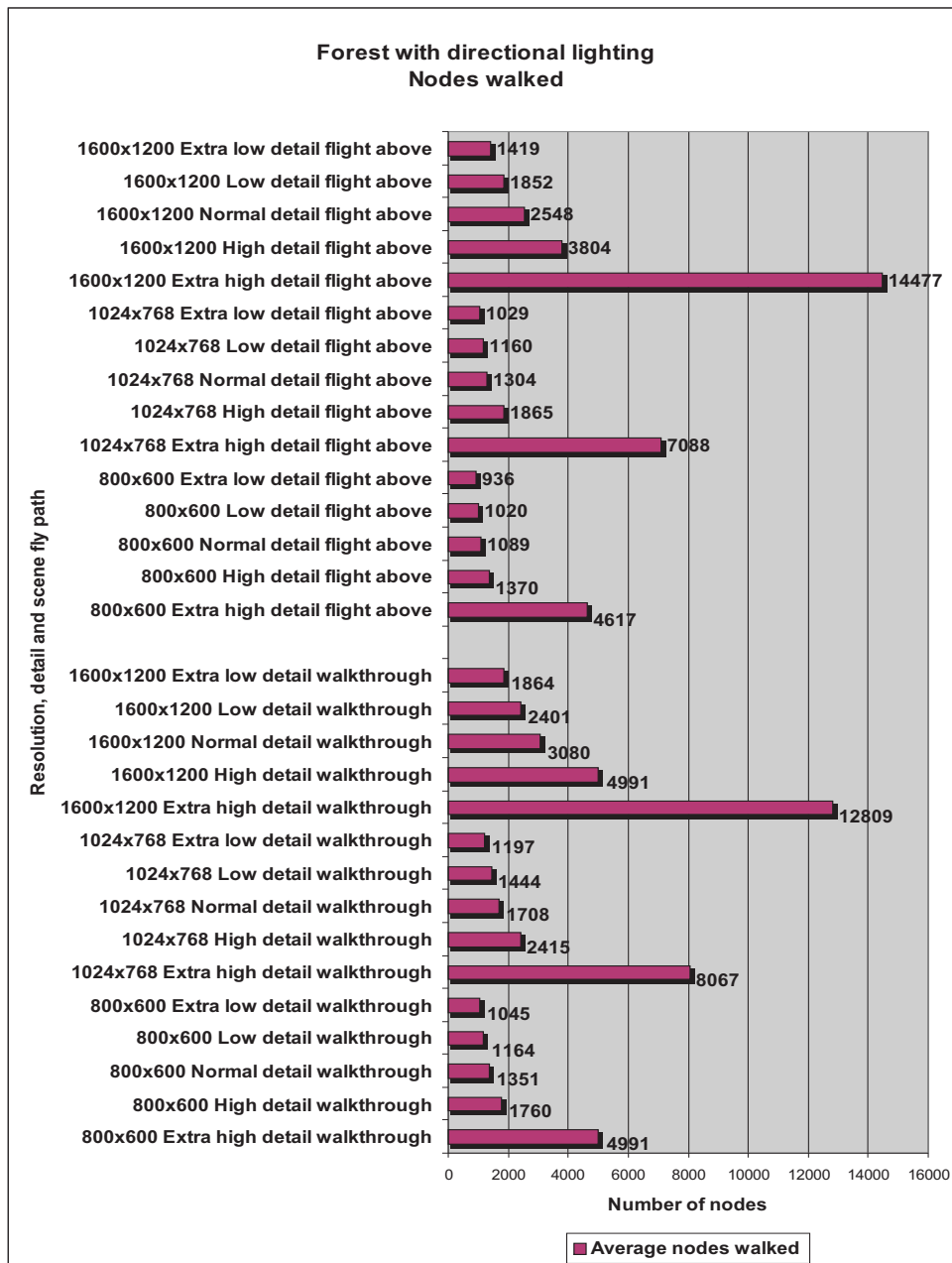


Figure 5.6: First series of tests. Nodes walked results of forest with directional lighting.

The same counts for average nodes walked results. The difference between impostor system used and not used is minimal.

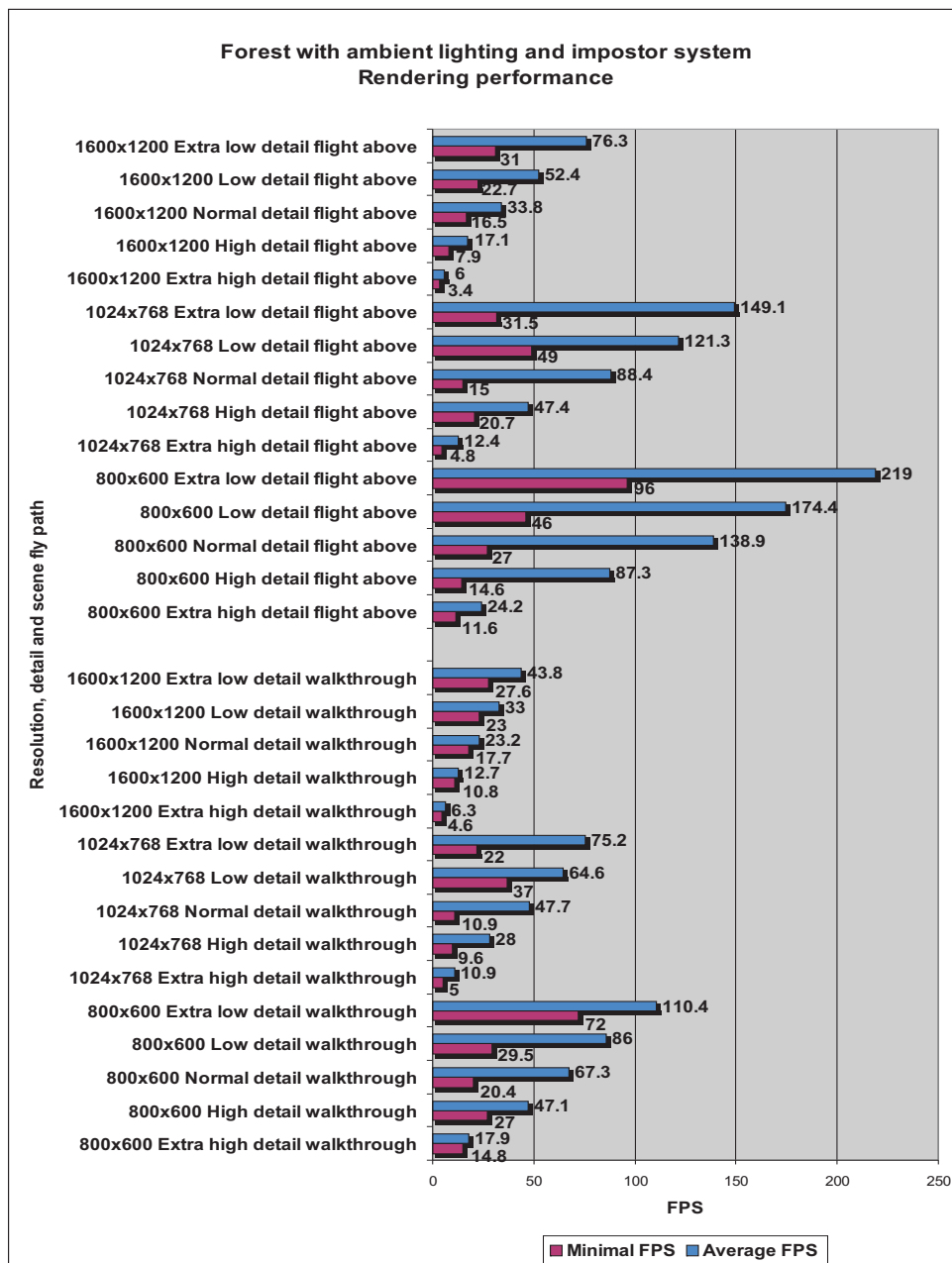


Figure 5.7: First series of tests. Rendering performance results of forest with ambient lighting and impostor system.

The ambient lighting only rendering path is about 20 percent faster than directional lighting with ambient rendering path.

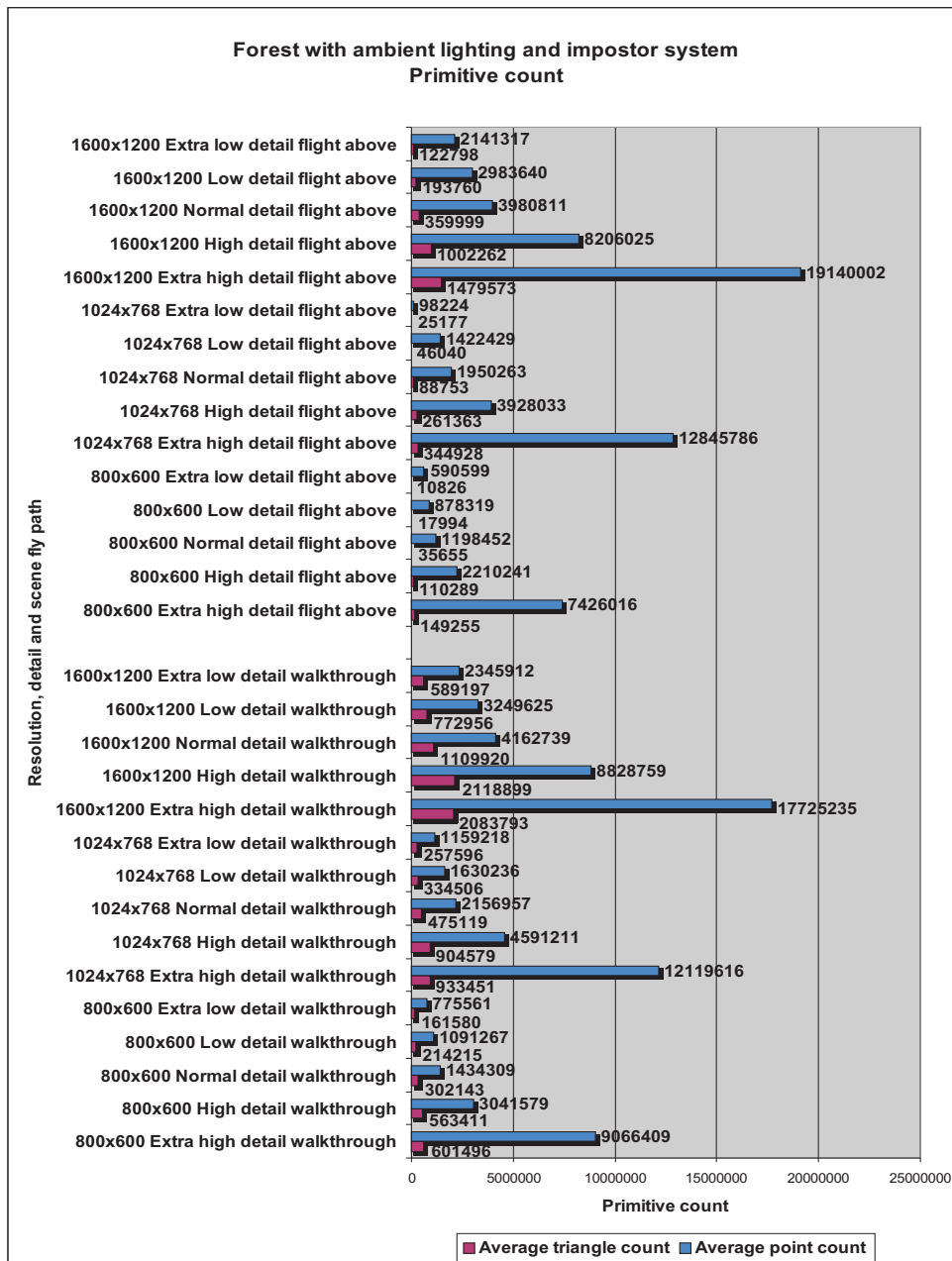


Figure 5.8: First series of tests. Primitive count results of forest with ambient lighting and impostor system.

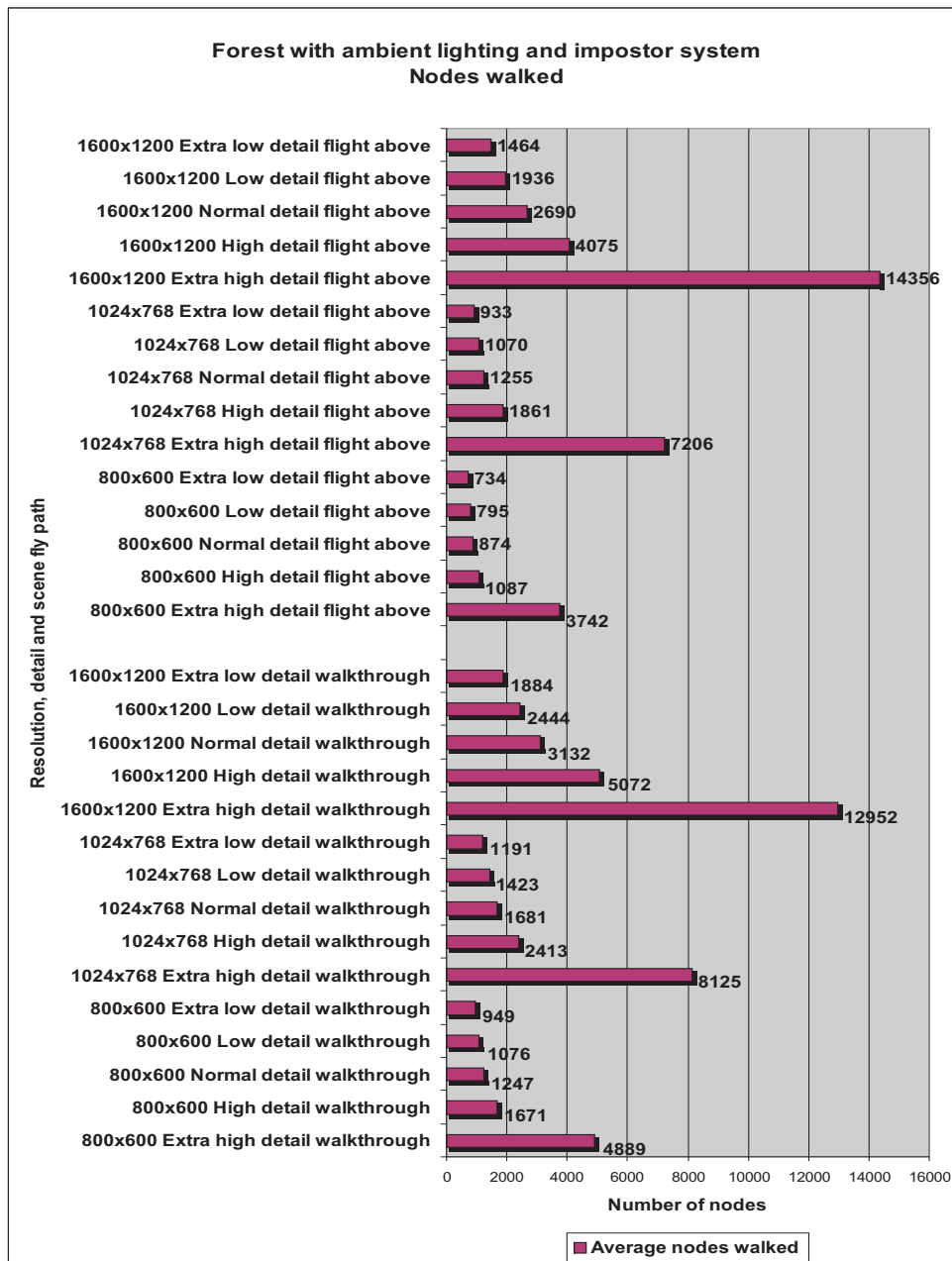


Figure 5.9: First series of tests. Nodes walked results of forest with ambient lighting and impostor system.

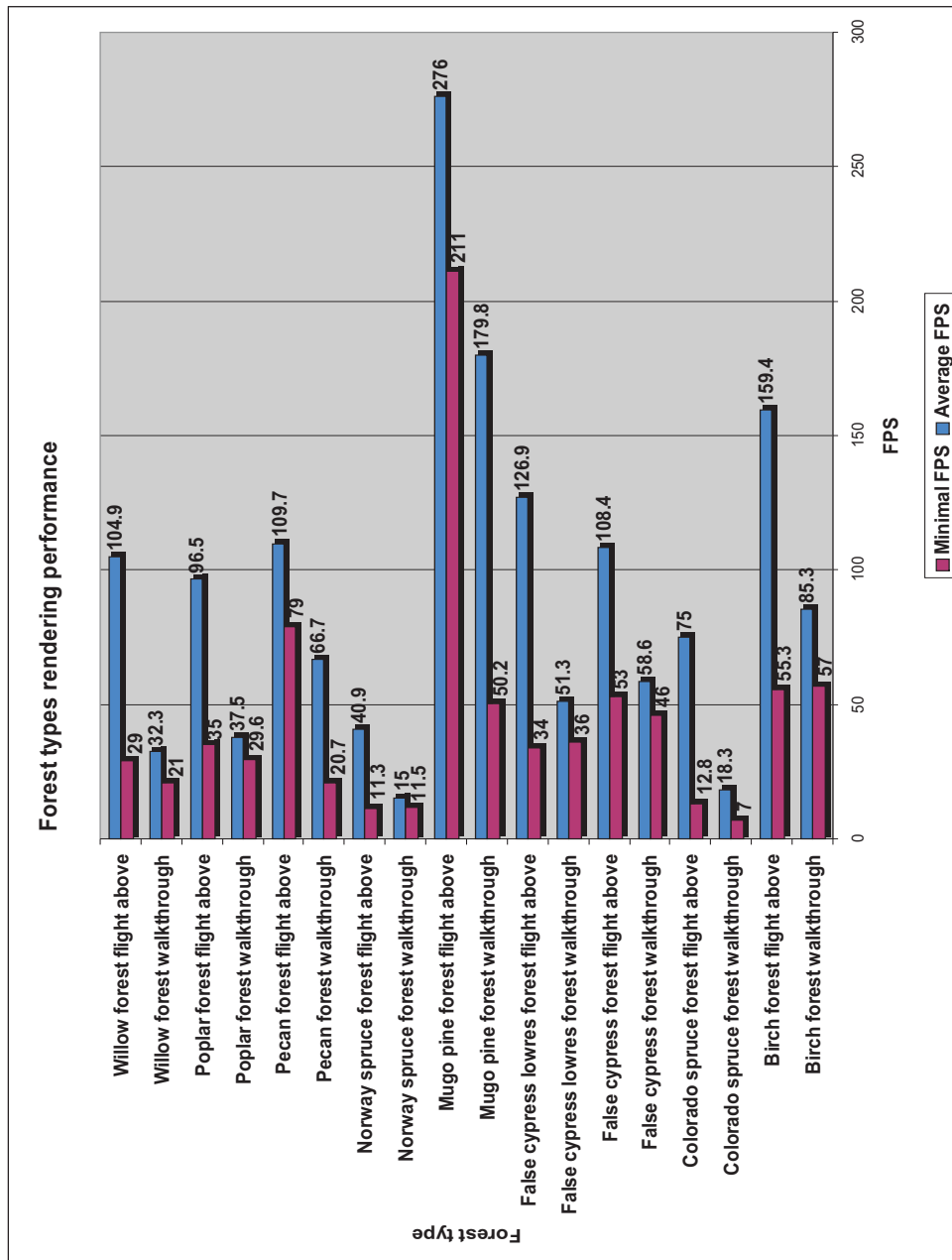


Figure 5.10: Second series of tests. Rendering performance results of various forest motives.

The results show that various forest motives can be also rendered in real-time or near real-time speed. The density used is 30 trees per 10000 square meters. The exceptional mugo pine performance is caused by the fact that it's shrub.

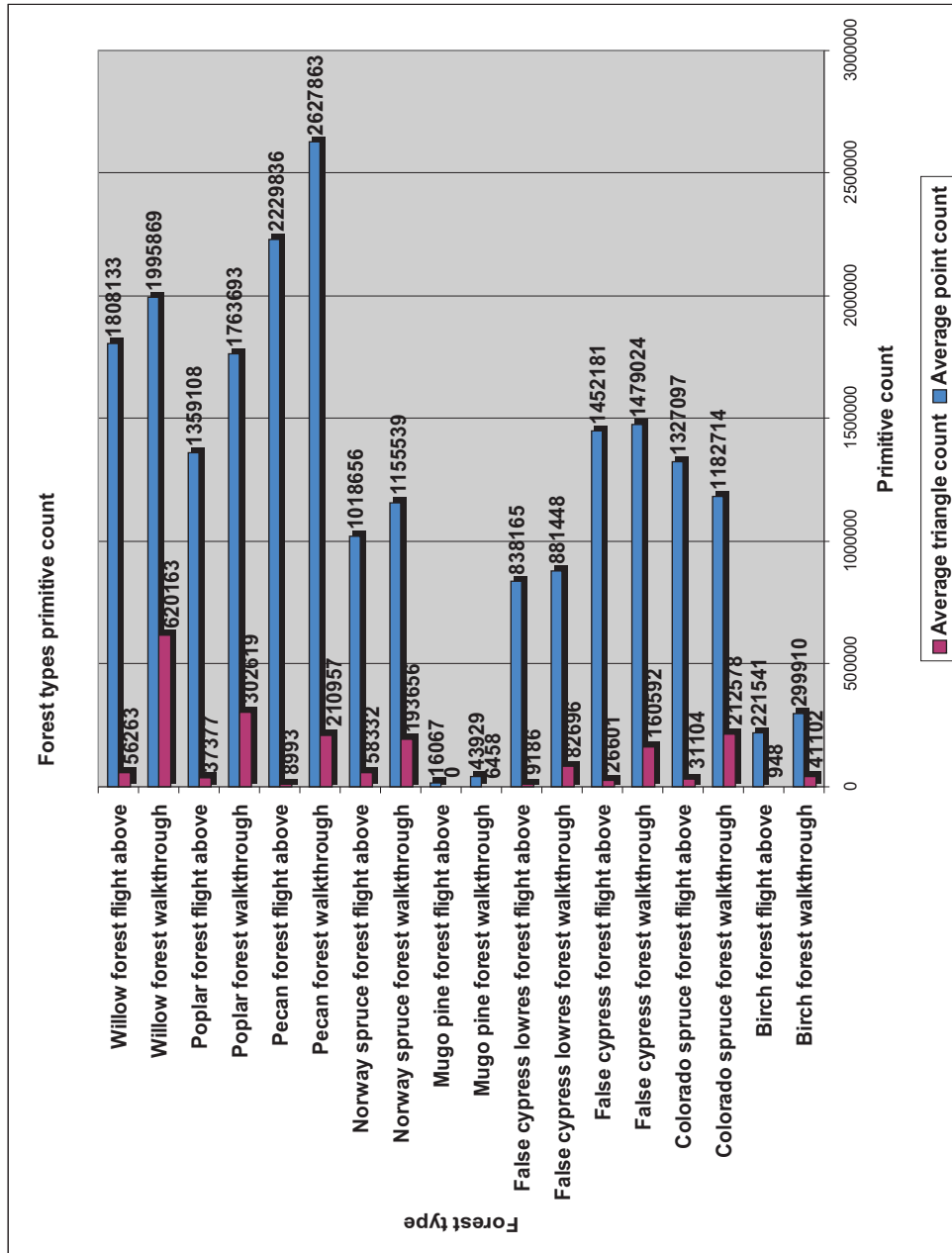


Figure 5.11: Second series of tests. Primitive count results of various forest motives.

Although the Colorado spruce forest doesn't have the highest average point count used for walkthrough, it's the slowest in FPS. This can be caused by the fact that Colorado spruce is rather big tree.

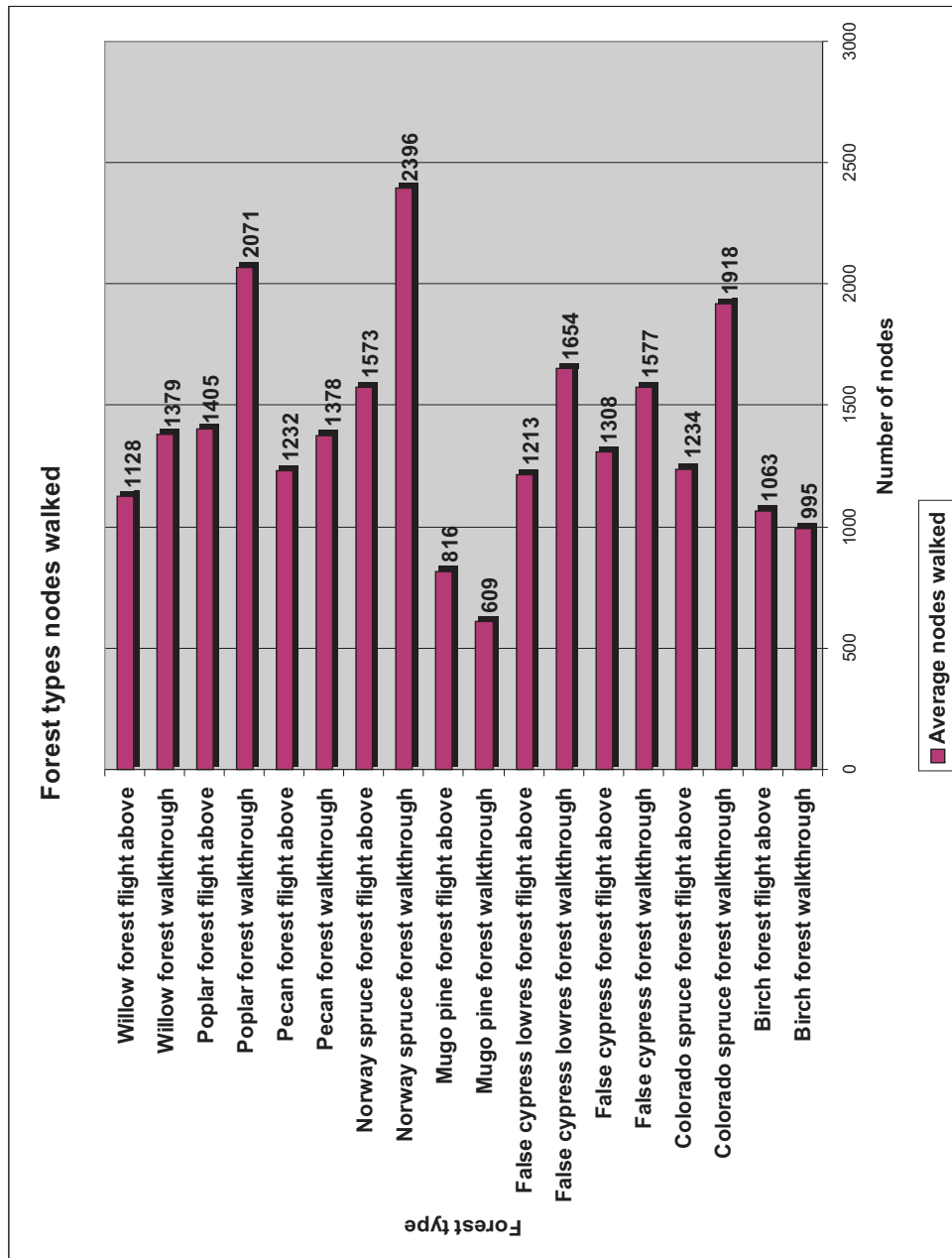


Figure 5.12: Second series of tests. Nodes walked results of various forest motives.

The average nodes walked are at affordable number for real-time rendering for nowadays hardware.

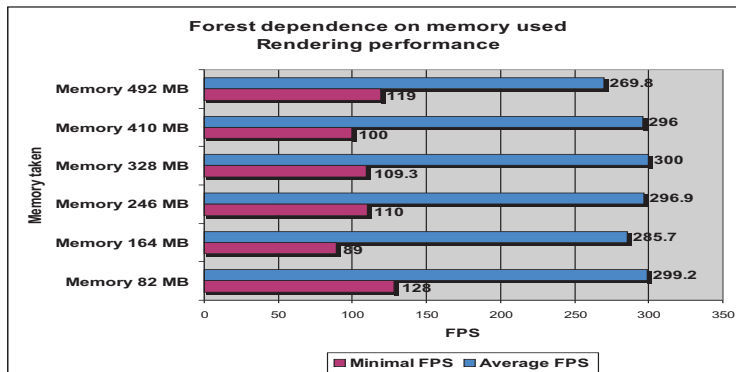


Figure 5.13: Third series of tests. Performance impact of used memory on application.

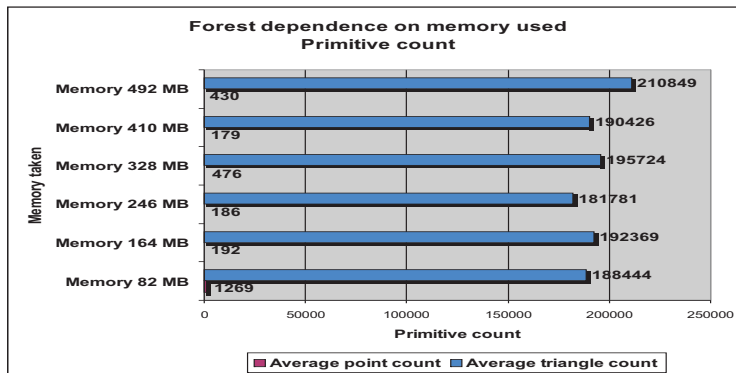


Figure 5.14: Third series of tests. Primitive count of memory tests.

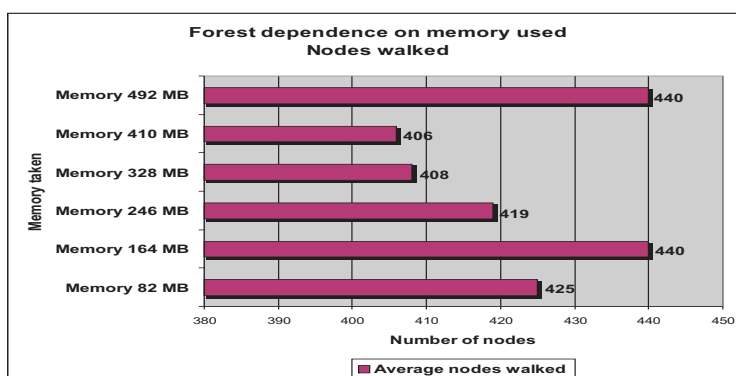


Figure 5.15: Third series of tests. Nodes walked of memory tests.

The dependence of application on memory used by graphic card is minimal. The different primitive count and nodes walked results are caused by another

forests used for every memory usage test. All forests however have the same density and consist of same tree type!

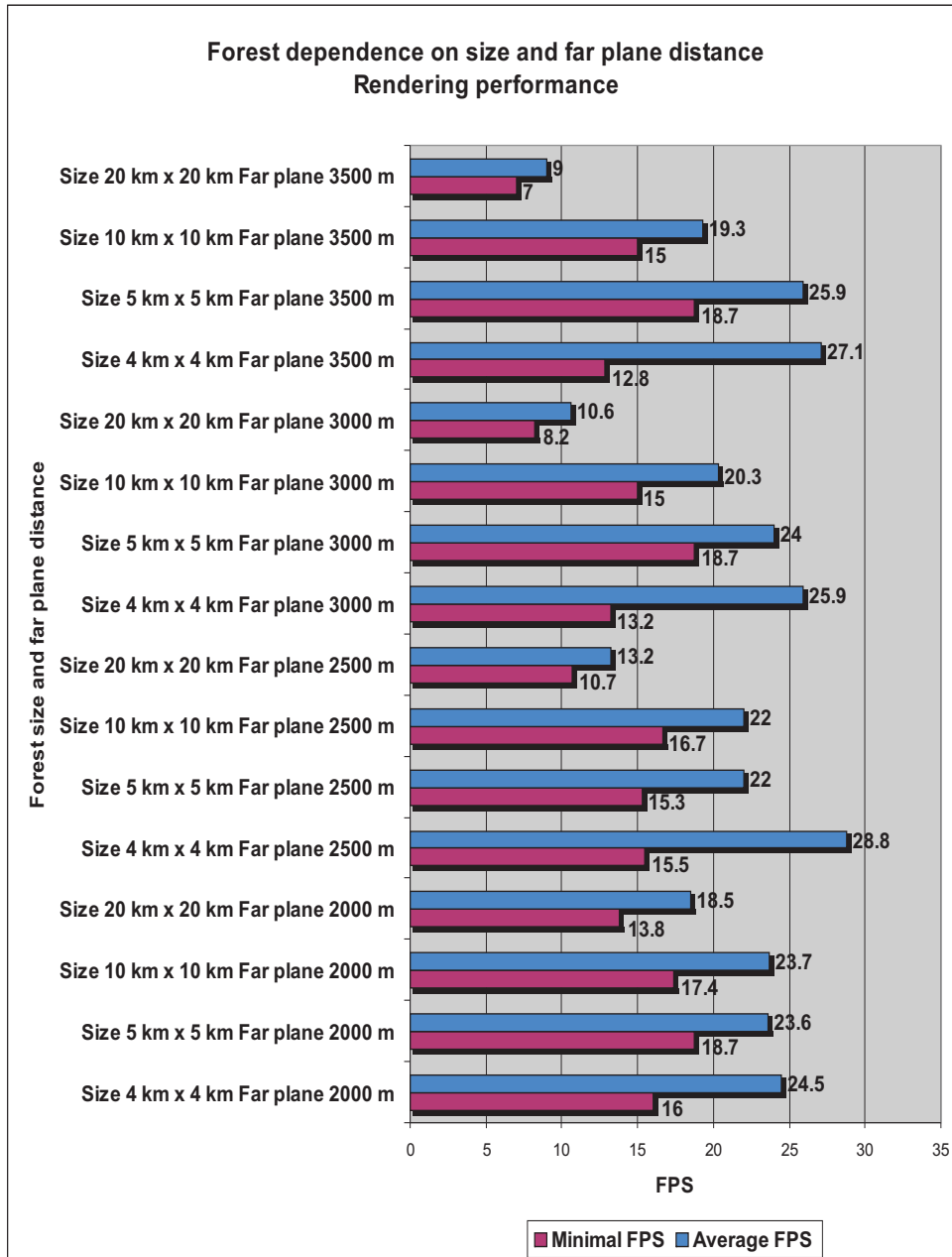


Figure 5.16: Fourth series of tests. Performance impact of forest size and far plane distance.

This shows that increasing far plane distance doesn't have any high performance impact. This is caused by low number of points used for trees rendered

far away from the camera. The 20 km x 20 km forest rendering is slow. This is caused by the increased number of forest level hierarchy nodes used for frustum culling. Managing about 400 000 trees uses a lot of node's hierarchy levels.

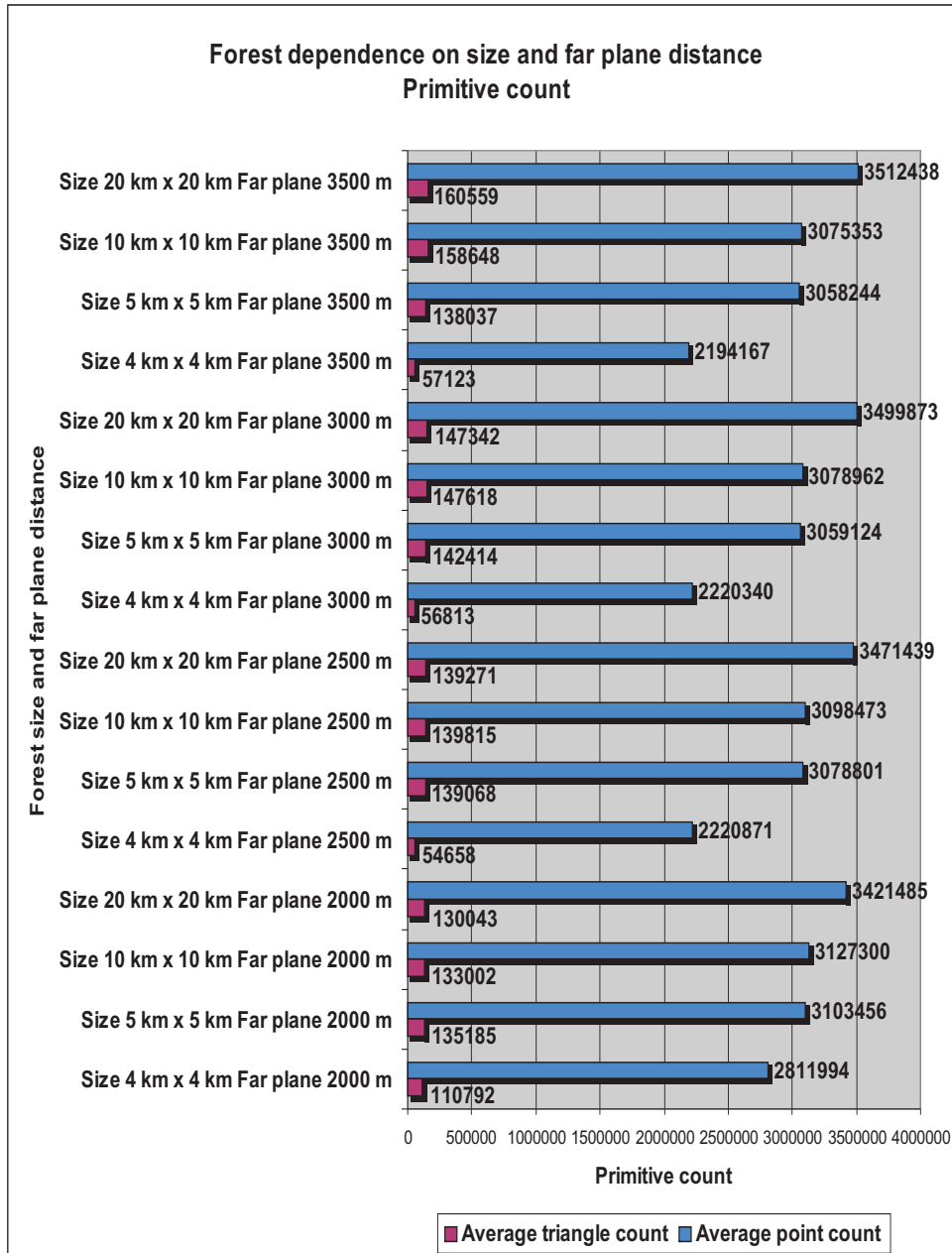


Figure 5.17: Fourth series of tests. Performance impact of forest size and far plane distance – primitive count.

There is not a big jump of primitive count between 10 km x 10 km and 20

km x 20 km forest which shows that increased primitive count doesn't cause the performance difference. However there is a big jump between the 4 km x 4 km and 5 km x 5 km, but still the performance is about the same. This can be caused by the fact that spruce forest used looks dense or still processor bound prevails.

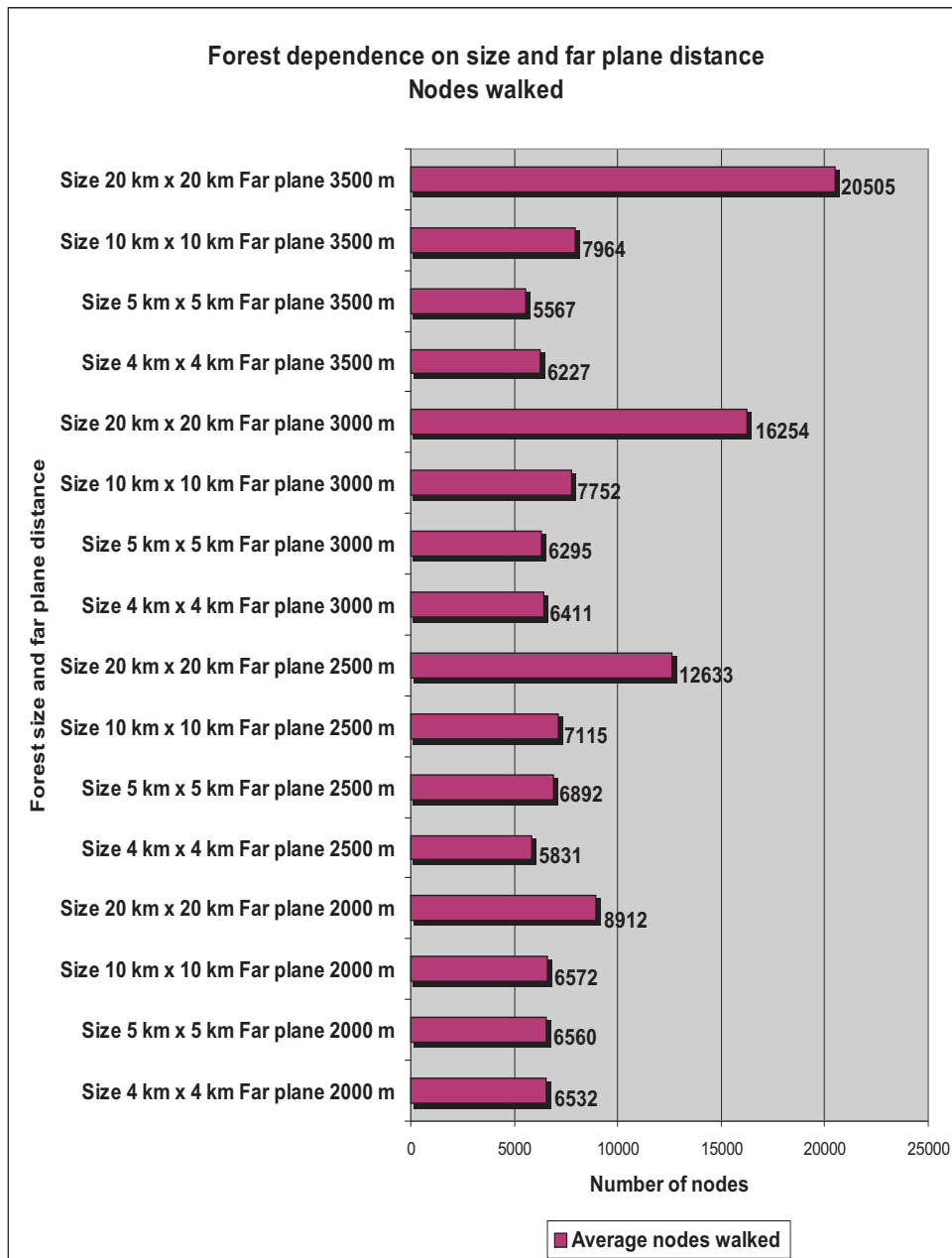


Figure 5.18: Fourth series of tests. Performance impact of forest size and far plane distance – nodes walked.

Doubled number of average nodes walked at 20 km x 20 km forest shows the reason why its rendering is so slow.

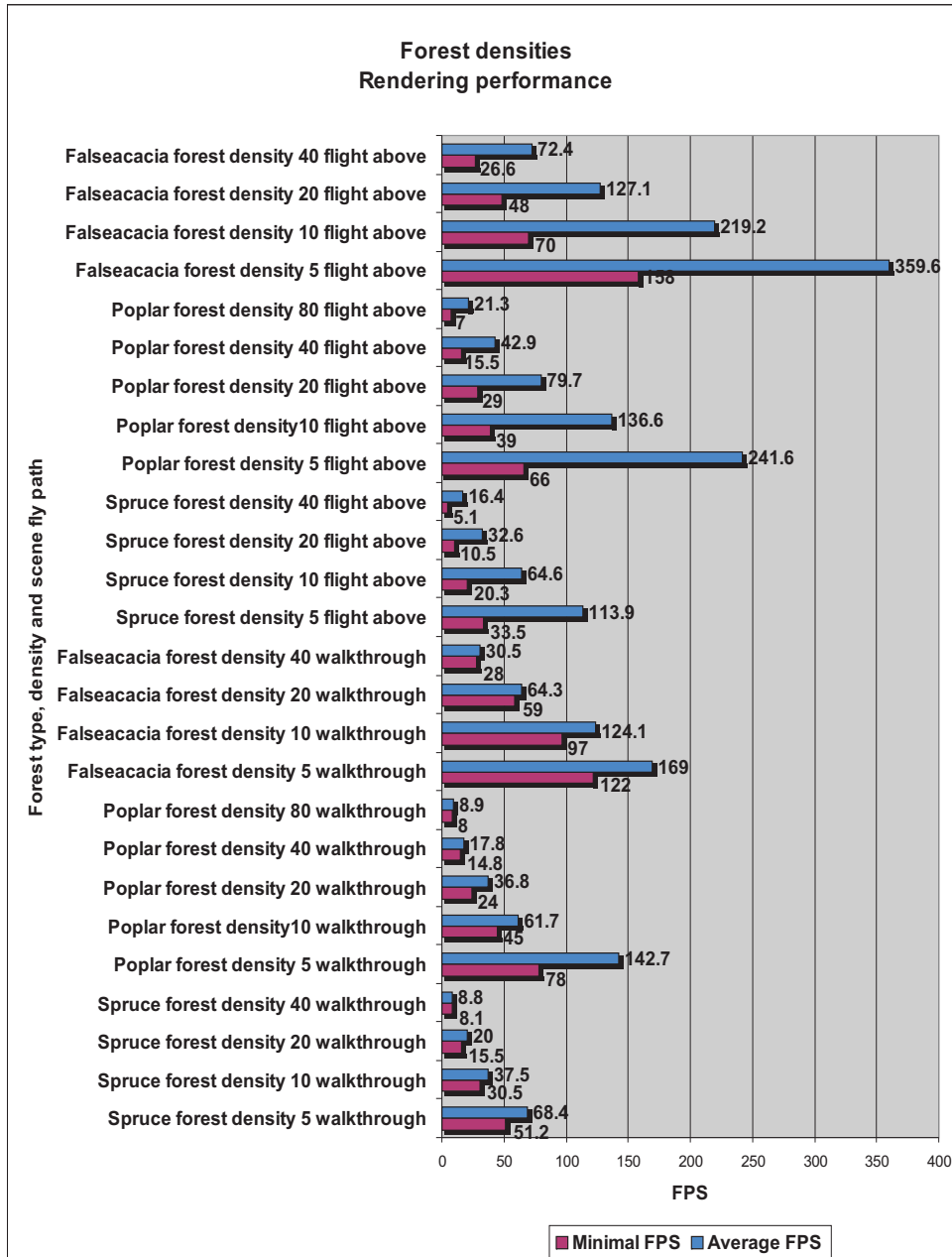


Figure 5.19: Fifth series of tests. Rendering performance of forests with various densities. Density is stated in number of trees per 10000 square meters.

The results show that relation between forest density and rendering performance is exponential. Still even high density forests can be viewed in interactive

speed. We note that density of real forest is even higher than ours in all cases.

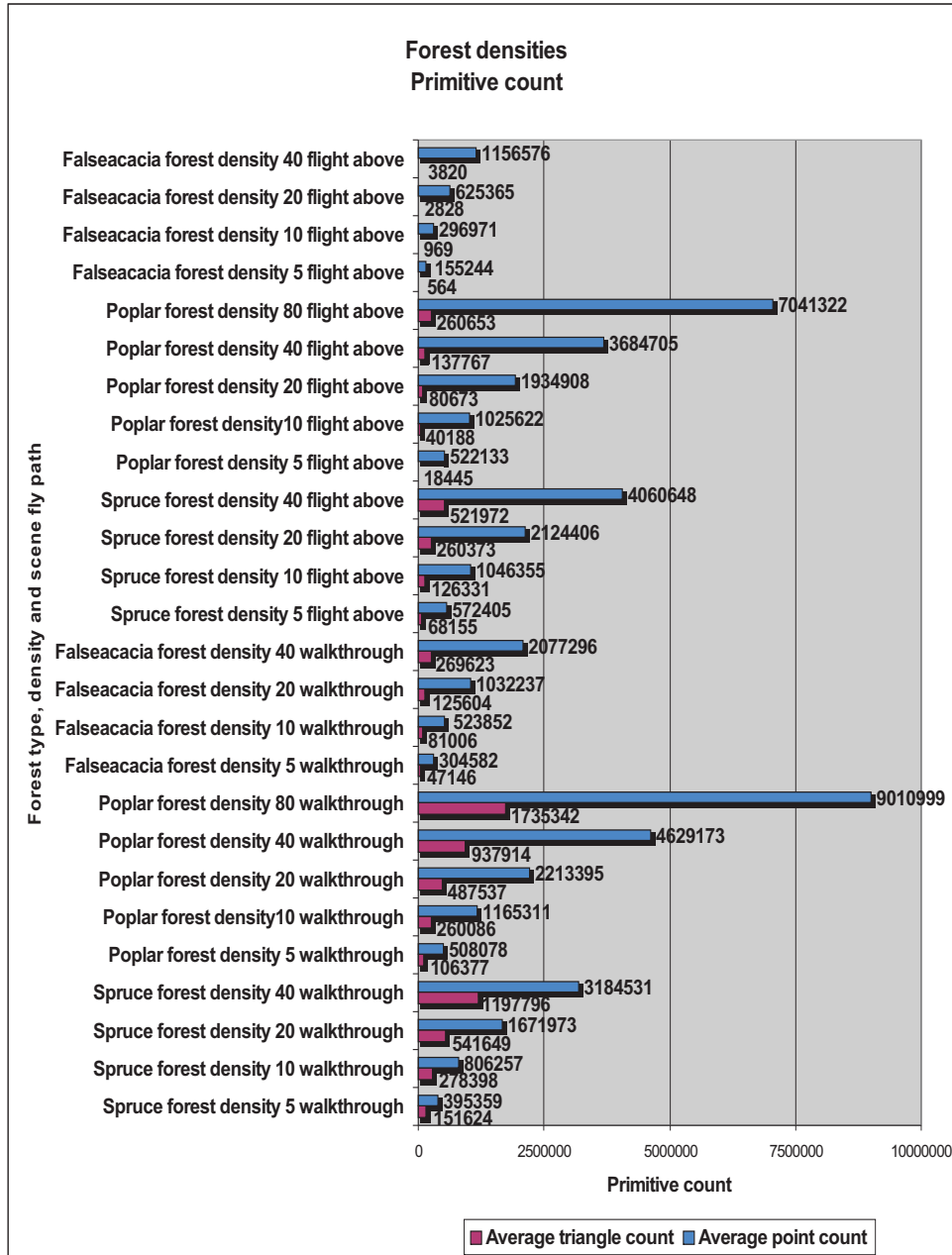


Figure 5.20: Fifth series of tests. Primitive count results of forests with various densities. Density is stated in number of trees per 10000 square meters.

Although falseacacia tree contains highest number of triangles and points, the falseacacia forest is not so demanding on primitive count because falseacacia tree is rather small.

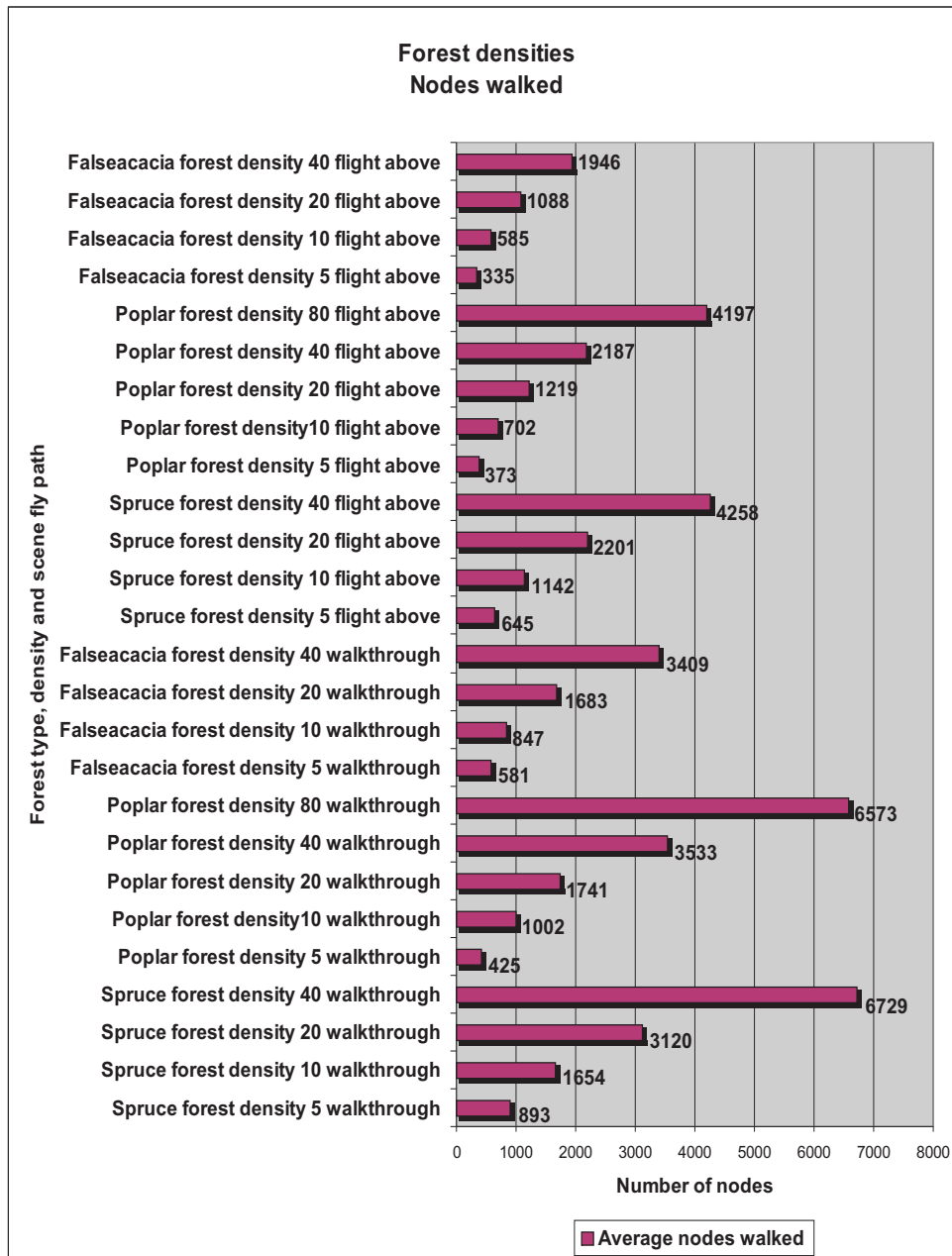


Figure 5.21: Fifth series of tests. Nodes walked of forests with various densities. Density is stated in number of trees per 10000 square meters.

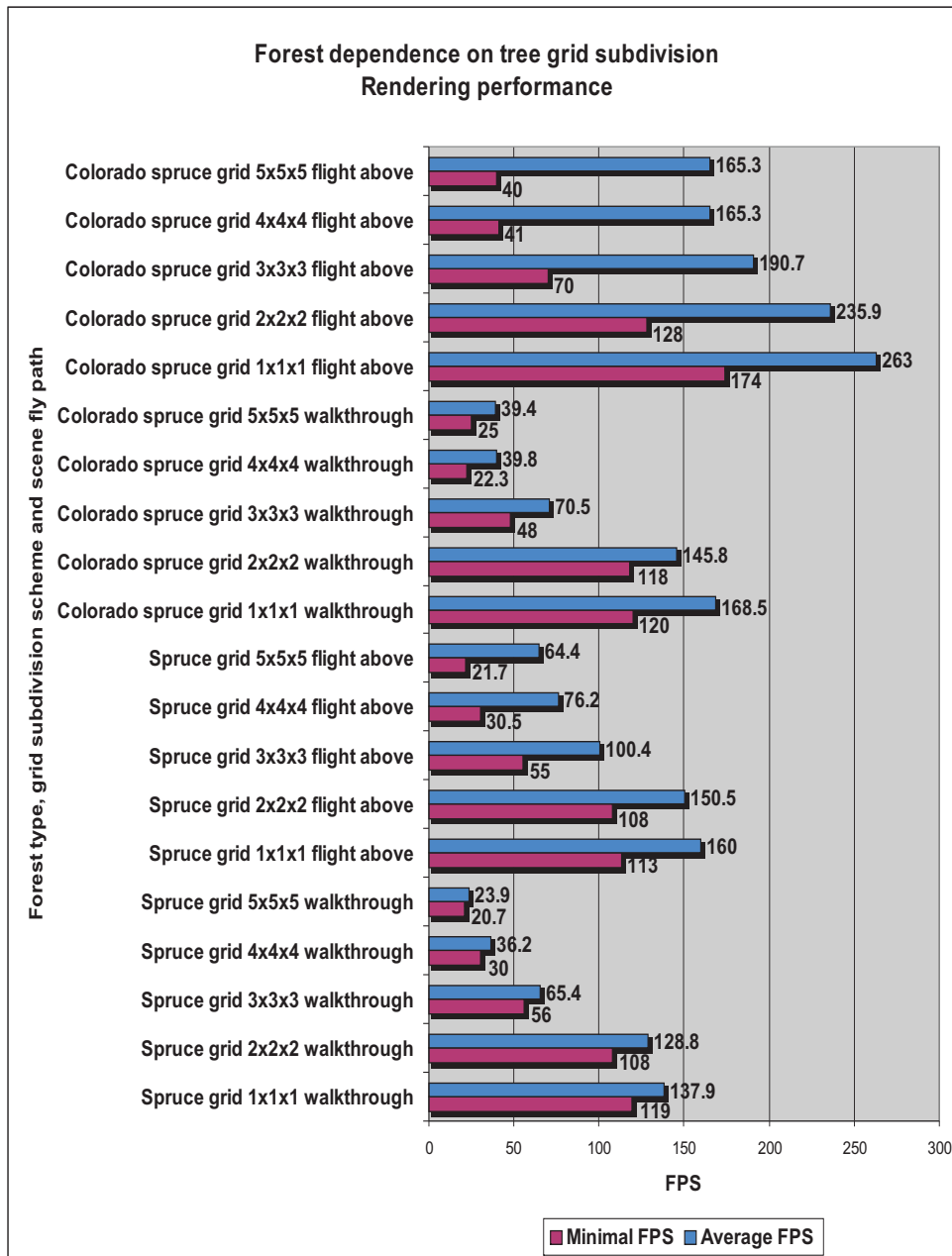


Figure 5.22: Sixth series of tests. Rendering performance of forests with various tree grid subdivisions.

The results show that using lower resolution grid is better and that application is surprisingly still heavily processor bounded. We did not expect so big performance penalty of using higher resolution grid subdivision schemes.

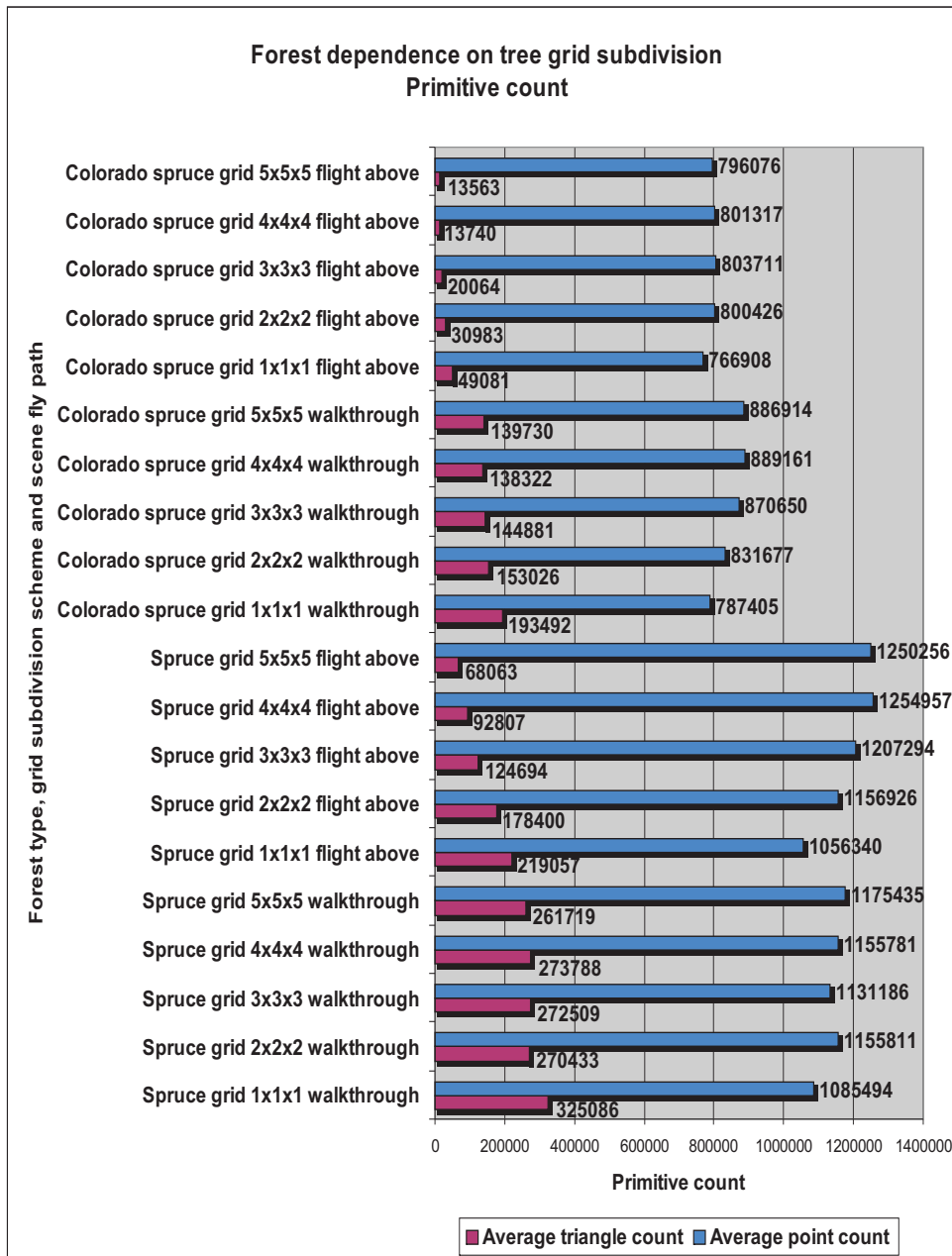


Figure 5.23: Sixth series of tests. Primitive count of forests with various tree grid subdivisions.

Notice that more triangles are used for lower resolution grids. This is in our opinion caused by only small number of node distance approximations which in turn cause to render more triangles in the back of the tree.

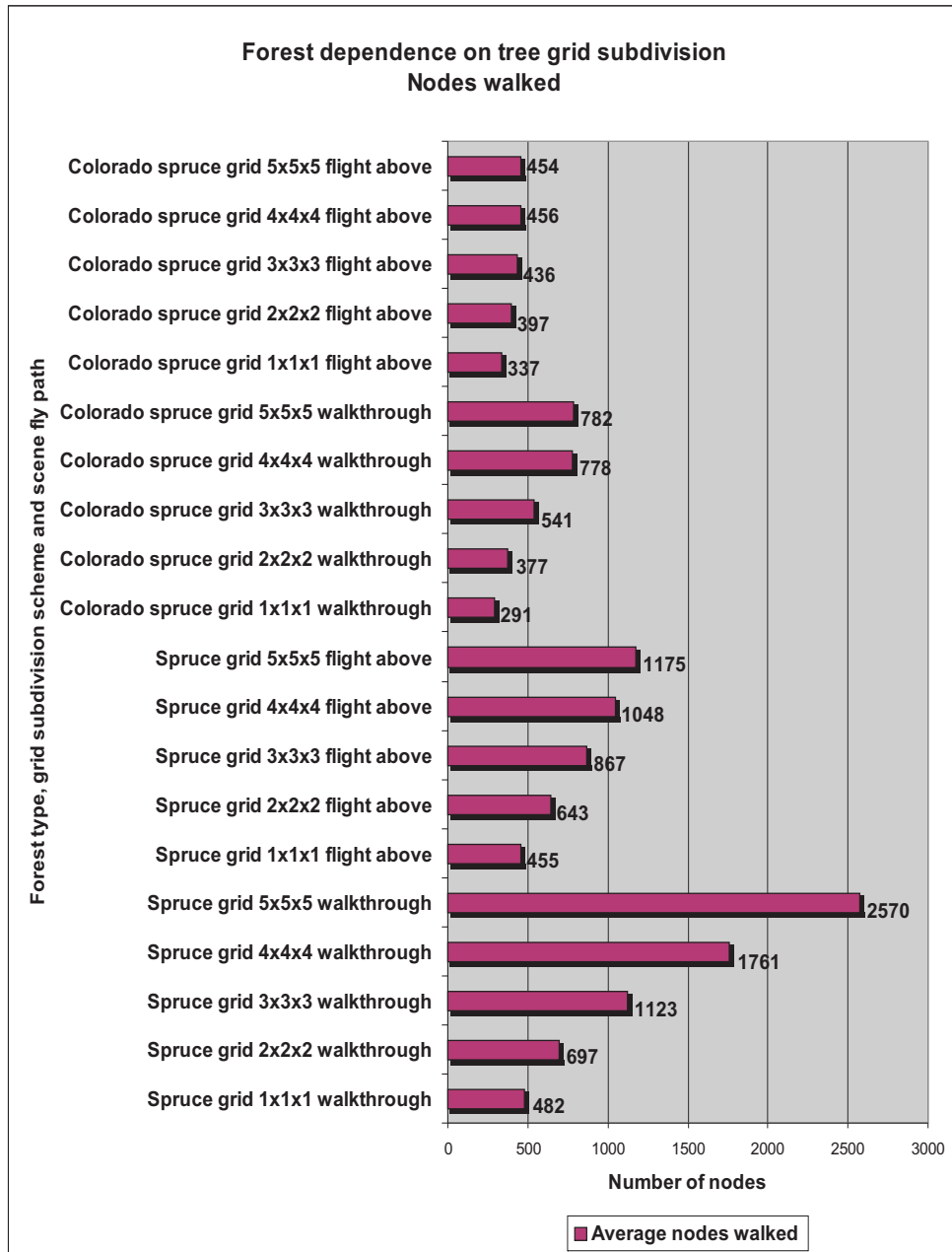


Figure 5.24: Sixth series of tests. Nodes walked of forests with various tree grid subdivisions.

The average number of nodes walked copies the grid subdivision scheme used, but not exactly. Practically this is caused by the fact that for higher resolution grid the point only representation rendering is started earlier because more grid cells mean smaller generated points in them.

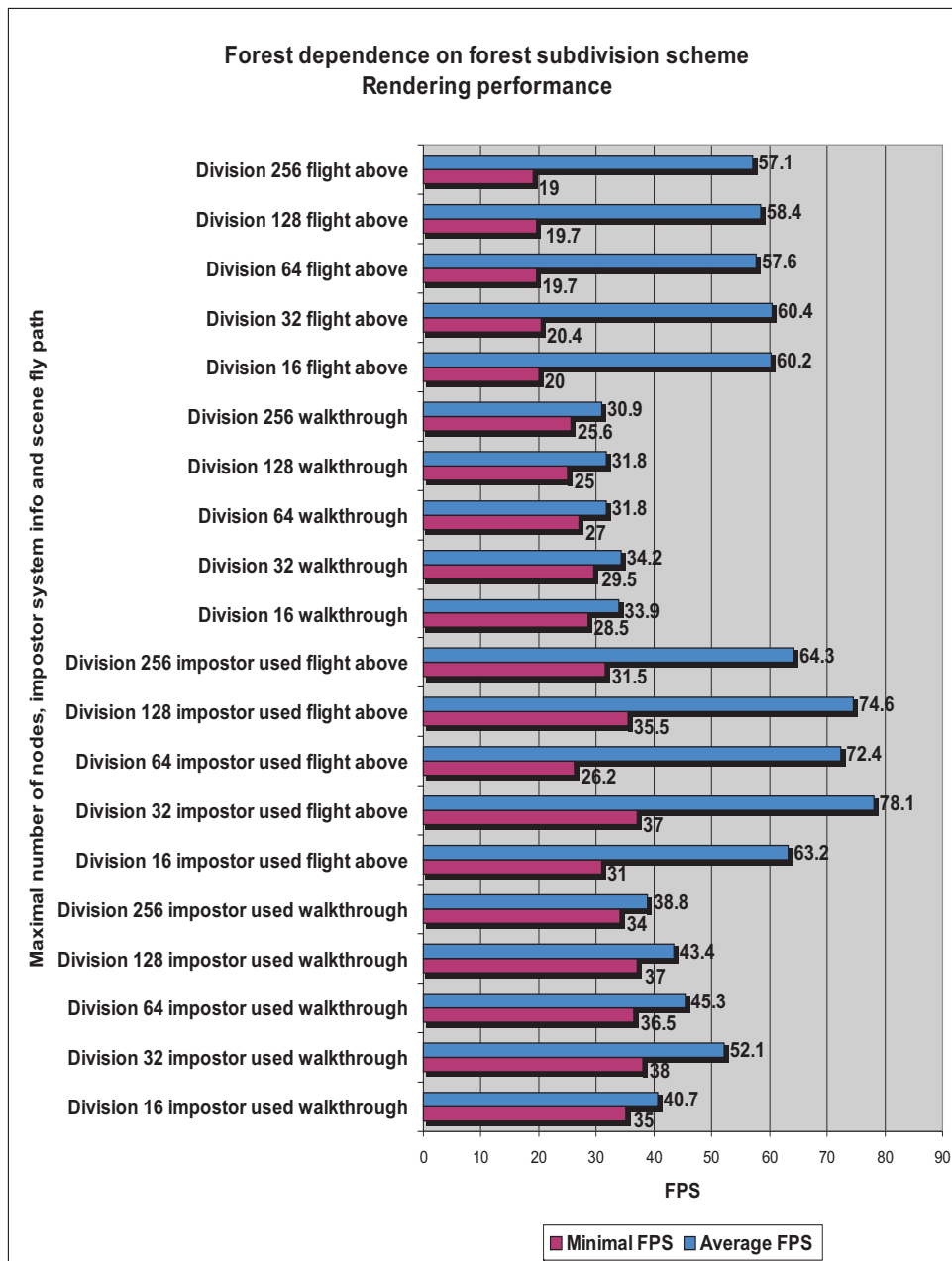


Figure 5.25: Seventh series of tests. Rendering performance of forest with various forest subdivision schemes.

The last test series showed that practically any forest subdivision scheme can be used. The results for this test series maybe slightly misleading because for every division scheme a different position of trees is used. With our current forest creation solution it isn't easy to generate same tree positions. This means that results are only accurate to show whether the dependence on forest subdivision

scheme is significant or not.

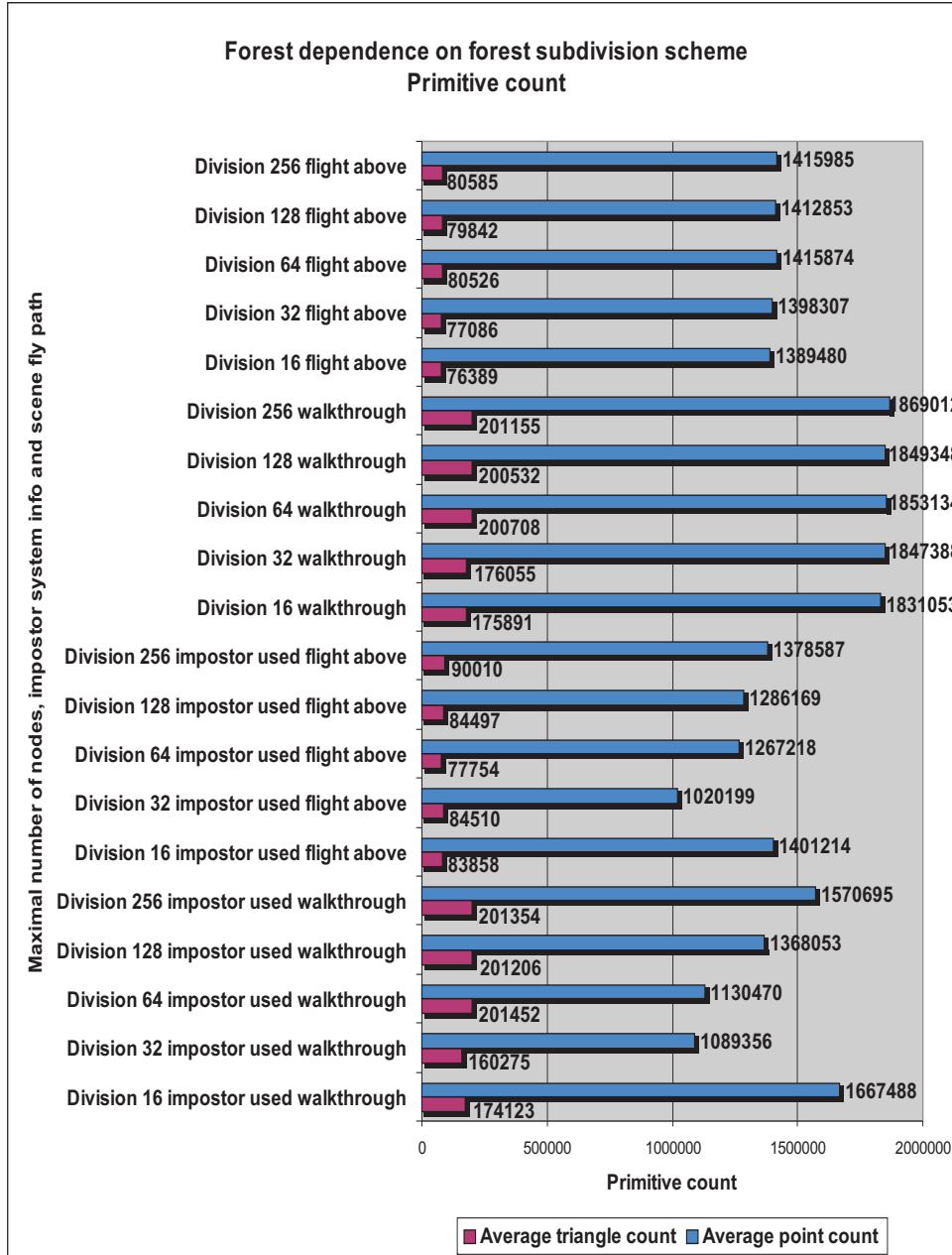


Figure 5.26: Seventh series of tests. Primitive count of forest with various forest subdivision schemes.

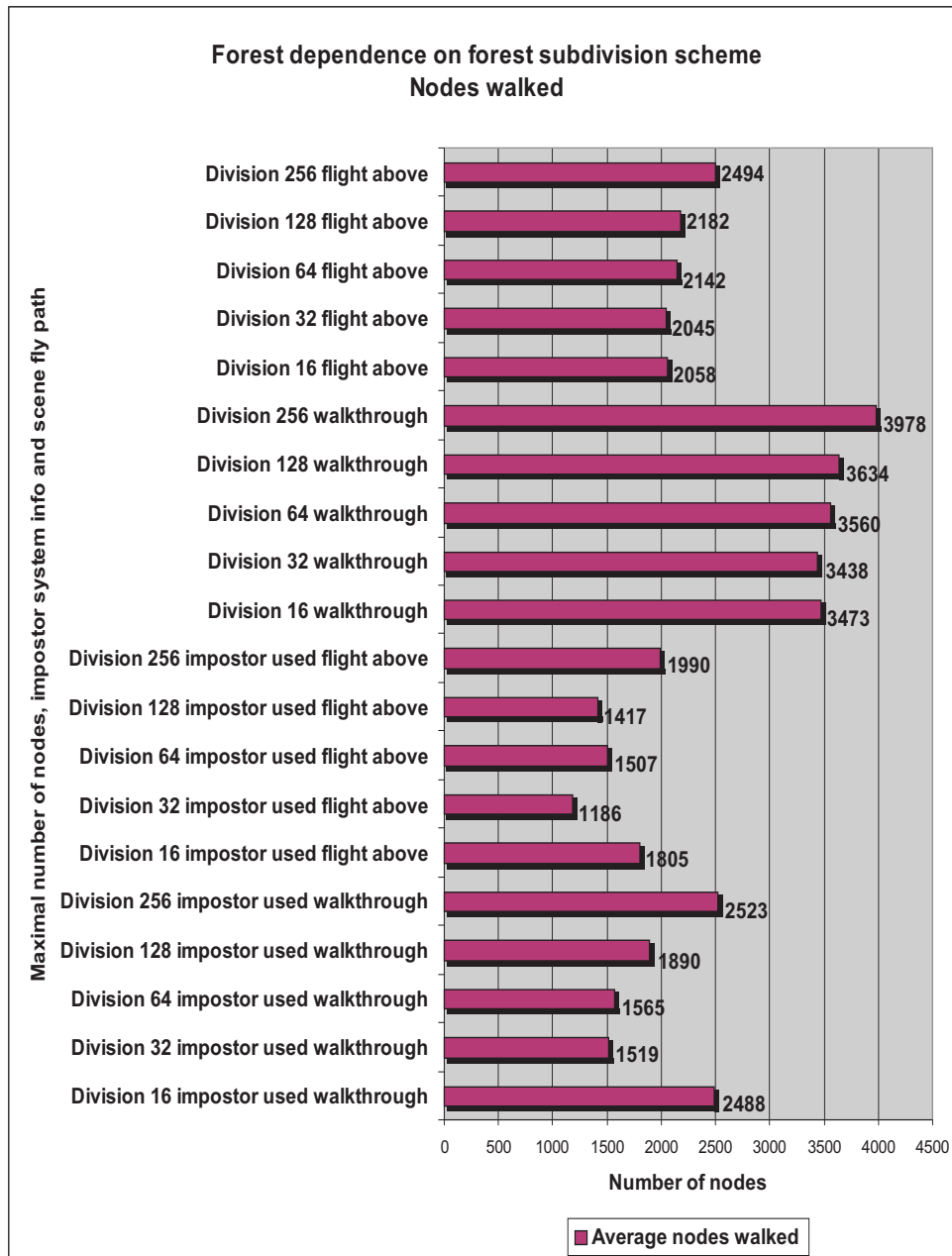


Figure 5.27: Seventh series of tests. Nodes walked of forest with various forest subdivision schemes.

5.4 Interpreting results

The first series of tests showed that most of our configurations can run in real-time or near real-time on average, but can potentially fallback to interactive speed in worst case. Unfortunately for this setup the impostor system doesn't

introduce any marginal speed up, but also don't slow rendering speed as well. We suspect that this is caused by the fact that the impostor system also refreshes actually not seen parts of forest. Therefore a more sophisticated impostor refreshing algorithm should be implemented to avoid updating parts of forest actually not seen by the camera. We also note that for forest walkthrough the impostor system can be much more relaxed than it's set in test suite because our walkthrough is in rather fast speed and even for that a much more relaxed impostor system refreshing rate is sufficient.

Nice speed up is between extra high and high detail setting. For forest walkthrough it is more than 100 percent speed up and for flight above the forest it is more than 300 percent speed up. This shows the importance of second metric settings for whole tree node rendering. Without changing the rendering representation from mixed triangle and point representation into point only representation such speed up wouldn't be possible.

The difference between the directional diffuse lighting with ambient rendering path and ambient only rendering path is about 20 percent on average. From this we can conclude that adding extra quality by using better lighting model is rather cheap.

The second series of tests proved that various forest motives can be viewed at 1024x768 normal detail setting in real-time or in near real-time. The exceptional mugo pine forest performance is caused by the fact that the mugo pine tree is shrub. The mugo pine is only 1 or 2 meters high and because of that the tree vanishes from view fast with added distance. There is indeed some dependence of rendering performance on tree size which is quite obvious because of the fact that small trees vanishes from scene quite fast.

The memory usage test proved that the application performance is minimally dependent on amount of used memory as long as whole data can fit into graphic card's memory. We note that we tested scenes which don't fit into memory of graphic card on other graphic cards and our Viewer application don't crush and still provides some rendering speed making interactive walkthrough often possible with some holes where user must wait a little time for data to be uploaded onto graphic card.

The forest size test series provided data showing that adding more trees to the far distance is rather cheap. We believe that this is caused by the fact that only few points are rendered for the trees in distance. Our representation is just not enough for trees far away. For 20 km x 20 km forest it showed that just walking the node's hierarchy is rather expensive and application is heavily processor bounded in this case. We have the data for smaller spruce forest of 1 km x 1 km and 2 km x 2 km which aren't included in the test series. This is because the used fly path will easily fly off these small forests. We can say that difference between 2 km x 2 km forest and 4 km x 4 km forest is that performance is cut approximately to half.

The fifth series of tests measure effect of various forest densities on rendering performance. It showed that both FPS and primitive count grew exponentially with our 5, 10, 20, 40 and 80 trees per 10000 square meters test set. This is expected result for our point-based rendering technique used. We believe that also for image-based techniques based on billboarding the forest density will have significant importance for performance measuring.

Our one but last series of tests showed surprisingly that our application is still heavily processor bounded. It's apparent from the results that it's best

to use 2x2x2 grid subdivision scheme which minimizes processor bound. Some effort should be required to speed up application in order to use more demanding subdivision schemes.

The last tests showed that using any forest subdivision scheme doesn't have any significant performance impact. However using 32 or 64 maximal nodes per node setting proved to have some small advantage.

From the overall results we can say that our method can render forest scenes in real-time or near real-time speed and that we therefore achieved the goal of our diploma thesis. However the tests showed that some things must be improved to fine tune our solution to achieve even bigger performance and visual quality. We believe that the potential of radical speed improvement is big within our solution.

Chapter 6

Conclusion and future work

This is concluding chapter. First conclusion about our rendering solution is written in section 6.1. Then author's subjective opinion is referred in section 6.2. Finally future work is outlined in section 6.3.

6.1 Conclusion

We have successfully implemented real-time forest rendering solution! While there is still some amount of errors and improvements left behind in current state of implementation, we are sure that our implementation realistically represent pros and cons of using our point-based forest rendering solution. The visual quality of trees for close look-ups is superior thanks to very nice models of trees used from XFrog utility. However, as these models are very detailed, it proved to be rather challenging to display whole forest of them in real-time or at least interactive speed. Only few series of latest graphic cards have potential to display whole and enough dense forests in real-time. This includes NVidia GeForce 6, GeForce 7 and GeForce 8 series and its ATI graphic card equivalents.

The point rendering solution showed to be extremely vertex program computationally intensive. This is problem for graphic cards before the latest series where most of computational power is devoted to pixel shader programs. Currently only NVidia's and ATI's latest series of graphic cards make dynamic assignment of streaming processors between vertex and pixel shader programs possible. We hope that this will improve in future.

Nevertheless our solution is meant to be used in future. The problems to solve still remain. Visual quality of distant trees is poor using point-based approach and point assignment scheme to triangles should be also improved. Another problem poses processor bound of our rendering solution. The pseudo-continuous view-dependent hierarchical level of detail for trees proved to be computationally intensive for processor. We believe that our frustum culling speed can be significantly improved. Maybe less computationally expensive spheres can be used for this purpose instead of arbitrary oriented bounding boxes. The level of detail selection alone now uses binary interval searching algorithm which is also not so fast when used 1000 times or more. We have however proposed to use linear array instead dependent on distance from camera in cost of lower number of levels of detail.

The real-time forest rendering is indeed difficult field to master. The tree solely is complex plant with many leaves and branches, and it's the complexity of nature and plants which places man before difficult task to render virtual forests similar with real forests with only limited resources and computational power available. Moreover in nature every plant is in its own way unique. This is even much more challenging than rendering forest with few selected trees of one type for each tree type. To achieve realistically looking unique trees in forest rendering, years of research would be required to study various tree types growing habits and look in order to mimic tree's uniqueness. This is totally out of scope of this diploma thesis and cannot be completed by only one person.

The importance of this work cannot be underestimated. We believe that we have extended the knowledge in point-based real-time forest rendering significantly. We pinpointed weak spots of this approach, researched several improvements and even were able to complete implementation. Unfortunately our novelty impostoring approach doesn't prove by our test suite to bring any advantage. On the contrary to test results we believe that the impostor system actually brings in many cases significant improvement. It's simply more an issue of fine tuning and implementing some small improvements which needs to be done on impostor system to unravel its real power. In fact our impostor system approach has nice properties regarding to hierarchy node structure. To impostor for example single trees only doesn't have sense.

We decided to left our test suite as it's once we have it completed. We could alter our test suite for example to provide better results for our impostor system, but we did not do so. By this some darker side of our implementation was revealed in best way and it may be useful in making our implementation better. It's much more useful than some nice results carefully chosen by several tests.

Before we'll proceed to subjective opinion, we would like to say that we threat our work as successful despite the number of issues which arises during development of various parts for our forest rendering solution.

6.2 Subjective opinion

I, author of this diploma thesis for forest rendering will now speak in first person for this section. I have written other parts of this diploma thesis in plural or in passive to comply with publication standards of research work. As much as plural maybe misleading to thought that this diploma thesis is made by more than one person, it's not so. The plural is my preferred publication style of choice because one must be aware of fact that every work is not only work of person in meaning of man, but also work of inspiration, clever advise and spirit of science. And in this light it's most appropriate to present work in plural of we as I, inspiration, spirit of science and everyone somehow involved in making this work even by small amount of for example an advise how to use text editor which in case of LaTeX is often really necessary. This is for clarifying of my presentation style.

It took me almost a year to finish this diploma thesis. The amount of work I have done is in my opinion rather large. I have four utilities and one Viewer application. The code is approximately 25 000 lines of unique code. This count includes free lines and commentaries, but doesn't include that I actually utilize

heavy reuse of coded parts among all four utilities and Viewer. During the development of my forest rendering solution I used approximately 35 trees to test the solution and most of them for final test suite.

I programmed most of implementation parts myself. I used third party libraries only for rendering and textures loading which I really don't want to write myself. Perhaps too much used for DirectX luxury from previous racing game project (see <http://www.crazyanimals.net>) I was working on, I found that OpenGL mathematical library support is miserable at best. So I developed my light-weight mathematical library on need to use basis during development. This is good example of how code evolved during project development. First the mathematical library has only few functions regarding to vector dot and cross products, points average and so on. Then when need arises I started to add some matrix support and at the end I added even quaternion support stuff. Actually mathematical library is now in state that I can completely throw off every OpenGL matrix operation call and even these calls for setting worldview and projection matrixes because I really don't need them any more when using shaders. Only the last step of generalizing some things such as vectors is not done for mathematical library and last step of refactoring is required to make mathematical library more general to use it elsewhere. I found this production system quite effective because everything worked with minimal programming effort. When a need arises, I refactored particular part of code to support what I needed.

Sometimes it proved that for example spending some time on serialization support will be more productive and will lead to better performance, but it was often difficult at the time I was doing on the parts of program to realize that I will need such thing so much. And sometimes I spent more time on some particular thing or algorithm. For example I have in my implementation a general templated implementation of quad tree which actually proved very useful and is easily portable to any other project. This templated implementation has an advantage that you can program the quad tree to store almost anything you want, the quad tree division scheme can be altered in any way and support for intersection with any number of classes can be also added easily.

Regarding to the visual quality of the solution I'm little bit unsatisfied with distant trees where the visual quality is not optimal. This was stated in paper *Point-based rendering of trees* [11] in rather misty way difficult to decipher. They also don't state what is their representation for even further trees rendering and in what conditions they carry their forest testing. The problem with holes in trees were not also stated or simply I wasn't able to handle it effectively myself. Maybe they also used simple solution of circumscribed circle area of triangle to deal with this problem. From my personal testing I'm almost sure that this can however cut performance to half in some cases.

From my opinion the most painful problem with this solution is that the detail is lost radically with increasing distance. It's difficult to produce good point average from set of points. For point position it would be perhaps best to test branches and leaves mass in the area to better position the point. With color and normal generation it's even more difficult to say what is right color and normal.

I'm satisfied with impostor system which I believe will work well when some minor changes are made. After I had seen the results from test suite I made some personal testing to pinpoint whether the problem of ineffectiveness of impostor

system is serious and it turned out that it isn't. For example for thematic1.forest it makes about 40 percent speed up with same fly paths used. I just decided not to make any relevant changes after I ended with developing implementation. I held to this and therefore I didn't try to make any implementation changes to bring impostor system back to help with rendering performance in these bad cases.

I hadn't made any heavy fly path testing before I tried my test suite and with subjective testing it seemed that impostor system brought performance advantage in every case. However the test suite testing proved this to be slippery. For upcoming projects it maybe good to develop some sort of test suite first in order to have more relevant data at hand earlier in the project when it comes to performance measuring.

The only thing I didn't do is to bring forest into motion. I calculated that for some solution it can take me two or three weeks to implement it and I was out of the time. The same it's with terrain rendering. When my project leader tells me that he would like to see the forest on the terrain, I responded that it will take me several weeks to implement something like this. And it's really true that it will take minimally three weeks to implement some sort of terrain rendering solution and after turning this feature on the performance will be cut to half. This is reality of time to take for implementation. Even 4 square kilometers terrain grid only solution will look ugly because the terrain cells will be 5x5 meters big. Not talking about 320 000 triangles needed for such grid. And for bigger forests such solution is not bearable. There is no easy way out of this.

The problem assignment for this diploma thesis covers several topics and therefore it's very extensive. To cover such wide area I decided not to look for minor issues and not to fix every visual quality defect or error. When you are staying before such big task as is real-time forest rendering without any implementation basis, utility support or something like this, then you are at position to make first time implementation. And after I completed this project, I actually have this utility support, structures lay out and 25 000 lines of code to build on. Moreover I have completely reconstructed tree hierarchy at hand when I decide to change my level of detail method to account for tree hierarchy, I can do it. It's only now after the first time implementation has been done that problematic areas are uncovered and that parts, which need refactoring, are pinpointed. It's not before! It's really hard to tell what will cause the problems before any implementation is written. For these reasons the actual codebase is very valuable although some parts of code can be really badly written which doesn't matter! They can be rewritten in better way and with relative easiness. The question which matters is: Would we know what parts to make more general and in what way, if there wasn't some implementation coded? And for that I answer that in most cases no.

6.3 Future work

We have outlined most of possible improvements in appropriate subsections in chapter 4. We will therefore now introduce some priority plan for improving our forest rendering solution.

The testing revealed that two things require immediate attention. First thing

is to fix impostor system to provide speed improvement in every situation. The second thing is to try to speed up hierarchy node's walkthrough and to measure application performance with some utility which tells what parts take the longest time on processor such as AMD CodeAnalyst.

Then some visual quality improvements should take place. Probably the easiest is to get rid of points flying solely in the space. This requires to position points only on the positions on triangle where the texture is not transparent overmuch. It will be also good to get rid of our error where oversized points are rendered although it may not be easy to find what is causing such error.

Some more serious changes should take place after. This includes rewriting parts of codes to be more general and to support more features. The best candidates are data structures to make them more flexible, rendering engine to easier task of rendering with flexible data structures and various shader programs and impostor system to support impostors of different sizes.

After these changes the doors are opened to make either big improvements to tree level of detail, for developing forest motion system, or to improve forest lighting by adding ambient occlusion. The tree level of detail improvements include rewriting point distribution scheme to get rid of the holes in tree when rendering with points and to divide tree according to its reconstructed tree hierarchy. Possible extension by forest motion system is discussed in whole section 4.7 and how to add ambient occlusion is relatively straightforward.

Probably the most difficult will be to add some terrain rendering support in the way not to harm application rendering speed overmuch and to try to make some rendering approach for trees in far distance. These tasks are not to be taken lightly and may prove difficult.

The area of improvements is large and many other small or big improvements can be made. For example second method for creating forest trees placement can be implemented for our forest creation utility to support either park trees placement or to designate areas where trees cannot be placed such as roads. We will left other improvements for decision of someone who will possibly continue on our work.

Chapter 7

Image gallery

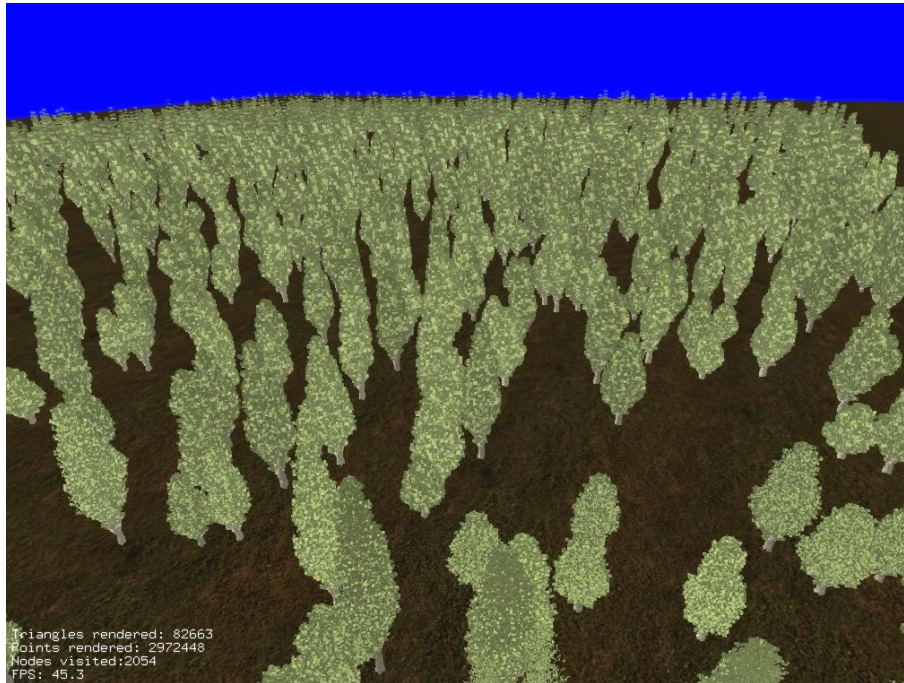


Figure 7.1: The poplar forest in normal detail used in first series of tests in test suite.

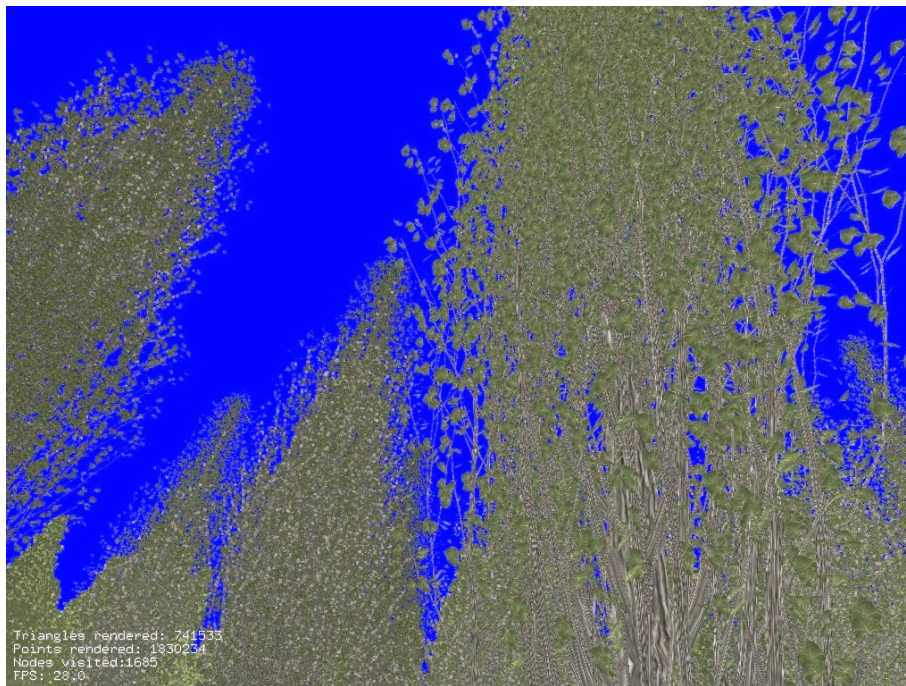


Figure 7.2: Detail of poplar forest in normal detail used in first series of tests in test suite.

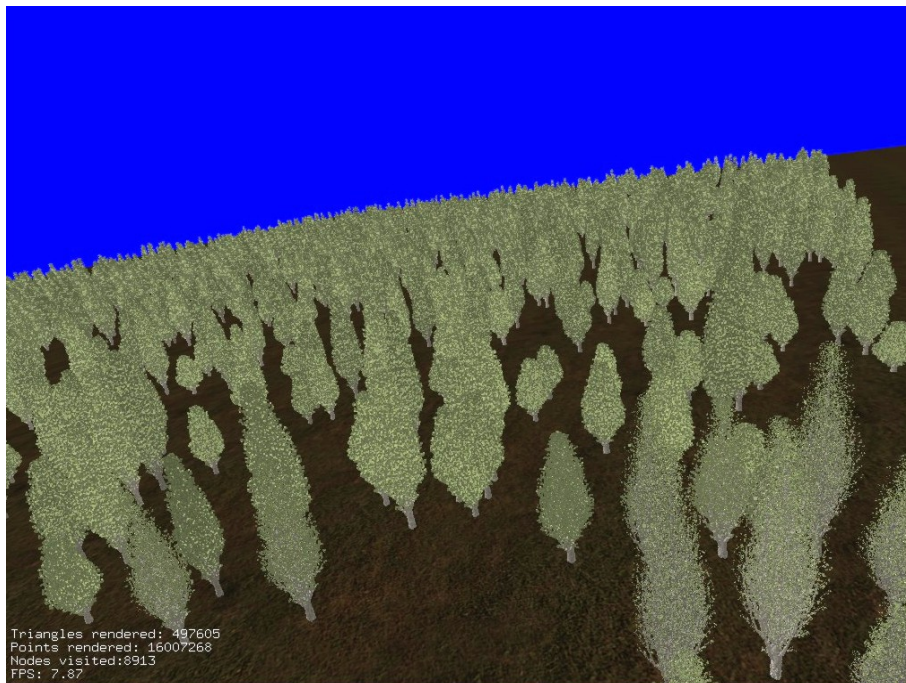


Figure 7.3: The poplar forest in extra high detail used in first series of tests in test suite.

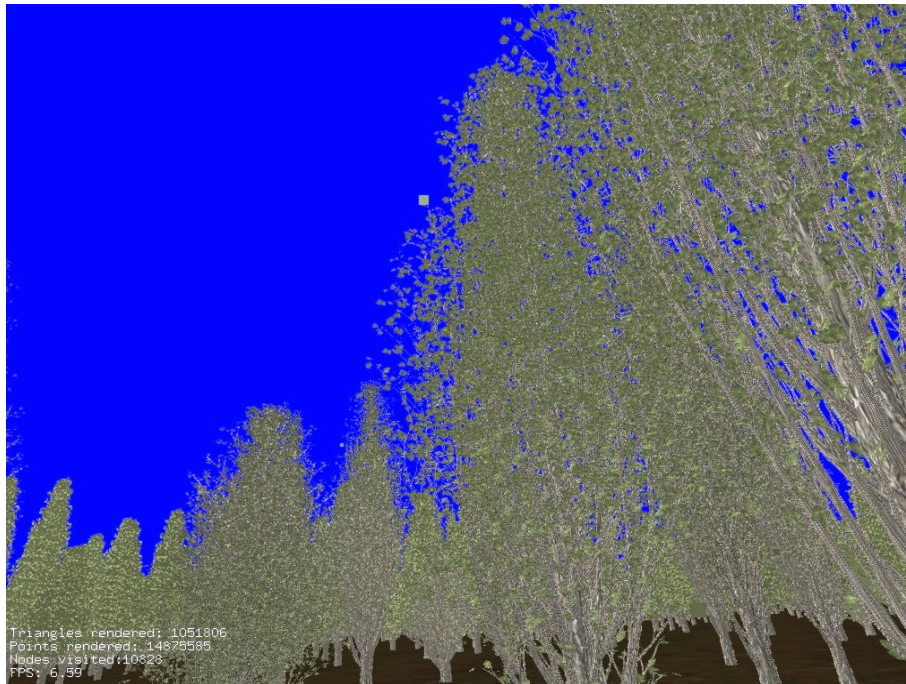


Figure 7.4: Detail of poplar forest in extra high detail used in first series of tests in test suite.

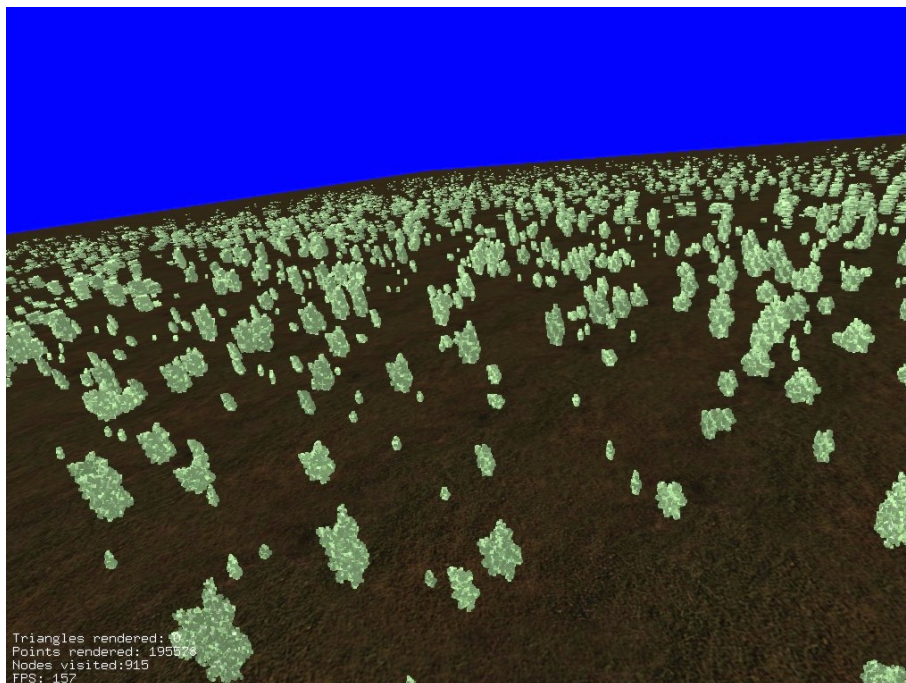


Figure 7.5: The birch forest used in second series of tests in test suite.

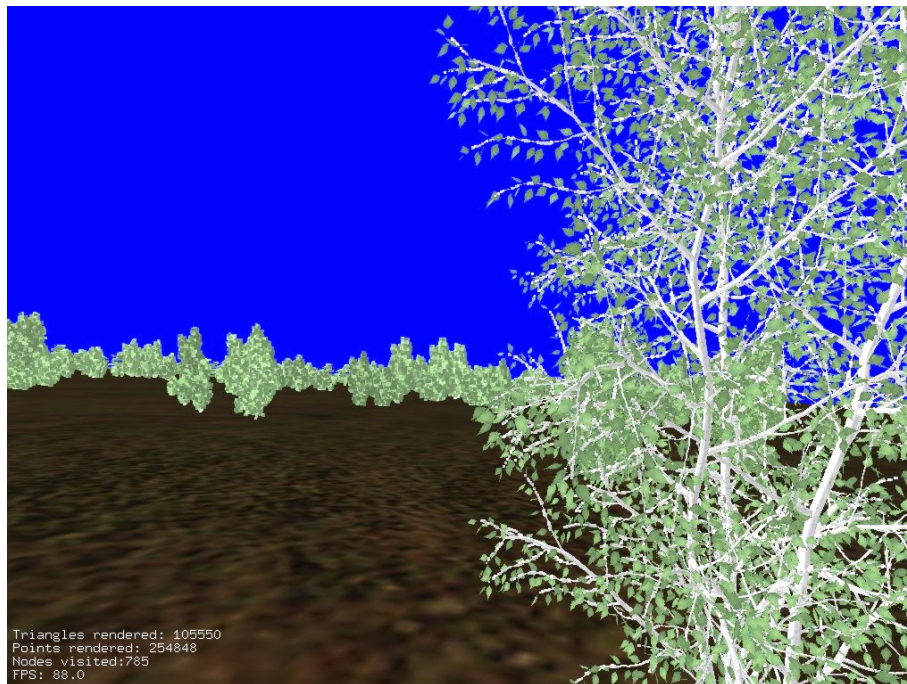


Figure 7.6: Detail of birch forest used in second series of tests in test suite.

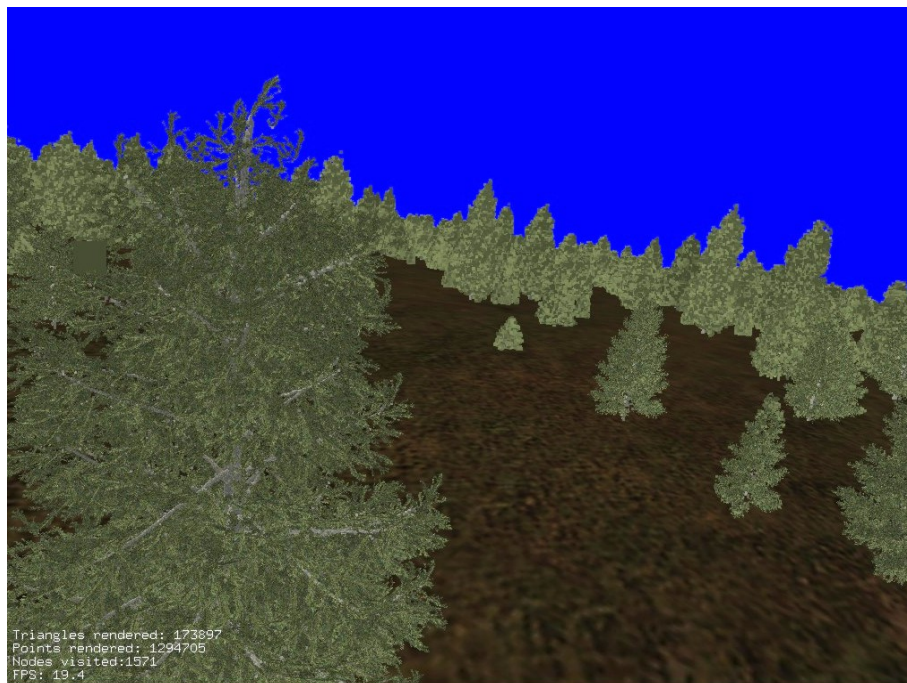


Figure 7.7: The Colorado spruce forest used in second series of tests in test suite.

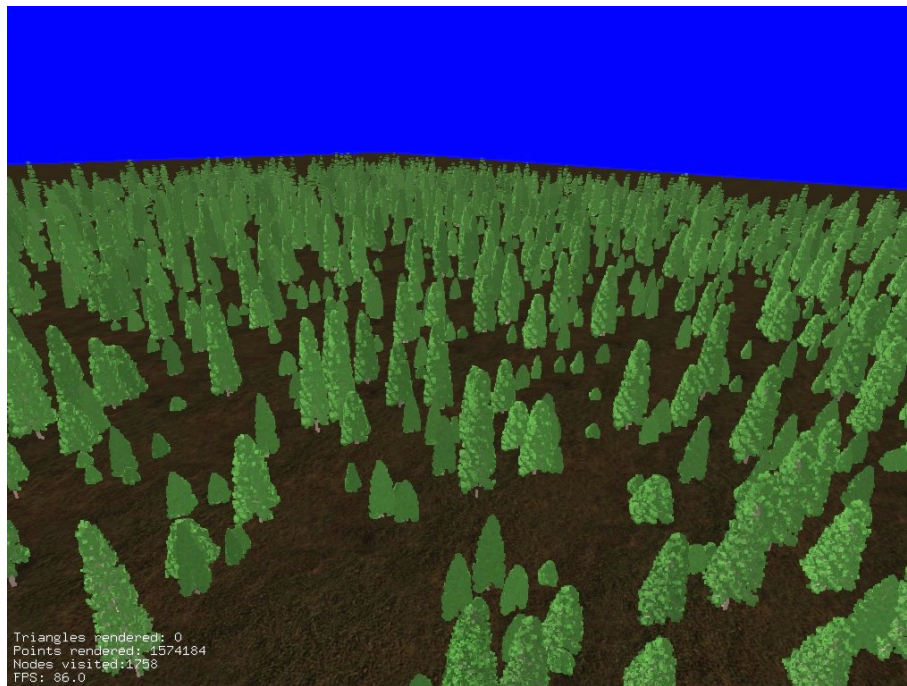


Figure 7.8: The Lawson false cypress forest used in second series of tests in test suite.

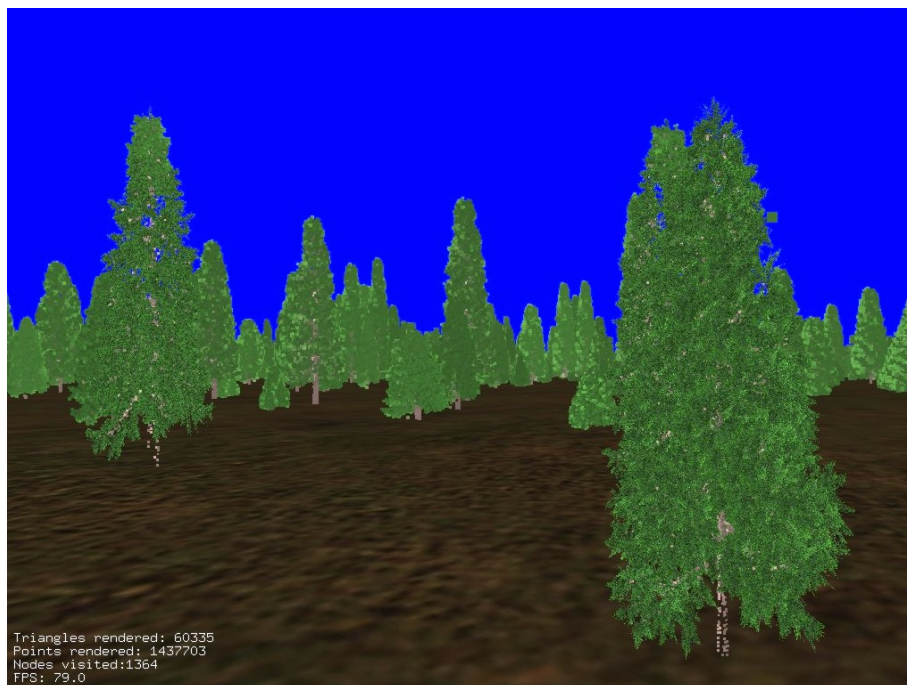


Figure 7.9: Detail of Lawson's Falsecypress forest used in second series of tests in test suite. Notice holes in the tree.

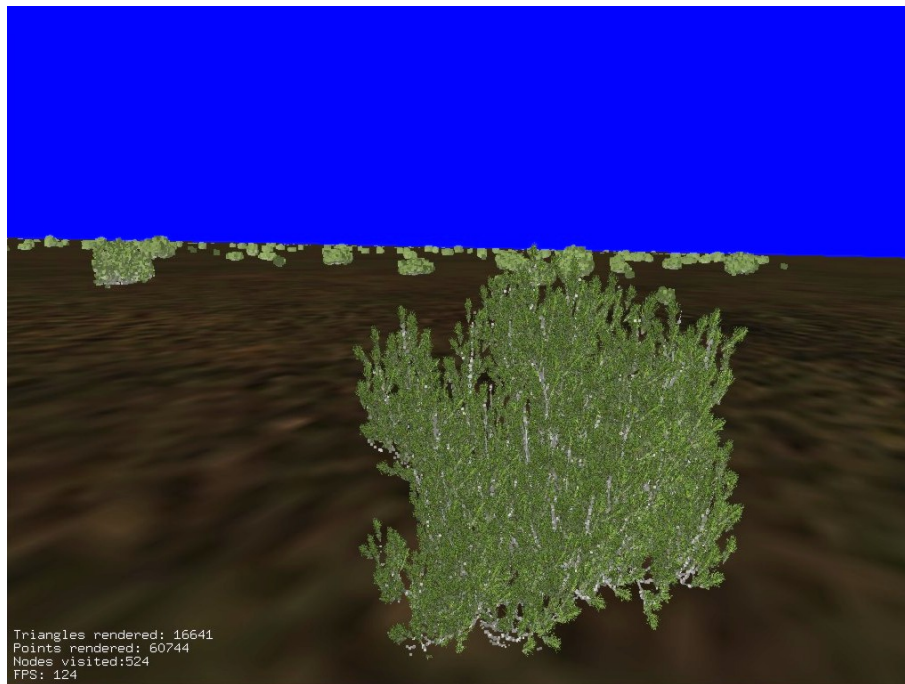


Figure 7.10: The mugo pine shrub forest used in second series of tests in test suite.

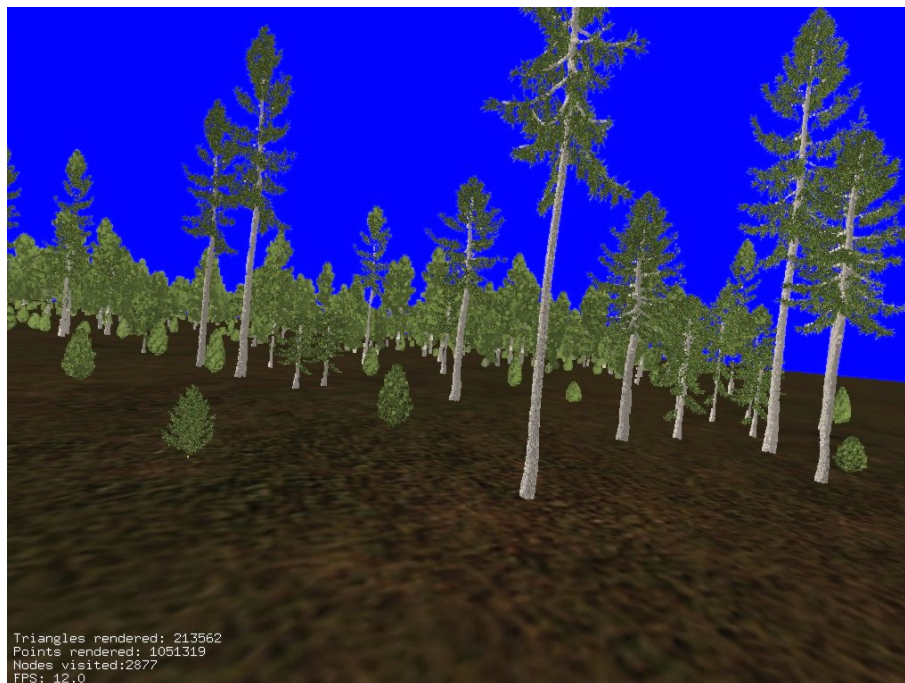


Figure 7.11: The norway spruce forest used in second series of tests in test suite.

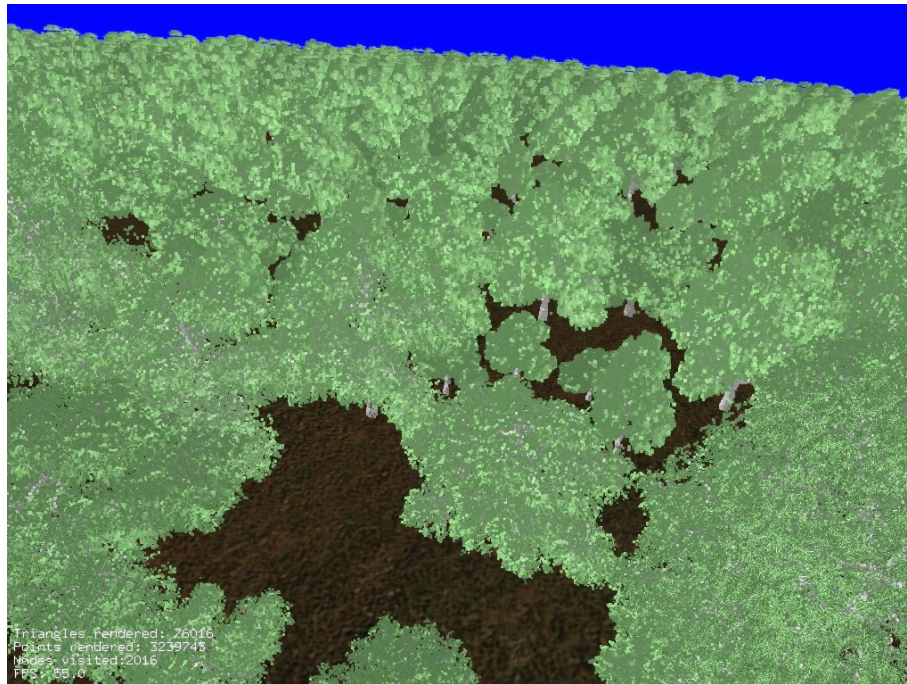


Figure 7.12: The pecan forest used in second series of tests in test suite.

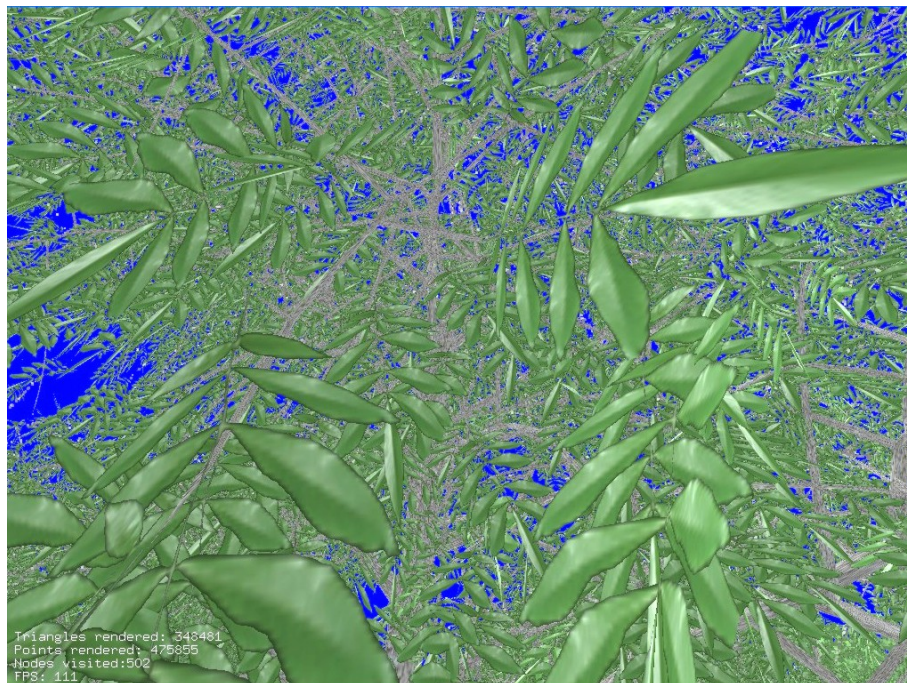


Figure 7.13: Extreme close detail of leaves on pecan tree.

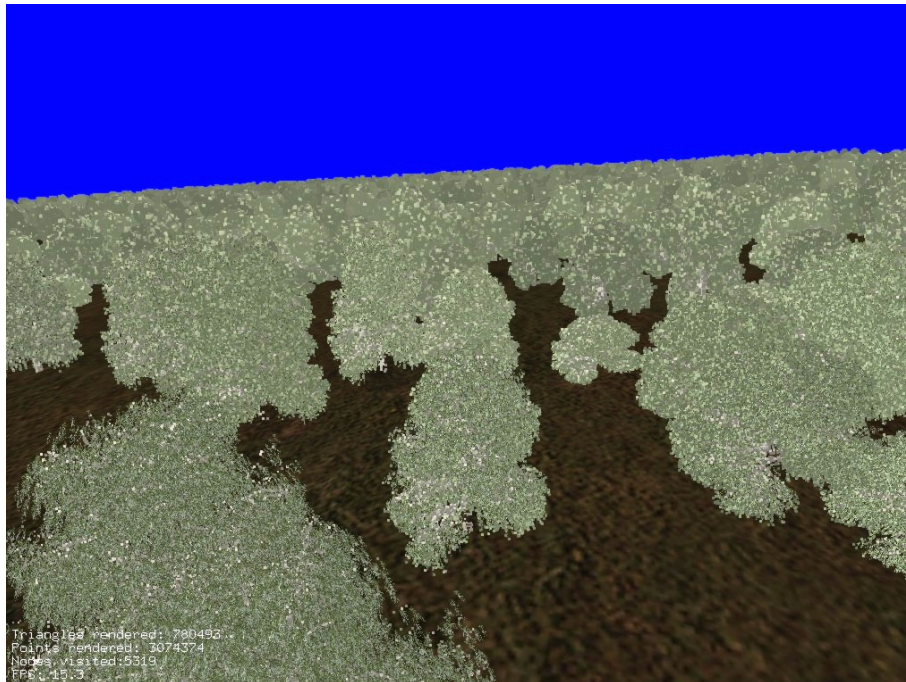


Figure 7.14: The willow forest used in second series of tests in test suite.



Figure 7.15: Detail of willow forest used in second series of tests in test suite.



Figure 7.16: The falseacacia forest used for memory usage testing in test suite.



Figure 7.17: Detail of falseacacia forest used for memory usage testing in test suite.

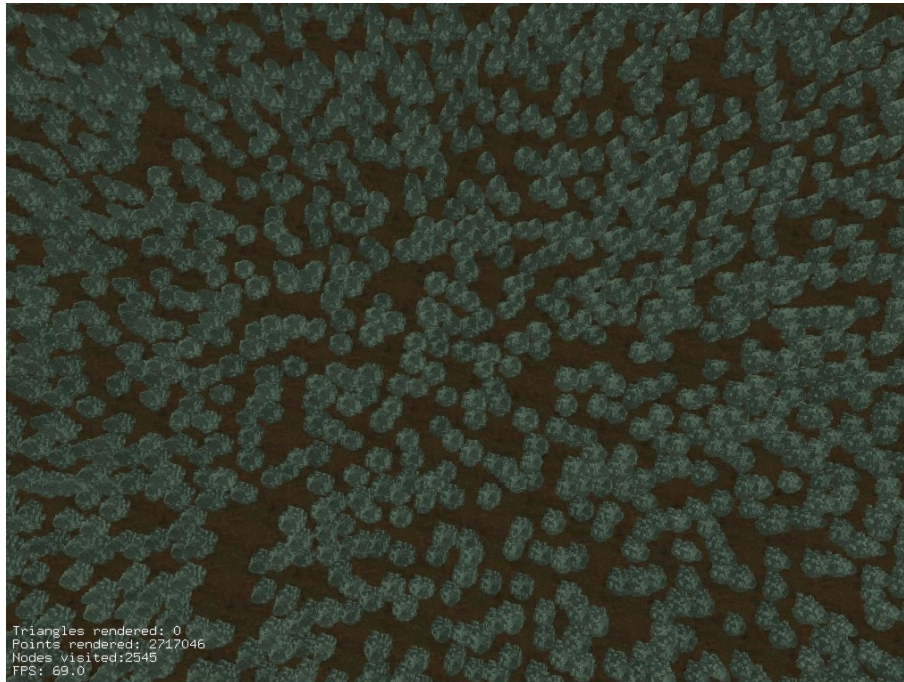


Figure 7.18: The spruce forest used for forest size testing in test suite. This is look-down from high above.



Figure 7.19: The spruce forest used for forest size testing in test suite.

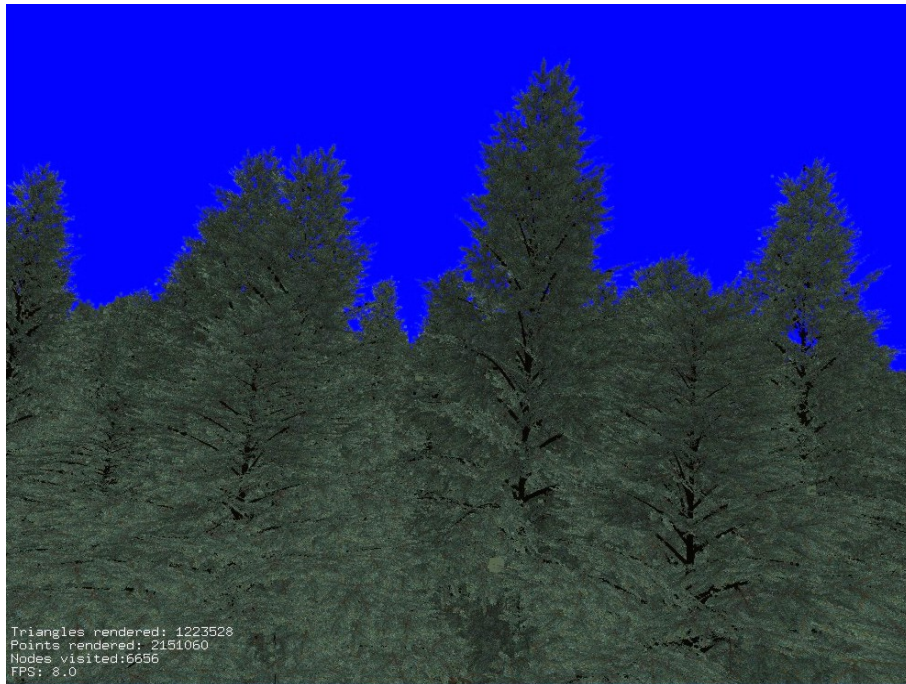


Figure 7.20: Detail of spruce forest used for density testing in test suite. This is with density of 40 trees per 10000 square meters.

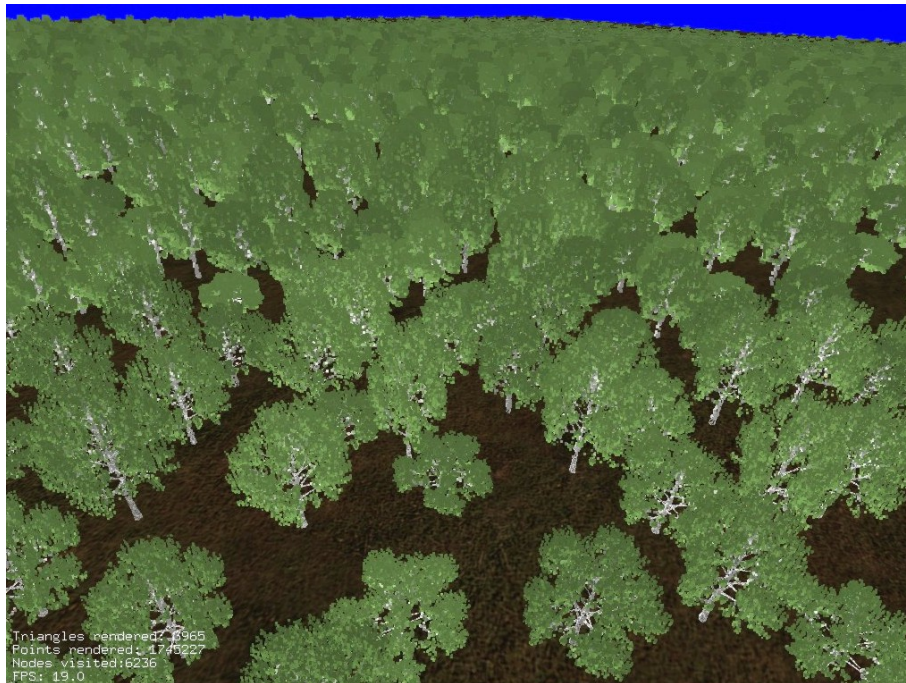


Figure 7.21: The falseacacia forest used for density testing in test suite. This is with density of 40 trees per 10000 square meters.

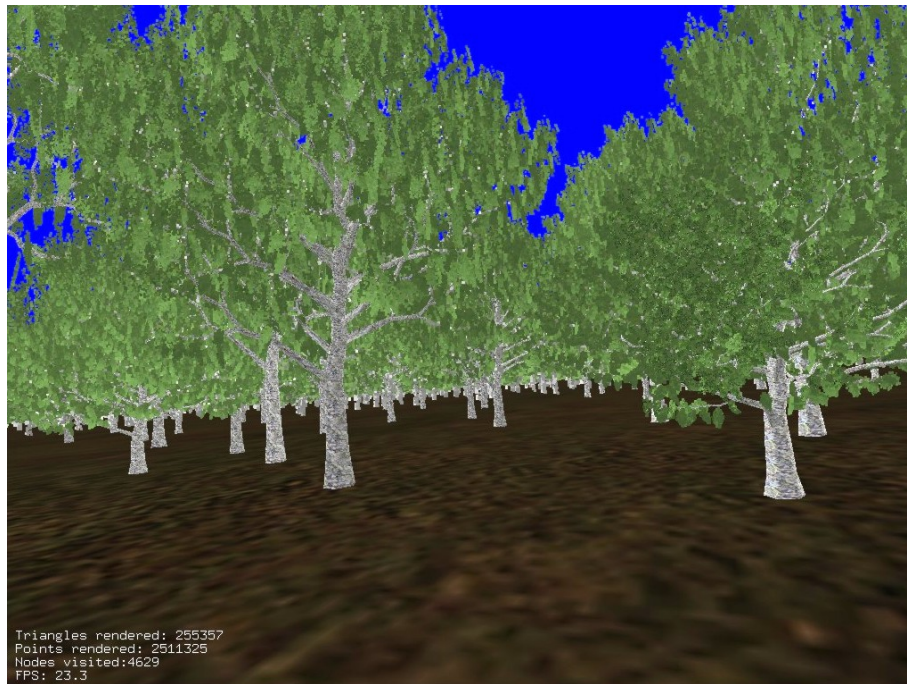


Figure 7.22: Detail of falseacacia forest used for density testing in test suite. This is with density of 40 trees per 10000 square meters.

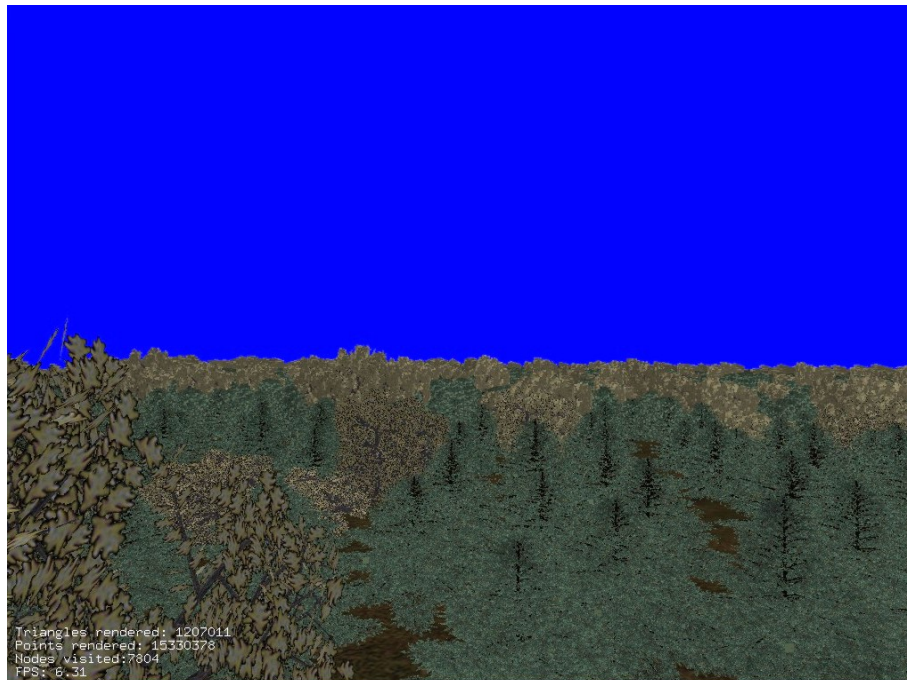


Figure 7.23: Thematic forest of intermixed oak and spruce.



Figure 7.24: Detail of oaks from thematic forest of intermixed oak and spruce.



Figure 7.25: Detail of maple tree. There is the problem with yellow points created from leaf petiole.

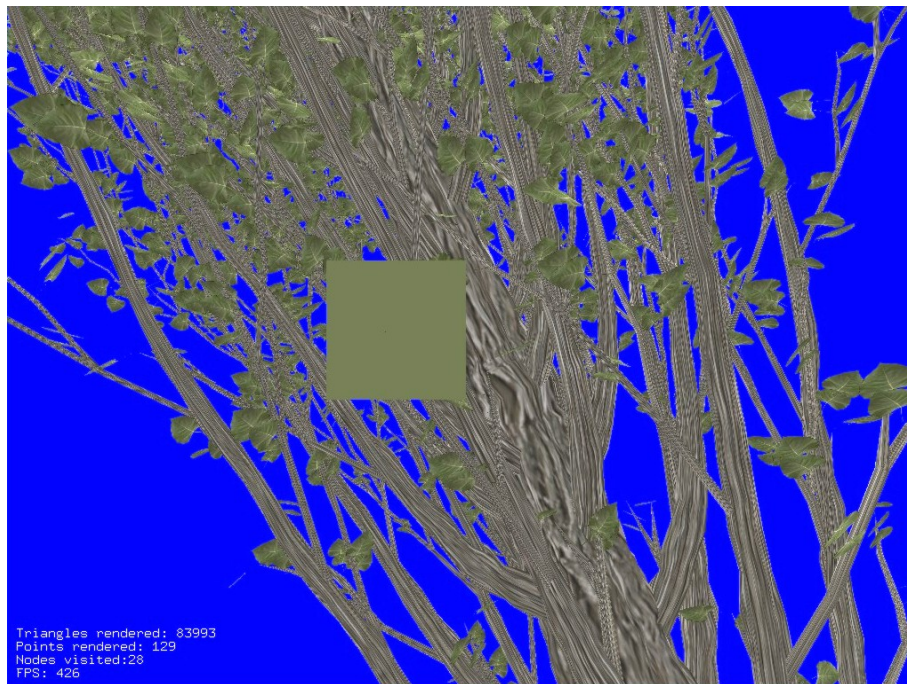


Figure 7.26: Detail of poplar tree with extra big point. This is error in our implementation.

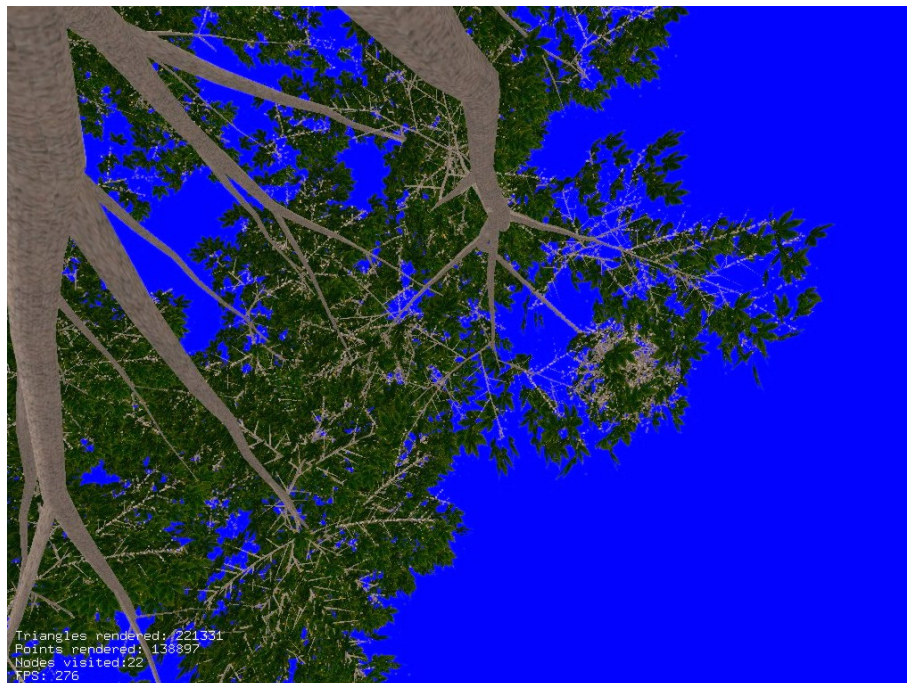


Figure 7.27: Problem with scattered points for small branches. This is detail of silver maple tree.

Appendix A

Installation

The installation is simple. Simply copy all DVD content to the hard-drive. If you don't have installed Visual Studio 2005, then you have to install Visual Studio 2005 redistributable files. The installation is contained in directory: Implementation/Runtime/

The DVD has following structure:

Text Contains text of diploma thesis in PDF format.

Implementation Contains our implementation.

ConvertToLODTree Contains conversion utility to LOD tree representation.

ConvertToSimpleTree Contains conversion utility to simple tree representation.

Forest Creation Contains forest creation utility along with all forests and forest creation configuration files. All forests can be recreated by running MakeForests.bat file.

FullTreeConvert Contains utility for conversion to our basic tree representation.

Generated Trees Contains all tree representations.

Runtime Contains installation of Visual Studio 2005 runtime.

Test suite Contains our test suite.

Utilities Contains source code for utilities along with solution files for Visual Studio 2005.

Viewer Contains our Viewer application.

Viewer project Contains source code for Viewer application along with solution file for Visual Studio 2005.

For viewing any forest with our Viewer application: Go to the "Implementation/Viewer" folder and type Viewer followed by forest name. For example: Viewer "../Forest creation/Thematic1.forest" The configuration parameters and usage of Viewer application is described in subsection 4.6.1.

To run test suite: Run RunAllTests.bat in "Implementation/Test suite/" folder for all tests. Or run SingleTreeTest.bat in "Implementation/Test suite/Single tree" folder for single tree tests only. Or run TestForest.bat in "Implementation/Test suite/Forests" folder for forest tests only.

To reconvert all trees to basic tree representation: Go to "Implementation/FullTreeConvert" folder and run ConvertAllTrees.bat. The usage of FullTreeConvert utility is described in subsection 4.2.3.

To reconvert all trees to simple tree representation: Go to "Implementation/ConvertToSimple" folder and run ConvertAllTrees.bat. The usage of ConvertToSimple utility is described in subsection 4.3.2.

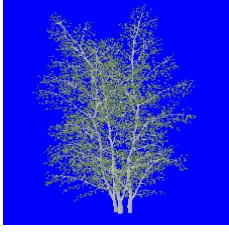
To reconvert all trees to LOD tree representation: Go to "Implementation/ConvertToLODTree" folder and run ConvertAllTrees.bat. The usage of ConvertToLODTree utility is described in subsection 4.4.2.

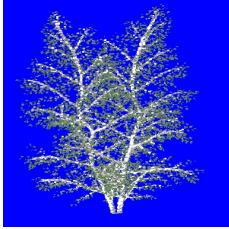
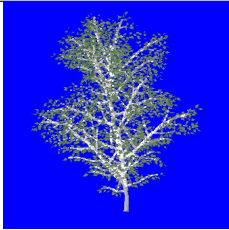
To recreate all forests: Go to "Implementation/Forest creation" folder and run MakeForests.bat. The usage of CreateForest utility is described in subsection 4.5.2.


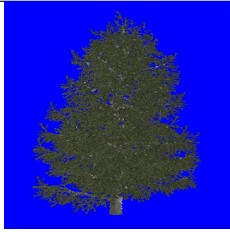
Appendix B

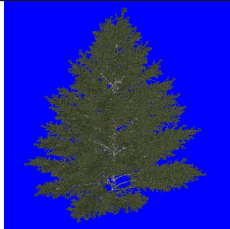
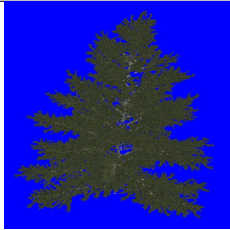
Catalogue of trees

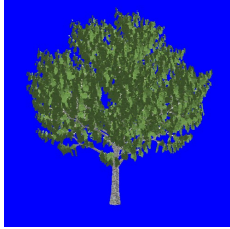

This catalogue contains every important tree with details. Some trees used for test suite may not be listed, but they are trees of listed types with modified conversion parameters.



Tree name:	Birch – adult
Picture:	
Location:	/Generated trees/Birch/
Full tree filename:	BirchA.FullTree
LOD tree filename:	BirchA.LODTree
Simple tree filename:	BirchA.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	147066
Number of triangles for leaves:	22110
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.03
Point scale factor:	2.0
Maximal point group size:	2

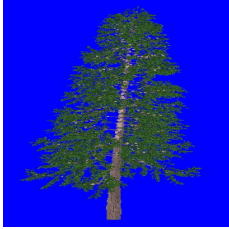

Tree name:	Birch – mediocre
Picture:	
Location:	/Generated trees/Birch/
Full tree filename:	BirchM.FullTree
LOD tree filename:	BirchM.LODTree
Simple tree filename:	BirchM.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	97653
Number of triangles for leaves:	14558
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.03
Point scale factor:	2.0
Maximal point group size:	2
Tree name:	Birch – young
Picture:	
Location:	/Generated trees/Birch/
Full tree filename:	BirchY.FullTree
LOD tree filename:	BirchY.LODTree
Simple tree filename:	BirchY.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	49932
Number of triangles for leaves:	7554
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.03
Point scale factor:	2.0
Maximal point group size:	2


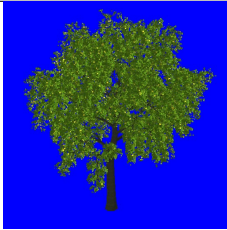
Tree name:	Chestnut
Picture:	
Location:	/Generated trees/Chestnut/
Full tree filename:	Castanea_sativa.FullTree
LOD tree filename:	Castanea_sativa.LODTree
Simple tree filename:	Castanea_sativa.SimpleTree
XFrog filename:	Castanea_sativa.xfr
Number of triangles for branches:	317926
Number of triangles for leaves:	112910
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Colorado spruce – adult
Picture:	
Location:	/Generated trees/Colorado spruce/
Full tree filename:	ColoradoSpruceA.FullTree
LOD tree filename:	ColoradoSpruceA.LODTree
Simple tree filename:	ColoradoSpruceA.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	65200
Number of triangles for leaves:	17500
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.28
Point scale factor:	1.0
Maximal point group size:	5

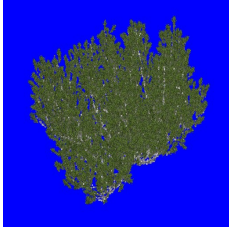
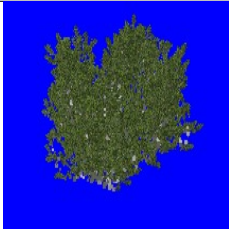
Tree name:	Colorado spruce – mediocre
Picture:	
Location:	/Generated trees/Colorado spruce/
Full tree filename:	ColoradoSpruceM.FullTree
LOD tree filename:	ColoradoSpruceM.LODTree
Simple tree filename:	ColoradoSpruceM.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	54496
Number of triangles for leaves:	9376
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.2
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Colorado spruce – young
Picture:	
Location:	/Generated trees/Colorado spruce/
Full tree filename:	ColoradoSpruceY.FullTree
LOD tree filename:	ColoradoSpruceY.LODTree
Simple tree filename:	ColoradoSpruceY.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	41620
Number of triangles for leaves:	6904
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.05
Point scale factor:	1.0
Maximal point group size:	5

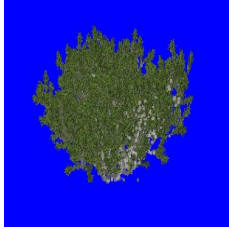
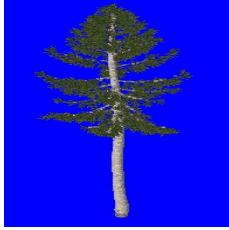
Tree name:	Falseacacia
Picture:	
Location:	/Generated trees/Falseacacia/
Full tree filename:	Robinia_pseudoacacia.FullTree
LOD tree filename:	Robinia_pseudoacacia.LODTree
Simple tree filename:	Robinia_pseudoacacia.SimpleTree
XFrog filename:	Robinia_pseudoacacia.xfr
Number of triangles for branches:	37338
Number of triangles for leaves:	1101276
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.02
Point scale factor:	1.0
Maximal point group size:	2
Tree name:	Lawson's falsecypress – adult
Picture:	
Location:	/Generated trees/Lawson Falsecypress/
Full tree filename:	LawsonFalsecypressA.FullTree
LOD tree filename:	LawsonFalsecypressA.LODTree
Simple tree filename:	LawsonFalsecypressA.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	204877
Number of triangles for leaves:	31444
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5

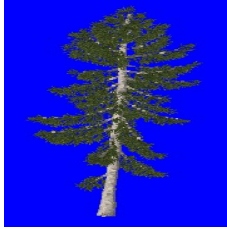
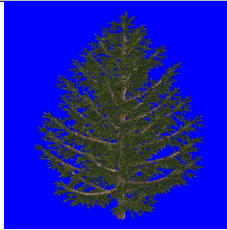
Tree name:	Lawson's falsecypress – mediocre
Picture:	
Location:	/Generated trees/Lawson Falsecypress/
Full tree filename:	LawsonFalsecypressM.FullTree
LOD tree filename:	LawsonFalsecypressM.LODTree
Simple tree filename:	LawsonFalsecypressM.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	235353
Number of triangles for leaves:	23972
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Lawson's falsecypress – young
Picture:	
Location:	/Generated trees/Lawson Falsecypress/
Full tree filename:	LawsonFalsecypressY.FullTree
LOD tree filename:	LawsonFalsecypressY.LODTree
Simple tree filename:	LawsonFalsecypressY.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	104186
Number of triangles for leaves:	18428
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5

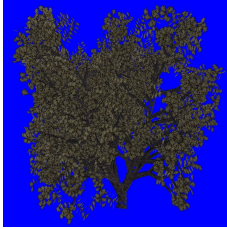
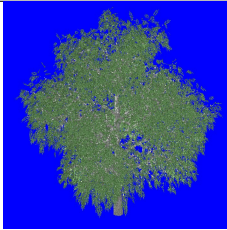
Tree name:	Lawson's falsecypress low resolution – adult
Picture:	
Location:	/Generated trees/Lawson Falsecypress/
Full tree filename:	LawsonFalsecypressALowres.FullTree
LOD tree filename:	LawsonFalsecypressALowres.LODTree
Simple tree filename:	LawsonFalsecypressALowres.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	36164
Number of triangles for leaves:	14796
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.3
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Lawson's falsecypress low resolution – mediocre
Picture:	
Location:	/Generated trees/Lawson Falsecypress/
Full tree filename:	LawsonFalsecypressMlowres.FullTree
LOD tree filename:	LawsonFalsecypressMlowres.LODTree
Simple tree filename:	LawsonFalsecypressMlowres.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	40830
Number of triangles for leaves:	10136
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5

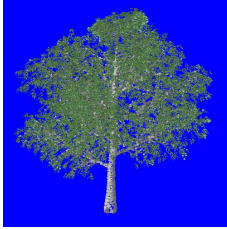
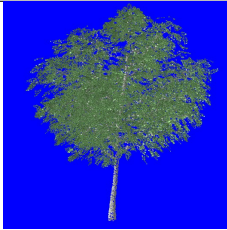
Tree name:	Lawson's falsecypress low resolution – young
Picture:	
Location:	/Generated trees/Lawson Falsecypress/
Full tree filename:	LawsonFalsecypressYlowres.FullTree
LOD tree filename:	LawsonFalsecypressYlowres.LODTree
Simple tree filename:	LawsonFalsecypressYlowres.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	22613
Number of triangles for leaves:	14252
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Maple
Picture:	
Location:	/Generated trees/Maple/
Full tree filename:	Acer_monspessulanum.FullTree
LOD tree filename:	Acer_monspessulanum.LODTree
Simple tree filename:	Acer_monspessulanum.SimpleTree
XFrog filename:	Acer_monspessulanum.xfr
Number of triangles for branches:	16472
Number of triangles for leaves:	780912
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5

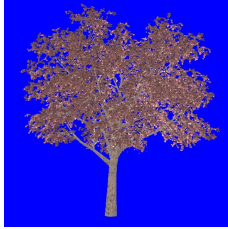
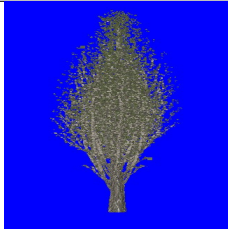
Tree name:	Mugo pine – adult
Picture:	
Location:	/Generated trees/Mugo pine/
Full tree filename:	MugoPineA.FullTree
LOD tree filename:	MugoPineA.LODTree
Simple tree filename:	MugoPineA.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	61610
Number of triangles for leaves:	11956
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.03
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Mugo pine – mediocre
Picture:	
Location:	/Generated trees/Mugo pine/
Full tree filename:	MugoPineM.FullTree
LOD tree filename:	MugoPineM.LODTree
Simple tree filename:	MugoPineM.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	47492
Number of triangles for leaves:	7472
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.03
Point scale factor:	1.0
Maximal point group size:	5

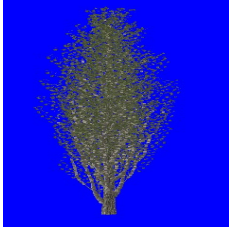
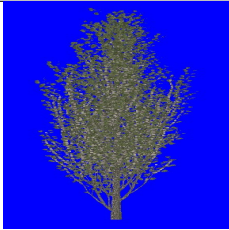
Tree name:	Mugo pine – young
Picture:	
Location:	/Generated trees/Mugo pine/
Full tree filename:	MugoPineY.FullTree
LOD tree filename:	MugoPineY.LODTree
Simple tree filename:	MugoPineY.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	29186
Number of triangles for leaves:	4356
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.03
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Norway spruce – adult
Picture:	
Location:	/Generated trees/Norway spruce/
Full tree filename:	NorwaySpruceA.FullTree
LOD tree filename:	NorwaySpruceA.LODTree
Simple tree filename:	NorwaySpruceA.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	71326
Number of triangles for leaves:	6228
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	2


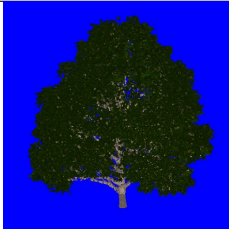
Tree name:	Norway spruce – mediocre
Picture:	
Location:	/Generated trees/Norway spruce/
Full tree filename:	NorwaySpruceM.FullTree
LOD tree filename:	NorwaySpruceM.LODTree
Simple tree filename:	NorwaySpruceM.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	71989
Number of triangles for leaves:	7368
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	2
Tree name:	Norway spruce – young
Picture:	
Location:	/Generated trees/Norway spruce/
Full tree filename:	NorwaySpruceY.FullTree
LOD tree filename:	NorwaySpruceY.LODTree
Simple tree filename:	NorwaySpruceY.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	36724
Number of triangles for leaves:	4656
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	2

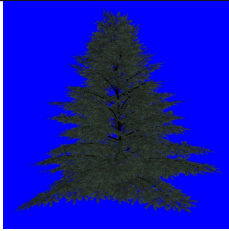
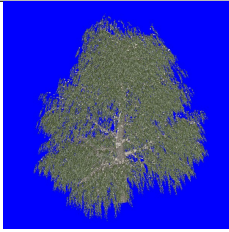
Tree name:	Oak
Picture:	
Location:	/Generated trees/Oak/
Full tree filename:	Oak1.FullTree
LOD tree filename:	Oak1.LODTree
Simple tree filename:	Oak1.SimpleTree
XFrog filename:	Oak1.xfr
Number of triangles for branches:	220000
Number of triangles for leaves:	29396
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.2
Point scale factor:	1.0
Maximal point group size:	2
Tree name:	Pecan – adult
Picture:	
Location:	/Generated trees/Pecan/
Full tree filename:	PecanA.FullTree
LOD tree filename:	PecanA.LODTree
Simple tree filename:	PecanA.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	263468
Number of triangles for leaves:	142750
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	2

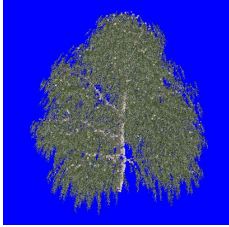
Tree name:	Pecan – mediocre
Picture:	
Location:	/Generated trees/Pecan/
Full tree filename:	PecanM.FullTree
LOD tree filename:	PecanM.LODTree
Simple tree filename:	PecanM.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	117678
Number of triangles for leaves:	52180
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	2
Tree name:	Pecan – young
Picture:	
Location:	/Generated trees/Pecan/
Full tree filename:	PecanY.FullTree
LOD tree filename:	PecanY.LODTree
Simple tree filename:	PecanY.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	102823
Number of triangles for leaves:	106150
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.02
Point scale factor:	1.0
Maximal point group size:	5

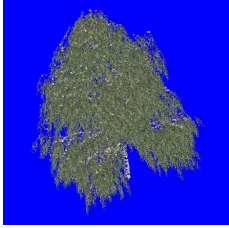
Tree name:	Plum
Picture:	
Location:	/Generated trees/Plum/
Full tree filename:	Prunus_cerasifera_nigra.FullTree
LOD tree filename:	Prunus_cerasifera_nigra.LODTree
Simple tree filename:	Prunus_cerasifera_nigra.SimpleTree
XFrog filename:	Prunus_cerasifera_nigra.xfr
Number of triangles for branches:	41000
Number of triangles for leaves:	164556
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Poplar – adult
Picture:	
Location:	/Generated trees/Poplar/
Full tree filename:	PoplarA.FullTree
LOD tree filename:	PoplarA.LODTree
Simple tree filename:	PoplarA.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	110912
Number of triangles for leaves:	37914
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	2.0
Maximal point group size:	2

Tree name:	Poplar – mediocre
Picture:	
Location:	/Generated trees/Poplar/
Full tree filename:	PoplarM.FullTree
LOD tree filename:	PoplarM.LODTree
Simple tree filename:	PoplarM.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	95308
Number of triangles for leaves:	29710
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	2.0
Maximal point group size:	2
Tree name:	Poplar – young
Picture:	
Location:	/Generated trees/Poplar/
Full tree filename:	PoplarY.FullTree
LOD tree filename:	PoplarY.LODTree
Simple tree filename:	PoplarY.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	48584
Number of triangles for leaves:	14790
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.02
Point scale factor:	2.0
Maximal point group size:	2

Tree name:	Red oak
Picture:	
Location:	/Generated trees/RedOak/
Full tree filename:	Quercus_rubra.FullTree
LOD tree filename:	Quercus_rubra.LODTree
Simple tree filename:	Quercus_rubra.SimpleTree
XFrog filename:	Quercus_rubra.xfr
Number of triangles for branches:	164985
Number of triangles for leaves:	643572
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.2
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Silver maple
Picture:	
Location:	/Generated trees/SilverMaple/
Full tree filename:	Acer_saccharinum.FullTree
LOD tree filename:	Acer_saccharinum.LODTree
Simple tree filename:	Acer_saccharinum.SimpleTree
XFrog filename:	Acer_saccharinum.xfr
Number of triangles for branches:	452432
Number of triangles for leaves:	393358
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.2
Point scale factor:	1.0
Maximal point group size:	5

Tree name:	Spruce
Picture:	
Location:	/Generated trees/Spruce/
Full tree filename:	Picea_pungens_glauca.FullTree
LOD tree filename:	Picea_pungens_glauca.LODTree
Simple tree filename:	Picea_pungens_glauca.SimpleTree
XFrog filename:	Picea_pungens_glauca.xfr
Number of triangles for branches:	9396
Number of triangles for leaves:	11700
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.6
Point scale factor:	1.0
Maximal point group size:	5
Tree name:	Willow – adult
Picture:	
Location:	/Generated trees/Willow/
Full tree filename:	WillowA.FullTree
LOD tree filename:	WillowA.LODTree
Simple tree filename:	WillowA.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	166963
Number of triangles for leaves:	59296
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5

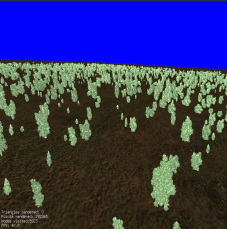
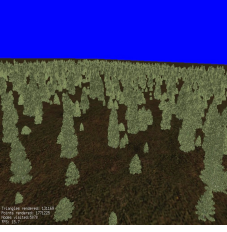
Tree name:	Willow – mediocre
Picture:	
Location:	/Generated trees/Willow/
Full tree filename:	WillowM.FullTree
LOD tree filename:	WillowM.LODTree
Simple tree filename:	WillowM.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	93535
Number of triangles for leaves:	30016
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5

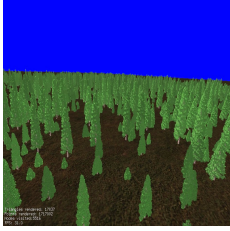
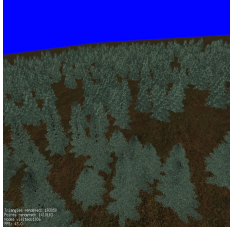
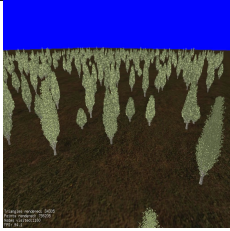
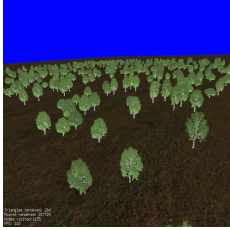
Tree name:	Willow – young
Picture:	
Location:	/Generated trees/Willow/
Full tree filename:	WillowY.FullTree
LOD tree filename:	WillowY.LODTree
Simple tree filename:	WillowY.SimpleTree
XFrog filename:	Not available – copyright issues
Number of triangles for branches:	63447
Number of triangles for leaves:	18216
Conversion parameters:	
Grid subdivision:	4x4x4
Percentage of uplift points:	0.1
Point scale factor:	1.0
Maximal point group size:	5

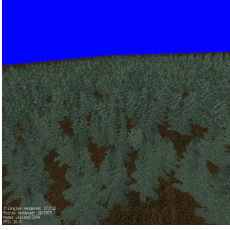
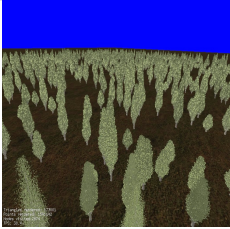
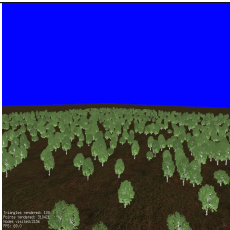

Appendix C

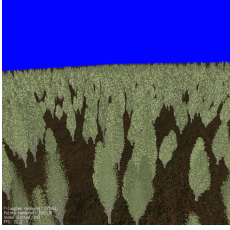
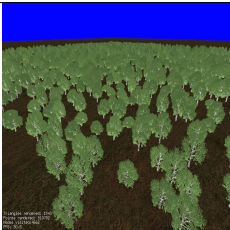
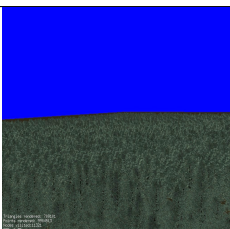

Catalogue of forests

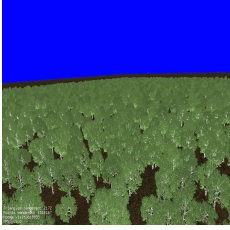
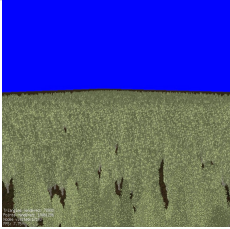
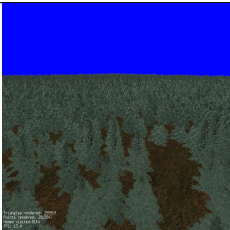
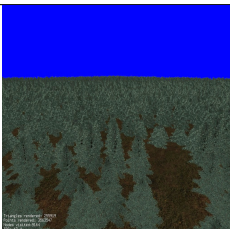
This catalogue contains list of all forests used in diploma thesis. All forests are saved in directory /Forest Creation/.

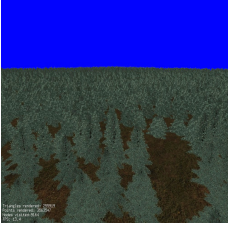
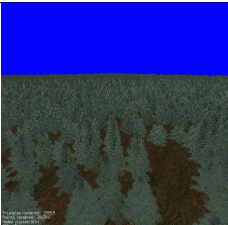
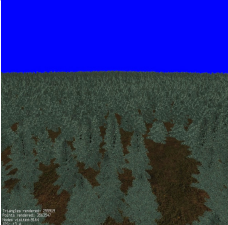
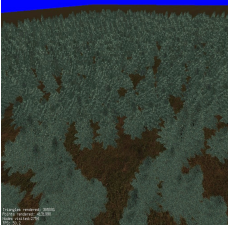
Forest filename:	BirchForest.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Birch – adult (10) Birch – mediocre (10) Birch – young (10)
Forest filename:	ColoradoSpruceForest.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Colorado spruce – adult (10) Colorado spruce – mediocre (10) Colorado spruce – young (10)

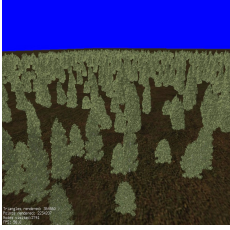

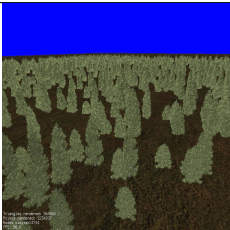

Forest filename:	FalsecypressForest.Forest
Size:	1 km x 1 km
Nodes for node:	64
Picture:	
Contained trees (density):	Falsecypress – adult (10) Falsecypress – mediocre (10) Falsecypress – young (10)
Forest filename:	ForestDensity5.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (5)
Forest filename:	ForestDensity5B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Poplar – adult (5)
Forest filename:	ForestDensity5C.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Falseacacia (5)

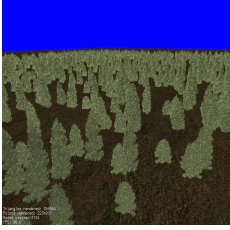

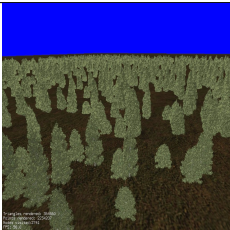

Forest filename:	ForestDensity10.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestDensity10B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Poplar – adult (10)
Forest filename:	ForestDensity10C.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Falseacacia (10)
Forest filename:	ForestDensity20.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (20)

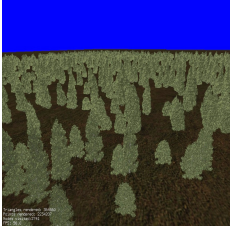
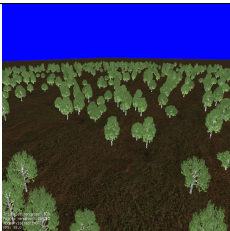

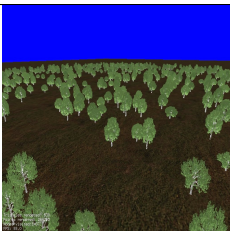
Forest filename:	ForestDensity20B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Poplar – adult (20)
Forest filename:	ForestDensity20C.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Falseacacia (20)
Forest filename:	ForestDensity40.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (40)
Forest filename:	ForestDensity40B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Poplar – adult (40)

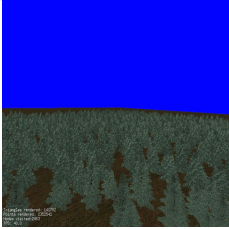
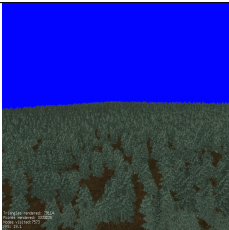
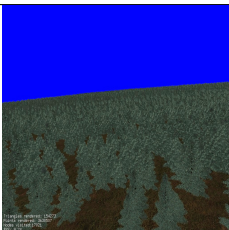
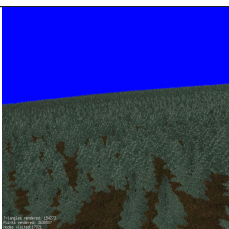
Forest filename:	ForestDensity40C.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Falseacacia (40)
Forest filename:	ForestDensity80B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Poplar – adult (80)
Forest filename:	ForestDivision16.Forest
Size:	2 km x 2 km
Nodes per node:	16
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestDivision32.Forest
Size:	2 km x 2 km
Nodes per node:	32
Picture:	
Contained trees (density):	Spruce (10)

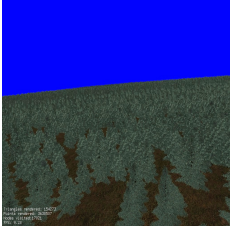
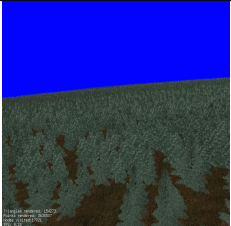
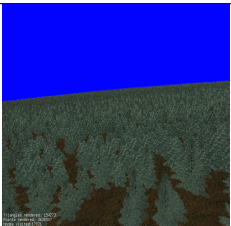
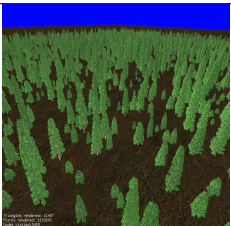
Forest filename:	ForestDivision64.Forest
Size:	2 km x 2 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestDivision128.Forest
Size:	2 km x 2 km
Nodes per node:	128
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestDivision256.Forest
Size:	2 km x 2 km
Nodes per node:	256
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestGrid1.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce with 1x1x1 grid (15)

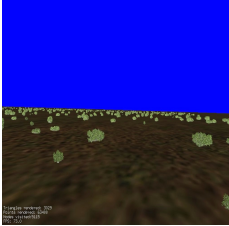
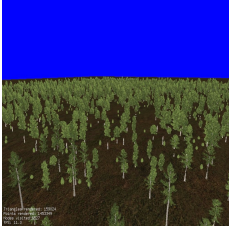
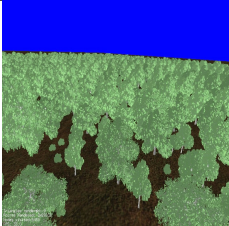
Forest filename:	ForestGrid1B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Colorado spruce with 1x1x1 grid (15)
Forest filename:	ForestGrid2.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce with 2x2x2 grid (15)
Forest filename:	ForestGrid2B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Colorado spruce with 2x2x2 grid (15)
Forest filename:	ForestGrid3.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce with 3x3x3 grid (15)

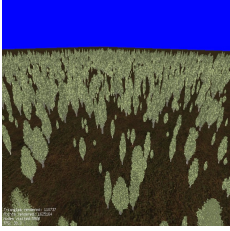
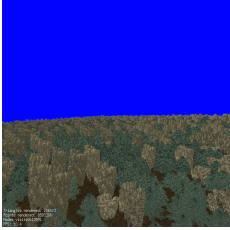
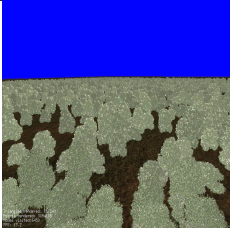
Forest filename:	ForestGrid3B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Colorado spruce with 3x3x3 grid (15)
Forest filename:	ForestGrid4.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce with 4x4x4 grid (15)
Forest filename:	ForestGrid4B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Colorado spruce with 4x4x4 grid (15)
Forest filename:	ForestGrid5.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce with 5x5x5 grid (15)

Forest filename:	ForestGrid5B.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Colorado spruce with 5x5x5 grid (15)
Forest filename:	ForestMemory1.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Falseacacia (6)
Forest filename:	ForestMemory2.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Falseacacia (5) Falseacacia (1)
Forest filename:	ForestMemory3.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Falseacacia (4) Falseacacia (1) Falseacacia (1)

Forest filename:	ForestSize11.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestSize22.Forest
Size:	2 km x 2 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestSize33.Forest
Size:	3 km x 3 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestSize44.Forest
Size:	4 km x 4 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (10)

Forest filename:	ForestSize55.Forest
Size:	5 km x 5 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestSize1010.Forest
Size:	10 km x 10 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	ForestSize2020.Forest
Size:	20 km x 20 km
Nodes per node:	64
Picture:	
Contained trees (density):	Spruce (10)
Forest filename:	LawsonFalsecypressLowresForest.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Lawson's falsecypress low resolution – adult (10) Lawson's falsecypress low resolution – mediocre (10) Lawson's falsecypress low resolution – young (10)

Forest filename:	MugoPineForest.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Mugo pine – adult (10) Mugo pine – mediocre (10) Mugo pine – young (10)
Forest filename:	NorwaySpruceForest.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Norway spruce – adult (10) Norway spruce – mediocre (10) Norway spruce – young (10)
Forest filename:	PecanForest.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Pecan – adult (10) Pecan – mediocre (10) Pecan – young (10)

Forest filename:	PoplarForest.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Poplar – adult (10) Poplar – mediocre (10) Poplar – young (10)
Forest filename:	Thematic1.Forest
Size:	2 km x 2 km
Nodes per node:	64
Picture:	
Contained trees (density):	Oak (3) Spruce (15)
Forest filename:	WillowForest.Forest
Size:	1 km x 1 km
Nodes per node:	64
Picture:	
Contained trees (density):	Willow – adult (10) Willow – mediocre (10) Willow – young (10)

Bibliography

- [1] Szymczak A. Linear interpolation. Published as part of teaching material on website http://www-static.cc.gatech.edu/classes/AY2004/cs4451a_spring/lininter.pdf. Found by searching on Google for triangle linear interpolation.
- [2] Akenine-Möller, Tomas and Eric Haines. *Real-Time Rendering Second Edition*. A K Peters, 2002.
- [3] Brooke Bakay, Paul Lalonde, and Wolfgang Heindrich. Real time animated grass. *Eurographics*, 2002.
- [4] J. Beaudoin and J. Keyser. Simulation levels of detail for plant motion. *Symposium on Computer Animation - Proceeding of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation, Session on Natural phenomena*, pages 297 – 304, 2004.
- [5] S. Benhrendt, C. Colditz, O. Franzke, J. Kopf, and O. Deussen. Realistic real-time rendering of landscapes using billboard clouds. *Eurographics 2005*, 24(3), 2005.
- [6] Philippe Decaudin and Fabrice Neyret. Rendering forest scenes in real-time. *Eurographics Symposium on Rendering*, 2004.
- [7] Oliver Deussen, Carsten Colditz, Marc Stamminger, and George Drettakis. Interactive visualisation of complex plant ecosystems. *Visualisation, Proceedings of the conference on Visualisation '02, Session nature visualization*, 2002.
- [8] Andreas Dietrich, Carsten Colditz, Oliver Deussen, and Philipp Slusallek. Realistic and interactive visualisation of high-density plant ecosystems. *Eurographics Workshop on Natural phenomena*, 2005.
- [9] Losasso F. and Hoppe H. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776. ACM Press, 2004.
- [10] Thomas Di Giacomo, Stéphane Capo, and François Faure. An interactive forest. *Proceedings of the Eurographic workshop on Computer animation and simulation*, 2001.
- [11] Guillaume Gilet, Alexandre Meyer, and Fabrice Neyret. Point-based rendering of trees. *Eurographics Workshop on Natural phenomena*, 2005.

- [12] Tan Kim Heok and Daut Daman. A review on level of detail. *International Conference on Computer Graphics, Imaging and Visualisation (CGIV'04)*, pages 70–75, 2004.
- [13] J. Lluch, E. Camahort, and R. Vivo. An image-based multiresolution model for interactive foliage rendering. *Journal of WSCG*, 12(1 - 3), 2004.
- [14] Javier Lluch, Emilio Camahort, and Roberto Vivó. Procedural multiresolution for plant and tree rendering. *Computer graphics, virtual reality, visualisation and interaction in Africa, Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa, Session on Natural phenomena*, pages 31 – 38, 2003.
- [15] David Luebke, Martin Reddy, D. Jonathan Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of Detail for 3D Graphics*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann, 2003.
- [16] Dana Marshall, Donald S. Fussell, and A. T. Campbell III. Multiresolution rendering of complex botanical scenes. *Graphics Interface*, pages 97 – 104, 1997.
- [17] Shin Ota, Machiko Tamura, Tadahiro Fujimoto, Kazunobu Muraoka, and Norishige Chiba. A hybrid method for real-time animation of trees swaying in wind fields. *The Visual Computer*, 20(10), 2004.
- [18] Frank Perber and Marie-Paule Cani. Animating prairies in real-time. *Symposium on Interactive 3D graphics, Proceedings of the 2001 symposium on Interactive 3D graphics*, 2001.
- [19] G. Szijártó and J. Kaloszá. Real-time hardware accelerated rendering of forests at human scale. *Journal of WSCG*, 12(1 - 3), 2004.
- [20] Daniel Wesslén and Stefan Seipel. Real-time visualisation of animated trees. *The Visual Computer*, 21(6), 2005.