UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Size Equivalent Cluster Trees - Rendering CAD Models in Industrial Scenes

Dissertation

by

Michael Kortenjan

Heinz Nixdorf Institute and Department of Computer Science
University of Paderborn
April 2008

Reviewers:

- Prof. Dr. Friedhelm Meyer auf der Heide, University of Paderborn
- Prof. Dr. Gitta Domik, University of Paderborn

CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

Visualizing industrial scenes containing diverse CAD models is a challenging task, as the individual models, and therefore the complete scenes, are highly complex. Thus, geometrical data exceeds rendering capabilities of common hardware, and scenes are larger than main memory.

Our application is the visualization of scenes simulated by a material flow simulation. Since CAD models of objects contained in this kind of scenes are available, our scenes feature such characteristics. For this application, we have developed the *Size Equivalent Cluster Tree* (SEC–Tree). This data structure enables the user to navigate smoothly through large scenes. Only portions of the tree reside in main memory at a time, which allows handling scenes far larger than available memory capacities. Further, our approach enables the user to interact with the 3-dimensional environment and manipulate the simulation from within the virtual world.

1.1 Motivation

Virtual scenes are widely used to visualize real world processes. Walkthrough systems allow the viewer to navigate through such 3-dimensional artificial worlds. These virtual scenes can provide the user with an impression of what the actual scene looks like or how it could be if this scene represents a project which still has to be realized. Planning of industrial facilities is one such application.

The graphical representation should concur to the real world scene as completely as possible, in order to give the most lifelike image to the user. For once, this means the graphical quality should be high enough to identify details of objects. Even more important is a frame rate high enough to allow interactive navigation through the scene.

These two goals contradict one another. Increasing the visual details of a scene representation results in additional computations. However, these computations slow down the overall performance, including the rendering speed. Therefore, a balanced approach keeps up the necessary frame rate, while at the same time showing as much details as possible under these conditions.

Despite the tremendous progress in graphics hardware during the last years, real-time rendering of large 3D scenes still requires adequate techniques. While especially computer games take advantage of features like vertex and fragment shaders, which allow computing transformations and special effects on the graphics card, other applications hardly benefit.

Shaders do not provide a solution in cases of 3D scenes consisting of millions of polygons. An example of such scenes can be found when looking at models of industrial facilities. Modern companies model their facilities in CAD systems, resulting in models of individual machines consisting of some hundred thousand triangles. Therefore, an entire facility contains some million triangles.

1.2 Material Flow Simulations as a Rendering Application

Planning industrial facilities is a resource intensive task. Machines need to have sufficient capacities, such that not one undersized machine restrains the entire production process. On the other hand, oversized machines tend to be more expansive while their additional capabilities can not be exploited. Trying just every type of machines until the best one is found would be expansive and time consuming and therefore is not practical. Instead, material flow simulations provide a suitable tool to simulate production processes.

In addition, modifications of existing facilities, as restructuring to assembling new products or increasing capacities of the current layout as examples, can be planned using simulations. Potential reactions to unexpected events like, a broken down machine, are tested by picturing the current situation in a simulation environment and projecting the future development. This way, appropriate retaliatory actions can be initiated before the complete production process is affected.

Commercial tools like EM Plant or Simple++ offer several statistics and diagrams to evaluate simulation results. While experts are used to this form of output, communicating the outcome to people who have to arrange the practical realization becomes difficult. 3D visualizations of simulations can help resolving this communication barrier.

A second possible application for 3D visualizations in material flow simulations is the verification of created simulations models. If the model looks exactly like a real facility, incorrect behavior becomes visible and can be corrected.

Existing tools offer only a rudimentary 3D visualization. Standard templates are used, which may vary from the actual items. On the other hand, complex CAD models of machines are available in large companies, which would allow creating an exact representation of the facility. The main problem is the complexity of these models each consisting of some thousand triangles. Therefore, sophisticated methods are needed to display huge plants at real-time.

Visualizing a simulation results in special requirements on the rendering system. The simulated environment contains multiple moving objects, examples are forklifts or products, which are produced by one machine and further processed by another one. Therefore, rendering algorithms, which allow only displaying static scenes, are not suited for this application.

1.3 Graphic Cards are not Everything

Graphics applications are limited by one of four potential bottlenecks. They are either *fill rate limited*, *geometry limited*, *bandwidth limited* or *cpu limited* (see [CW02]). In fill rate limited applications, the per pixel operations, like coloring and processing textures, present the bottleneck. Per vertex operations constitute restrictions in geometry limited cases. Examples are transformations which occur especially in animations, but also projections from an object's coordinate system to the screen coordinate system. Bandwidth limited applications suffer from the large amount of data that has to be transferred from main memory to the graphics hardware. Despite the increase of memory available on modern graphics cards, its size is not sufficient to store large scenes completely on the graphics hardware. If the bottleneck is not related to graphics hardware but computations on the cpu, an application is called cpu limited.

Over the past few years graphics hardware has made tremendous improvements. While especially computer games benefit from these advancements, some other applications, like rendering CAD models, can hardly capitalize on these developments.

New features like programmable vertex and pixel shaders allow manipulating data to create impressive visual effects. Instead of polygons, textures can be used to show details of the scene, resulting in less geometry which has to be processed. Sophisticated lighting can further increase the visual quality of the impressions made by textures.

Computer games rely heavily on these techniques. The main difference to rendering CAD models is that scenes in computer games are created with this application in mind. Models consist of few polygons and detailed textures. The complexity of models and textures are adjusted to the rendering engine. Therefore, games are usually fill rate limited.

CAD models are created, in order to produce an exact image of the object they represent. Every detail, like single screws as an example, are modeled such that it is possible to create the actual object by producing it as a copy of the CAD model. While objects in computer games only need to look like an object, CAD models also have to have correct geometry. This large amount of geometry results in bandwidth limited applications. Therefore, cases in which CAD models are rendered hardly benefit from modern graphics cards.

1.4 Contributions

If the user wants to observe a simulation in a 3D environment, he must be able to move in real-time to any place he is interested in, requiring a sufficiently high frame rate. We have developed a rendering algorithm as part of the material flow simulation tool *d^3FACT* for realtime visualization of simulated processes. d^{3}FACT features several abilities, which are beyond possibilities of concurring tools ([MDL+04]).

For rendering these large simulated worlds under the conditions arising from our application, we have developed the Size Equivalent Cluster Tree (SEC–Tree, [KS06]). This data structure and its application for rendering will be the focus of this work. As its main properties, the SEC–Tree offers

- rendering at a chosen frame-rate with only small fluctuations at different viewing positions in the scene, adjustable to capabilities of the given hardware
- organizing the triangles of objects to obtain a subset of triangles for rendering, while making only small sacrifices on quality of rendered images
- rendering of large dynamic scenes, containing some thousand moving objects. Forklifts or packets created by the simulation, representing produced goods, are examples of such objects
- organizing static scenes on object level, which allows a more efficient handling compared to dynamic scenes
- keeping up object identities, which enables the altering of object parameters or positions at runtime
- displaying scenes larger than memory capacities by applying an out-of-core rendering approach, tested with a scene occupying 66 GB
- adaptive regulation of object details depending on the location of the viewer. Parts of objects near to the viewer are automatically rendered at a higher degree of detail
- rendering of important objects at a higher level of detail. These important objects may be identified by the d^{3}FACT simulation tool

We have analyzed the construction of the SEC–Tree and will show that these costs mainly depend on the costs of a clustering algorithm utilized within the construction process.

Further, we have implemented the SEC–Tree as part of a prototypical rendering system. This system was used to obtain experimental results, proving that our approach works well in practice.

SEC–Trees fulfill the requirements raised by our simulation tool d^{3}FACT. d^{3}FACT allows multiple users to work commonly at the same simulation model, by providing a 3-dimensional multi-user environment. This 3D-world permits manipulating the underlying simulation model. Such interactions with the simulation result in additional requirements. If the user wants to see parameters of a machine or make changes to the virtual world, he must be able to select an object. Therefore, object identity has to be preserved in the rendering system. In contrast to some other rendering systems, our approach fulfills this requirement.

Further, d^{3}FACT supports significant events, which can be reported to the renderer and are specially treated by the rendering system. Such event can be defect machines or filled depots. This supports the user by directing him to locations where his attention is required.

The user is lead to the places, where significant events occurred, by showing a path in the virtual world. This path is automatically determined, started at current position of the viewer and leading around obstacles to the place of the significant by displaying arrows on floor.

Path finding is used for moving simulation objects, forklifts as an example, as well ([FMM$^+$05]). While common simulation tools require the developer to specify paths explicitly, d3FACT determines them automatically. Further, adjusting simulation models no longer requires updating paths manually to consider changed object locations, as this is done by the application. We have developed a new motion planning algorithm to compute the paths in the virtual environment ([MLD$^+$05]).

Recent progress of d^{3}FACT includes the simulation and rendering of multiple experiments in parallel ([DHL$^+$06], [FLH$^+$07]). Since this exceeds the computation power of a single CPU, a distributed approach is used. These multiple simulations are rendered in one single 3D-image, thus allowing a direct comparison of the various parameter sets. At any time, the user can terminate simulations, or clone them to create an additional simulation run, where altered parameters can be tested. Since rendering multiple simulations at once is an even more complex task than rendering just one simulation, the rendering algorithm used also works in parallel.

1.5 The Basic Approach

In order to achieve a rendering system, which is able to display huge scenes at interactive frame rates, we limit the amount of rendered geometry depending on capabilities of the given hardware. The SEC-tree is created for each object as a preprocessing.

At runtime, each object is assigned a weight corresponding to its size and distance to the viewer. According to this weight, the number of triangles chosen from this object for rendering is determined. The SEC-tree is used to select which triangles are rendered. This allows not only displaying objects with different degrees of detail, but further details can vary within one object.

The SEC-tree organizes each object into a hierarchy of details. Larger triangles are at higher levels of the tree, while smaller triangles are stored further down. The tree is created bottom up by finding clusters of the smallest triangles initially. These clusters are combined with triangles of size equal to the accumulated size of contained triangles forming new clusters. Continuing this approach, a hierarchy of all triangles is created.

In static scenes another SEC–Tree is built on object level and used at runtime for frustum culling and weight distribution. While this is not possible in dynamic scenes, as SEC–Trees are a static data structure, scenes containing some thousand moving objects can be handled without any sophisticated global data structure managing the objects, using only SEC–Trees for the single objects.

To allow scenes larger than main memory, only parts of the objects possibly needed within the next frames are loaded from hard drive. A separate thread is responsible for prefetching and deleting geometry to avoid waiting times caused by hard drive access.

Object details can be increased at points of interest determined by a simulation [FMM$^+$05]. Depending on the level of significance, such objects obtain additional triangles. These triangles are either redistributed from other, less important, objects or assigned in addition, allowing exceeding the budget in favor of details. Further, instead of emphasizing just the one significant object, other objects surrounding it can be increased in detail well.

1.6 Overview

The further is structured as follows: In Chapter 2 we cite work of other authors related to the topic and compare them to our approach. Chapter 3 introduces the SEC–Tree in detail, describing how it constructed, followed by analyzing the costs of this process. In Chapter 4 we will show how to use the SEC–Tree to render a scene. Then, some details on the implementation are presented in Chapter 5. In Chapter 6 we will present practical results obtained from the prototypical system. Finally, we draw our conclusions and give an outlook on future possibilities in Chapter 7.

CHAPTER 2 PREVIOUS WORK

By the increasing use of CAD modeling to support industrial purposes like visualization of product flows or industrial plants, the object and thus the scene complexity has grown tremendously. Different strategies appeared to overcome the problem of rendering large scenes, which are too complex for obtaining interactive frame-rates, when rendering the complete scene.

Neglecting complexity or brute force rendering will not consider the complexity of the modeled scene with all its objects and will likely fail, even if the scene is small enough to fit into main memory. Current graphic cards may feature more than 130 million triangles per second in theory as peak performance, while this performance is not archievable in practice. Thus, one challenge in displaying complex scenes with interactive frame rates is to find an appropriate reduction of the scene complexity in the number of primitives.

For a brief context overview we give a short introduction in some general techniques in complex scene visualization and give a more detailed description on some work closer related to our approach.

2.1 Level of Detail Modeling

Level of Detail proposed by Clark [Cla76] creates multiple versions of every 3D object as a preprocessing step, each of different complexity. These different representatives are used within the rendering system, depending on the distance from the viewer. One difficulty of this approach is to generate the various coarse-level representations of an object.

The visual discontinuity appearing while switching between different representations motivates the continuous LOD approach [LRC$^+$03]. Rather than creating a few LOD's for each object, the simplification system creates a data structure encoding a stream of continuous details.

Chhugani et. al. subdiveded the view space into *view-cells*, similar to an octree based partitioning scheme and generated *vLODs* [CPK$^+$05], based on the given LODs as view dependent representatives of view-cells. Usually, generated vLODs are much larger than the original model and to large to be stored directly on disk.

2.1.1 Surface Simplification

While creating differently detailed models of an object by hand results in high quality approximations when created by an expert, this is incorporated with significant additional work.

Surface simplification describes a process of automatically obtaining a model with less complexity from a single highly detailed model, and can be deployed to obtain multiple models with different degrees of detail as mentioned LOD models. To obtain this, mainly edge collapsing (contraction), vertex splitting, or methods combining both as described by Garland [Gar99] and Rossignac [Ros04] are used. Several differently detailed models may be obtained and stored as discrete LODs, as constantly redefining objects during rendering may result in a significant computation overhead at runtime.

Hoppe introduced *progressive meshes* [Hop96] to create a sequence of operations refining a simplified mesh stepwise up to the original model. An energy function is used to determine the order, in which edges are processed. Garland and Shaffer combined an initial uniform clustering on the input model (of size n) to produce an intermediate approximation (of size r) with an iterative edge contraction phase [GS02]. Their system produces results comparable to the QSlim [GH97] approach. In advance their new method provides approximations in $1/5$ the time or less and uses less memory.

For deeper insights in the matter of multi-resolution modeling, LOD, surface simplification, and visibility techniques we would like to refer to the following articles Puppo and Scopigno [PS97], Erikson [Eri96] and Luebke [Lue97].

2.1.2 LOD Management

If LODs are selected simply depending on their distance to the viewer, rendering performance may depend heavily on the user's viewing direction and position in the scene. A user located at the border of the scene might see the complete scene at one moment, but by turning his viewing direction all objects may become invisible after a few frames.

Funkhouser and Séquin introduced a heuristic to choose LODs for any viewpoint [FS93], such that rendering costs do not exceed rendering capabilities. A benefit function rates every detail level of any object. A cost function estimates the rendering costs, given a selected detail level for every object. LODs should be selected, such that the benefit function is maximized, while at the same time rendering costs are below some threshold. Selecting LODs satisfying these conditions corresponds to solving a version of the knapsack problem, which is NP-complete.

Therefore an approximation algorithm is used to find a solution which is at least half as good as the optimal solution and has worst case costs of only $O(n \log n)$. Benefiting from temporal coherence solutions of succeeding frames tend to be quite similar. Typically, given the solution for one frame a result for the next frame can be obtained very fast. However, for the first frame or in case the user teleports to a distant viewpoint, selecting LODs might slow down the rendering algorithm significantly and may be even slower than just rendering the complete scene at a high complexity.

The estimation of rendering costs depends on the number of rendered polygons, their vertices and an approximation on the number of covered pixels. Similar to our approach, the benefit function considers the projected size, accuracy at a given distance and importance of an object. In addition, speed of moving objects, projected screen position and detail differences to the previous frame have an impact. For optimization reasons a PVS system is used to identify occluded objects. However, since PVS systems are static, moving objects are not regarded for occlusion culling.

A drawback of static LODs is that selecting LODs requires creating several instances of each object with different detail levels beforehand. This can be done automatically but does not neccesarily lead to good results and is limited by restrictions on the type of models such algorithms can be applied to. On the other hand, modeling different versions of a model by hand leads to good results, but requires enormous effort. Another drawback of fixed LODs is that the degree of details can only be customized for the entire model, while our approach allows increasing details within one model at locations near to the viewer and choose less refined representations at portions further away at the same time.

Out of core algorithms have not been considered within the LOD management system, every LOD of each model is kept in main memory simultaneously. It remains unclear if continuity in the cost and benefit functions could be utilized to efficiently integrate out-of-core rendering.

2.1.3 HLODs

Simple LOD approaches consider only one object at a time and not the overall complexity of the scene. To overcome this, the approach by Erikson et. al. enhanced the classical LOD approach by introducing *hierarchical levels of detail (HOLDs)* [EMWVB01] and presented its performance as a part of a system called *SHAPE*. HLOD supports scenes with limited dynamics and features target frame rendering with a constant frame rate.

The algorithm works on scenes organized as a scene graph [Cla76], [RH94]. At first, this scene graph is reorganized for better support of spatial criteria, i.e. nodes near to each other in the graph should represent geometry located near to each other. For each node containing geometry, discrete LODs are created using the *GAPS* [EM99] algorithm. HLODs represent not only the geometry of one node, but also all succeeding nodes. The finest representation of a node consists of its coarsest LOD and the coarsest HLODs of its children. Using GAPS again, coarser HLODs are created for the node. When an HLOD is rendered, an error bound on each node measures the deviation from the correct image, depending on the distance from the viewer.

Rendering the scene graph, either the frame rate or image quality can be fixed. If image quality is the main concern, the scene graph is traversed and at each node the error bound is used to determine if rendering HLODs of this node results in an appropriate image. If so, following nodes can be discarded, as HLODs represent a node well enough as a replacecment of succeeding nodes. Otherwise, an LOD is selected for the node and children are processed recursively.

When rendering with a constant frame rate is preferred over correct images, a polygon budget is applied. Starting with the root, nodes are recursively traversed to select HLODs from the graph with a combined number of polygons below the budget. As long as the budget is not reached, the scene graph is traversed further, continuing at the node of largest error.

Dynamic scenes may require updating the scene graph structure in order to keep spatial locality of nodes. This goes along with anew generation of HLODs. The authors used a multiprocessor platform to perform these updates parallel to rendering the scene, as regenerating HLODs was more expansive than the actual rendering. Therefore, this approach can handle scenes with few moving objects, but does not cope with their increasing number. Further, the authors used a SGI Reality Monster with 16 GB of main memory, sufficient to store all their test scenes in main memory.

2.2 Sampling

Point based rendering uses points as rendering primitives. In order to reduce the polygon rendering complexity, points can be used to represent as set of polygons. An alternative to using sample points is rendering only sample set of polygons.

These techniques to choose representatives have been early introduced by Catmull [Cat74] and Levoy and Whitted [LW85] and later been picked up by Grossman and Dally [GD98]. Further enhancements providing realtime rendering, including scalability of polygonal scenes, are made by Zwicker et. al. [ZGP00], Rusinkiewicz and Levoy [RL00], and Klein et. al. [KKF$^+$02].

2.2.1 Survels

Pfister et al presented *survels* [PZvBG00] to represent objects by sampled points. In addition to the coordinates of a point, survels contain information about the normal, color and textures of the model at the given position.

In a preprocessing step, survels are created and organized in an octree by a bottom up approach. Initially, the model is projected by three orthographic projections from different directions. Using ray casting, the intersections of rays with the model are calculated and stored as survels. The density of rays determines the granularity of object representation. The set of survels is used to create an octree, which stores survels in its leafs. Every parent node contains half the survels of its children. The survels associated with a node are referred to as a block.

Rendering an object is performed by traversing the octree and projecting blocks to screen coordinates. The number of survels projected to one pixel under an orthographic projection can be estimated and is used as abbortion criteria. The traversal stops if at least a certain number of survels cover a pixel. This may be one survel per pixel for fast rendering or more than one, to improve image quality. Since the estimation is correct for orthographic projections but perspective projections are commonly utilized, holes may appear which require special consideration. Further, anti aliasing and texture filter techniques can be applied to improve image quality.

The scenes considered by the authors contained only one single object. Rendering a scene of a single model with originally about 81.100 polygons on a 700 MHz Pentium III at a resolution of 480 $\times$ 480 resulted in 4.6 frames per second. While this definitely would run smoothly on current hardware, a scene consisting or multiple objects might be problematic. It remains unclear, how the algorithm performs if the complete scene was treated like one object, as this might results in a very different topology. Further, such an approach could only be applied to static scenes. On the other hand, generating and rendering a separate tree for every object might not be fast enough for a scene containing multiple objects.

2.2.2 The Randomized Sample Tree

While survels perform well for objects with lots of small details, large, flat areas are much better represented by polygons. The *randomized Z-buffer* [WFP$^+$01] can be classified as a hybrid method, which overcomes this disadvantage. Its main idea is to represent geometry projected to only a small screen area by reconstructing it from an set of random surface sample points, while accurately rendering geometry covering larger portions of the screen.

The *randomized sample tree* [KKF+02] extends the randomized Z-buffer with out-of-core techniques.

As a preprocessing step, initially an octree is created from the input triangles. Then, for each node u triangles are randomly selected and stored in the parent node with probability of the triangles area relative to the summed area of all triangles located in u. At rendering time the tree is traversed by a depth first search. The triangles contained in each node encountered are rendered until the projected size of a node is smaller than one pixel. Rendering is aborted at such locations and continued with the next large node found by the traversal.

A client server architecture allows rendering scenes which may not fit into main memory. The server loads geometry as needed and sends it to the client. This allows more general scenes than or implementation which loads geometry only from local hard drive. On the other hand, the SEC–Tree prefetches geometry to avoid slow downs during rendering due to missing geometry, which in contrast become visible in case of the sample tree. However, it should be possible to integrate prefetching without severe problems.

The authors show that for a certain sample size selected during the tree construction, correct images are obtained with high probability. However, their definition of correctness varies from the result actual rendering delivers. Still such errors occur only at nodes of small projected size. Rendering such nodes, aliasing becomes the main problem and even an approach rendering every triangle might result an image with pixels not representing the scene accordingly. As an alternative to rendering the sample points, a single color value can be precomputed for each sample tree node by averaging the colors of all triangles. However, this does not consider occlusion or orientation towards the viewer.

This approach is related to rendering SEC–Trees as both methods choose a subset of the overall geometry when rendering the image. The main differences are the underlying data structure and the number of regarded triangles. The randomized sample tree selects sufficient triangles from an octree to ensure image quality, while a constant number of triangles is selected from a SEC–Tree to keep up a constant frame rate. At locations in a scene, where many large triangles are near to the viewer, the sample tree has to render all of them, which might lead to frame rate decreases unsuitable for interactive walkthroughs.

The randomized sample tree considers the complete scene as a triangle soup, i.e. the input is a set of unstructured triangles. Object identity is not considered and therefore can not be preserved. If objects are not known by the data structure, removing or relocating objects is not possible and limits the randomized sample tree to static scenes.

2.2.3 Far Voxels

Gobbetti et al presented *far voxels* [GM05], an approach for out-of-core construction and view-dependent rendering of large models on commodity graphics platforms, instead of choosing points randomly they look for points visible from viewing positions on the outside of the model. Their method integrates visibility culling and out-of-core data management with level-of-detail construction and rendering. Triangles as well as volumetric data are considered by their approach. Nodes of an underlying *BSP tree* [FKN80] contain triangles in case of leaf nodes, while inner nodes are discretized into cubical voxels. Voxels are associated with points obtained from intersections of rays, originating from multiple directions, with the model's surface.

At rendering time, the tree is rendered in front to back order, using hardware occlusion culling to avoid rendering invisible parts of the object. Triangles stored in leaf nodes are rendered, while inner nodes are traversed until projected voxel size falls beneath a threshold. When this case occurs, points associated with voxels are rendered and subtrees are ignored. Only portions of the model that have to be rendered are kept in main memory and are loaded on demand when not present.

The disadvantages are similar to those of surfels, that is dynamic scenes or scenes consisting of multiple objects are not considered.

2.3 Occlusion Culling

Occlusion culling algorithms discard polygons which are not visible within the rendered image due to occlusion. Two basic approaches can be distinguished. From region visibility algorithms organize the space into cells and determine visibility relations between these cells as a preprocessing step. For each cell a set of potentially visible cells (*potentially visible sets, PVS*) is computed. Teller and Séquin [TS91] presented one example of such an approach.

While PVS are calculated in object space, other approaches like *hierarchical occlusion maps* [ZMHH97] or the *hierarchical Z-buffer* [GKM93] work in screen space. Such point visibility approaches identify visible portions of the scene at runtime and depend on the viewer's current position in the scene. With the hierarchical Z-buffer [GKM93], regions of a scene are culled, when their closest depth value is greater then that of the pixel which is already displayed at the projected scene location. Finding occluded polygons is supported by advances in current graphics hardware, see Bittner et. al. [BWPP04]. An overview on different occlusion culling algorithms can be found in the survey of Aliaga et. al. [COCSD03].

2.3.1 The Prioritized-Layered Projection Algorithm

The *prioritized-layered projection (PLP)* algorithm [KS00] has been introduced by Klosowski and Silva. This approach is strongly related to occlusion culling, as it traverses a data structure in an order preferring nodes that are more likely not to be occluded. The algorithm is not conservative, i.e. visible portions of a scene may not be recognized. This results from a limited polygon budget available for rendering each frame, similar to our approach. An extension resulting in conservative visibility is presented in cPLP [KS01].

As a preprocessing step, the polygons of the scene are organized in a data structure which leads to convex cells, such that polygons within a cell are evenly distributed. The authors proposed an octree as a well as a delauny triangulation as possible data structures to obtain such cells. Each cell is assigned a solidity value, indicating how large the contained geometry is compared to the cell.

At runtime, the data structure is traversed in an order similar to a front-to-back traversal, starting with the cell containing the user's current viewpoint. All nodes which might be visited next are stored in a priority queue of candidates called front. Initially, the front is empty. Whenever a node is rendered, the neighboring nodes which have not been rendered yet are updated in the front. Depending on the solidity of the rendered node, the direction to the neighbor compared to the direction to viewer and depending on the orientation of the boundary between the node and its neighbor, the solidity value is determined. The intention is to reach a high solidity value if much geometry between a node and the viewer has been rendered and instead continue working on the front at less occluded positions.

For each node encountered during the traversal, the number of rendered polygons is counted. If this number exceeds the given budget, the traversal is aborted. The differences to rendering SEC–Trees are the underlying data structure and how triangles are chosen. The PLP algorithm tries to select triangles which are most likely to be not occluded and moves in a direction away from the viewer.

In contrast, the SEC–Tree selects triangles depending on distance and size, and therefore prefers components defining the approximate structure over details, even if those are facing the viewer. Figure 2.1 shows the image of an example object, Figure 2.1(b) shows how it is rendered using the SEC–Tree. The lattices are partly drawn, indicating that they exist but not wasting that much of the budget. For approaches utilizing a triangle budget, choosing the nearest triangles is not necessarily an optimal strategy. The lattices in the front would be encountered and drawn by the PLP algorithm early, while they contribute only little to the final image.

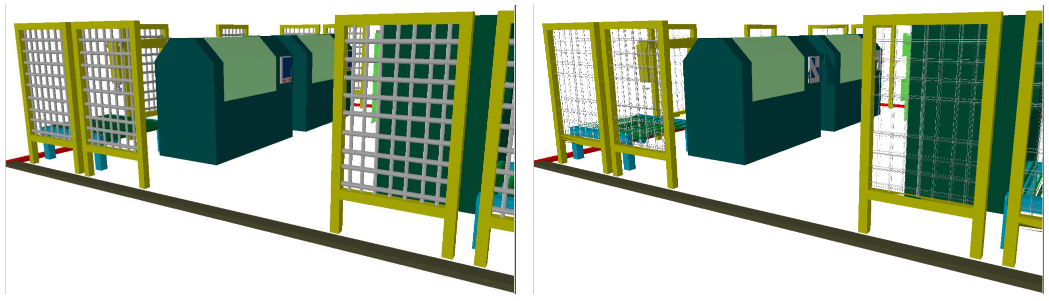

(a) The model with all triangles (b) and rendered with a SEC–Tree

Figure 2.1: Triangles in the front may not be the most important

Since the PLP approach works on a polygon level rather than object level, object identity can not be prevailed. Further, dynamic scenes are not supported. Out-of-core rendering is not supported or discussed in the proposed algorithm, however the walkthrough system iWalk [CKS03] integrates the PLP algorithm in an out-of-core rendering system.

2.4 Hybrid Rendering Systems

Some systems use hybrid approaches, relying on more than one single techique for rendering scenes, thus combining several methods. The *UC Berkeley Architecture Walkthrough* system [FTSK96] combined hierarchical algorithms with object-space visibility computations [Tel92] and LODs for architectural models.

A framework presented by Andujar et. al. [ASVNB00] integrated occlusion culling and LODs. This approach estimated the degree of visibility of each object by using PVS. Then, this value is used for selecting appropriate LODs and culling. The method relies on decomposing scene objects into overlapping convex regions (axis-aligned boxes) which are considered as occluders.

Another integrated approach uses the prioritized-layered projection visibility approximation with view-dependent LOD rendering [ESSS01]. Instead of discarding scene portions which are likely to be occluded, they use its visibility estimation to determine their level of detail.

The *UNC Massive Model Rendering (MMR)* system [ACW$^+$99] combined LODs with image-based impostors and occlusion culling to deliver interactive walkthroughs of complex models.

2.4.1 Giga Walk

Giga walk [BSGM02] combined a complex mixture of different technologies to obtain 12-37 frames per second within complex environments on dedicated SGI workstations. Their approach uses two graphics pipelines in parallel, the first for occlusion culling, while the second renders visible geometry. An additional process loads geometry on demand, such that only geometry currently required is kept main memory which allows displaying scenes larger than main memory.

An axis aligned bounding box (aabb) hierarchy for managing LODs is built up as a preprocessing. Creating this data structure, at first larger objects are split up and a clustering approach combines these smaller object fragments into new objects. While these redefined objects feature a larger spactal togetherness, information about original input objects is lost. After building the aabb hierarchy for the new objects, HLODs are computed using GAPS

[EM99]. Leaf nodes contain traditional static LODs, while LODs of inner nodes approximate the geometry of all children.

Rendering the data structure, objects which were visible in the last frame are rendered as occluders. Using a hierarchical Z-buffer [GKM93], the aabb hierarchy is traversed and objects are tested for visibility. Then, visible objects are rendered in a second graphics pipeline. For each node visited, the HLODs are checked for the resulting errors if they are used for rendering instead of the original geometry. Adjusting this threshold allows to sacrifice image quality for higher frame rates. However, this approach still keeps image quality fixed and offers no possibility to predict the image quality necessary to guaranty a certain frame rate.

2.4.2 iWalk

Correa et. al. presented the out-of-core rendering system *iWalk* [CKS03]. It uses the PLP algorithm for rendering and to perform visibility based prefetching of geometry. Two rendering modes are supported, either approximate rendering, limiting the displayed geometry by a triangle budget to keep up a constant frame rate comparable to rendering SEC–Trees, or conservative rendering, displaying all visible triangles and thus favoring image quality over speed.

Since the PLP algorithm is integrated into their system, their basic data structure is an octree, which is created in an out-of-core preprocessing. The iWalk walkthrough system traverses this octree at runtime to render the scene.

There are two threads involved in the rendering process, first the actual rendering thread and second a lookahead thread for prefetching geometry. The rendering thread uses PLP in approximate rendering mode or cPLP in conservative rendering mode to determine visible portions of the scene. A geometry cache loads the necessary geometry from hard drive if it is not currently available. Then, this geometry is passed to the graphics hardware for rendering. In order to avoid delays while waiting for geometry to be loaded, the lookahead thread predicts future positions of the viewer and uses the visibility algorithm to identify geometry which might be rendered in one of the next frames.

iWalk combines the PLP algorithm with out-of-core rendering, removing one problem from the original PLP approach when displaying large scenes. However, the other drawbacks inheritet from PLP as described in Section 2.3.1 still remain unsolved in iWalk, namely object identity is still lost and changing the scene at runtime is not possible.

2.5 Clustering

One step within the construction of SEC–Trees will be clustering center points of similar sized triangles. A variety of different clustering approaches have been developed, like k-means, k-medoids or hierarchical clustering, an extensive overview is given in the surveys by Pavel [Ber02] and Xu and Wunsch [XW05]. Our approach is similar to the density based clustering algorithms *DBSCAN* [EKSX96] and *OPTICS* [ABKS99].

2.5.1 DBSCAN

Within the precprocessing, we use a density based clustering algorithm similar to DBSCAN [EKSX96]. DBSCAN searches for clusters by defining an ϵ-neighborhood around each input point and identifying neighbors located within this region.

The algorithm DBSCAN discovers clusters and identifies noise in a database. It starts by defining core objects, which are points containing a number above a given threshold of other points inside an ϵ-neighborhood. An point $\tilde{P}$ is called density reachable by P if there exists a chain of core objects between them, such that each of these points is contained in an ϵ-neighborhood of the previous one, starting with P and ending with $\tilde{P}$. Then, a cluster is a maximal set of points, where each point is density reachable from the others.

The retrieval of density reachable points is performed by iteratively searching for them. DBSCAN checks the ϵ-neighborhood of each point in the database. If the neighborhood contains sufficient points, a new cluster is created from these points. Then, the ϵ-neighborhood of all added points which have not been processed is checked. If this neighborhood contains at least the required number of points, the neighbors which are not already contained in the cluster are added and their neighborhood is checked in the next step. This procedure is repeated until no new point can be added to the current cluster.

While the construction process of SEC–Trees contains similar approach, the intentions for our clustering algorithm are different from those in DBSCAN. DBSCAN is created for data mining in databases, while we are interested in spatial organization of triangles to create a rendering hierarchy. As a result, DBSCAN tries to identify noise inside the point set, which are points not contained inside any cluster, while SEC–Trees on the other hand do not discard any triangles from the input set. The authors state that their algorithm has costs of $O(n \log n)$ if the input consists of n points. However, their assumption is that the number of points reported by a range query is bounded by a constant, as query regions are small. SEC–Trees need every point to be contained in a cluster, so neighborhoods might become large. We will present a version of our clustering algorithm which guaranties that the number points each range query reports will be limited by a constant. Still, our algorithm has higher overall costs, as we enlarge neighborhoods until every point is contained in a cluster, while DBSCAN in contrast considers a fixed value of ϵ.

An enhancement of DBSCAN, called OPTICS [ABKS99], overcomes this limitation of DBSCAN that only one fixed value of ϵ is considered. Instead of computing clusters explicitly, they determine an order which allows, with few additional information per point, to apply an algorithm based on DBSCAN finding clusters for any $\epsilon\prime \leq \epsilon$. However, neighborhoods larger than ϵ are not supported.

CHAPTER 3 SEC–TREES

In the following we will give a detailed on description on the construction of SEC–Trees. We start giving a short overview on this process and a simple example in Section 3.1. In Section 3.2 we describe how groups of similar sized triangles are created. Section 3.3 introduces our clustering algorithm. Then, we present our approach on creating SEC–trees in Section 3.4, using the two previous algorithms. While these methods are applied to single objects, we introduce our global data structure in Section 3.5. In Section 3.6, an analysis of the algorithms is given. However, while the algorithms have been implemented as described before and perform well in practice, we will analyse alternatives which are showing a better worst case behavior.

3.1 Outline of the Approach

The main idea behind the SEC–Tree is to create a data structure, in which every node represents a cluster of nearly equally sized smaller clusters and triangles. The input data for the rendering system are objects $J_1, \ldots, J_m$ consisting of triangles. For each object the position of the bounding box center is given, additionally a 3D model is specified defining the geometry of the object.

For every 3D model a SEC–Tree is constructed individually, containing all triangles in object coordinate space. Then, transformation matrix defines how to transform the SEC–Tree of the 3D model corresponding to an object into world coordinate space. If positions of objects do not change, the scene is static, and in addition to the trees created for each model, a further SEC-tree can be built on object level. We will call this tree a global SEC-tree. The SEC–Tree of all 3D models as well as a global SEC-tree are built once in a preprocessing step and stored on hard drive. Instead of a complete scene, we will consider a single object J consisting of n triangles $T_1, \dots, T_n$ as our input data first, before dealing with static scenes containing multiple objects later on in Section 3.5.

For each triangle T_i its area $A(T_i)$ and center point z_i are the main information when organizing these triangles in a SEC–Tree. The SEC–Tree differs from most spatial data structures used for rendering, as the main characteristic of triangles is their area, while position is only secondary. In contrast most other approaches, like octrees as an example, consider position to be the primary attribute. Inserting a triangle into an octree, at first one has to decide which octant contains this triangle. The triangle's size is only important to find the appropriate depth in the tree.

The SEC–Tree construction process groups triangles of approximately equal area first, the primary attribute is a triangle's area. Then, spatial criteria become of importance, as clusters within groups of equally sized triangles are searched for, but this criteria is not considered in the first step.

Another difference to most other spatial data structures lies in the avoidance of a strict distinction between data structure and stored data. As an example, in an octree each node has exactly eight children and contains a varying amount of geometry, clearly delimiting nodes from triangles. The SEC–Tree, on the other hand, is build up in bottom up manner, combining triangles with clusters. The combinations are based on size and the position of the center point of triangles and clusters. Only definitions of these parameters are different, beyond that triangles and clusters are treated equally. At this point, a subtree of the SEC–Tree is handled exactly like a triangle.

We assume that objects are not animated, in the sense that objects are static in themselves, i.e. turning wheels on forklifts are not supported, thus a data structure organizing this object can be constructed in a preprocessing stage and no reorganization has to be performed at runtime. Complete objects however may be inserted, deleted or moved in the scene.

The goal is to get an approximation of a scene suited for interactive rendering regardless of the object's complexity. An object's impact on the rendered image is depending on the number of pixels covered after its projection to screen coordinates. Considering the object's size and distance to the viewer allows to rate the importance of an object for displaying, because these parameters mainly determine its projected size. The SEC–Tree organizes all triangles by size and position, offering information needed for high quality approximations. A detailed description of the deployment of SEC–Trees in the rendering process will be given in Chapter 4.

Our approach consists of two main phases which are iterated. First, triangles are sorted by size in ascending order. After that, those positions inside the sorted list are identified where consecutive triangles show most difference. By splitting the list at these positions groups of nearly equally sized triangles result.

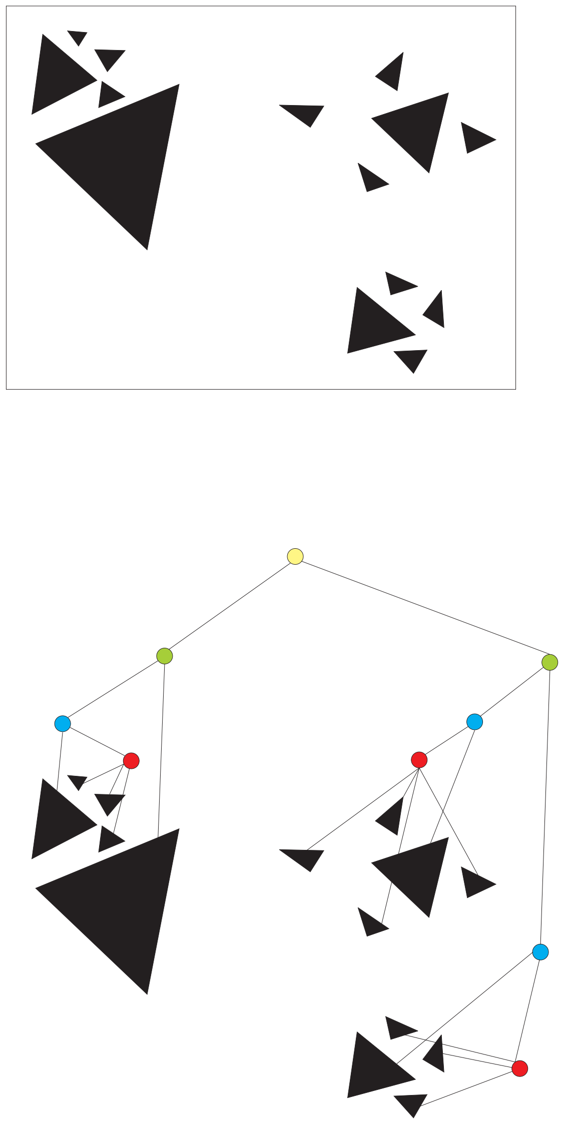
To structure these groups we take spatial criteria into account. More precisely, we use a clustering approach for finding triangles located near to each other in every group. Groups are treated in a sequence consecutive of contained triangles' size, starting with the group of smallest triangles. The obtained clusters are treated like larger triangles and reinserted into the triangles groups. Since clusters can be inserted into groups of triangles and take part in the clustering processes, doing so results in a hierarchy of clusters.

In addition to groups constructed as described above, an empty group is created, into which we insert clusters larger than original triangles of the input data set. Instead of clustering this group we start at the beginning, sorting the clusters contained in this group by size again. We iterate these two phases of grouping by size and clustering groups, until we finally obtain one cluster, which defines the root of the SEC–Tree.

3.1.1 A Simple Example

To improve clarity, Figure 3.1 illustrates the following example: The SEC–Tree is constructed from an object of fourteen triangles (top of Figure 3.1(a)). Those are divided into three groups of similar sized triangles (Figure 3.1(b)). An empty fourth group will receive clusters which are larger than the input triangles but is not listed in the figure.

We will visualize different levels in the SEC–Tree by colors. Triangles are


(a) An example object with 14 triangles. Below the SEC–Tree created for this object is shown.

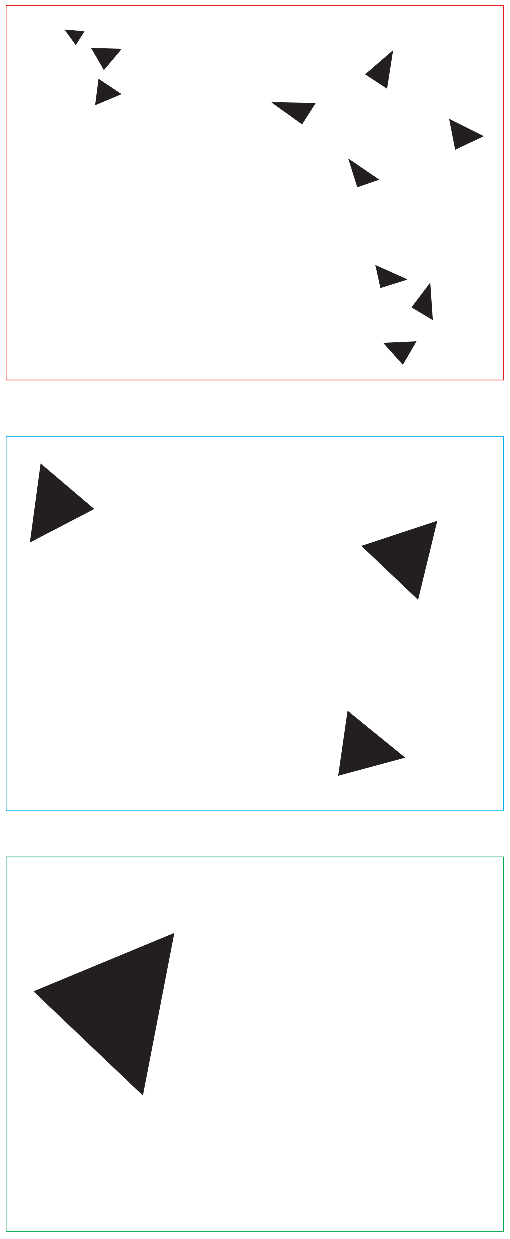
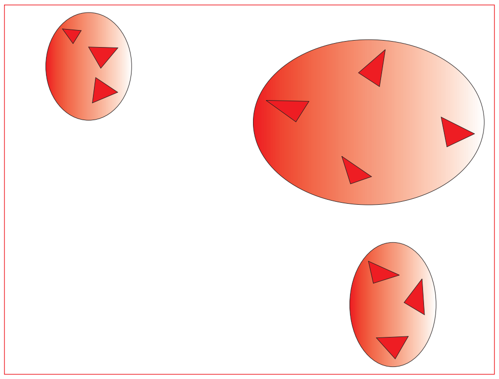
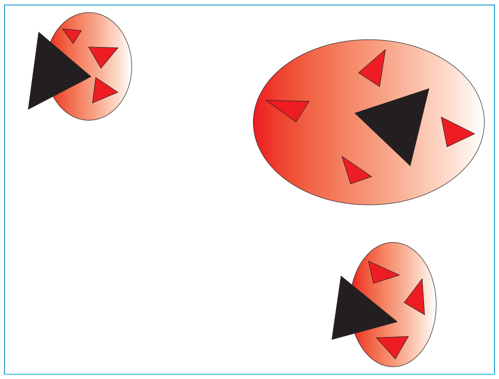

(b) Triangles are arranged in 3 groups

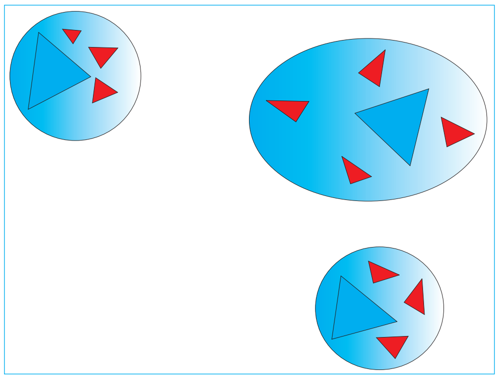
Figure 3.1: An example of a SEC–Tree



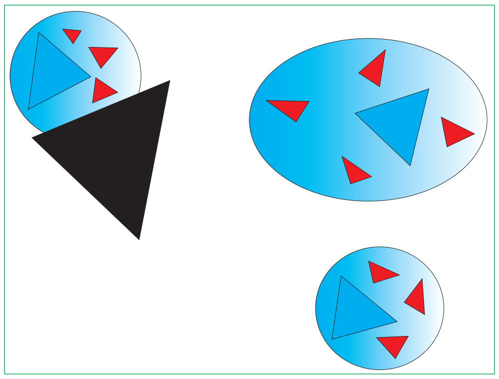
(a) The first group is clustered



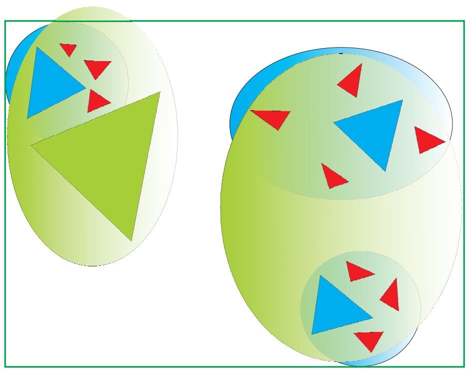
(b) Clusters are inserted into the group of larger triangles



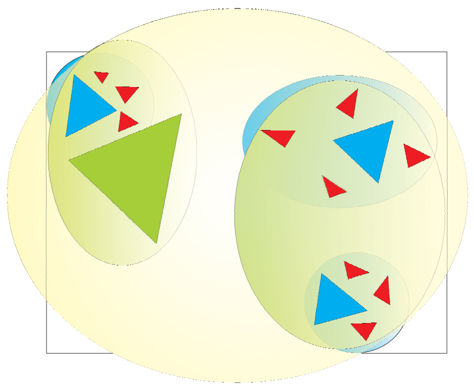
(c) The second group is clustered



(d) Clusters are inserted into the 3rd group



(e) The clustering alorithm generates 2 clusters from the 3rd group



(f) One group is created from the two clusters, resulting in a single root cluster

Figure 3.2: The SEC–Tree construction process

colored black. In Figure 3.1(a) red, blue, green and yellow are used for nodes with ascending levels in this order. We will draw triangles in Figure 3.2(a) to 3.2(f) in the same color as the representing node, once they are assigned to a cluster.

At first, we cluster the group containing the smallest triangles, resulting in three clusters (Figure 3.2(a)), which are inserted into the second group. Figure 3.2(b) shows the the situation of the second group afterwards, this is it contains three original triangles as well as the newly created clusters. Again, we use our clustering algorithm (Figure 3.2(c)), combining triangles with equally large clusters and insert the three new clusters into the third group (Figure 3.2(d)). We continue with clustering this third group, resulting in two clusters larger than all out triangles. These clusters are now inserted into the empty fourth group, which contains the input data for a new iteration, i.e. we generate new groups (Figure 3.2(e)) from these clusters. Since we only have two clusters, which are nearly equally sized, only one group results. When we search for clusters inside this group, we find one cluster (Figure 3.2(f)) which is the root of the SEC–Tree. The complete tree is illustrated at the bottom of Figure 3.1(a).

3.2 Grouping Triangles by Size

We assume that a list of triangles $M = \{T_1, \ldots, T_n\}$ is given, sorted in ascending order. This set is split at a position j, such that the size difference of two consecutive triangles is as large as possible at position j. More interesting than the absolute value of size differences is a relative measure. Therefore, we choose j as the position of largest size ratio, i.e. such that

$$A\left(T_{j+1)}\right)/A\left(T_j\right) = \max_{1\leq k\leq n-1} A\left(T_{k+1}\right)/A\left(T_k\right).$$

We divide the set of triangles at position j into a set of larger triangles $L := \{T_{j+1}, \ldots, T_n\}$ and smaller triangles $S := \{T_1, \ldots, T_j\}$.

This procedure is repeated recursively with sets S and L. However, sorting only has to take place once, after that order can be assumed. We abort if one the following three conditions is true for a set $T_k, T_{k+1}, \ldots, T_l, k < l$:

$l - k \leq c_1$, $\frac{A(T_l)}{A(T_l)-A(T_k)} > c_2$ or $A\left(T_l\right) < c_3$ for appropriate values of c_1, c_2, c_3. Groups of less than c_1 elements are no longer divided to avoid groups containing too few triangles. If the size difference between triangles is small compared to the overall size of the triangles, we can consider the triangles to be of nearly equal size and stop splitting the set. Additionally, we do not continue if the size of all triangles is below c_3, since a relative comparison of triangles sizes is of limited use regarding small values. Triangles of such a small area present a numerical problem, computing their area becomes challenging, and therefore comparing these values has only limited expressiveness.

To organize generated groups we create a search tree. Every time a set M of triangles gets split into sets S and L, two new nodes appear representing the sets S respectively L. These nodes become children of the node representing the complete set M. All leaf nodes are storing groups of nearly equally sized triangles, while inner nodes store the value v of smallest triangle in the subset of larger triangles, i.e. $v = \min_{T \in L} A(T)$. When this tree is completed, we finally add an empty leaf at the most right position. This leaf will contain clusters which are larger than the original input triangles or cluster. These are the input data for a next iteration. Algorithm 1 summarizes this process.

Algorithm 1 Construction of the Searchtree

```
Require: sorted set of triangles $M = \{T_1, \dots, T_n\}$ with $A(T_j) \leq A(T_i)$ for all $j \leq i$
1: if $n > c_1$ && $A(T_m) / (A(T_1) - A(T_n)) \leq c_2$ && $A(T_n) \geq c_3$ then
2:     divide $M$ at position $j$ of largest size ratio $A(T_{j+1}) / A(T_j)$ into sets $S = \{T_1, \dots, T_j\}$ of smaller and $L = \{T_{j+1}, \dots, T_n\}$ of larger triangles
3:     store $v = A(T_{j+1})$
4:     left = new searchtree($S$)
5:     right = new searchtree($L$)
6: else
7:     store $M$
8: end if
```

3.3 Clustering

In the following section we will describe the algorithm used to cluster triangles inside a leaf of the search tree. Until now, we have organized sets of triangles by size, next we take spatial aspects into account. Given a set

S of similar sized triangles, we associate triangles nearby resulting in clusters. On the one hand these clusters should feature high density, and on the other hand cover a sufficient number of triangles. Clusters of few triangles provide sparse information about neighborhood relations. Otherwise, clusters embracing large sets may contain subsets of sufficient size to form a cluster on their own, thus preventing denser clusters. This way, information about immediate neighborhood of these triangles is lost. Therefore, we will determine a lower limit c cluster size should not fall below, and search for clusters containing no denser sub-clusters of relevance.

We want to group triangles positioned near to each other. Instead of examining complete triangles we treat them as points. All triangles stored in one node of the search tree are of nearly equal size. Therefore, a single point specifying the position contains sufficient information to characterize an triangle and we do not lose too much information. Given a triangle T_i, we will refer to this point as the center z_i.

The approach applied follows a flooding strategy: We start with a triangle and try to find other triangles near to it. This is continued with the found triangles until no longer new triangles are discovered. For every triangle a sphere defining a neighborhood is determined and the triangle with the smallest neighborhood is selected as the starting point, since a cluster should be as dense as possible. The sphere size determines the density of the cluster. If the sphere is small, all neighbors have to be near to the center and we obtain a dense cluster, but if we start with a larger neighborhood, there might be a real subset of triangles contained forming a denser cluster. On the other hand, the number of triangles in a cluster has to be sufficient in order to prevent an inefficient data structure with only little information about direct neighbors. Thus, we only accept large clusters and discard those, which are to small.

The neighborhood of a triangle $T = (P_1, P_2, P_3)$ depends on its extensions. Larger triangles result in larger spheres as the radius of the neighborhood sphere is determined by its span $d(T) := \max_{1\leq k,l\leq 3} ||P_k - P_l||$, which is the longest distance between two of its vertices. If a triangles has a large span, points far away from the center can be near to another point inside the triangle. If the span is small, points near to the triangle have to be near to the center. As a neighborhood we define a sphere with radius proportional to the span.

Triangles are considered to be neighbors if their centers are contained in a neighborhood. Continuing with these neighbors, the same method is applied for further searching. This is repeated until no more triangles can be found. If there have been spotted a sufficient number of triangles, the cluster is accepted. In case that the number is too low, the initial neighborhood is enlarged and the procedure is repeated. Triangles, which have been inserted into a cluster, are removed from the set S, therefore every triangle is inserted into exactly one cluster.

Algorithm 2 reproduces our approach to create clusters of triangles. In the following we will describe the calculations carried out in the k-th iteration of the while loop starting at line 5. We define the neighborhood of T_i in iteration k by $K\left(z_i, 2^{s_k(T_i)+1}d\left(T_i\right)\right)$, whereas z_i is the center of triangle T_i, $d\left(T_i\right)$ refers to the span of T_i, which is half the initial neighborhood radius, and $K\left(p, r\right) := \{q \in \mathbb{R}^3 | \parallel p - q \parallel \leq r\}$ for $p \in \mathbb{R}^3, q \in \mathbb{R}$ describes a sphere with center p and radius r. $s_k\left(T_i\right)$ indicates how often T_i has been this starting point of the search for a cluster. We will denote this starting point as *seed*. Flooding always starts with a triangle as seed, which has a neighborhood of minimal volume.

Let C_k denote the cluster determined during the k-th iteration. S_k refers to the set of triangles not integrated into a cluster of sufficient size c before iteration k, so

$$S_k := S \setminus \bigcup_{j=1}^{k-1} \{C_j | \parallel C_j \parallel \geq c\}.$$

In Algorithm 2 S is replaced by the set S_k (line 20). $u_j^k := u_k\left(T_j\right) := 2^{s_k(T_j)+1}d\left(T_j\right)$ describes the neighborhood of triangle T_j to regard in iteration k (line 1-3 and 20) if T_j is choosen as seed. The function $s_k\left(T\right)$ with

$$s_1\left(T\right) := 0 \; \forall T \in S,$$

$$s_{k+1}\left(T_{t_k}\right) := s_k\left(T_{t_k}\right) + 1, s_{k+1}\left(T\right) := s_k\left(T\right) \forall T \in S \setminus \{T_{t_k}\}$$

indicates, how often clustering has started with triangle T in iterations 1 to $k-1$, i.e. how often T has been the seed. Thereby $T_{t_k} \in S_k$ is an arbitrary triangle with

$$u_k\left(T_{t_k}\right) = \min_{T \in S_k} u_k\left(T\right),$$

i. e. T_{t_k} is a triangle with smallest neighborhood. Therefore, neighboring triangles are comparatively near to T_{t_k}, such that generated clusters feature

Algorithm 2 Cluster construction

```
Require: Set of triangle centers $S = \{T_1, \ldots, T_n\}$
 1: for all $1 \leq j \leq n$ do
 2:     $u[j] := 2 * d(T_j)$    {initial neighborhood}
 3: end for
 4: $F = \emptyset$    {set of clusters}
 5: Outer loop:
 6: while $\| S \| > 2 * c$ do
 7:     {enough triangles to create clusters}
 8:     find triangle $T_i \in S$ such that $u[i] = \min_{T_j \in S} u[j]$ {triangle with minimal
        neighborhood}
 9:     $C_{Tmp} := \{T_i\}$ {triangles that are to be examined}
10:     $C := \emptyset$ {set of processed objects}
11:     Inner loop: flooding
12:     while $C_{Tmp} \neq \emptyset$ do
13:         select arbitrary $T_t \in C_{Tmp}$ {search for neighbors of $T_t$ next}
14:         $C := C \cup \{T_t\}$
15:         find set $M = \{T_j \in S \setminus C | z_j \in K(z_t, u[i])\}$ {search for neighbors of $T_t$}
16:         $C_{Tmp} := (C_{Tmp} \cup M) \setminus \{T_t\}$ {add neighbors of $T_t$ to cluster, $T_t$ is pro-
            cessed}
17:     end while
18:     if $\| C \| \geq c$ then
19:         $F := F \cup \{C\}$ {add $C$ to set of clusters}
20:         $S := S \setminus C$ {make sure, each triangle is contained in only one cluster}
21:     else
22:         $u[i] *= 2$ {enlarge neighborhood of $T_i$ in case there were not sufficient
            triangles inside}
23:     end if
24: end while
25: $F := F \cup \{S\}$ {consider remaining triangles to be a cluster}
```

high density. Starting with T_{t_k}, triangles of the next cluster are searched (starting at line 7).

A cluster is build by flooding. Beginning with T_{t_k} all neighbors within S_k are acquired, and proceeding those afterwards search is continued (lines 10-16). A triangle is a neighbor if its center is positioned inside a sphere defining a neighborhood. Every triangle found is inserted into a temporary cluster C_{Tmp}, containing triangles to be processed (line 14). We continue the search for new triangles by examining neighborhoods of these newly added triangles. The considered neighborhoods are spheres positioned at the centers of the added triangles with radius u_k equally to the initial sphere. This search for neighborhoods of triangles in T_{TMP} is repeated until no more triangles are found (line 12). So, the resulting cluster is

$$\begin{gathered} C_k = \{T \in S_k |\ \exists T_{i_1}, \ldots T_{i_l} \in S_k, T_{i_1} = T_{t_k}, T_{i_l} = T, \\ z_{i_j} \in K\left(z_{i_{j-1}}, u_k\left(T_{t_k}\right)\right), 2 \leq j \leq l\} \end{gathered}$$

We define

$$C_i^0 := \{z_i\} \text{ and } C_i^k := \begin{cases} C_k, & \text{if } i = t_k \\ C_i^{k-1}, & \text{else} \end{cases}$$

to be temporary clusters we have found up to iteration k with seed T_i.

If the found cluster C_k contains a sufficient number of triangles, i.e. if there are more than c triangle centers located in C_k, the cluster is added to a set of final clusters F and inserted into the search tree as described in Section 3.4 (line 18). If $\| C_k \| < c$, the radius u_j^k of T_{t_k}'s neighborhood is doubled (line 22), such that additional neighbors of T_{t_k} might be found in an iteration later on and C_k is discarded. After this, the smallest neighborhood might belong to another triangle, which is then chosen as the seed of the next iteration.

We continue constructing clusters C_k this way until the number of remaining triangles becomes too low to form more than one cluster satisfying our size criteria, i.e. until $\| S_k \| < 2c$ (lines 5, 6). Finally, this remaining set of triangles is considered to be a cluster of lowest density (line 24), since further splitting would unavoidably lead to clusters not satisfying the size criteria.

3.4 Construction of the SEC–Tree

A SEC–Tree is constructed by repeating two phases, the arranging and the clustering phase (Algorithm 3). At first triangles are arranged in groups of similar sized triangles during an arranging phase, and a search tree managing these groups is constructed as described in Section 3.2 (line 5 of Algorithm 3).

Let $L_1, \ldots, L_m$ denote the leafs of this searchtree, whereas L_i represents a set of triangles M_i. We assume a numeration of leafs is given in an ascending order, i.e. $A(T) < A(Q) \forall T \in L_k, Q \in L_l, k < l$. We perform traversals of the searchtree by a depth search, such that leaves are visited in sequence $L_1, \ldots, L_m$.

Phase two is the clustering phase, where the tree created in the arranging phase is traversed and clusters inside the groups are searched for. Leafs are visited in an order such that leafs containing triangles of smallest area are visited first. This starts at the most left leaf L_1 of the tree. If a leave L_i is reached, we check if there is a sufficient number of triangles, i.e. $\| M_i \| \geq c_4$. If this condition holds, the clustering algorithm described in Section 3.3 is applied to M_i resulting in clusters $C_{i,1}, \ldots C_{i,h_i}$ (line 15). For each cluster $C_{i,j}$ we define its size to be the summed area of triangle areas contained in $C_{i,j}$, so $A(C_{i,j}) := \sum_{T \in C_{i,j}} A(T)$.

Now, a cluster of triangles will be treated like new triangles. This cluster is inserted into the search tree according to its area. Then we have $A(C_{i,j}) \geq \min_{T \in M_i} A(T) > A\left(\tilde{T}\right) \forall \tilde{T} \in M_j, j < i$, i.e. the cluster is at least as large as every triangle in $L_1, \ldots, L_{i-1}$. Therefore, if we insert $C_{i,j}$ into the searchtree, it will be included in a leaf $L \in L_i, \ldots L_m$ (line 17). However, if the cluster is not larger than the largest triangle in the current leaf L_i, it would be inserted into the same leaf we are just handling. This situation occurs when there are only few triangles left after the last iteration within the clustering process. In this case we choose the next leaf $L = L_{i+1}$ instead. Now, L is a leaf we will encounter later on while traversing the search tree. So, if the traversal reaches L, not only triangles $T_k, \ldots, T_l$ will be found, but at least one cluster too. Note, that M_i might change here, as it will represent inserted clusters as well. More precisely, we have to consider

$$\tilde{M_i} := M_i \cup \{C_{j,k}, 1 \leq j \leq i-1, 1 \leq k \leq h_j \mid C_{j,k} \text{ is inserted into } L_i\}.$$

Algorithm 3 Construction of the SEC–Tree

```
Require: set $S$ of triangles $S = \{T_1, \ldots, T_n\}$, sorted in ascending order by
    their size $A(T_i)$
1:  {start of arranging phase}
2:  if $n$ < minimum then
3:     cluster triangles and add them under the root of the SEC–Tree
4:  else
5:     generate searchtree from $S$ with leafs $L_1, \ldots, L_m$
6:     {start of clustering phase}
7:     for all leaves $L_i$ from $L_1$ up to $L_m$ do
8:        if $L_i = L_m$ then
9:           new SEC–Tree(triangles and clusters of $L_m$) {start phase 1 again}
10:       else
11:          if ($\| M_i \| < c_4$) then
12:             $M_{i+1} := M_{i+1} \cup M_i$
13:          else
14:             generate clusters $C_{i,1} \ldots, C_{i,h_i}$ from set $M_i$ of triangles and clus-
                ters of $L_i$
15:             for all clusters $C_{i,j}$ do
16:                insert $C_{i,j}$ into set $M_k$ of a leaf $L_k \in \{L_{i+1}, \ldots, L_m\}$ according to
                   its size $A(C_{i,j}) := \sum_{T \in C_{i,j}} A(O)$
17:             end for
18:          end if
19:       end if
20:    end for
21: end if
```

So, input data to the clustering algorithm (Algorithm 2) will not only consist of triangles, but additionally may contain clusters. Since the algorithm only depends on triangle centers and extensions and not on accurate geometry, we can extend the approach to sets containing objects as well as clusters. We will refer in the following to triangles, although there might be clusters as well.

Small clusters should be avoided. They reduce the breadth of the SEC–Tree and increase depth. We need more time to traverse the tree without a gain of information. Therefore, we do not start the clustering algorithm if the number of triangles in a leaf L_i is below a minimum of c_4. Instead, the triangles are inserted into the next leaf L_{i+1} (line 12) if their number is low. We could avoid the creation of leaves containing less than c_4 triangles if we checked for the minimal number of triangles while building the search tree. This way sets of triangles always could be divided only into subsets greater than a minimal number of $||M_i|| > c_4$. But since clusters of triangles might be inserted later into leaves containing few triangles, their number of triangles $||\tilde{M}_i||$ can increase on more than c_4 elements. In the beginning we do not know the number of triangles $||\tilde{M}_i||$ that will be in a leaf when it is reached by the traversal.

Phase two ends by the traversal of the search tree reaching L_m. This leaf is treated differently than $L_1, \ldots, L_{m-1}$. Since we know that there exits a $T \in M_{m-1}$ with $A\left(C_{i,j}\right) < A\left(T\right)$ for all clusters $C_{i,j}$ inserted into a leaf $L_1, \ldots, L_{m-1}$, all these clusters fitted somewhere between the initial data, i.e. they were smaller than the largest triangle and could be inserted at an appropriate place inside the tree. This is not true for clusters inserted into L_m, since they can be arbitrarily large. We know that $A\left(C_{i,j}\right) \geq min_{T\in\tilde{M}_i} A\left(T\right)$ holds for them, but they might be significantly larger than all triangles. Nothing about the ratio between sizes of elements in L_m can be stated, they could differ by the order of several magnitudes. If we continued as before by clustering the data, the result might be a degenerated data structure, we might cluster elements without similarity. Therefore, we start again with phase one with the triangles and clusters contained in this leaf and group them by size again, i.e. a new arranging phase is started with triangles and clusters contained in M_m as input data.

If phase one starts with less triangles or clusters than a given minimal number, repetition of the two phases is aborted and we only start one last clustering on the given set of triangles. The SEC–Tree root is created and the newly generated clusters are inserted beneath this root node (line 3). Since we wanted groups which are clustered to have a size of at least c_4 elements, we choose the minimal number within the abortion criteria not smaller than c_4.

After finishing the construction of the tree structure we traverse it and sort the triangles inside every node. Actual triangles are stored first, followed by clusters of smaller triangles. Triangles are sorted by their area, large triangles first, while clusters are sorted by the number of triangles they contain, those including least triangles are first. This ordering will be beneficial when the object is rendered.

3.5 Global SEC–Trees

We have described how to build the SEC–Tree for a single object. Given a static scene, an additional SEC–Tree is created organizing the objects contained in the scene.

Now our input data is a set of n objects $J_1, \ldots, J_n$, each object J_i is modeled by a number of triangles $T_{i,1}, \ldots, T_{i,n_i}$. Additionally, we compute an axis aligned bounding box $B_i = \left(P_1^i, P_2^i\right), P_1^i, P_2^i \in \mathbb{R}^3$, enclosing O_i, where P_j^i contains the minimal respectively maximal coordinates of all triangles $T_{i,k}, 1 \leq k \leq n_i$. The size $A\left(J_i\right)$ of an object J_i is defined by the area of its triangles: $A\left(J_i\right) := \sum_{j=1}^{n_i} A\left(T_{i,j}\right)$.

Instead of considering accurate geometry, only the bounding box enclosing an object is regarded. Via this bounding box we define the center $z_i := \frac{1}{2}\left(P_1^i + P_2^i\right)$ and diameter $d\left(J_i\right) :=\parallel P_1^i - P_2^i \parallel$ of an object J_i. Bounding box position and diameter already provide sufficient information to determine objects located near to each other. Construction of the SEC–Tree does not rely on accurate geometry of an object but only the center point and span, which are used to define an initial neighborhood within the clustering, so all information needed to apply the algorithm described above are present.

Within each node of a global SEC–Tree, objects and clusters are sorted in a descending order by the number of triangles they contain. While triangles and clusters have been separated in SEC–Trees of singles objects, as triangles were followed by clusters, objects within the global SEC–Tree are more similar to clusters than a single triangle. Therefore, clusters and objects are mixed in the global SEC–Tree.

Even in scenes containing multiple objects we do not store the triangles on a per object base, but instead for object types. Every object type refers to a 3D-model and every object is specified by its type and a transformation matrix, describing how to obtain object triangles in world coordinates from object type triangles given in a local coordinate system.

Summarizing, for the complete scene a global SEC–Tree is build also, organizing its objects the same way triangles of a single object were structured. We only had to define an object's center and span.

3.6 Construction Costs of SEC–Trees

The time needed to build a SEC–Tree mainly depends on the clustering costs. We will present two variations with costs of $O\left(n\left(\log n+\epsilon^{-2}\right)\log\frac{D}{d_{min}}\right)$ respectively $O\left(n^2\log n\right)$, whereas D is the diameter of the scene, d_{min} is the minimal span of a triangle and ϵ an error which is permitted when neighbors are searched for. Further, we will show that the multiple repition of clustering phases does not lead to increased costs.

In the following, we will look at the costs of the arranging phase at first. Then, we will look at the clustering algorithm itself and show that one clustering phase will not take more time even if there is more than one group in which clustering has to take place, and finally we will prove that the overall costs do not differ from the costs of the clustering algorithm.

3.6.1 Costs of an Arranging Phase

The group creation approach presented in algorithm 1 needs to find the position of largest ratio between consecutive triangles' size (line 2). A naive solution would be to iterate through the complete list, resulting in a linear runtime. This leads to similar worst cases as the well known quicksort algorithm [Sed92], i.e. if the splitting position is always at the end position, resulting in quadratic costs. Modifying the algorithm as described in the following, costs of $O\left(n\log n\right)$ are possible.

Algorithm 1 divides a set of triangles, each time looking for the largest gap within the set. On the other hand, gaps do not change. An initial sorting of these gaps allows to establish an order of splitting positions, but algorithm 1 does not benefit from this information. Instead of looking at a set of triangles and finding a splitting position within, algorithm 4 starts with a splitting position and finds the set which has to be divided. Using the order of gaps, the next position is always known. An AVL-tree [Knu98] can be used to efficiently manage the sets by storing the end index of each set.

Algorithm 4 Alternative Construction of the Searchtree

```
Require: sorted set of triangles $M = \{T_1, \dots, T_n\}$ with $A(T_j) \leq A(T_i)$ for all
    $j \leq i$
 1: insert $n$ into AVL tree
 2: compute permutation $\psi$ such that $\frac{A(T_{\psi(i)})}{A(T_{\psi(i+1)})} \geq \frac{A(T_{\psi(j)})}{A(T_{\psi(j+1)})}$ for $i \leq j$
 3: for all $1 \leq i \leq n$ do
 4:     if $\psi(i)$ has been marked then
 5:         continue
 6:     end if
 7:     find next smaller $a_i := a(\psi(i))$ and larger $b_i := b(\psi(i))$ values than $\psi(i)$
        in AVL tree
 8:     if $b(i) - a(i) < c_1$ || $A(T_{a_i}) / (A(T_{a_1}) - A(T_{b_i})) > c_2$ || $A(T_{b_i}) < c_3$ then
 9:         mark $T_{a_i}, T_{a_i+1}, \dots, T_{b_i}$
10:     else
11:         insert $\psi(i)$ into AVL tree
12:     end if
13: end for
14: get groups from AVL tree
```

Given a set of triangles $T_1, \dots, T_n$, within an arranging phase we will find groups of similar sized triangles. W.l.o.g. we assume $A(T_i) \leq A(T_{i+1})$. Otherwise sorting the triangles would result in the required condition. Since sorting takes $O(n \log n)$ and we will see, that costs of algorithm 4 are $O(n \log n)$ as well, we can consider the triangles to be already sorted.

We define $r_i := \frac{A(T_i)}{A(T_{i+1})}, i = 1, \dots, n-1$, to be the size ratio of two consecutive triangles. Let $\sigma \in S_{n-1}$ be a permutation such that $\sigma(i) \leq \sigma(j)$ for $r_i \geq r_j$, i.e. σ sorts the ratios $r_{\sigma(i)}$ in a descending order. In the following we will only use its inverse, which we denote by $\psi := \sigma^{-1}$. ψ can be obtained by sorting the r_i values and keeping track of the initial position of each element. Then, the i-th largest ratio is between positions $\psi(i)$ and $\psi(i) + 1$.

Groups are created by splitting temporary groups at the points of the largest fraction. A temporary group is a set of consecutive triangles, which does not fulfill the conditions stated in line 8 of algorithm 4 and thus has to be divided further. Each group stores the indices of the triangles it contains. We used a search tree when describing the algorithm in section 3.2, however such a tree is not necessarily balanced. Therefore, the temporary groups are managed by an AVL-tree storing the index of the largest triangle contained in the temporary group. Initially one group containing all triangles is given. Thus, the AVL-tree contains only the value n. Splitting groups will be stopped at final groups, at which point contained triangles are tagged, indicating that we do not have to process them any further.

For $i \in \{1, \ldots, n-1\}$ we denote the starting point of the temporary group containing i by $a(i)$ and the endpoint by $b(i)$. Using the AVL-tree, these values can be received efficiently in $O(\log n)$.

The algorithm examines the positions $r_{\psi(1)}, \ldots, r_{\psi(n-1)}$ in order to decide where a group has to be split. After $r_{\psi(1)}, \ldots, r_{\psi(i-1)}$ have been evaluated, $r_{\psi(i)}$ is considered. If the triangle $T_{\psi(i)}$ has been tagged, the position $\psi(i)$ has been processed. Otherwise, the temporary group is split, resulting in the new groups from $a(\psi(i))$ to $\psi(i)$ and from $\psi(i)+1$ to $b(\psi(i))$, which corresponds to the sets L and R in algorithm 1. Then, the value $\psi(i)$ is inserted into the AVL-tree. Looking at the endpoints, one can decide if one or both of these two groups are a final group, i.e. the condition in line 1 of algorithm 1 is not met, in which case all triangles contained in the affected group are tagged.

This algorithm creates the triangle groups in $O(n \log n)$. This holds for the initial sorting of the r_i values utilizing a common sorting algorithm. The following loop considers every r_i exactly once, therefore passing through $n-1$ iterations. Within each iteration, finding the end positions of the temporary group containing the currently regarded element within the AVL-tree takes $O(\log n)$. The decision if one or both of the newly generated groups are final groups can be made in constant time. In addition, we have to consider the costs of tagging the triangles. While these costs within one iteration of the loop depend on the size of a created group, the algorithm tags each triangle exactly once, requiring $O(n)$ time. Therefore, all iterations of the loop combined take $O(n \log n)$ time, just like an initial sorting.

3.6.2 Clustering

In the following we will look at the time required to generate clusters from a group of triangles $T_1, \ldots, T_n$. However, instead of considering triangles, we only use their centers for clusters. Therefore, we can restrict our examination to the set of their center points $z_1, \ldots, z_n$.

The efficiency of the clustering algorithm depends on the efficiency of range queries. Epstein et al introduced *skip octrees* ([EGS05]), which we will utilize in our analasys. Given a set of n points in $\mathbb{R}^d$, skip octrees can be constructed in $O(n \log n)$. Insertions, deletions and point location can be performed in $O(\log n)$.

Given a point p and radius r, a $(1+\epsilon)$-approximate range query reports all points contained in $K(p,r)$ and no points located beyond $K(p, r+\epsilon)$. Points contained in $K(p, r+\epsilon) \setminus K(p,r)$ may or may not be reported. Such $(1+\epsilon)$-approximate range queries are supported. Skip octrees can answer $(1+\epsilon)$-approximate range queries in $O\left(\log n + \epsilon^{1-d} + k\right)$ with k being the number of reported points.

An $(1+\epsilon)$-approximate nearest neighbor of p is a point is a point q, such that $\| p - q \| \leq (1+\epsilon) \| p - v \|$, with v being a nearest neighbor of p. The skip octree reports an $(1+\epsilon)$-approximate nearest neighbor in $O\left(\epsilon^{1-d}\left(\log n + \log \epsilon^{-1}\right)\right)$. In our case, we have $d = 3$ as we are interested in center points of triangles in a 3-dimensional scene.

Clustering with Approximate Range queries

In the following we will analyse the costs of a variation of algorithm 2. One part of searching for neighbors within this algorithm was performing range queries. A naive approach to perform range queries is simply checking every point if it is located in the requested area. However, this leads to a linear runtime of $O(n)$ for each query. Various more efficient algorithms have been developed, see the surveys of Matousek [Mat94] and Agarwal and Erikson [AE99]. In our implementation we use kd-trees, which leads to worst case costs of $O\left(n^{\frac{2}{3}} + k\right)$ with k being the number of reported points. While this performs well in practice, we will allow $(1+\epsilon)$-approximate requests instead of exact ones as originally proposed. Using the skip octree, this leads to only logarithmic costs.

At first, we have a look at the number of outer loop iterations k. Let D be diameter of the bounding box surrounding the complete scene, and $d_{min} := min\{d(T_i) \,|1 \leq i \leq n\}$ the minimal span of all triangles. Then the initial neighborhood of each triangle is a sphere with a radius of at least $2d_{min}$. Every time a triangle T_i is chosen as a seed and we did not find a cluster, we double the neighborhood we have to consider the next T_i is chosen as a seed. After $\log \frac{D}{d_{min}}$ iterations the neighborhood covers the complete bounding box. Since this holds for every triangle, no more than $n \log \frac{D}{d_{min}}$ iterations are necessary.

At the beginning of each iteration (line 8), we have to find a triangle with a minimal neighborhood. The appropriate data structure to find this triangle is a heap. The initial construction of the heap requires $O(n \log n)$ ([CLR90]). In every iteration of the outer loop, the minimum can be found in constant time. Removing it and inserting this triangle with an enlarged neighborhood can be done in $O(\log n)$.

Inserting a cluster into the set F (line 19) corresponds to simply inserting a pointer into a list and therefore can be done in constant time, while we will update S (line 20) during the inner loop. Checking if $\| C \| \geq c$ can also be done in constant time if we count the number of iterations within the inner loop.

Now, we will consider the inner loop. The sets C and C_{TMP} are organized as double linked lists. Selecting and deleting an element from C_{TMP} and inserting it into C can be done in constant time. Finding a set M requires $O(\log n + \epsilon^{-2} + r)$ with r being the number of results reported by the skip octree. Instead of removing the set C from S after a cluster has been found as described in the algorithm, we will remove M immediately from the skip octree. This also means we have to reinsert C if we have not found a cluster, i.e. the condition in line 18 is not met after leaving the loop. Since these insertions take as long as removals before, we do not have to consider the reinsertions when looking at the overall costs. On the other hand, if we have found a cluster, we have to update the heap by removing the appropriate triangles. Since each triangles can not be contained in more than one cluster, heap updates take $O(n \log n)$ in total.

At the beginning of the outer loop the elements within the skip octree correspond to the set S_k. Within each iteration of the inner loop, we have to perform a range query and delete the reported elements from the skip octree. If r elements have been found, a range query and the following deletions require $O(\log n + \epsilon^{-2} + r \log n)$. Let r_i be the number of triangles found during all iterations of the inner loop within the ith iteration of the outer loop. The combined runtime of all inner loop iterations is then $O\left(\sum_{i=1}^{k} r_i \log n + r_i\epsilon^{-2}\right)$.

We can differentiate the iterations of the outer loop by those, which resulted in a cluster, and those, which did not find a sufficient number of triangles. Let be $I_1 := \{j|r_j \le c\}$ and $I_2 := \{j|r_j > c\}$. Then, we have

$$\begin{aligned}
\sum_{i=1}^{k}\left(r_i \log n + r_i\epsilon^{-2}\right) &= \left(\log n + \epsilon^{-2}\right)\sum_{i=1}^{k} r_i \\
&= \left(\log n + \epsilon^{-2}\right)\left(\sum_{i\in I_1} r_i + \sum_{i\in I_2} r_i\right) \\
&\le \left(\log n + \epsilon^{-2}\right)\left(\sum_{i\in I_1} c + n\right) \\
&\le \left(\log n + \epsilon^{-2}\right)\left(n \log \frac{D}{d_{min}} c + n\right) \\
&= O\left(n \log \frac{D}{d_{min}}\left(\log n + \epsilon^{-2}\right)\right)
\end{aligned}$$

In total, we have costs of $O(n)$ for determination the initial neighborhoods, $O(n\log n)$ to remove elements from the heap that have been inserted into a cluster, $n \log \frac{D}{d_{min}}$ iterations which need $O(\log n)$ to remove and if applicable insert a triangle into the heap and $O\left(n \log \frac{D}{d_{min}}\left(\log n + \epsilon^{-2}\right)\right)$ for all iterations of the inner loop combined and updating the set S. Since the overall costs are determined by the maximal value, in the worst case we have costs of $O\left(n \log \frac{D}{d_{min}}\left(\log n + \epsilon^{-2}\right)\right)$ for creating a cluster.

A Modified Clustering Approach

With some modifications on the clustering algorithm and the resulting clusters, we can achieve clustering costs which only depend on n and are indepent of ϵ and $\frac{D}{d_{min}}$.

Basically, algorithm 5 works like algorithm 2 did, by taking a triangle with minimal neighborhood and starting flooding at its center point. The main differences are the size of the initial neighborhood and how neighborhoods are enlarged. Further, some details have been concretized in section 3.6.2. These are the application of a heap and skip octree, which are integrated into algorithm 5.

We can use the approximate range queries and nearest neighbor searches for a fixed ϵ, in order to find lower bounds of neighborhoods we have to examine. This will enable us to perform exact range queries in logarithmic time. First, we will alter the initial neighborhoods by searching for a 2-approximate nearest neighbor. Finding such a neighbor for every triangle can be done in $O(n \log n)$. Then, we will use a fixed $\epsilon = 1$ for range queries, resulting in costs of $O(\log n) + r$. Furthermore, we will show that r does not exceed a constant for every query we perform. Then, we can check every one of the r reported points for being contained in the exact query region, requiring only constant time altogether. Therefore, each exact region query can be performed in $O(\log n)$. The function *range query* in algorithm 5 performs these operations.

The reason enabling us to perform exact queries efficiently, is that points will be removed from the set we have to consider, as soon as a cluster has been found. If the number points reported from a range query would exceed a certain constant, there has to be a subset of higher density, which forms a cluster and had to be found in a previous step.

In the following we will make some observations, which will enable us to analyse the costs of the modified clustering algorithm. We will use a skip octree Q synonymous to the set of points it contains.

Lemma 1 *u_i^k is lower bound on the distance from z_i to all points, which are neither already contained in a cluster nor have been found as a neighbor during a previous iteration, i.e. $\| z_i - \tilde{z} \| \geq u_i^k \forall \tilde{z} \in Q \setminus C_i^k$.*

Proof: Initially u_i^k is half the distance to a 2-approximate nearest neighbor and therefore a lower bound on the distance to the nearest neighbor. When u_i^k is updated, it becomes either the distance to the nearest neighbor of all points remaining in Q (line 12) or half the distance to a 2-approximate nearest neighbor of all points that are still contained in Q (lines 15 and 32). Since all points that have been neighbors in previous iterations or have been inserted into clusters are no longer contained in Q at these times (lines 11, 26), the lemma follows. $\square$

Remark: It might be possible, that started with a radius $r < u_i^k$ and seed z_i a cluster is encountered. However, in this case the additional points are no

Algorithm 5 Modified cluster construction

```
Require: Set of points $S = \{z_1, \ldots, z_n\} = \{z(T_1), \ldots, z(T_n)\}$
 1: build skip octree $Q$ from $S$
 2: for all $1 \leq i \leq n$ do
 3:    $u_i := \frac{1}{2}$ distance to 2-appr NN of $z_i$      {initial neighborhood}
 4:    $C_i := \{z_i\}$
 5: end for
 6: build Heap $H$ from $z_1, \ldots, z_n$ with keys $u_1, \ldots, u_n$
 7: $F = \emptyset$      {set of clusters}
 8: Outer loop:
 9: while $\| Q \| > 2 * c$ do
10:    receive and remove seed $z_i$ with minimal $u_i$ from $H$
11:    remove $C_i$ from $Q$
12:    $u_i :=$ distance to exact NN of $z_i$ within $K(z_i, 2 * u_i)$
13:    if $u_i > \min(H)$ or no neighbor has been found then
14:       if no neighbor has been found then
15:          $u_i := \frac{1}{2}$ 2-appr-NN search($Q, z_i$)
16:       end if
17:       reinsert $z_i$ into $H$ with key $u_i$, $C_i$ into $Q$
18:       continue outer loop
19:    end if
20:    $C_{Tmp} := \{z_i\}$
21:    Inner loop:
22:    while $C_{Tmp} \neq \emptyset$ do
23:       choose arbitrary $z_j$ from $C_{Tmp}$
24:       $C_{Tmp} := C_{Tmp} \setminus \{z_j\}$
25:       $C := rangequery(Q, K(z_j, u_i))$; $C_i := C_i \cup C$; $C_{Tmp} := C_i \cup C$
26:       remove $C$ from $Q$
27:    end while
28:    if $\| C_i \| \geq c$ then
29:       $F := F \cup \{C_i\}$ {add $C_i$ to set of clusters}
30:       $H := H \setminus C_i$ {make sure, each element is contained in only one cluster}
31:    else
32:       $u_i := \frac{1}{2}$ 2-appr-NN search($Q, z_i$)
33:       insert $z_i$ into $H$ with key $u_i$
34:       insert $C_i$ into $Q$
35:    end if
36: end while
37: $F := F \cup$ elements of $Q$ {consider remaining points to be a cluster}
```

direct neighbors of z_i. If $\tilde{z} \in C_i^k \setminus C_i^{k-1}$ is a new point we found when searching with radius r, then there is a chain of points $z_{i_2}, \ldots z_{i_l} \in C_i^k \setminus C_i^{k-1}$, $z_{i_1} \in C_i^{k-1}$ with $z_{i_l} = \tilde{z}, z_{i_j} \in K\left(z_{i_{j-1}}, r\right)$. Therefore, we have $u_{i_l}^k \leq r < u_i^k$ and z_{i_l} is chosen as a seed before z_i.

Also note that the seed is only significant for choosing the size of neighborhoods and as it is one known point of the cluster. Flooding could start with an abritrary point of the cluster without generating a different result as long as the same radius is used in the range queries. This is true because $z_{j_1} \in K\left(z_{j_2}, r\right) \Leftrightarrow z_{j_2} \in K\left(z_{j_1}, r\right)$ for any r.

Lemma 2 *Let u_i^k and u_j^l be the radius of the considered neighborhood in iterations k and $l, k < l$. Then $u_i^k \leq u_j^l$ holds.*

Proof: The neighborhood's radius u_i is always the minimal value on the heap. Since u_i is always replaced by a larger value, the claim follows. □

If we have not found a cluster during an iteration of the outer loop, the set C_i is reinserted into Q (line 34). However, at no time points are removed from a set C_i. Therefore, we have to ensure that all elements contained in C_i are elements of Q at the beginning of the iteration, otherwise we would insert points already contained in a cluster. The following lemma states that this does not happen.

Lemma 3 *If $\tilde{z}$ has been inserted into C_i, then $\tilde{z}$ and z_i will be elements of the same cluster.*

Proof: From $\tilde{z} \in C_i^l$ during an iteration l follows the existence of a chain of points $z_{i_1}, \ldots, z_{i_t}$, such that $z_i = z_{i_1}$, $\tilde{z} = z_{i_t}$ and $z_{i_r} \in K\left(z_{i_{r-1}}, u_l^k\right)$. Now consider a iteration $\tilde{l} > l$. From lemma 2 we know $u_l \leq u_{\tilde{l}}$, and therefore we have $K\left(z_{i_r}, u_l\right) \subseteq K\left(z_{i_r}, u_{\tilde{l}}\right)$. Now, if we encounter one of the points $z_{i_1}, \ldots, z_{i_t}$ during the flooding in iteration $\tilde{l}$ we will find the other points as well. If one point is inserted into a cluster, this holds for all these points. □

Lemma 4 *Let z_i be the seed in iteration k of the outer loop in algorithm 5 and u_i^k the radius of considered neighborhood. Then every range query within the inner loop (line 25) returns $O(1)$ results.*

Proof: z_i is selected such that $u_i^k = \min(H)$ (line 25). We have to show that $K\left(z_j, 2u_i^k\right)$ does not contain more than a constant number of points, which have not been inserted into clusters. Those points already contained in clusters have been removed from Q and therefore will not be encountered again.

Let be $z_{j_1}, \dots, z_{j_l} \in K\left(z_j, 2u_i^k\right)$, such that these points are not contained in a cluster before iteration k. We have $z_{j_r} \in K\left(z_{j_s}, \frac{u_i^k}{2}\right) \Leftrightarrow K\left(z_{j_r}, \frac{u_i^k}{4}\right) \cap K\left(z_{j_s}, \frac{u_i^k}{4}\right) \neq \emptyset$. At least an eighth of $K\left(z_{j_s}, \frac{u_i^k}{4}\right)$ is located within $K\left(z_j, 2u_i^k\right)$, since it contains the center z_{j_s}. If we consider the ratio of volumes between $K\left(z_j, 2u_i^k\right)$ and $\frac{1}{8}K\left(z_{j_s}, \frac{u_i^k}{4}\right)$, we get

$$\frac{\frac{4}{3}\pi 8 \left(u_i^k\right)^3}{\frac{1}{4}\frac{4}{3}\pi\frac{1}{512}\left(u_i^k\right)^3} = 16384$$

Therefore, given 16384 Spheres of radius $\frac{u_i^k}{4}$ and with a center contained in $K\left(z_j, 2u_i^k\right)$, at least one point is located within two of the smaller spheres.

Now assume $l \geq 16384c$. We will show that this leads to a contradiction and therefore only $O(1)$ points can be located within $K\left(z_j, 2u_i^k\right)$.

Given $16384c$ spheres of radius $\frac{u_i^k}{4}$ and with a center located within $K\left(z_j, 2u_i^k\right)$ there exists at least one point $\tilde{z}$ located within c of the smaller spheres. W.l.o.g. we assume $\tilde{z} \in K\left(z_{j_1}, \frac{u_i^k}{4}\right), \dots, K\left(z_{j_c}, \frac{u_i^k}{4}\right)$. Then, $z_{j_1}, \dots, z_{j_c} \in K\left(z_{j_1}, \frac{u_i^k}{2}\right)$ holds.

According to lemma 1, $u_{j_1}^k$ is a lower bound on the distance from z_{j_1} to all points, which have not been inserted into a cluster or C_{j_1}. Since we found $z_{j_1}, \dots, z_{j_c}$ during a range query, we know $z_{j_1}, \dots, z_{j_c}$ have not been inserted into a cluster yet. This also ensures that we have $\| C_{j_1}^k \| < c$, otherwise we would have found a cluster containing z_{j_1}. Therefore, one of the points $z_{j_1}, \dots, z_{j_c}$ is no element of $C_{j_1}^k$, w.l.o.g. this is true for z_{j_2}. Then, we get $u_{j_1}^k \leq \| z_{j_1} - z_{j_2} \| \leq \frac{u_i^k}{2} < u_i^k$. This is a contradiction, since u_i^k was the minimal element of the heap. Therefore, we know that the number reported points for one range query is $l < 2048c$. □

Consequence: The exact nearest neighbor searches in line 12 can be performed in $O(\log n)$. First, we perform a range query, finding points contained in $K\left(z_i, 2u_i^k\right)$. This requires $O(\log n)$ and returns up to a constant number of results. The proof is analogous to the one above. Then, we can find the nearest neighbor of z_i within the result set in constant time.

Now, we will have a closer at the number of iterations the algorithm passes. The following lemma considers the iterations of the outer loop.

Lemma 5 *The outer loop of the modified clustering algorithm is iterated at* *most* $O\left(n^2\right)$ *times.*

Proof: Consider all iterations with a fixed seed z_i. Regarding the conditions in lines 13 and 14, there are three different cases which might occur.

Case 1: No neighbor was found in $K\left(z_i, 2u_i^k\right)$. When u_i^k was assigned its value, there was a neighbor located within this neighborhood as $2u_i^k$ was the distance to a 2-approximate nearest neighbor. If this point is no longer available, it must have been inserted into a cluster. Since each point is contained in only one cluster, this can only happen $n-1$ times.

Case 2: There is a neighbor in $K\left(z_i, 2u_i^{k-1}\right)$ but $u_i^k > min\left(H\right)$ after updating u_i^k in line 12. Now, there are two possibilities which can occur the next time z_i is chosen as the seed. After line 12, we have either $u_i^k = min\left(H\right)$, which will be case 3, or $u_i^k > min\left(H\right)$. If $u_i^k > min\left(H\right)$, the nearest neighbor must have been removed, i.e. it has been inserted into a cluster. Therefore, case 2 can only occur as often as case 3 plus $n-1$ times.

3. Now, u_i^k is assigned no different value after finding the nearest neighbor $\tilde{z}_i$ of z_i in $K\left(z_i, 2u_i^k\right)$. Then, $\tilde{z} \in K\left(z_i, u_i^k\right)$ holds. On the other hand, we know that $\tilde{z} \notin C_i^{k-1}$, because all elements of C_i^{k-1} have been removed from Q before performing the range queries (line 11). Therefore, $C_i^k \supsetneq C_i^{k-1}$ follows. This event occurs at most $n-1$ times.

Combining all three cases, we have at most $4\left(n-1\right)$ iterations with seed z_i in total. Since this holds for every i, we have $O\left(n^2\right)$ iterations of the outer loop in total. □

Now, we will use this knowledge on the number of outer loop iterations to have a closer look at the inner loop. Similar to the previous clustering algorithm, we will not consider the cost of the inner loop within one iteration of the outer loop, but rather the costs of all inner loop iterations combined.

Lemma 6 *All inner loop iterations of the modified clustering algorithm combined have costs of* $O\left(n^2 \log n\right)$.

Proof: Let r_i be the number of triangles found during all iterations of the inner loop within the i-th iteration of the outer loop and l the number of overall outer loop iterations, analogous to the runtime consideration of the previous clustering approach.

The costs of the operations within one inner loop iterations are as follows: Removing z_j from C_{TMP} (line 24) takes constant time if C_{TMP} is organized as a list and z_j is chosen the first element.

A single range query (line 25) takes $O(\log n)$, since a 2-approximate range query can be performed in $O(\log n)$ first, which reports only $O(1)$ results according to lemma 4. Then, we can check in constant time, which of the reported points are contained in the exact query region.

The costs of the range queries in all iterations combined are $\sum_{i=1}^{l} r_i \log n = O(n^2 \log n)$. The proof is analogous to the one we gave for the combined runtime of inner loop iterations for the previous clustering approach.

The costs of removing one point reported by the range query from Q (line 26) are $O(\log n)$. Therefore, removing the results of the range queries from Q costs as much as the queries themselves. □

Combining these results we can now determine the costs of the modified clustering algorithm:

Theorem 1 *The modified clustering algorithm has costs of* $O(n^2 \log n)$.

Proof: The intitialization consists of building the skip octree Q, the heap H and calculating the starting radii of neighborhoods u_i^k, which take $O(n \log n)$ each (lines 1-7).

Within each iteration of the outer loop, we have to select and remove an element from the heap ($O(\log n)$), remove C_i from Q ($O(\log n)$ since $\| C_i \| < c$), and find the nearest neighbor of z_i ($O(\log n)$).

When checking conditional clauses in lines 9 and 28, we have to know the number elements currently contained in the data structure. While counting these elements might take linear time, we can avoid this by keeping track of set cardinalities whenever we insert or remove an element from Q or C_i. Then, we can always decide in constant time if a set becomes larger or smaller than some threshold (lines 9, 28). Obviously, modifying a counter is faster than altering one of the sets and therefore has no influence on the overall costs.

In case the if-clause in line 13 is met, a nearest neighbor search might have to be performed (line 15), which takes $O(\log n)$. This holds for the insertion of z_i into H and C_i into Q as well (line 17), while checking the if-clause itself only requires constant time.

As we have seen in lemma 6, all inner loop iterations have costs of $O(n^2 \log n)$ combined, so we do not have to regard lines 22 to 27 when examining the costs of the outer loop.

Depending on the result of the if-clause in line 28, we either have to perform a 2-approximate range query ($O(\log n)$), insert z_i into H ($O(\log n)$) and reinsert C_i into Q ($O(\log n)$ since $\| C_i \| < c$, lines 32-34), or we have to insert C_i into F which corresponds to inserting a pointer into a list ($O(1)$) and remove the elements of C_i from the heap. The costs of these heap updates depend on the number of elements the cluster contains. However, since each element is a member of exactly one cluster, not more than n delete operations will we performed in total.

Summarized, we have $O(n^2)$ iterations of the outer loop with costs $O(\log n)$ and in addition costs of $O(n^2 \log n)$ for all iterations of the inner loop combined, as stated in the theorem. □

3.6.3 Overall Costs

We have seen before, that modifications on the clustering algorithm have an influence on the construction costs. In both cases, the costs of creating the complete SEC–Tree will be identical to the costs of the clustering algorithm. We will see that this holds for any clustering algorithm, if it produces clusters with an average size of at least $\frac{n}{c}$ and its costs are specified as follows: $O(nf(n,m))$, whereby $f(n,m) = \log n + g(n) h_1(m) + h_2(m)$ with a monotonously ascending function g. h_1 and h_2 are arbitrary positive functions independent of n. Through the remaining chapter, we will only consider this more general function instead of the two concrete examples $n(\log n + \epsilon^{-2}) \log \frac{D}{d_{min}}$ and $n^2 \log n$.

Both examples are covered by f. If we define $g(n) = \log n$, $m = (\epsilon, D, d_{min})$, $h_1(m) = \log \frac{D}{d_{min}}$ and $h_2(m) = \epsilon^{-2} \log \frac{D}{d_{min}}$ we the result of the analysis of the first clustering algorithm. In case of the modified version, we have $g(n) = n \log n$ and $h_1 = h_2 = 1$. The single $\log n$ term of f has no effect in these cases, but we have seen in 3.6.1 that the creation of groups might take $O(n \log n)$ and therefore the assumption that this holds for the clustering algorithm as well is no drawback. It will simplify notations in the following sections, as we will get costs of $O(n \log n)$ anyways.

In this section we will show that the overall costs of building a SEC–Tree do not differ from the time needed to create a cluster, which is $O(nf(n,m))$.

First, we will look at a single clustering phase.

Costs of one Clustering Phase

Within a clustering phase, groups created in an arranging phase are visited and each group is clustered. Then, the created clusters are inserted the groups again, see lines 7 - 20 of algorithm 3.

Consider k groups consisting of $n_1, \ldots, n_k$ triangles with $\sum_{i=1}^{k} n_i = n$. Clusters are inserted into the a group of larger triangles according to the their size. Let $m_{i,j}, 1 \leq i, j \leq k$ be the number of clusters created in group j and inserted into group i. Then $m_{i,j} = 0$ for $j \geq i$, as clusters are inserted into groups of triangles larger than those the cluster is derived from. Let $\tilde{n}_i = n_i + \sum_{j=1}^{k} m_{i,j}$ be number total number of elements encountered in group i, when it is clustered. Every cluster we create within one group has at least c elements, besides the last one, which consists of the last elements if there are less than $2c$. However, if this last cluster contains less than c elements, together with the second but last there are at least $2c$ elements, since we consider all elements to be one cluster if there are only $2c$ or less elements left (see algorithm 2 line 6 and algorithm 5 line 9). So, on average every cluster has at least c elements, and therefore for the number of clusters created in group j holds: $\sum_{i=1}^{k} m_{i,j} \leq \lfloor \frac{\tilde{n}_j}{c} \rfloor$.

As we have seen in the previous section, the time needed to create the clusters within all groups is $O\left(\sum_{i=1}^{k} \tilde{n}_i f\left(\tilde{n}_i, m\right)\right)$. We will show that this does not exceed $O\left(nf\left(n, m\right)\right)$.

At first we have a closer look at

$$\sum_{i=1}^{k} \tilde{n}_i = \sum_{i=1}^{k} \left(n_i + \sum_{j=1}^{k} m_{i,j} \right)$$

$$= n + \sum_{i=1}^{k} \sum_{j=1}^{k} m_{i,j} = n + \sum_{j=1}^{k} \sum_{i=1}^{k} m_{i,j}$$

$$\leq n + \sum_{j=1}^{k} \lfloor \frac{\tilde{n}_j}{c} \rfloor \leq n + \sum_{j=1}^{k} \frac{\tilde{n}_j}{c} \leq n + \sum_{j=1}^{k} \frac{\tilde{n}_j}{2}$$

Looking at the start and end of the inequality we have

$$\sum_{i=1}^{k} \tilde{n}_i \leq n + \frac{1}{2}\sum_{j=1}^{k} \tilde{n}_j$$

$$\Leftrightarrow \sum_{i=1}^{k} \tilde{n}_i \leq 2n$$

Since a group can not contain more elements than the original number of triangles, we have $\tilde{n}_i \leq n$ for every i. With these results we get

$$\sum_{i=1}^{k} \tilde{n}_i f\left(\tilde{n}_i, m\right)$$

$$\leq f\left(n, m\right) \sum_{i=1}^{k} \tilde{n}_i$$

$$\leq 2nf\left(n, m\right)$$

$$= O\left(nf\left(n, m\right)\right)$$

Within one clustering phase, we do not only have create the clusters, but also have to insert them into the appropriate groups (line 17). The number of these insertions is

$$\sum_{i=1}^{k}\sum_{j=1}^{k} m_{i,j} = \sum_{i=1}^{k} \tilde{n}_i - n \leq n$$

The groups are organized in an AVL tree, therefore inserting one cluster into a group takes $O\left(\log n\right)$, so all insertions combined need at most $O\left(n \log n\right)$. At this point the single $\log n$ term in f comes in handy, and we finally get that the overall time necessary for a clustering phase is $O\left(nf\left(n, m\right)\right)$.

Overall Costs of all Phases

Arranging and clustering phases are repeated until we start an arranging phase with a number of clusters below our threshold (algorithm 3, line 2). Now, let k denote the number these phases are iterated and n_i be the number of triangles or clusters considered as the input of the i-th iteration, whereby

$n_1 = n$ is the number initial triangles. Then the time needed to build the complete SEC–Tree is $O\left(\sum_{i=1}^{k} n_i f\left(n_i, m\right)\right)$.

The triangles in the last group within iteration i are the input for iteration $i+1$. There are two possibilities, when triangles are inserted into this last group. Either by creating a cluster which is larger than the triangles contained in the input set, or by moving triangles into the next group because there were less than c_4 triangles, until the last group is reached.

As we have already seen in before, every cluster contains on average at least c triangles, therefore at most $\frac{n_i}{c}$ clusters can be inserted the last group. Further, not more than c_4 triangles can be moved into this last group without creating clusters, otherwise these triangles would have been elements in the second but last group and we would have clustered this group instead pushing the triangles forward. Further, a triangle inserted into a cluster can not be one of the triangles inserted into the last group without being part of a clustering process. Therefore, we have

$$n_{i+1} \leq \frac{n_i - c_4}{c} + c_4$$

With $n_1 = n$ this leads to $n_i \leq \frac{n-c_4}{c^{i-1}} + c_4$. This can easily be proven by induction: The induction beginning is already stated above if we replace i by one: $n_2 \leq \frac{n_1 - c_4}{c} + c_4$. Now consider $i \geq 2$. Then

$$n_i \leq \frac{n_{i-1} - c_4}{c} + c_4 \leq \frac{\frac{n-c_4}{c^{i-2}} + c_4 - c_4}{c} + c_4 = \frac{n - c_4}{c^{i-1}} + c_4.$$

Therefore, after $i = \log_c n$ iterations n_i is at most c_4 and thus the number of iterations is $k \leq 1 + \log_c n$.

Since the number of triangles inserted into the last group during a clustering phase can not be larger than the number of triangles the corresponding arranging phase was started with, it follows that $n_i \leq n_{i-1}$ and especially $n_i \leq n$. Now, we can now look at the overall costs utilizing the preceding estimations:

$$\sum_{i=1}^{k} n_i f\left(n_i, m\right)$$

$$\leq f\left(n, m\right) \sum_{i=1}^{k} n_i$$

$$\begin{aligned}
&\leq f(n,m)\sum_{i=1}^{k}\left(\frac{n-c_4}{c^{i-1}}+c_4\right)\\
&= f(n,m)\left(kc_4+(n-c_4)\sum_{i=1}^{k}\frac{1}{c^{i-1}}\right)\\
&\leq f(n,m)\left(c_4(\log_c n+1)+(n-c_4)\frac{c}{c-1}\right)\\
&\leq (2n+c_4(\log_c n+1))\,f(n,m)\\
&= O(nf(n,m))
\end{aligned}$$

Finally, our result is that the complete SEC–Tree can be build in $O(nf(n,m))$. Note, how the algorithm used to perform the range queries influences the overall costs. Under reasonable assumptions on the performance of the clustering algorithm, the costs do not differ from the time needed to create to complete tree, besides a constant factor. The only assumption necessary was, that clusters created within one group have at least c elements on average and that the clustering algorithm has costs of $O(n\log n)$ or more.

CHAPTER 4

RENDERING WITH TRIANGLE BUDGETS

If the visualization of a scene becomes too slow, navigating through the virtual environment is nearly impossible. Therefore, rendering has to be fast and the frame rate should not fall below a minimal number. Navigation in a scene becomes possible if ten frames per second are rendered. In order to achieve a smooth visualization, even higher frame rates are necessary.

Since we do not use expensive effects for lighting or similar operations, per pixel operations claim only small part of computation power and do not limit rendering speed, therefore we are not fill rate limited. Further, only little cpu effort is needed since there are no expensive computations during rendering. However, in dynamic scenes we will have to touch every object. This results in cpu limited rendering for large scenes, with scenes of 20.000 objects requiring as much time to process objects than for actually rendering the scene with 300.000 triangles. We need the cpu to perform frustum culling, move objects to new positions and assign weights to them. Navigation in such scenes is still possible, while in static scenes this disadvantage is abated. Traversing the SEC–Tree of objects itself is much less expensive than rendering the triangles we encounter during the traversal.

The main limiting factor is the amount of triangles in the scene. Since the graphics card is only capable of a certain number of operations per second, rendering necessarily will be too slow if the number of triangles exceeds a limit, depending on the actual hardware. If we want to render always at high frame rates, we have to limit the number of processed primitives. Therefore, only a subset of all triangles is considered by our rendering algorithm. The size of this subset is given in advance, depending on the hardware. There are never more primitives touched than the given limit, such that the number of graphics operations is independent of the scene's size. This may result in image errors, as more triangles might be visible than we have chosen. Additionally, if triangles would be chosen arbitrarily we might not catch those with the most impact on the correct image, since the chosen triangles might be occluded, their area might be small, or they could be located completely outside the view frustum. To get high image quality, we have to pick the triangles with most impact on the correct image and reduce errors this way. On the other hand, finding exactly those triangles would be to expansive again, so what we are looking for is a good estimation.

Generally, large primitives contribute more to the image than smaller ones. But not only actual area is crucial, more important is the area of the projected primitive on the screen. This depends on the actual area, the distance from the viewer's position, the orientation and the position relative to the view frustum, i.e. whether a triangle is completely, partially or not at all inside the view frustum.

We do not consider orientation of triangles as this would be to expansive. However, clusters and objects contain several primitives of different orientation, thus changing the viewing angle to an object some triangles might turn away from the viewer, while other triangles turn towards the user, such that the effects of different orientations are at least partially compensated. The SEC–Tree gives information about size and area of triangles. In the data structure, all triangles stored in one node are of approximately equal size. The clustering approach groups triangles, which are near to each other, and therefore are nearly at the same distance from the viewer.

A frame is rendered in two phases. In a first phase, each object is assigned a number of triangles it is allowed to render. We will describe, how this number is determined for static scenes in Section 4.2 and consider dynamic scenes in Section 4.4. Given this number, the second rendering phase starts. Now, a SEC–Tree is used to decide which triangles of an object are chosen.

4.1 Selecting Triangles from Objects

3D models created in CAD systems are highly detailed and therefore complex. Given an object by a set of triangles, SEC–Trees create a hierarchy of details. Objects like machines are highly regular and symmetric. By searching for clusters of equally sized triangles, details can be reconstructed. As an example, buttons may be modeled as a cylinder and consist of a group of small triangles with equal distances between them.

Figure 4.1 shows an example of a SEC–Tree. On the left the complete tree is reproduced, while the right shows the geometry contained in some example nodes. For each of these nodes, all triangles in the complete subtree beneath are pictured. The root represents the complete model. On a path to a leave an object is decomposed into details. At level two, the model is divided into the left and the right part. Level three contains the instrument panel as an example and the leaves represent e.g. buttons or wheels.

In the following, we describe the rendering of a single object. A triangle budget is assigned to nodes of the SEC-tree, limiting the number of rendered primitives. At first, triangles at higher levels defining the structure are rendered. Then, spare triangles are distributed on more detailed parts of the object. Given node N_i of a SEC–Tree, we denote its triangles by $T_1^i, \ldots, T_{k_i}^i$ and succeeding nodes by $N_1^i, \ldots, N_{l_i}^i$, sorted in the order determined during the preprocessing. Rendering a cluster is done as follows. At first, we render the triangles stored in this cluster up to the given limit. Thus, if a N_i is assigned t_i triangles, first $T_1^i, \ldots, T_{min\{t_i,k_i\}}^i$ are rendered. All triangles contained in a cluster are of nearly equal size, while a triangle is of approximately the same area as the sum of triangles in a succeeding cluster N_k^i. Hence, one triangle in this node has greater impact than triangles on deeper levels of the tree. Triangles were sorted according to their area, thus we choose the largest triangles within a node. If we have rendered the triangles and did not reach the limit, the remaining number of primitives is distributed on all subclusters. In this case we have $t_i > k_i$ and the $t_i - k_i$ remaining triangles are assigned to the succeeding nodes. Recursively node N_k^i receives and renders $w_k^i\,(t_i - k_i)$ triangles, whereby the weight w_k^i is defined as follows: we denote the viewer's position by V, the center of a node N by $Z\,(N)$, the summed area of triangles

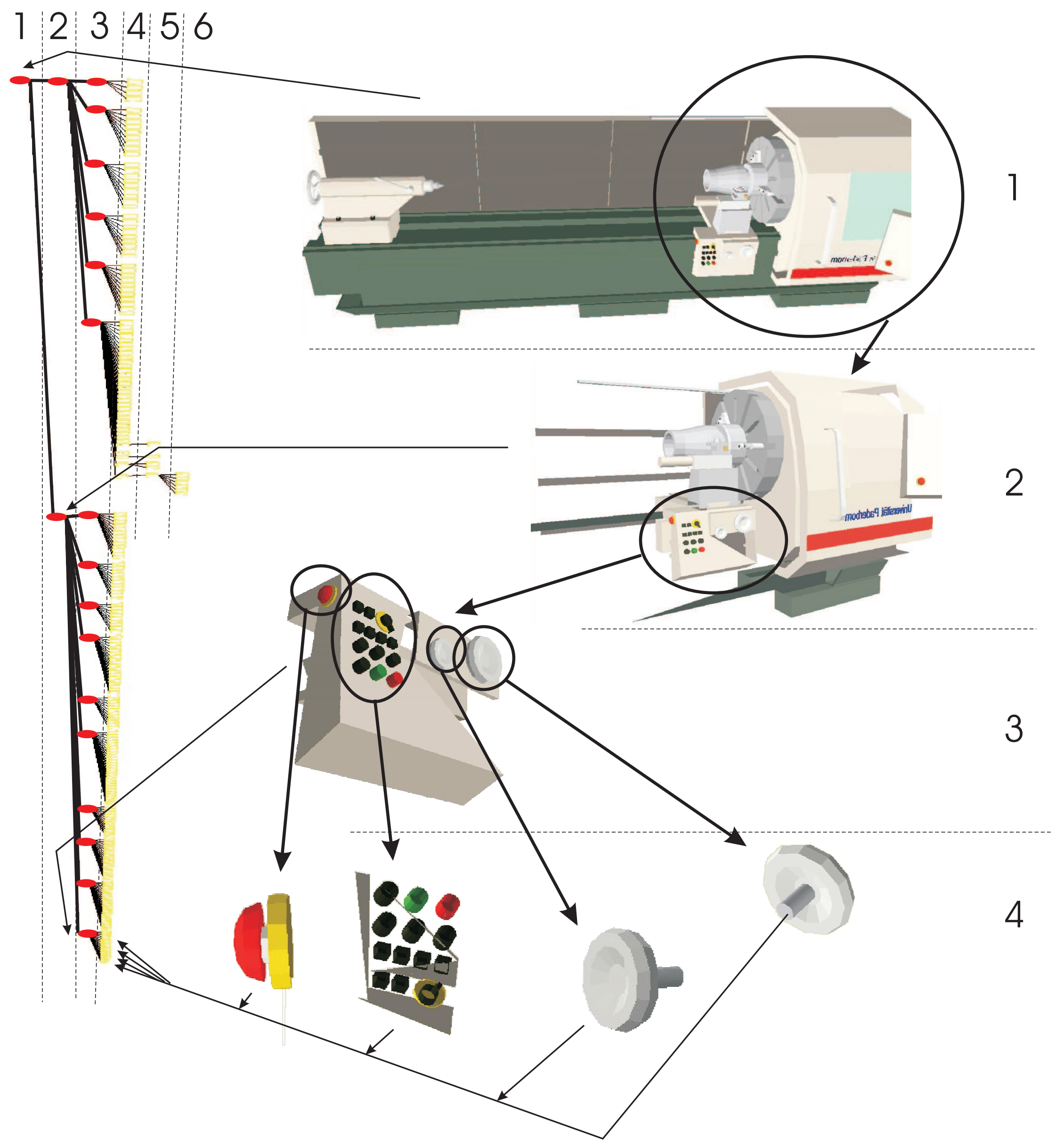


Figure 4.1: SEC–Tree of a lathe

contained in N and the subtree beneath by $A(N)$ and define

$$w_k^i := \frac{A\left(N_k^i\right)/||V - Z\left(N_k^i\right)||^2}{\sum_{j=1}^{l_i} A\left(N_j^i\right)/||V - Z\left(N_j^i\right)||^2}$$

Weights are chosen this way, because the projected size of a triangle increases linear with its actual area and decreases quadratic to the distance from the viewing point. Normalizing weights ensures that they sum up to one and therefore $\sum_{j=1}^{l_i} w_j^i \left(t_i - k_i\right) = \left(t_i - k_i\right)$.

Since only few objects partly intersect the view frustum and most are entirely in or outside, we do not consider frustum culling within one object. Testing every node for its position relative to the frustum would be to expensive to justify the benefits, so if an object intersects the view frustum, we consider every node of the corresponding SEC–Tree to be within the frustum as well.

Clusters are sorted in a descending order according to the number of triangles inside that subtree. If a cluster is assigned a limit greater than the number of primitives it contains at all, this will probably happen to a cluster we encounter early, and we can assign these waste primitives to other nodes later. Precisely, we add them to the next cluster's limit. If we have rendered all triangles but did not exhaust the budget of this node, we report the redundant number back to the predecessor in the SEC–Tree, where it is assigned to the next cluster.

If the clusters were not sorted this way, we might encounter clusters with few triangles late, such that none of the upcoming clusters can make use of more primitives. These waste triangles had to be reassigned to clusters we have already rendered. Furthermore, we had to know which triangles have been rendered and where to spend these additional triangles.

The number of primitives has to be an integer, since we can not render half triangles, but we might get float values because of non integer weights. In order to achieve high rendering speed, we never render more triangles than the given limit. If this is a float value, we round down. If we always round down, there might be a count difference between the overall limit and the sum of rounded limits. These left triangles are assigned to the next cluster we encounter. Due to our ordering, this is the cluster containing the next smallest number of triangles. On the other hand this means we can expect large and therefore important triangles there. Thus, raising the triangle limit of this cluster is most promising.

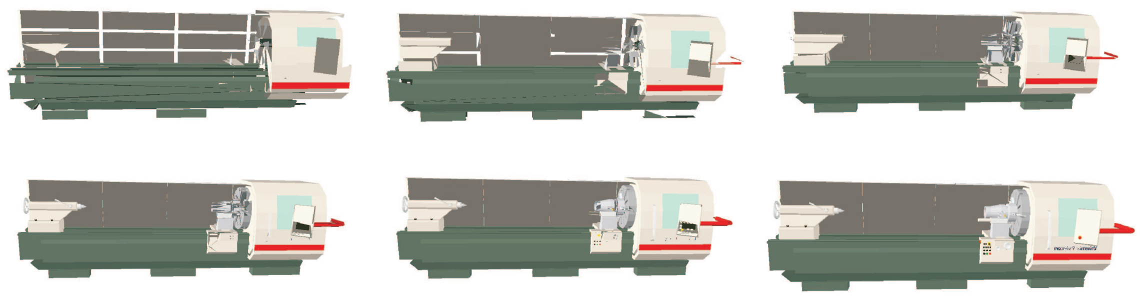


Figure 4.2: Lathe rendered with 300, 500, 1000, 3000, 5000 and all 7794 triangles

Figure 4.2 shows a lathe rendered from one viewpoint with a different number of triangles assigned to the root node. This model consists of 7794 triangles in total. Even with only 300 triangles rendered, the model is well recognizable, although more than 96% of all triangles are missing. If the distance to the viewer increases, no difference to the complete model is identifiable. Details increase with the number of rendered triangles, stepwise refining the representation of the object.

SEC–Trees control the degree of details depending on the viewer's position within different parts of one object. This is different from many other approaches, discrete LODs as an example, which can alter the degrees of detail for a complete model, but do not allow to change details for only a part of an object. Since weights assigned to SEC–tree nodes are view dependent, parts within an object gain more triangles if they are located near to the view point. Figure 4.3 shows one model rendered from a position on the top right compared to a viewpoint at the center. Large triangles dominating the structure are displayed from both viewpoints, giving a good impression of the model. Details like lattices and parts of robots disappear at the other end of the model but near to the viewer's position errors are avoided.

4.2 Global SEC–Trees in Static Scenes

Now, that we have seen how triangles are chosen from an object, we still have to determine how many triangles each object receives. In static scenes we use an object hierarchy for this, the global SEC–Tree.

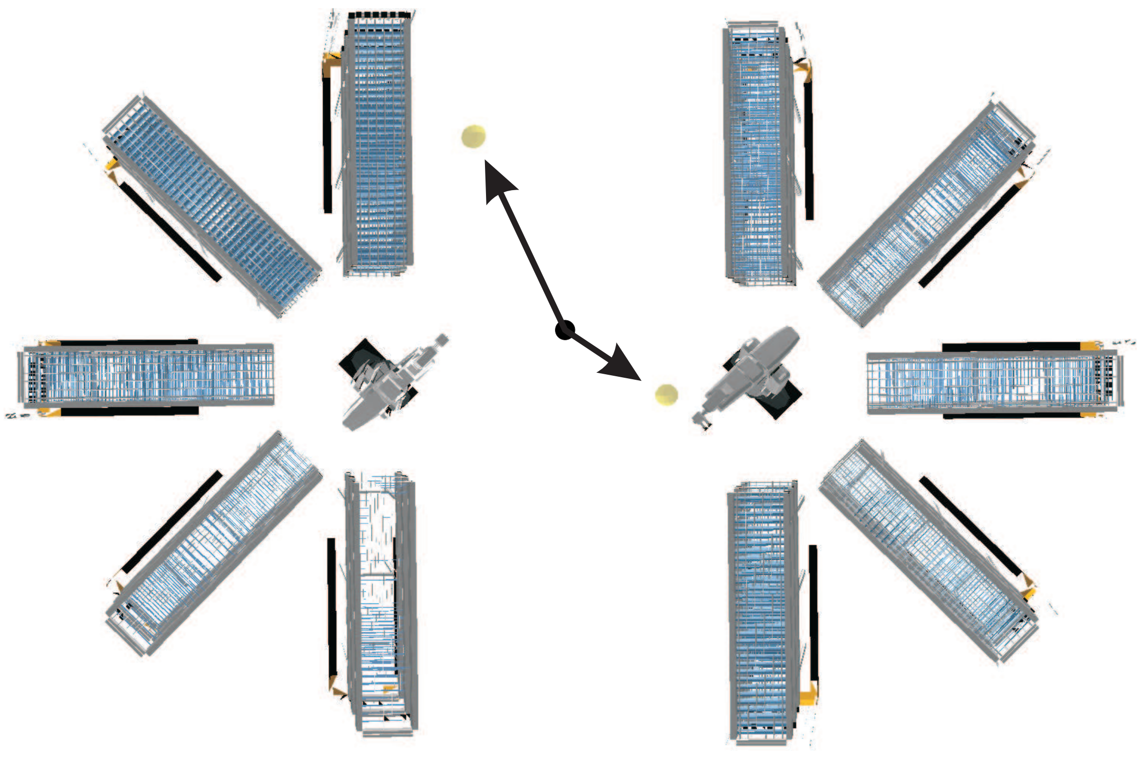


Figure 4.3: Large objects are displayed highly detailed at positions near to the viewer

Rendering the global SEC–Tree basically works the same way as rendering the tree of a single object. However, while we rendered triangles directly when we encountered them in a SEC–Tree node, objects are more similar to clusters, i.e. they contain multiple triangles, and therefore are treated similarly.

Consider a global SEC–Tree T with nodes $\mathcal{N} = \{N_1, \ldots, N_n\}$, created from a set of objects $\mathcal{J} = \{J_1, \ldots, J_m\}$. The viewer is located at position V. We will assign a limit L_i to each node N_i, which is determined recursivly and given for the root node depending on hardware capabilities. At most L_i triangles contained in the subtree with the root N_i are rendered.

For each node N we define $e(N) \subseteq \mathcal{N} \cup \mathcal{J}$ to be the children and objects of N. We will refer to $e(N_i) = \{E_{i,1}, \ldots E_{i,n_i}\}$ as elements of N_i, since we do not have to differ between objects and subtrees. Then, the size of node N is $A(N) = \sum_{M \in e(N)} A(M)$.

The maximal number of rendered triangles L_r is given in advance to the root N_r as a limit, depending on hardware capability. The limit L_i of node N_i is distributed and each element of $E_{i,j} \in e(N_i)$ is given a smaller limit $l_{i,j} = w_{ij} L_i$ with $\sum_{k=1}^{n_i} l_{i,k} \leq L_i$.

The area of a triangle projected to screen depends quadratic on its distance to the viewer. Therefore, we choose weights w_{ij} just like before depending on the distance of an element's center $Z(E_{i,j})$ to the viewer:

$$w_{ij} := \frac{A(E_{i,j}) \,/\, \| V - Z(E_{i,j}) \|^2}{\sum_{k=1}^{n_i} A(E_{i,k}) \,/\, \| V - Z(E_{i,k}) \|^2} \tag{4.1}$$

Let $t(J_j)$ denote the numbers of triangles represented by object $J_i \in \mathcal{J}$, $t(N) := \sum_{M \in e(N)} t(M)$ the number of triangles within a subtree with root $N \in \mathcal{N}$. During preprocessing, elements are numbered such that we can assume $t(E_{i,j}) \leq t(E_{i,k})$ for $j \leq k$. If $l_{i,j} > t(E_{i,j})$ for an element $E_{i,j}$, there a more triangles assigned to $E_{i,j}$ than available. These $l_{i,j} - t(E_{i,j})$ additional triangles can be distributed on elements, which have not been visited yet. This happens especially to elements with few successors, i. e. to elements at the beginning of the list. Therefore, we render these elements first and redistribute remaining triangles to elements visited later. We define additional weights $\tilde{w}_{ijk}$ to assign remaining triangles of element $E_{i,k}$ to $E_{i,j}$:

$$\tilde{w}_{ijk} := \frac{A(E_{i,j}) \,/\, \| V - Z(E_{i,j}) \|^2}{\sum_{l=k+1}^{n_i} A(E_{i,l}) \,/\, \| V - Z(E_{i,l}) \|^2}$$

Then each element $E_{i,j}$ is assigned $l_{i,j}$ triangles with

$$l_{i,j} := w_{ij} L_i + \sum_{k=1}^{j-1} \tilde{w}_{ijk} \max\{0, l_{i,k} - t(E_{i,k})\})$$

Object SEC–Trees were treated exactly the same way. Instead of objects, triangles $T_{i,j}$ with $t(T_{i,j}) = 1$ were considered. If a triangle $T_{i,j}$ is assigned a limit $l_{i,j} \geq 1$, it is drawn.

Errors due to rounding can be considered. A simple example is a situation of a node containing to clusters of equal size and distance from the viewer, with a triangle limit of one. Then, each cluster receives a weight of 0.5, and therefore nothing is rendered. In general, only up to $\sum_{j=1}^{n_i} \lfloor l_{i,j} \rfloor \leq L_i$ triangles are rendered. For a large value of n_i and small L_i this can be a significant difference. Therefore, values of $l_{i,j}$ are rounded to integers, such that no more than

L_i triangles are assigned to the totality of elements, but on the other hand as few triangles as possible are lost. This can be done by adding the truncated decimal places and round up the limit if the sum reaches one. Especially at lower levels of a SEC–Trees this becomes important. This is even more important for SEC–Trees created for single objects. In this case, triangle limits may turn to the value of one, hence they will be displayed. At higher levels, one triangle more or less is not of such importance.

4.2.1 Integrating Frustum Culling

Now, let us assume that not the complete scene is visible, so we have to take frustum culling into account. Then, our rendering approach traverses the global SEC–Tree twice for every frame. We do not consider occlusions or backfacing geometry, but regard every object intersecting the view frustum as visible. Frustum culling is done to determine visible parts of the scene before the actual rendering takes place. We modify the weights to take the visible area into account. The new weights are proportional to the area of visible objects.

For every cluster or object in the global SEC–Tree, we have an axis aligned bounding box. We traverse the hierarchy and test every bounding box if it is contained in the view frustum, and stop the recursion if a node is not visible. We report the overall area of all objects contained in this node back to the predecessor. In the end, every node intersecting the frustum knows the visible area it contains. The area of an object has to be computed only once as a step of the preprocessing, since we do not consider partial visibility. The visible area of an object is either zero if it is located completely outside the view frustum, or the complete area of all primitives it contains.

Let $F \subseteq \mathcal{N} \cup \mathcal{J}$ denote those nodes and objects, which are at least partially inside the view frustum. We consider each object that intersects the view frustum to be completely contained. In fact, this is not true, but only a small portion of all objects is partially inside as well as outside, and checking the exact intersection of these few cases is too expansive. Further, we denote the elements of an SEC–Tree node N inside the view frustum by $e_F(N) := e(N) \cap F$. The visible area of N is

$$A_F(N) := \sum_{J \in e_F(N)} A(J) \text{ for } N \in \mathcal{N} \text{ and}$$

$$A_F(J) := A(J) \text{ for } J \in \mathcal{J} \cap F \text{ and } A_F(J) := 0 \text{ for } J \in \mathcal{J} \setminus F.$$

Now, weights used to distribute triangles on elements take this visible area into account. We define

$$w_{ij}, \tilde{w}_{ijk} := 0, \text{ for } E_{i,j} \notin F, \text{otherwise:}$$

$$w_{ij} := \frac{A_F(E_{i,j}) \,/\, \| V - Z(E_{i,j}) \|^2}{\sum_{k=1}^{n_i} A_F(E_{i,k}) \,/\, \| V - Z(E_{i,k}) \|^2}$$

$$\tilde{w}_{ijk} := \frac{A_F(E_{i,j}) \,/\, \| V - Z(E_{i,j}) \|^2}{\sum_{l=k+1}^{n_i} A_F(E_{i,l}) \,/\, \| V - Z(E_{i,l}) \|^2}$$

For each frame rendered, the SEC–Tree is traversed twice. The first time only the global SEC–Tree is considered. The values $A_F(N_i)$ are view dependent, and therefore have to be determined every frame. Because of their recursive definition, computation must proceed bottom-up, and triangle limits can not be assigned while descending the tree. During the second traversal, the image is rendered. Now, the weights $w_{ij}, \tilde{w}_{ijk}$ and triangle limits can be calculated. With these alternative definitions, the global SEC–Tree is traversed and nodes and objects are assigned weights analogous to the previous case of rendering without frustum culling.

4.3 From the Randomized Sample Tree to the SEC–Tree

Now, that we have seen how SEC–Trees can be used for rendering static scenes, we will show for a better classification how rendering SEC–Trees is related to the Randomized Sample Tree [KKF+02]. Therefore, we will integrate a rendering budget into the Sample Tree and look at arising problems and possible solutions. The basic approach of the Randomized Sample Tree has already been described in Section 2.2.2. The Sample Tree and SEC–Tree share the basic concept of selecting a subset the overall geometry and rendering only this portion. However, there are differences, which we will discuss in the following.

Rendering large scenes is essentially about balancing. On the one hand, visualization should be smooth, and therefore as the rendering has to be fast, on the other hand image quality should be high. The Randomized Sample Tree and the SEC–Tree prioritize different aspects in this regard. This motivates the main difference between these two approaches, which is the number of rendered triangles. The Randomized Sample Tree chooses a subset, which is sufficient large for being representative for the complete geometry, thus rendering correct images with high probability. While this reduces the number of rendered primitives significantly compared to the set of all triangles, the selected subset may still be too large for smooth rendering. We will show an example for this in our practical results (Section 6.5.1). In contrast, only a constant number of triangles are selected from the SEC–Tree. Therefore, errors in the image might occur, while the frame rate is preserved.

There are some further differences. For once, triangles are chosen deterministically from the SEC–Tree. Then, the underlying data structure is another difference, as the Randomized Sample Tree uses an octree. The last main difference concerns object identities. These are preserved by the SEC–Tree by building two separate hierarchies, the first on triangle and the second on object level. The Randomized Sample Tree, in contrast, considers triangles in the scene without differentiating which object they belong to.

Now, assume we wanted to modify the Randomized Sample Tree, such that the frame rate is preserved for rendering by applying a triangle budget, as we did with the SEC–Tree. Then, a constant number had to be chosen. The obvious modification of the approach is selecting these triangles at random. While this subset would no longer be representative, this method allows smooth rendering at any position of the viewer. However, the question how to select triangles rises. We will discuss one possible approach.

The Randomized Sample Tree chooses a triangle T from an octree node N with probability $\frac{A(T)}{A(N)}$, whereby $A\left(N\right)$ denoted the summed area of all triangles contained in node N or one of its subtrees. Note that this does not depend on the viewer's position. This is not critical for the Randomized Sample Tree, as octree nodes of a projected size larger than one pixel are always rendered completely, and sampling influenced only nodes that are entirely in the distance. Therefore, different distances of triangles within one node are not considered. Choosing a subset of constant size at random, this selection should depend on the viewer's position, as we sample in the foreground as well. This can be done by estimating the projected size of triangles and nodes by integrating their distances to the viewer.

Since the projected area of a triangle decreases quadratically with its distance to the viewer, we adjust the probabilities accordingly. However, considering the distance from every triangle of the subtree is expansive. Therefore, we a hierarchical approach is recommendable. So, let $N_1, \ldots, N_8$ be the eight children of octree node N, and $Z(N_i)$ their center. Then, the probability of triangle T would be

$$P(T) = \frac{A(T) / ||V - Z(T)||^2}{\sum_{R \in N} A(R) / ||V - Z(R)||^2 + \sum_{i=1}^{8} A(N_i) / ||V - Z(N_i)||^2}, \tag{4.2}$$

whereas $R \in N$ denotes all triangles stored in N without considering triangles contained in subnodes. The probability $P(N_i)$ of node N_i is defined analogous.

Still, this approach has its drawbacks. One problem is that the same node might be selected more often than the number of triangles it contains. As an example, let us consider a scene containing a plant near to viewer. Walls or the floor of this plant consist of only few triangles with a large area, thus obtaining a high probability. This is desirable, as these triangles have a large impact on the rendered image. But if this probability becomes too high, the corresponding octree nodes might be selected more often than the number of triangles they contain. This could be avoided by adjusting the probabilities of a node. Every time a triangle T is selected from N, the probabilities of all preceding nodes $N_1, ldots, n_8$ could be adjusted, such that they no longer consider the area of T. However, this has the drawback of altering the probabilities of every triangle and node each time a triangle has been chosen, and therefore induces a significant computational overhead. Further, it is not clear how probabilities have to be updated, as simple obvious strategies may induce an imbalance. As an alternative, we could discard a chosen triangle every time it has been selected before, and repeat this process until a new triangle is encountered. However, in the presence of triangles with large probability multiple tries might be necessary until a new triangle is found, and this approach favors small triangles disproportionally high [KV07].

Note, that the original Sample Tree defined probabilities only once independent of the user's position. Therefore, the sampling process could be shifted into the preprocessing, thus having no influence on rendering times. Now, we have to adjust the probabilities for every rendered triangle, based on the position of the viewer in the scene and the triangles selected before. These problems can be avoided using a deterministic approach.

The probabilities defined in Formula 4.2 are similar to the weights defined for the SEC–Tree rendering in Formula 4.1. The difference between the weights and the probabilities is that the weights consider only subtrees, while the probability considers subnodes and triangles in stored in node N as well. The reason for this is that triangles are chosen from a SEC–Tree node first and only a remaining budget is assigned to subnodes, while the randomized approach selects from triangles and subnodes as well. As a consequence, a triangle T is selected with the same probability as a subtree with all triangles combined of equal area as T if they are the same distance from the viewer. An algorithm selecting a representative subset, the Randomized Sample Tree as an example, should show exactly this behavior. However, if only a constant number of primitives are chosen, we can not expect a representative subset, and thus selecting those triangles with the largest impact is preferable. Thus, we alter the randomized algorithm, such that triangles are rendered first. If all triangles have been rendered, a subnode N_i is chosen at random with probability of

$$P\left(N_i\right) = \frac{A\left(N_i\right)/||V - Z\left(N_i\right)||^2}{\sum_{j=1}^{8} A\left(N_j\right)/||V - Z\left(N_j\right)||^2},$$

Now, the probability corresponds exactly to the weights defined for rendering SEC–Trees. Consider a node N containing t triangles, with N being chosen k times. Next, assume $k > t$. Then, each node N_i is chosen $P\left(N_i\right)\left(k - t\right)$ times in expectation. Now, if we assign N_i this number of triangles deterministically, we can check if this value exceeds the number of triangles contained in the complete subtree and assign those waste primitives to other nodes.

What we have just described is exactly how our rendering algorithm works, with the difference of using an octree instead of a SEC–Tree. However, using a SEC–Tree has advantages for this approach, as our practical results will show (Section 6.5.2). One reasons for this is that SEC–Trees are more adaptive to the distribution of geometry in the scene than octrees with their fixed center point. Further, octrees store triangles with small area but large span on higher levels, though they contribute little to the final image. In contrast, the SEC–Tree organizes triangles by size, such that these triangles are stored at deeper levels.

Summarizing, the Randomized Sample Tree works well as it renders the geometry of large projected size entirely, and chooses a representative sample set of smaller geometry. However, selecting from the larger triangles at random as well with a fixed triangle budget induces some difficulties, which are avoided by our deterministic approach.

4.4 Weighting Objects to Assign Triangle Budgets in Dynamic Scenes

Next, we will consider rendering dynamic scenes with SEC–Trees. We organize these scenes in a list without any hierarchy. While organizing objects in SEC–Trees allows efficient rendering of scenes consisting of thousands of objects, this is only possible in static scenes, since SEC–Trees do not support dynamic updates, while on the other hand lists allow inserting or deleting objects in constant time and do not rely on spatial information, allowing fully dynamic scenes. However, this comes with the costs of needing to touch each object for every frame.

We assume that scenes are given by a set of objects $J_1, \ldots, J_n$. Like before, a frame is rendered in two phases. First, the list of objects is traversed. The positions of moving objects are updated and each object is assigned a number of triangles. During the second phase, a number of triangles corresponding to the object's budget is selected and actually rendered. This decision, which triangles of an object are considered, is made by utilizing a SEC–Tree as explained in Section 4.1.

In phase 1 of rendering a frame, the triangle budget L is distributed on the objects. Therefore, each object J_i is assigned a weight w_i and w_iL triangles will be selected from this object.

Objects near to the viewer with few triangles may be assigned more triangles than they contain. In this case, the spare triangles are redistributed on the other objects. Within one node of a global SEC–Tree, objects were sorted by their number of triangles. We avoid this in dynamic scenes. Since objects containing fewer triangles are no longer necessarily rendered before objects with more triangles, we do not distribute waste triangles on following objects only. Instead, the complete list is traversed a second time. The weights are modified, such that objects are not considered if all their triangles have been selected. Triangles wasted during the first pass are redistributed and assigned in addition to triangles previously distributed. This can be repeated multiple times if there are still wasted triangles. However, most of the time we needed two and never more than three passes to assign all triangle limits.

Analogous to our approach on static scenes, weighting of objects to determine their contingent of triangles considers their importance to the final image.

This is identified by an estimation of the area an object covers on the screen after passing the viewing pipeline. The area a triangle covers on the screen increases with its actual area. As before, if object J_i intersects the view frustum, we define its size $A_F(J_i)$ by the summed area of all triangles J_i contains. If on the other hand J_i is located outside the frustum, we define $A_F(J_i) := 0$. We denote the viewer's position by V and the center of object O_i by $Z(O_i)$. Since the area a triangle covers on the screen decreases proportional to the squared distance from the viewer, we consider this factor and each object is assigned the weight

$$w_i = \frac{A_F(J_i)/||V - Z(J_i)||^2}{\sum_{j=1}^{n} A_F(J_j)/||V - Z(J_j)||^2}$$

Dynamic scenes with thousands of moving objects are supported. Since objects are organized as a list of objects and no additional data structure is necessary, moving, inserting and deleting objects only causes costs linear in the number of modified objects.

Assignment of weights does not rely on any kind of coherence. If an object far away becomes important, the user might be interested in instantly examining what has happened at that location, so he just teleports to that place. The view on the scene can suddenly change completely. These events are not known beforehand and can not be predicted. Since weights depending on the current position are assigned every frame independently from previous values, such knowledge is not needed and our approach can handle these situations, as long as scenes are stored completely in main memory, while out-of-core rendering might require to load geometry which is to be rendered from the new viewpoint. We will describe out-of-core rendering in detail in the following section.

4.5 Out-Of-Core Rendering

Large scenes rise an additional challenge to rendering at sufficient frame rates, which is memory consumption. Even though the number of triangles which can be stored in main memory tops the number that can be rendered in realtime, the size of the complete scene may still be exceeding memory capacities by far.

The easiest solution is using instantiation. If a scene contains several objects of with identical geometrical representation, e. g. several forklifts of the same type, it is sufficient to store the geometry only once. The triangles are given in a local coordinate system and each instance of an object type has the transformation information available, which describes how to position the geometry in world coordinates. While this approach allows to keep large scenes in main memory, it is limited to situations where there are only few different object types which are replicated multiple times.

The more general solution is using an out-of-core algorithm. Only a fraction of the complete scene is kept in main memory, while the majority is stored on external memory and loaded as it is needed. Our rendering approach disregards the entire scene for rendering, except for a constant number of triangles. Therefore, only this amount of geometry is needed at a time. However, in order to avoid latencies arising from loading geometry, it is recommendable to prefetch geometry which may be needed within the next frames and keep it in main memory as well.

We introduce an additional constraint to rendering the SEC–Tree of an object, which allows discarding the majority of triangles and only depends on the distance of this object to the viewer, but is not influenced by the position of the other objects. If all triangles of subtree become too thin, we regard their impact on the final image as negligible, and the weight of the subtree's root node is set to 0. Triangles are considered to be thin if their height after projecting them to screen coordinates is below some threshold.

In order to determine if a subtree below a node N is ignored, we look at the heights of all contained triangles. Let $N_1, \ldots, N_{n_N}$ be the subnodes and $T_1, \ldots, T_{m_N}$ the triangles contained in node N. Then, let h_1^i, h_2^i, h_3^i be the heights of triangle T_i. We define $h_i := \min_{j=1,2,3} h_j^i$ to be the minimal height of T_i and $H_N := \max\{h_i, H_j | 1 \leq i \leq m_N, 1 \leq j \leq n_N\}$ the largest minimal height of all triangles contained in the subtree beneath N.

For a node N, a distance of d between the viewer and the bounding box of N indicates the minimal distance of all triangles contained in N. Knowing the height of a triangle, an estimation on the projected height can be given. Depending on the shape of pixels, we look at the width or the height of the image. If pixels are quadratic, it is irrelevant which dimension we choose, else we consider the direction in which pixels are shorter. W.l.o.g. let this be the height, otherwise the examination is analogous. Now, let p be the height of the rendered image in pixels and α the vertical opening angle of the view frustum. Consider a triangle orientated parallel to the projection plane. The height of the view frustum at a distance of at least d is never smaller than $2d\sin\frac{\alpha}{2}$ (see Figure 4.4). Therefore, the fraction of the viewing plane covered by the projection of the triangle in this dimension is at most $\frac{H_N}{2d\sin\frac{\alpha}{2}}$. Since the height is p pixels, $\lceil\frac{p \cdot H_N}{2d \cdot \sin\frac{\alpha}{2}}\rceil$ is an upper bound on the projected size of the triangle in at least one dimension. N is discarded for rendering if this value falls below some threshold.

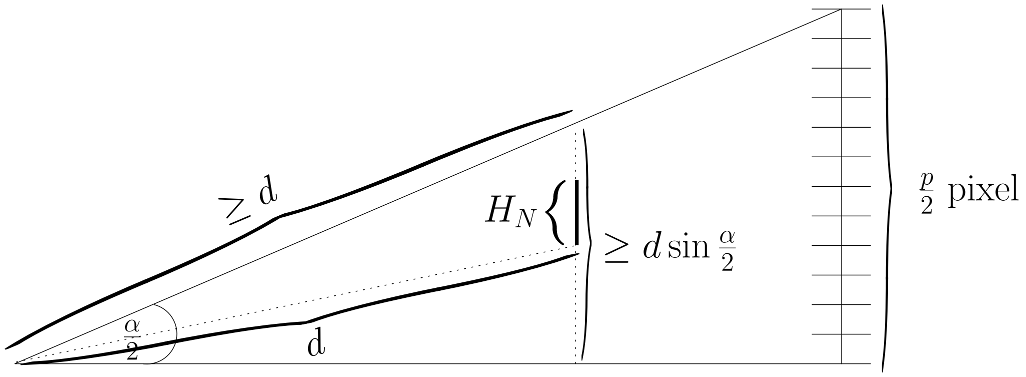


Figure 4.4: The upper half of the view frustum

Knowing beforehand which nodes will necessarily be rendered the next frames, we will store only such nodes in main memory. If object O_i is positioned at a point P_i with a bounding sphere radius of r_i, and the user is located at position V, the distance from the viewer to any triangle of the object is at least $||V - P_i|| - r_i$. If the user is moving with a maximal speed of t_V, the distance to O_i within the next t steps can not become smaller than $\max\{||V - P_i|| - r_i - t \cdot t_V, 0\}$, as long as O_i does not move. Hence, if

$$\frac{pH_N}{2\max\{||V - P_i|| - r_i - t \cdot t_V, 0\}\sin\frac{\alpha}{2}} \tag{4.3}$$

becomes larger than our threshold, it is possible that we have to render node N within the next t steps, and therefore load N into main memory.

In scenes with arbitrary dynamics, there are no limitations on the changes within the transition to the next frame. Since every object could enter or leave the view frustum at any distance to the viewer, we need to restrict possible movements in order to determine reasonable bounds on the weights. Therefore, we assume that movement speeds of objects and the viewer are limited.

In addition, we do not regard insertions of objects for prefetching, since objects can be created at any time, and we can not predict if a simulation introduces new objects. New goods, which are produced by a machine, are one example of such objects. Thus, necessary geometry of new objects is loaded as soon as they are created.

In scenes containing moving objects we have to adjust the estimation of projected size, considering possible distances within future frames. Let the speed of object J_i be limited by t_i. Then, the minimal distance within the next t frames is $\max\{||V - P_i|| - r_i - t \cdot (t_V + t_i)\,, 0\}$. Updating the expressions in Formula 4.3 accordingly ensures, that nodes are loaded in time if projection size will not top the threshold within t steps.

4.6 Generating Multi-Point Level of Detail

We assume that information about significant objects is given by an external application to our rendering system, e.g. a material flow simulation which evaluates the processes it simulates. Such an application might identify overflowing depots or broken machines, which require special attention of the user. Our rendering system can increase the visual details at such locations. We call this multi-point level of detail, since details are not simply depending on the distance to viewer, as in classical LOD apporaches, but also on points of special interest.

Multi-point level of detail can be realized by modifying the weights of objects according to their significance. The simulation identifies points of interests and assigns a significance value $s\left(J_i\right)$ to each object. This value varies between 0 and 1, for objects which are not significant $s\left(J_i\right)$ is equal to zero.

We utilize three approaches to realize multi-point level of detail. The first one redistributes triangles in favor of significant objects, the second one allows exceeding the triangle budget, and the third possibility is to increase details at regions around significant objects and not only on these objects themselves.

4.6.1 Emphasizing Significant Objects

Rendering SEC–Trees allows increasing details at significant objects by assigning additional triangles. In this section, we will present two approaches of modifying wheights, differing by their concerns on the triangle budget. The first approach respects the triangle budget and redistributes triangles on significant objects, taking them away from less important objects. The second approach allows exceeding the triangle budget, and significant objects receive additional triangles, while other objects remain unaltered. This allows choosing if the current frame rate will be maintained or the degree of detail on all objects is preserved.

In addition to automatically changing details of significant objects, a semi-automatic control is available. By altering a parameter r, the user can also influence the appearance of an object by assigning more or less triangles to it. The integration of significance values is achieved by altering weights w_i. We add a factor $1 + r \cdot s\left(J_i\right)$ and adjust the denominator accordingly.

In our first approach, we redistribute triangles on significant objects. Weights of such objects are increased, while at the same time weights of other objects are decreased by modifying the weights as follows:

$$w_i = \frac{\left(1 + r \cdot s\left(J_i\right)\right) \cdot A_F\left(J_i\right) / ||V - Z\left(J_i\right)||^2}{\sum_{j=1}^{n} \left(1 + r \cdot s\left(J_j\right)\right) \cdot A_F\left(J_j\right) / ||V - Z\left(J_j\right)||^2}$$

On non significant objects, this increases the denumerator, while the numerator is untouched, decreasing the weight if significant objects are present. For significant objects, the numerator is enlarged, corresponding to their degree importance. Thus, objects with increased significance attract triangles from other objects. This way we achieve a more detailed representation of significant objects without influencing rendering performance.

Alternatively, we allow significant objects to exceed the triangle budget. Now, the significance of objects is ignored when normalizing weights, so we get:

$$w_i = \frac{\left(1 + r \cdot s\left(J_i\right)\right) \cdot A_F\left(J_i\right) / ||V - Z\left(J_i\right)||^2}{\sum_{j=1}^{n} A_F\left(J_j\right) / ||V - Z\left(J_j\right)||^2}$$

In this case objects get additional triangles if their significance increases. For objects O_i, which are not significant, the value of $s\left(J_i\right)$ is equal to zero, therefore they obtain the same weight they would get in a scene without significant objects, thus all details are preserved. However, this approach might decrease rendering performance, as more triangles are rendered than the original budget allowed.

4.6.2 Emphasizing Regions Surrounding Significant Objects

Instead of enhancing details of single significant objects, the details can be increased for regions surrounding significant objects. This can be helpful if not only the object is to be examined, but if the user wants to recognize correlations of events resulting in the significance of this object.

Given $k-1$ significant objects $J_{l_2}, \dots J_{l_k}$ we denote their centers by $P_i := Z\left(J_{l_i}\right)$. In addition, we consider the viewpoint $P_1 := V$. For each object J_i, we look at its quadratic distances $||P_h - Z\left(J_i\right)||^2$ to all significant objects. However, if objects are far away, the user can not recognize details anyway. Even if objects are very near to significant events, there is no point in giving them the same degree of detail objects near to the viewer receive. Therefore, the influence of a significant object J_{l_h} is weighted by its distance to the viewer $||P_h - V||$.

The higher the significance of an object is, the more important this area becomes, and therefore the significance value $s\left(J_{l_h}\right)$ influences the weight. Now, the weights are defined by:

$$w_i = \frac{A_F\left(J_i\right) \sum_{h=1}^{k} s\left(J_{l_h}\right) / \left(||P_h - Z\left(J_i\right)||^2 \cdot ||P_h - V||\right)}{\sum_{j=1}^{n} A_F\left(J_j\right) \sum_{h=1}^{k} s\left(J_{l_h}\right) / \left(||P_h - Z\left(J_j\right)||^2 \cdot ||P_h - V||\right)}$$

whereby $s\left(J_{l_1}\right) := 1 + \sum_{j=2}^{k} s\left(J_{l_h}\right)$ represents the importance of the viewer's position and is larger than the combined significance of all objects, assuring that the viewpoint is still more important than objects in the scene.

At all points, a minimal distance of 1 is assumed, even when regarding the distance between identical points, to avoid undefined expressions. So, more exactly we do not just consider the distances, but the maximum of the distance and one, but left it out of the formula for reasons of clarity.

CHAPTER 5
IMPLEMENTATION DETAILS

The rendering system has been implemented using Microsoft Visual C++ and openGL. Since C++ is an object oriented programming language, the use of the word object can become ambiguous in this chapter. On the one hand, objects are objects of the simulated world, for example machines or forklifts, on the other hand, object can refer to C++ objects. If necessary, we will refer to objects of the simulated world as world objects, however most of the times this differentiation is nonessential, since each world object is represented by exactly one C++ object.

5.1 Preprocessing

For each 3D file, which has to be in *vrml 97* or *obj* format, a SEC–Tree is created in a preprocessing step. This is also done for the global SEC–Tree if a scene is static. A scene is described in an xml-file, defining the positions of objects. An xml scheme regulates how this xml file has to be composed.

In contrast to our analysis, we did not use skip octrees in the implementation. They were an adequate tool to consider costs, but are not necessary in practice. Instead, we used a kd tree for range queries during the clustering phase. While k-trees have worst case costs of $O\left(n^{\frac{2}{3}} + r\right)$ for a rectangular range queries reporting r points, they behave fairly good in practice and are easier to implement. Further, we organized groups during an arranging phase only in a search tree as originally proposed, instead of using an AVL tree, which would guaranty a balanced tree. In practice, we did not find more than 20 groups during an arranging phase, and therefore a more sophisticated data structure proposes only overhead in the implementation.

While our rendering algorithm performs out-of-core rendering, our preprocessing is only an in-core approach. Since we create SEC–Trees on a per object basis, we only need the geometry of one object at a time. An object has been modeled using a conventional CAD application, which typically does not support out-of-core representations. Therefore, we can assume that a single object fits into main memory. Similarly, a global SEC–Tree only depends on object positions, but not on actual geometry, and therefore we can assume that all necessary points can be stored in memory. Since a SEC–Tree has to be created only once and is stored on hard drive afterwards, hardware demands higher than those of the rendering algorithm are acceptable, and memory constraints become less important.

5.2 Triangle Centers

When performing clustering, while constructing the SEC–Tree of an object, a triangle's center is used as the regarded point of the clustering algorithm. A first choice could be the center of gravity. This center is contained in the triangle and easy to compute. If the triangle is equilateral, this is a adequate point. However, in cases of acute-angled triangles, this point is no longer appropriate.

Consider a cylinder as an example. Its superficies surface is approximated by a set of triangles. If we look at their centers of gravity, we get two clusters, while the triangles constitute a continous area, which should be one cluster. Figure 5.1 shows an example of this situation.

The solution is to use a weighted average of triangle points. Each point P_i of a triangle T is weighted by $w_i\left(T\right)$, with is proportional to the point's distance from the triangle's centroid $z\left(T\right)$. This reduces the maximal distance from the center to the triangle's vertices.

More precisely, the following center point $Z\left(T\right)$ is used for a triangle T: Let

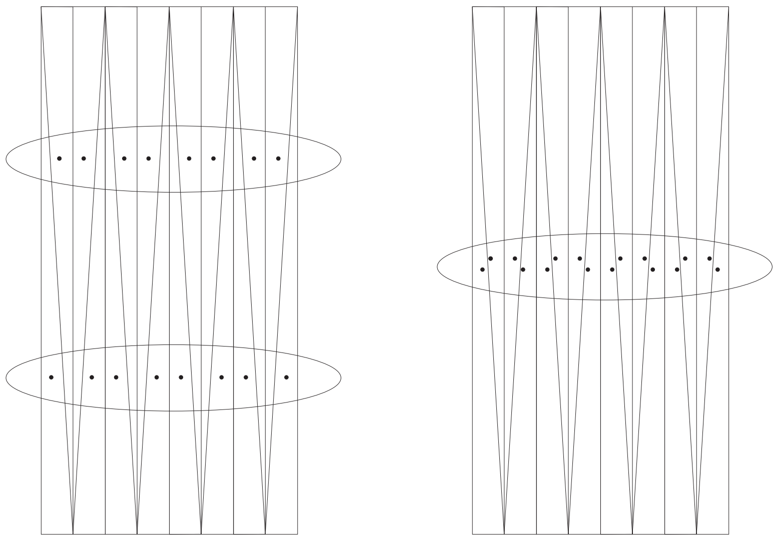


Figure 5.1: The triangles form two clusters if centers of gravity are clustered, which can be seen on the left example. The desirable result would be to obtain only one cluster. To achieve this, we have to obtain points located nearer to each other, like the points on the right.

$T = (P_1, P_2, P_3)$ be a triangle. Then, we define its center by

$$Z(T) := \sum_{i=1}^{3} w_i(T)\, P_i, \text{with } w_i(T) := \frac{||P_i - z(T)||}{\sum_{j=1}^{3} ||P_j - z(T)||}, \text{and } z(T) := \frac{1}{3}\sum_{k=1}^{3} P_k.$$

5.3 SEC–Tree Structure

Out-of-core algorithms require the data to be stored externally, such that it can easily be loaded at runtime. In a preprocessing step, a SEC-tree is created for each 3D object which might occur in a scene and is stored in a single file.

The structure defined as Algorithm 6 contains all information necessary for one node. At first, the number of triangles and subnodes are given, which are needed to determine the amount of data associated with this node. An array of triangles contains the actual geometry information. The second array contains instances of the *SubclusterData* structure, which covers the data concerning the subnodes.

Within each *SubclusterData* object, first an index states the position in the file where the data of a subnode starts. A bounding sphere is needed to bound the distance of the node to the viewer. In combination with the value H_N, it is possible to decide if the projection becomes large enough within the next steps, such that the node has to be loaded, as described in Section 4.5. The area $A_N := A(N)$ is needed to determine the weight node N is given for rendering.

Algorithm 6 SEC–Tree Node Structure on Hard drive

```
Class SectreeNode
{
        int numberOfTriangles
        int numberOfClusters
        Triangle[numberOfTriangles] tringles
        SubclusterData[numberOfClusters] subcluster
}
Class SubclusterData
{
        int clusterFileIndex
        float bounding spheresRadius
        Vector bounding spheresCenter
        float $H_N$
        float $A_N$
}
```

Subnodes of a given node N constitute a cluster, therefore their data should be located near to earch other in a file. Since they are of similar size, they should obtain similar weights, besides a few exceptions near to the viewer, where small changes in the distance can have a larger impact on weights. As a result, if one subnode of N has to be loaded, most of the times this holds for the other subnodes of N as well. Therefore, the SEC–Tree is traversed in a breadth first search when storing the tree. This allows loading a complete sublevel at once, instead of getting every node for itself, as all nodes are stored consecutive in the file. For further increased loading speed, the representation on hard drive is binary identical to the representation in memory. This way, data has not to be converted after it has been read, but can be used instantly.

5.4 System Architecture

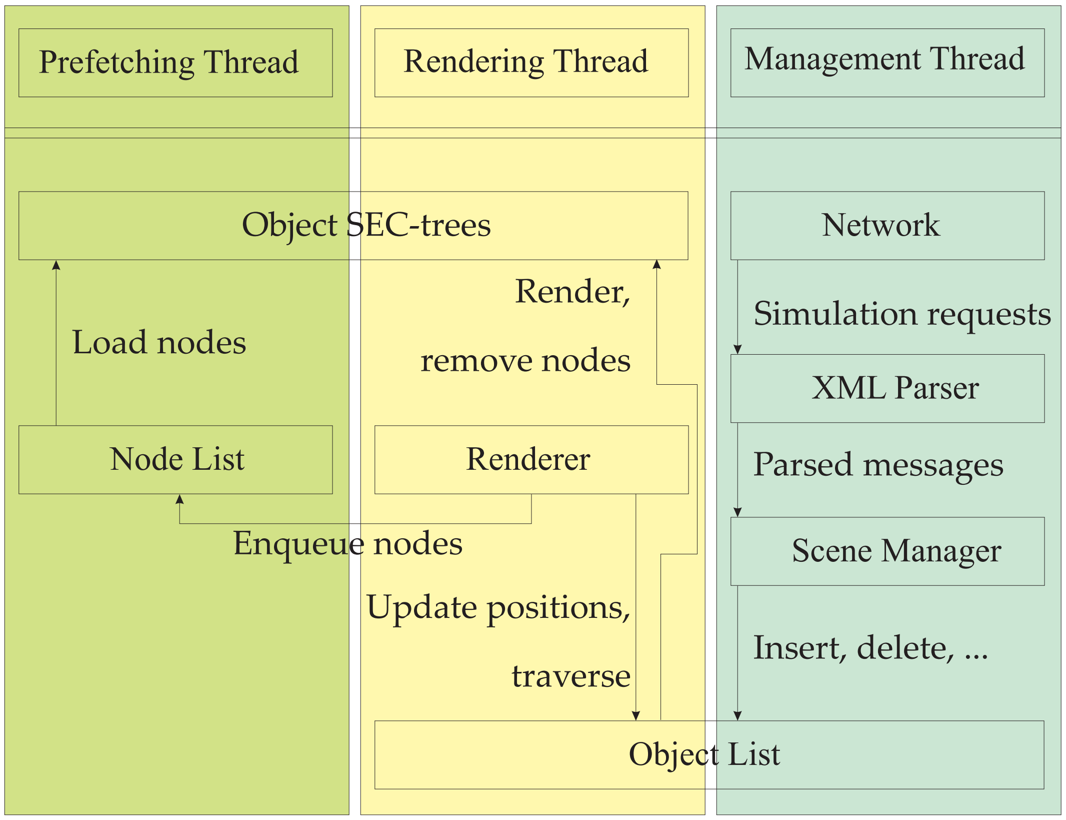


Figure 5.2: Architecture of the rendering system

In dynamic scenes, the system consists of three threads. The first thread is responsible for prefetching geometry into main memory, the second thread renders the scene, and the third thread manages the world objects of the scene. Static scenes do not allow changes at runtime, thus the managing thread becomes redundant.

Each world object is represented by exactly one C++ object stored either in a global SEC–Tree or in a list. This C++ object memorizes the position, movement speed and target position if the object is moving, a transformation matrix to project from object coordinates to world coordinates and the inverse matrix, a bounding box, the significance level and the object's name. The actual geometry is not stored with the object itself, instead a pointer refers to a geometry object holding the SEC–Tree with the triangles of the object in a local coordinate system. Only one geometry object is needed for each file with geometry information, which may be shared by multiple objects having the same geometrical representation.

5.4.1 The Management Thread

Objects can be manipulated by external applications, like a simulation as an example. The operations cover inserting, deleting and rotating objects, assigning a target position and speed for movements, and altering the significance level of an object. In order to able to communicate with arbitrary applications, commands are accepted via a network interface. These commands have to be composed in xml, specified by a scheme, and are interpreted by a xerces parser.

Each object can be identified by a string, which has to be included in every message. A hashmap allows to find the object targeted by the message and apply the appropriate changes to the object's paramters.

5.4.2 The Rendering Thread

In static scenes, the rendering thread traverses the global SEC–Tree performing frustum culling and assigning a weight to each object, while in dynamic scenes the object list is processed, additionally updating the positions of all moving objects as described in Sections 4.2 and 4.4. Then, the SEC–Trees of all objects, which are assigned at least one triangle, are traversed. This traversal selects the triangles from the tree according to the limit determined by the object's weight. The chosen triangles are sent to the graphics hardware, where they are rendered.

Some consideration has to be given on the calculation of weights. When determining a weight w_i, we devide by the distance of a node N_i to the viewer, which might be close or even equal to zero if the viewer's position is located near to the center of the node. On the other hand, we want our weights to represent an estimation on projected screen size. Since this screen size is limited, we also limite the minimal distance we consider, avoiding the problem of divisions by zero.

When rendering an object's SEC–Tree, each subnode referred to by a parent node visited during the traversal, which has not already been loaded into main memory, is checked if it may be needed within the next steps. If so, and the node has not been marked, the associated SubClusterData structure containing the necessary informations to load the node is inserted into the queue, together with a reference to this node. A marking indicates, that this subnode has already been queued and has not to be considered again.

We use a caching strategy for efficiency. Nodes in main memory, which gained no triangles, are checked for last time they have been considered to be possibly necessary. If at least ten frames have passed since then, we remove the node and subtree beneath from main memory. Waiting for some frames provides two advantages over immediately deleting nodes which are not needed. First, multiple instances of one object type might use the same SEC-tree. If a node is not necessary for one object, this might not hold for other instances. Ten frames without considering this node implies that this node was not possibly necessary for any other instance within at least the last nine frames. The second advantage is, that backward movements require the same nodes to be loaded as previous frames. Loading these same nodes, which have just been deleted, can be avoided if those are kept in main memory for some additional frames.

5.4.3 The Prefetching Thread

Synchronous prefetching of geometry within the rendering thread, when encountering nodes that have to be loaded, would stall the CPU, and therefore slow down the complete rendering. On the other hand these data are not required immediately. Since accessing the hard drive in parallel to the rendering is more efficient, loading is realized by an additional thread.

The prefetching thread processes the nodes inserted into the queue by the rendering thread. At first, we check if the node still has to be loaded or if the distance has increased due to movements of the object or the viewer during the time other nodes in the queue have been processed, such that it no longer has to be loaded.

After loading a node, its children are processed. Instead of inserting them into the queue, they are directly checked if loading them is requiered. If all children are necessary, the complete sublevel is loaded at once, otherwise only the required nodes are accessed. This continues recursively, until no further nodes are found, which might be rendered in the near future.

5.5 Coordinate Systems

We use instantiation, i.e. geometrical representations of objects are stored only once for each geometry file, instead of once for every world object in the scene. The obvious benefit is, that less memory is consumped if one 3D model represents more than one world object. Therefore, the geometrical data has to be independent of an object's actual position. Instead, a local coordinate system is used, with the center of the object's bounding box as the origin. For each object, a transformation matrix defines how to convert local coordinates into world coordinates.

An additional advantage is, that moving and rotating objects only requires to update the matrix, while the geometry itself is left unchanged. All matrices are stored as 4×4 matrices, which can be as well used for internal calculations as be pushed directly on the openGL model-view matrix stack. On the other hand, this results in a drawback against using world coordinates in the beginning, as every time the rendering system encounters a new object, the transformation matrix on the openGL stack has to be changed. However, the benefit prevails perspicuously.

Assigning weights to nodes of a SEC–Tree, while traversing it for rendering, includes calculating the distance between the viewer's position and the center of the node. Since the geometry is only available in object coordinates, this holds for the complete tree, while the viewers's position is given in world coordinates. In order to get comparable coordinates, one position has to be translated into the other coordinate system. Converting the node center into world coordinates would have to be done everytime a node is visited during the traversal, while translating the viewer's position into object coordinates using the inverse transformation matrix has to be done only once for every object, which is at least partially rendered. Since objects are only rotated and translated, distances between the transformed points do not depend on the chosen coordinate system.

CHAPTER 6
PRACTICAL RESULTS

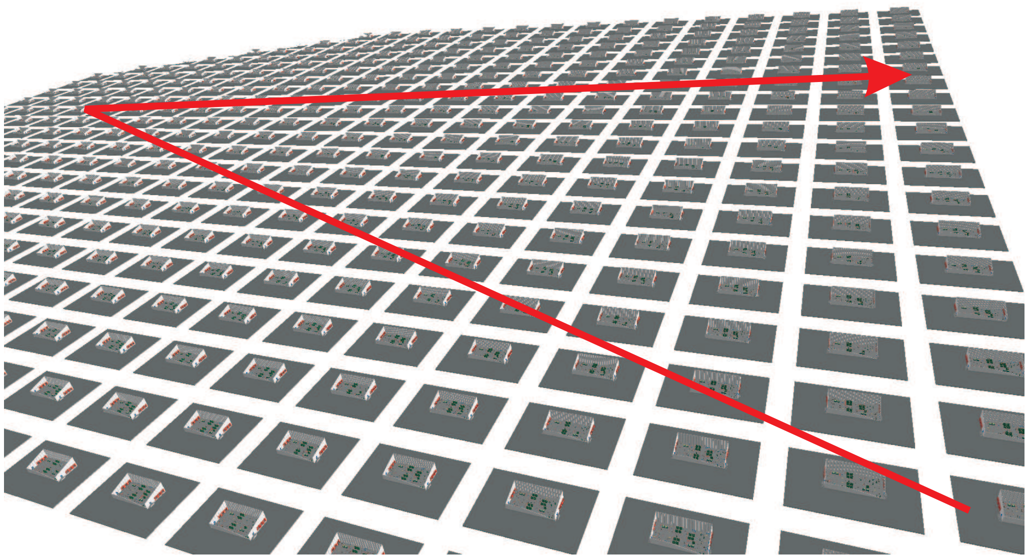
We tested our approach in a prototypical implementation in C++ without optimizing the code. OpenGL was used as low level rendering API. The test equipment was an AMD Athlon XP 2600+ computer with 1GB Ram and an Ati Radeon 9600 EZ graphics card, running windows XP.

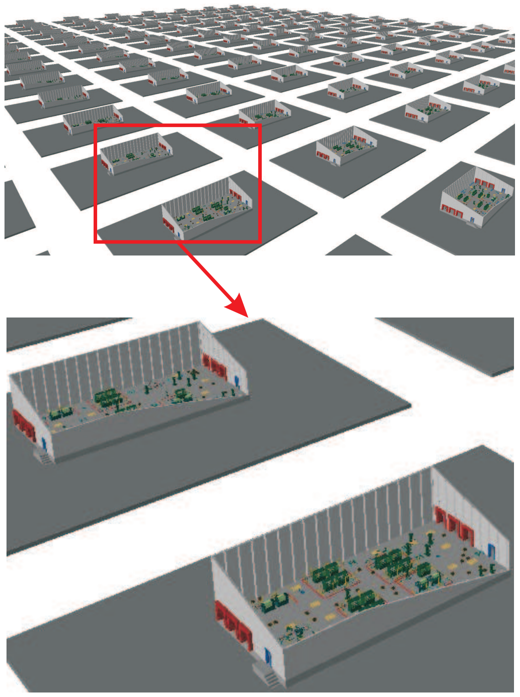
We tested static scenes as well as dynamic ones, looking at different properties like frame rates, memory consumption, pixel errors or the influence of emphasizing significant objects. Further, we compared SEC–Trees to an octree based rendering approach.

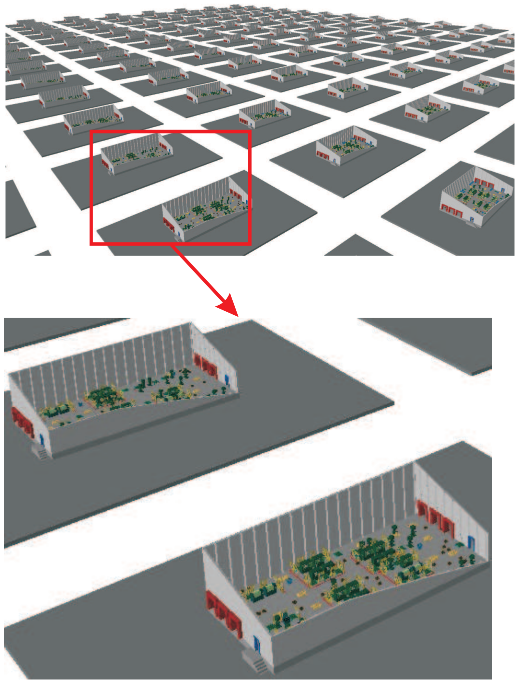
6.1 Rendering Static Scenes

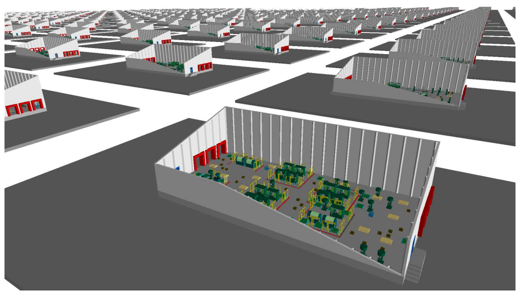
We used static test scenes of different sizes, all showing similar results. In the following, we will refer to the largest scene. This test scene consists of 23.880 objects with 180.371.756 triangles altogether and is pictured in Figure 6.1. Different kinds of objects were used like halls, machines, machine cells or boxes.

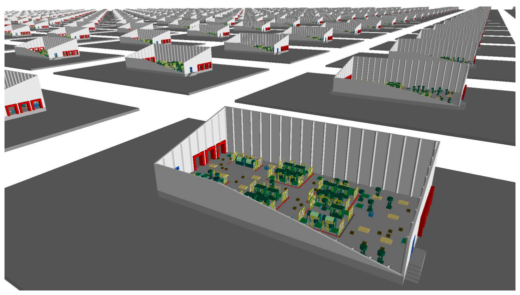
The number of triangles per object varied from a few hundred to several thousands. The rendering limit for each frame was set to 300.000 triangles. Preprocessing of this scene took about 12 minutes in total to obtain a global SEC–Tree as well as those for all different 3D models.

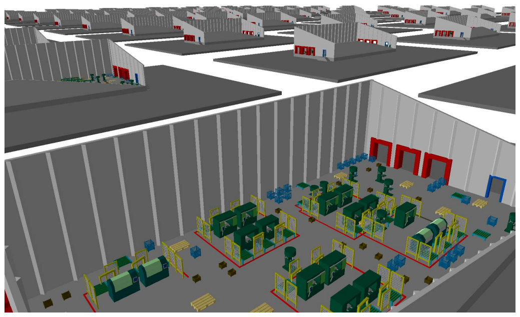

(a) Measurements were made following the indicated path

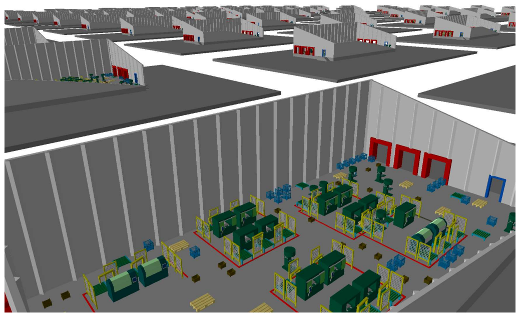

(b) This image was rendered using the SEC–Tree with a detail view on a part of the image below

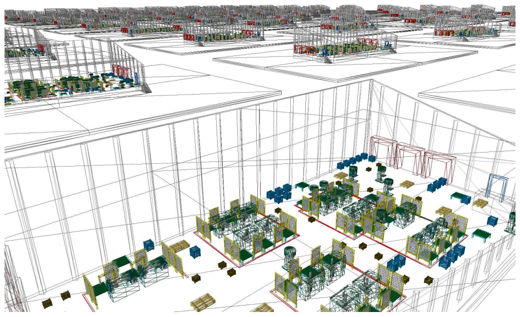

(c) All triangles are rendered from the same position as before. The detail view demonstrates image errors that occurred on the left when the SEC–Tree was used


(d) A further image rendered with a SEC–Tree . . .


(e) and the corresponding exact image with all triangles


(f) Now, the viewing position is closer to one of the plants. Again, the SEC–Tree is and used first . . .


(g) and the correct image is given for comparison


(h) This wireframe image indicates the complexity of the models

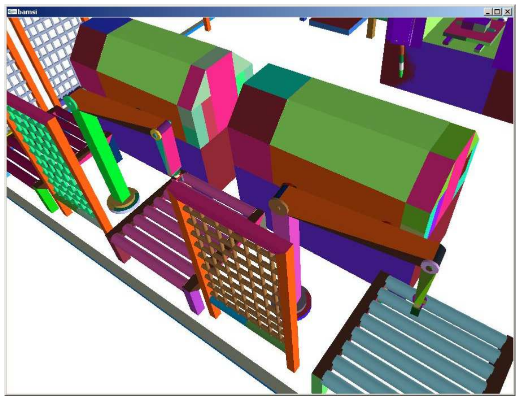

(i) Triangles in one cluster are drawn in the same color

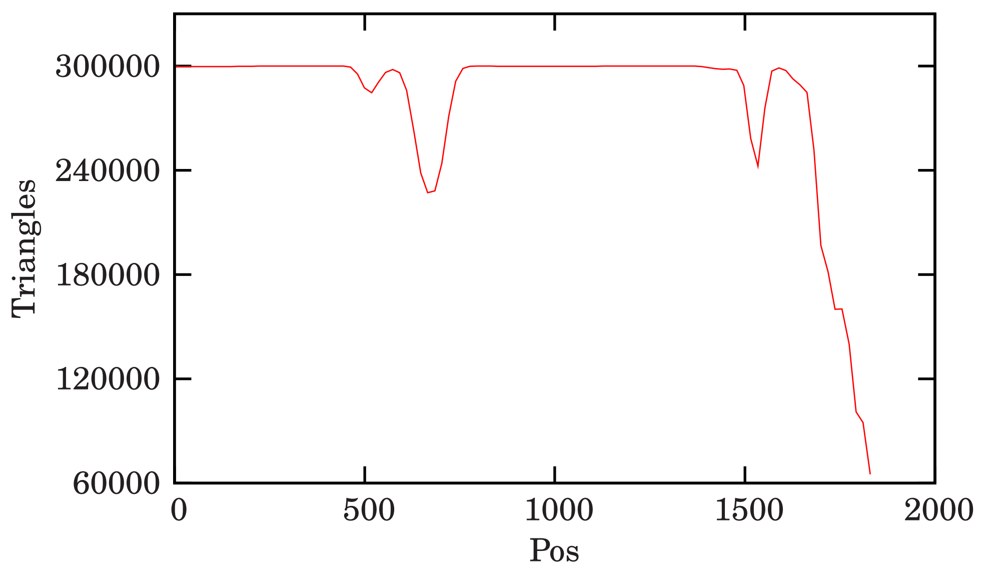
Figure 6.1: Screenshots taken from the static scene with about 23000 objects and 180 million triangles

Figure 6.1(a) was taken from a position which allows seeing most parts of the scene. There are about 20.000 objects visible, with only few objects being hidden by other objects. In such situations, approaches like occlusion culling are no solution. Reducing the complexity of single objects by a conventional level of detail algorithm will still result in a large number of triangles, simply because of the large number of objects. While most other approaches consider objects independently of each other, the SEC–Tree in contrast reduces the number of triangles for an object if it is increased for another one. This results in a balanced rendering system.

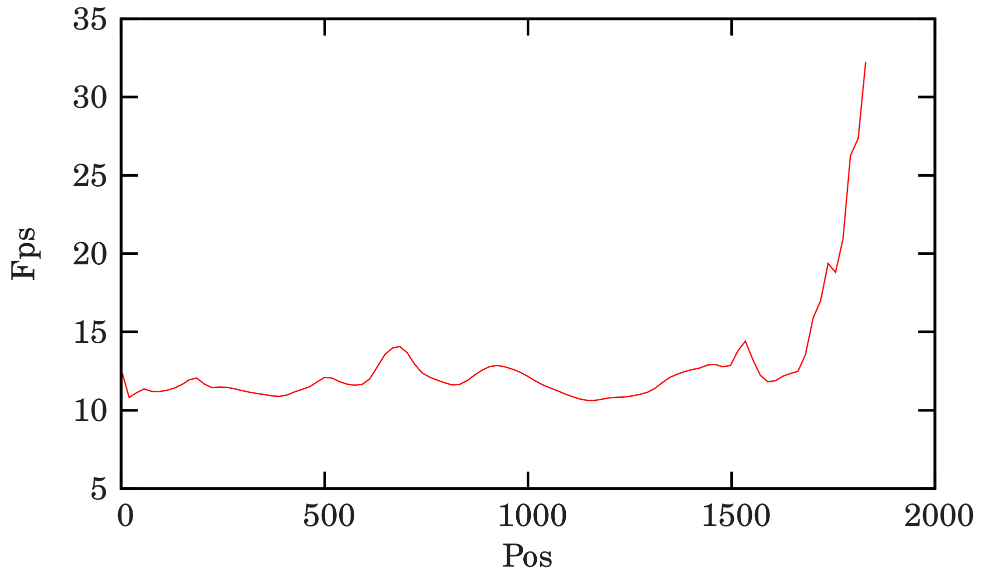
A more detailed view on a single hall can be seen in Figure 6.1(e). The wireframe in Figure 6.1(g) demonstrates the complexity of single objects. The clustering of an object can be seen in Figure 6.1(i). Triangles in one cluster are drawn in the same color, visualizing leaves nodes of the SEC–Tree. The lattices and rolls in the foreground are examples, where one can see that triangles of equal size are inserted into groups of neighboring primitives, reconstructing components of the object.

6.1.1 Rendering Statistics

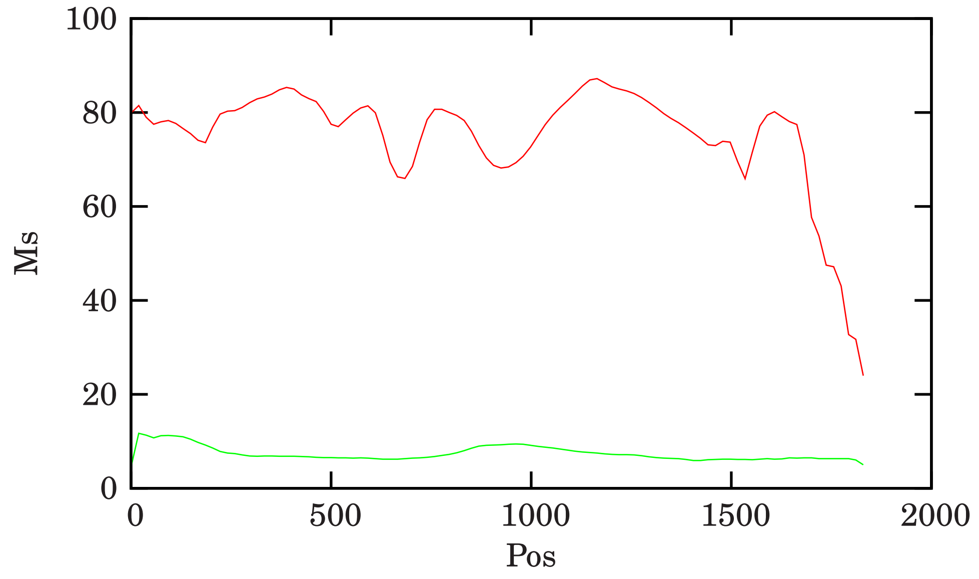
Figure 6.2 shows some statistics on rendering, following the sample path indicated in Figure 6.1(a). Measuring data at about 2000 positions, rendering time was almost constant. The visibility changed from views showing nearly the complete scene to others, such that only a few objects were visible. Figure 6.2(a) shows the number of rendered triangles at each position. The limit of 300.000 triangles was nearly exceeded at every position, except for those, where the number of triangles contained in the view frustum was lower than 300.000. The frame rate was about 12 fps, even when the complete scene was visible (Figure 6.2(b)). Only at viewing positions near the boundary of the scene rendering was faster, since the number of triangles inside the view frustum was less than the triangle limit. Figure 6.2(d) shows this number of triangles inside the view frustum for every sample position. The number of visible objects results in nearly the same diagram except for the scale on y-axis, and therefore is not shown. Independent of the number of visible objects and triangles, the frame rate never became too slow to interact with the scene. In contrast, rendering an image of the complete scene without limiting the number of triangles by sending every triangle to the graphics hardware took more than 30 seconds for one frame.



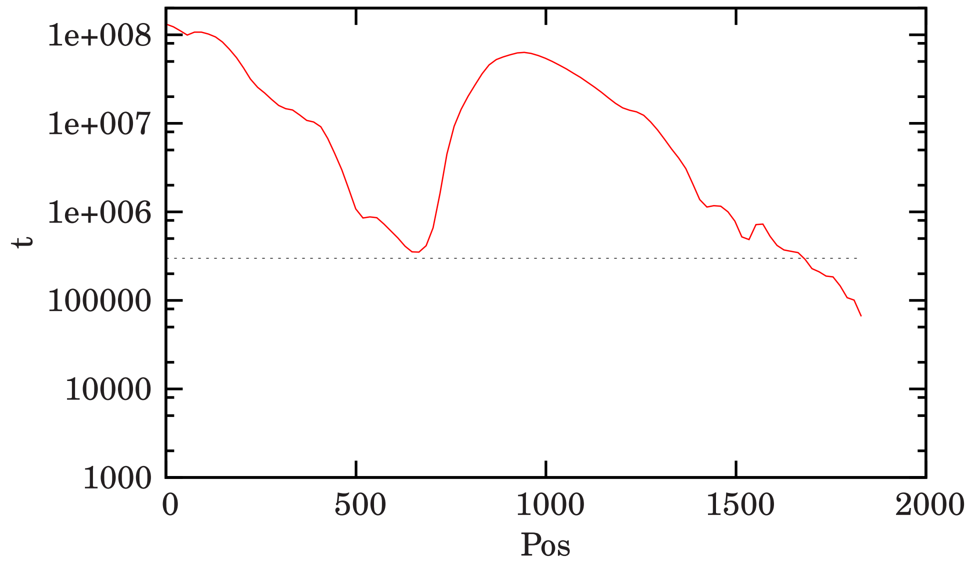
(a) The number of rendered triangles



(b) Frames per second if rendered constantly with current speed



(c) Milliseconds needed for the two rendering phases of frustum culling (green line) and rendering according to weights (red line)



(d) Number of triangles t inside the view frustum (red) and the triangle limit (dotted line). Note that the y-axis is scaled logarithmically

Figure 6.2: Rendering statistics have been collected along a sample path through the scene. The diagrams above show the results for each of about 2000 frames

The time needed to render one frame was dominated by the second traversal of the data structure, where weights are assigned and triangles are rendered. Even if all objects were inside the view frustum, checking visibility was always fast and did not preponderate. Figure 6.2(c) shows the time needed for the two rendering phases.

6.1.2 Image Quality

We compared correct images with results created by the SEC–Tree. By correct, we mean that we did not spare any triangles, but rendered the complete scene contained in the view frustum. Walking through the scene to obtain these values took several hours due to the high rendering time of correct images, demonstrating the necessity of approaches speeding up rendering. Figure 6.1(b) to Figure 6.1(f) show example comparisons of images rendered using the SEC–Tree to correct images. Though errors are recognizable, especially when looking at at the extract in Figures 6.1(b) and 6.1(c), this regards only small details.

Let I_i denote a correct RGB image of resolution $w \times h$ at the i-th frame, J_i the image rendered using a SEC–Tree. We measured errors by comparing color distances (Figure 6.3(a))

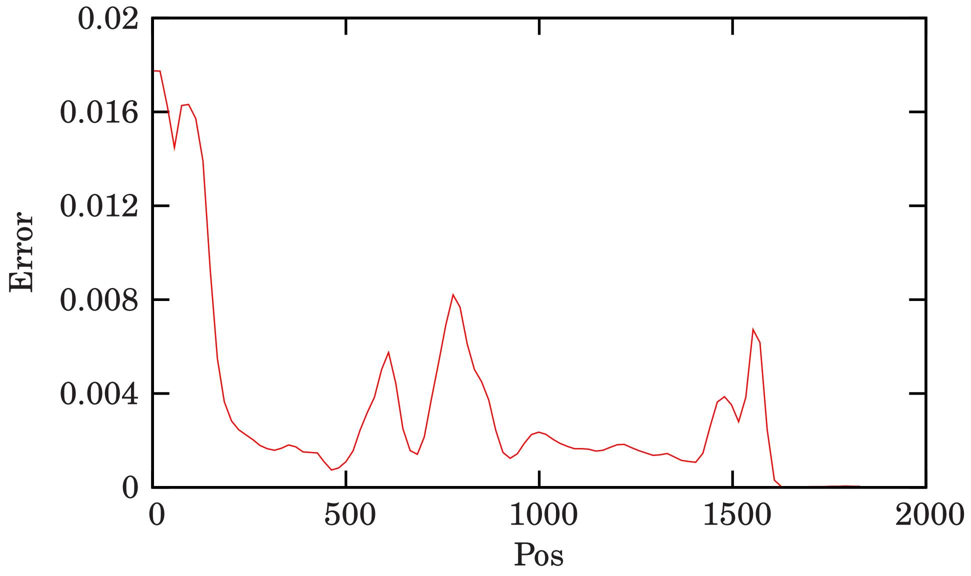
$$\| I_i - J_o \|_{color} := \frac{1}{wh} \sum_u \sum_v \| I_i(u,v) - J_i(u,v) \|^2$$

as well as the fraction of differing pixels (Figure 6.3(b))

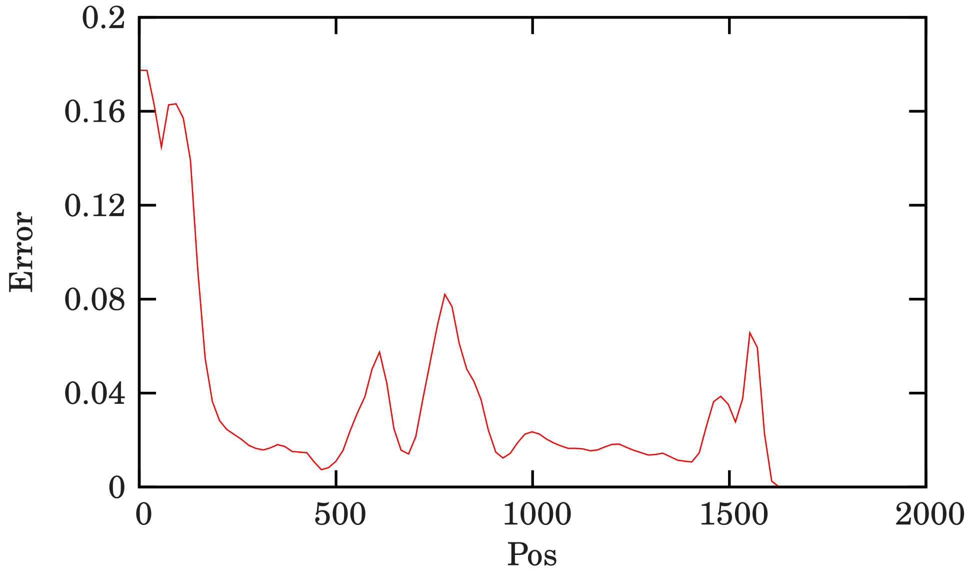
$$\| I_i - J_i \|_{pixel} := \frac{1}{wh} \sum_u \sum_v sgn\left(\| I_i(u,v) - J_i(u,v) \|\right).$$

Both curves show the same gradient and differ only in scale, i.e. visual impression only depends on the number of different pixels, the influence of actual color is of less importance.

Figure 6.3(a) demonstrates the quality of images rendered by the SEC–Tree, following the same path shown Figure 6.1(a) again. During the first 500 frames, image quality improved, while the number of visible triangles decreased, since selecting the same number of triangles from a smaller set results in a better representation. However, not only the number of triangles



(a) Averaged squared color differences



(b) Fraction of pixel differing in SEC–Tree rendered image from a correct image

Figure 6.3: Pixel errors on the path through the scene.

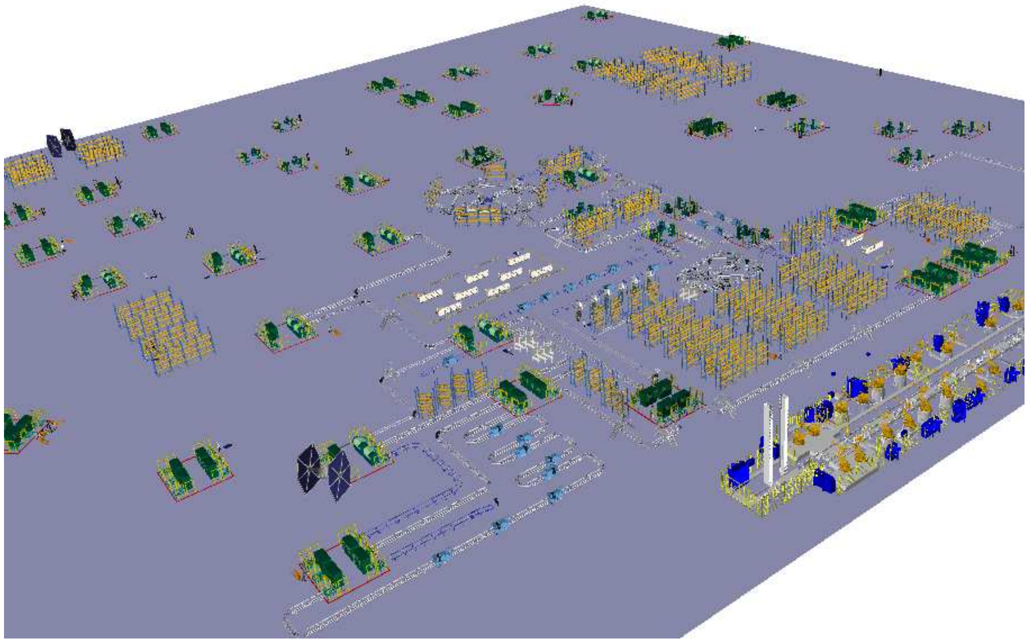
is of importance. At about frame 500, errors increased, while the number of visible primitives decreased. At these positions, some objects were near to the viewing position and small details were visible in the correct image, while the SEC–Tree preferred larger triangles. Partially, chosen triangles were occluded, thus making no contribution to the image. Still, more than 90 percent of all pixels were correct, so the resulting image was appropriate. At frame 900, nearly the complete scene was contained in the view frustum again, just as in the beginning, but there were much less errors. This occurred due to a different viewing angle. In contrast to the first frames, now a lot of objects were occluded by larger triangles. Great part of these occluders were large triangles, and therefore chosen as representatives, resulting in correct pixels. Towards the last frames, the number of visible triangles became less than the overall limit of rendered primitives. When rendering these frames, all these visible triangles were selected, hence a correct image was produced.

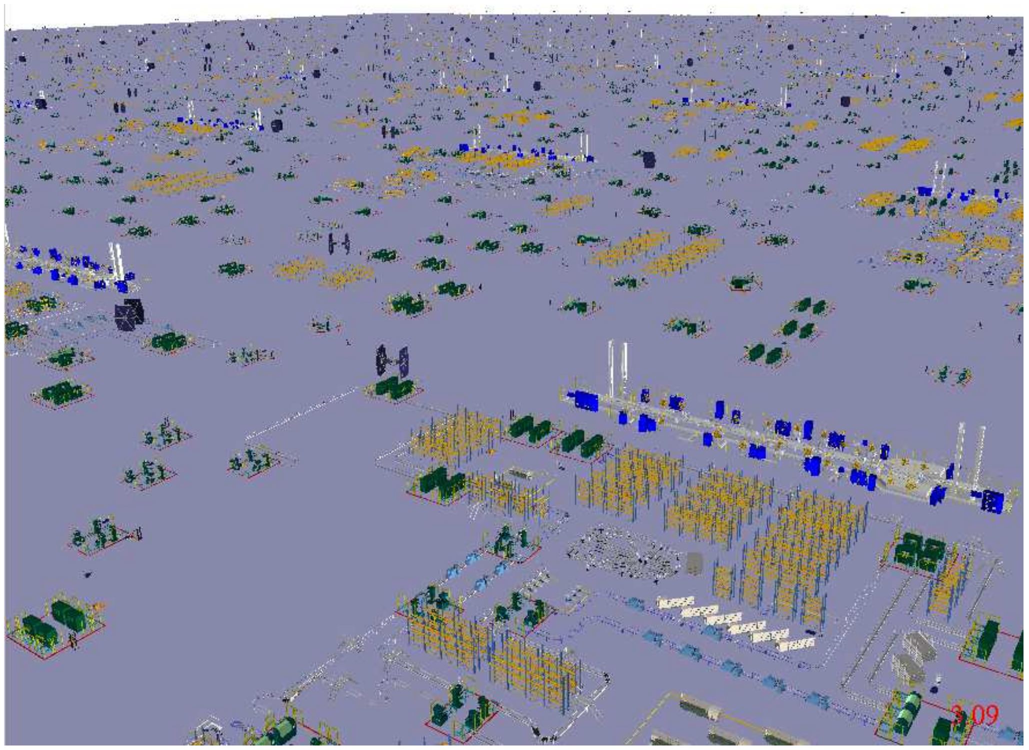
6.2 Rendering Dynamic Scenes

Figure 6.4(a) shows our dynamic scene, again rendered with 300.000 triangles. The complete scene contains about 14.5 million triangles and 892 objects in total. The complexity of different models varies from simpler models with a few hundred polygons to a complex model of a production line with more than 220.000 triangles. We increased the complexity of the scene by inserting additional replicas, resulting in six and 100 times the original scene. We will refer to them as Scene 1, 2 and 3, whereby Scene 1 is the original and Scene 3 the largest scene, shown in Figure 6.4(b).

Figure 6.5 shows some measurements along a path through each of these three scenes. At each point we looked at the number of objects intersecting the view frustum, shown in Figure 6.5(a) to 6.5(c). In addition, we considered the time needed to render a frame. In Figures 6.5(d) to 6.5(f), one can see the milliseconds needed to update the position of objects, determine the weights and actually render the scene.

In Scene 1, every object was moving the whole time, in Scene 2 and 3 we started with all objects staying at their position in the beginning and at one point they all began to move. While in an actual environment objects representing machines would not move but stay at one place, there are various moving objects like fork lifts or packets. By moving every object, we show


(a) A single instance of the scene



(b) The same scene as in Figure 6.4(a) but 100 times replicated

Figure 6.4: Overview on the dynamic test scene

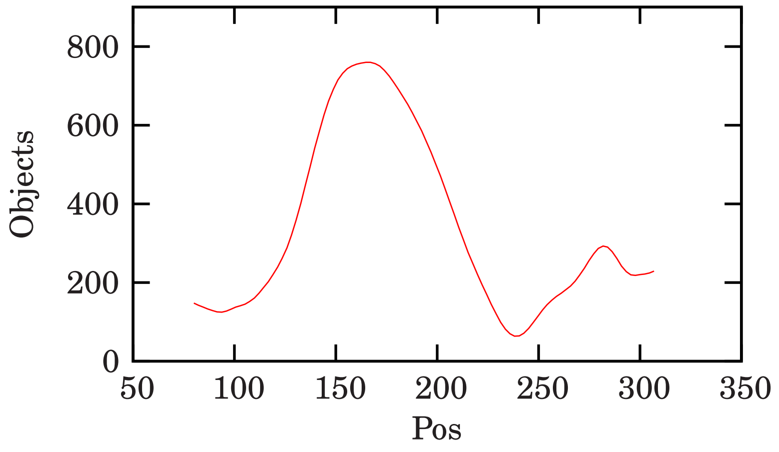
that our approach can handle scenes with a large amount of moving objects. In addition, this demonstrates that no object has to stay at one place. If the user wants to change the layout of the plant by moving a machine to another place, this can be done.

The time needed to update the position of an object is independent of the viewer's position and the number of objects in the view frustum. Comparing the time needed to update positions in Scene 1 to 3, one can see at the dotted lines in Figures 6.5(d) to 6.5(f), that this times increases linear with the number of objects in the scene from about 5ms to 30ms and 500ms.

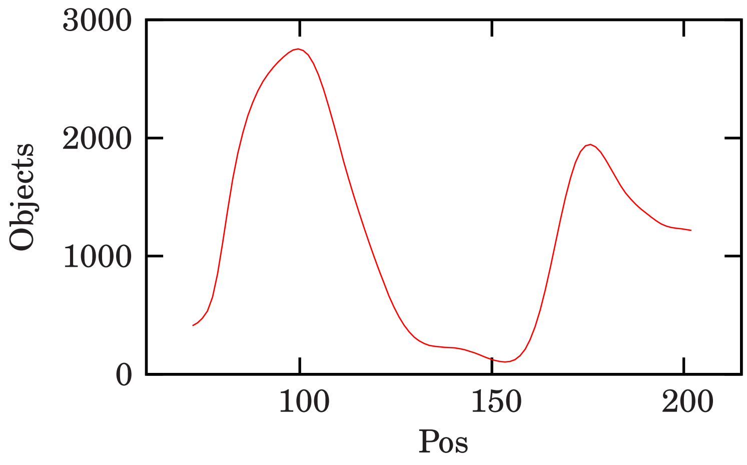
The results are similar regarding the assignment of weights. While there are no differences in the time needed to move the objects, there are some fluctuations in the time spent on assigning weights, because at some positions the object list was passed three times to avoid wasting triangles, while at other positions we traversed the object list only twice. One pass was never sufficient, since the floor consists of objects containing two very large triangles. Therefore, these objects received large weights and were assigned too many triangles, which had to be redistributed.

We see a dependency on the viewer's position, when we take a look at the actual rendering time. These curves in Figures 6.5(d) to 6.5(f) correspond to the number of objects intersecting the frustum, shown in 6.5(a) to 6.5(c). Every frame, up to a predefined number of triangles is rendered. But there is an additional overhead to rendering primitives, if the number of objects, which these triangles are assigned to, increases. For example, we have to alter the transformation matrix each time the rendered object changes. However, since the number of triangles to regard is limited, this holds for the number of objects which are actually rendered, i.e. receive a positive weight. In Figure 6.5(f) and 6.5(c) one can see that the number of visible objects increases at the end of the walkthrough, while the rendering time stays the same. Figure 6.5(g) shows the number of objects which are visible but do not receive any triangles. Even though this means that visible objects are not displayed, this has only small impact on the resulting image and is not recognizable due to aliasing anyways. These objects are small and far away from the viewer, covering only a few pixels. While results gained from the smaller scenes indicate that rendering time increases with the number of objects intersecting the view frustum, we can see now that this is only true up to a certain point.

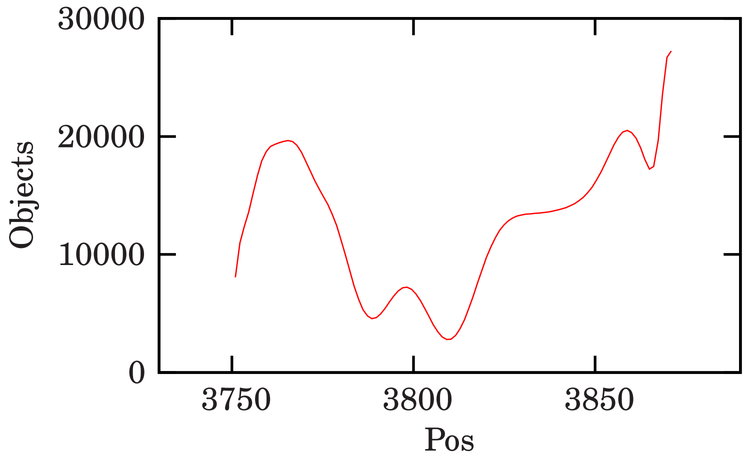
Rendering was fast enough to navigate through Scene 1 and 2. In Scene 3, we



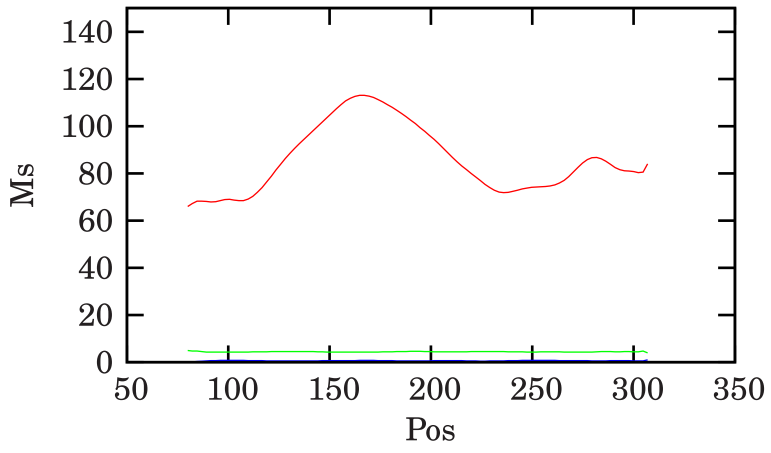
(a) Number of objects intersecting the view frustum at a path through Scene 1,



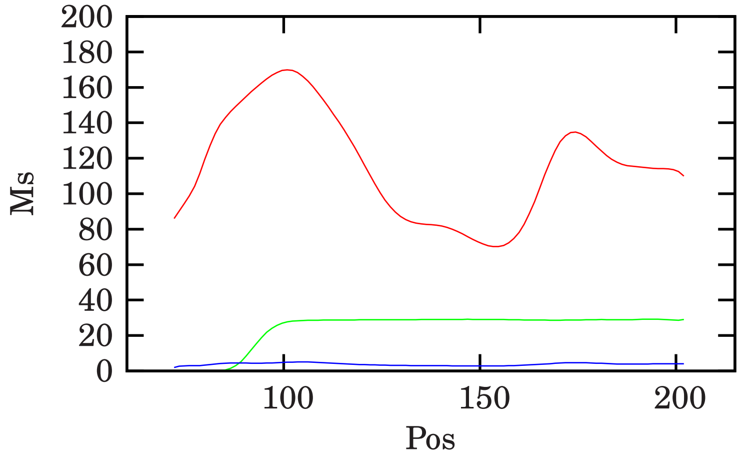
(b) . . . Scene 2



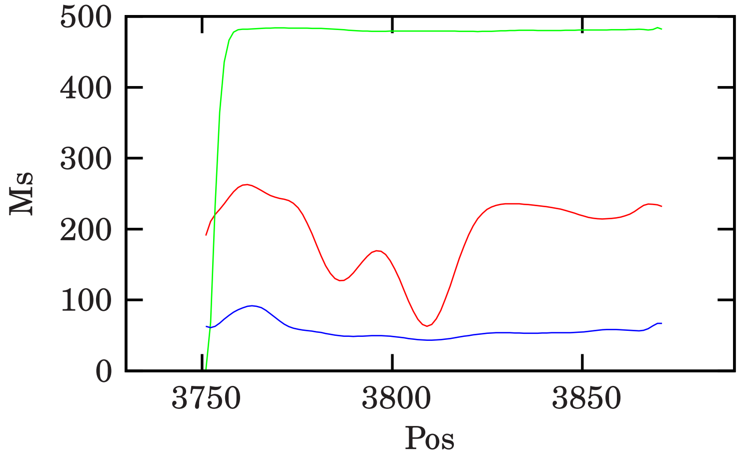
(c) . . . and Scene 3



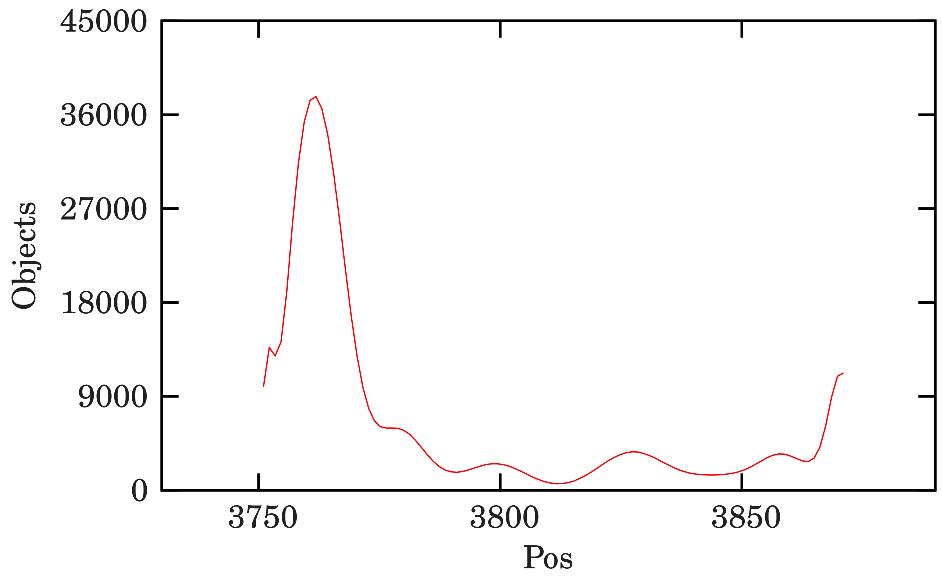
(d) Ms needed to updates position (green), distribute weights (blue) and render (red) Scene 1,



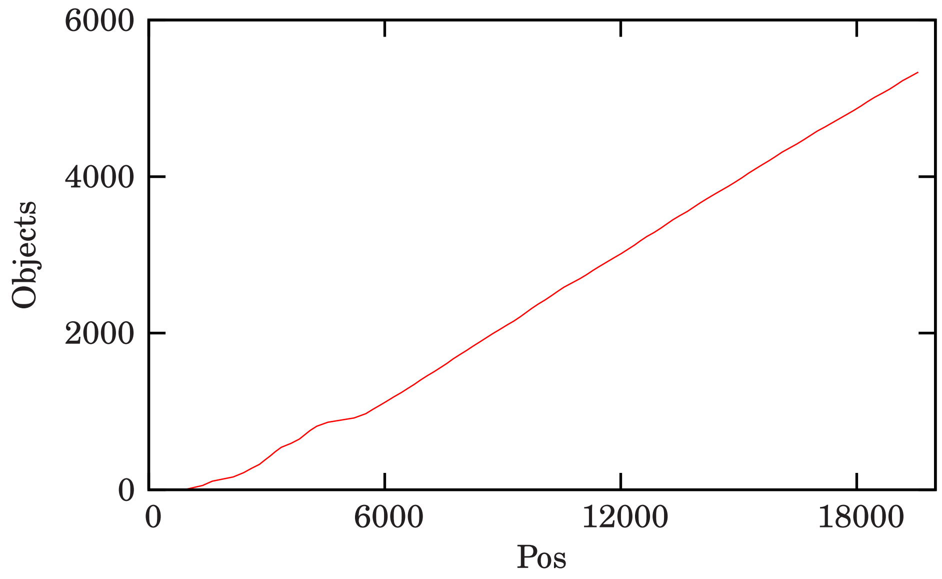
(e) . . . Scene 2,



(f) . . . and Scene 3 on the same path



(g) Number of objects intersecting the view frustum but not receiving triangles because of their low weight on the walk through Scene 3



(h) Number of objects inserted into Scene 2 at time after start

Figure 6.5: Measurements along a path through different scales of the dynamic scenes

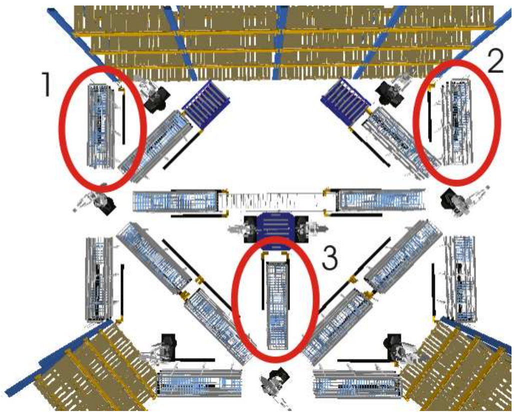
got only about one frame per second, half of the time was needed to update object positions. But this scene shows that a triangle budget leads to a limited amount of objects that have to be rendered, and therefore rendering time is bounded, while traversals of the object list depend on the actual number of objects in the scene.

Scenes are fully dynamically, inserting additional objects is possible any time. Figure 6.5(h) shows the process of loading Scene 2. Loading the complete scene takes about 18 seconds. At any time the objects already inserted can be rendered showing the scene loaded so far. As one can see in Figure 6.5(h) the time needed to insert additional objects does not depend on the size of the scene. This is necessary since new objects are created frequently, as during manufacturing processes new objects are produced and placed in the scene.

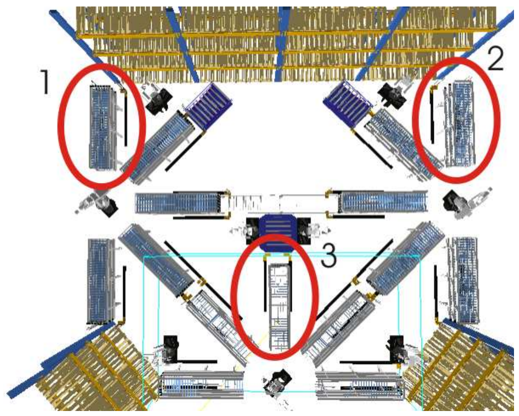
6.3 Multi-Point Level of Detail

Figure 6.6 shows the effect of applying multi-point level of detail to a scene. In addition to some other objects, this scene contains three instances of the object displayed in Figure 6.6(d). In Figure 6.6(a), no object is significant, i.e. all objects have a significance of 0. Since the viewpoint is located at a similar distance to all three instances, their representation is nearly equal. Altering the significance of objects changes their weights, and therefore their appearance. In Figure 6.6(b), instance 3 at the bottom is still not significant. However, the significance of instances 1 and 2 has been modified. Now, instance 1 on the left has a significance of one, instance 2 on the right a received value of 0.5. Triangles formerly assigned to object 3 at the bottom are now assigned to the significant objects 1 and 2. While the degree of detail of the two significant objects has increased, it decreased at instance 3. This becomes especially recognizable when looking at the marked rolls of the conveyor belt. Figure 6.6(c) shows a scene with the same significance as in Figure 6.6(b), the difference is that this time budget exceeding is allowed. The significant objects 1 and 2 are assigned additional triangles and their representations in Figure 6.6(b) and 6.6(c) correspond. However, instance 3 is not influenced by their significance. Its degree of detail in Figure 6.6(c) is the same as in Figure 6.6(a).

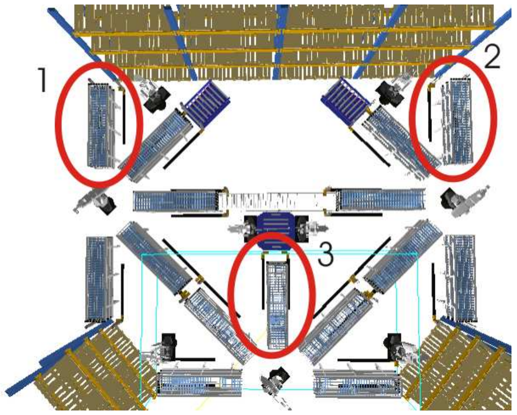
Figure 6.6 shows only a small scene. If we want to look at regions around significant objects with increased details, we have to consider larger scenes. Such a scene can be seen in Figure 6.7. The significance of the machine cell



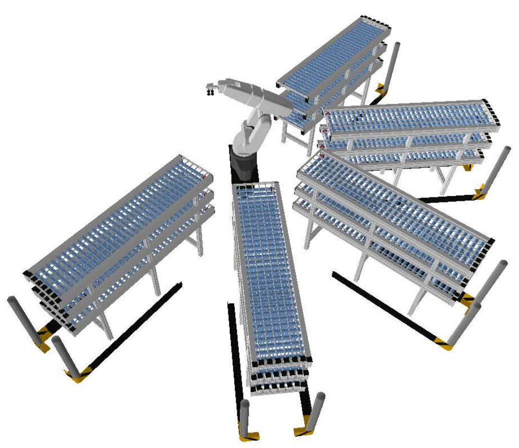
(a) None of the objects is significant



(b) Significance value of objects 1 is now 1, object 2 received a value of 0.5 while 3 still is not significant



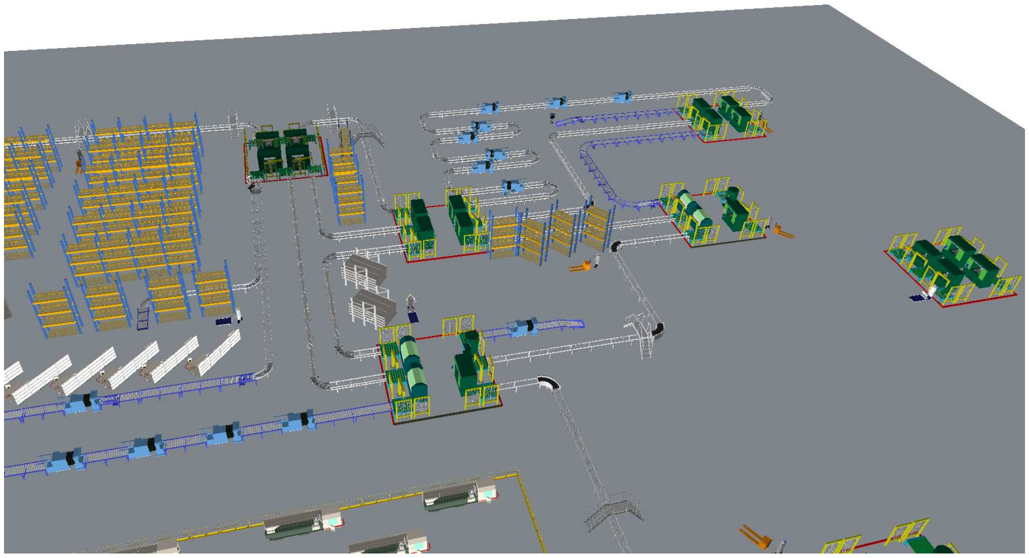
(c) Now objects have the same significance as in Figure 6.6(b), but this budget exceeding is allowed



(d) A single instance of the complete object we are interested in

Figure 6.6: Views on one scene with different representations of significant objects. Objects 1, 2, 3 are identically modeled and differ only in significance. The marked conveyor belt is a part of this model, at which differences in representations are easy to identify

located at the top right has been changed. In Figure 6.7(a) the scene has been rendered without any significance, while in Figure 6.7(b) an image has been created from the same position, now containing a significant machine cell. The degree of detail around this object was increased. Figure 6.8(a) and 6.8(b) show an enlarged image of the section around the machine cell. The


(a) No object is significant

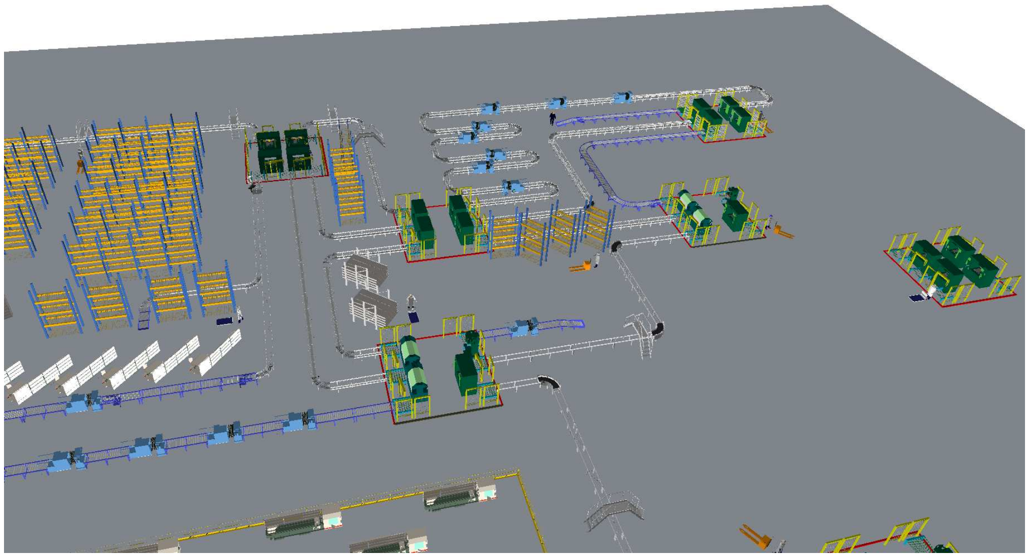
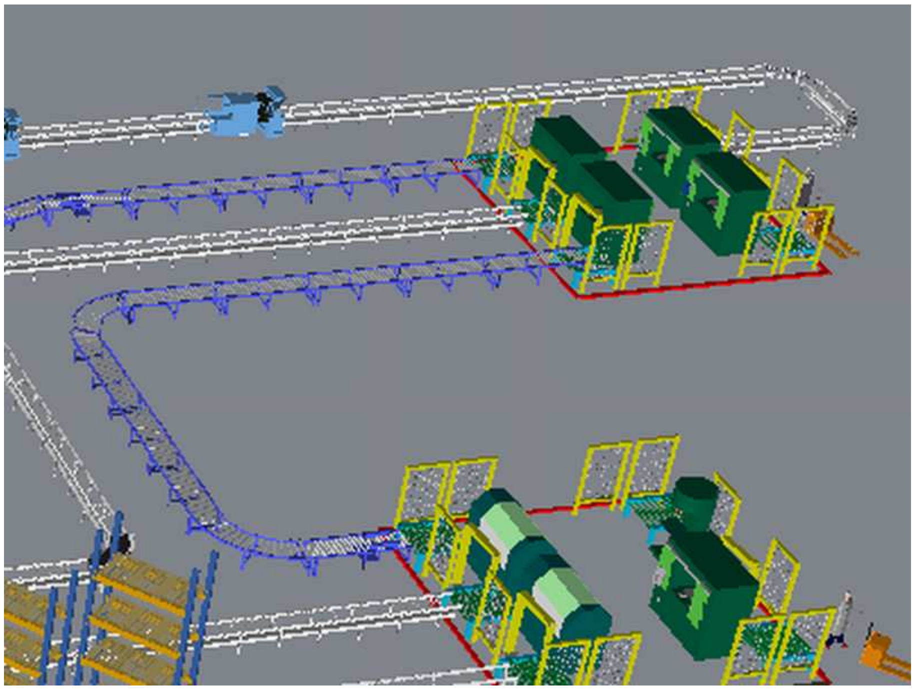

(b) The machine cell at the top right became significant

Figure 6.7: Example of increased details due to a significant object positioned nearby


(a) A closer look at the region where significance is about to change

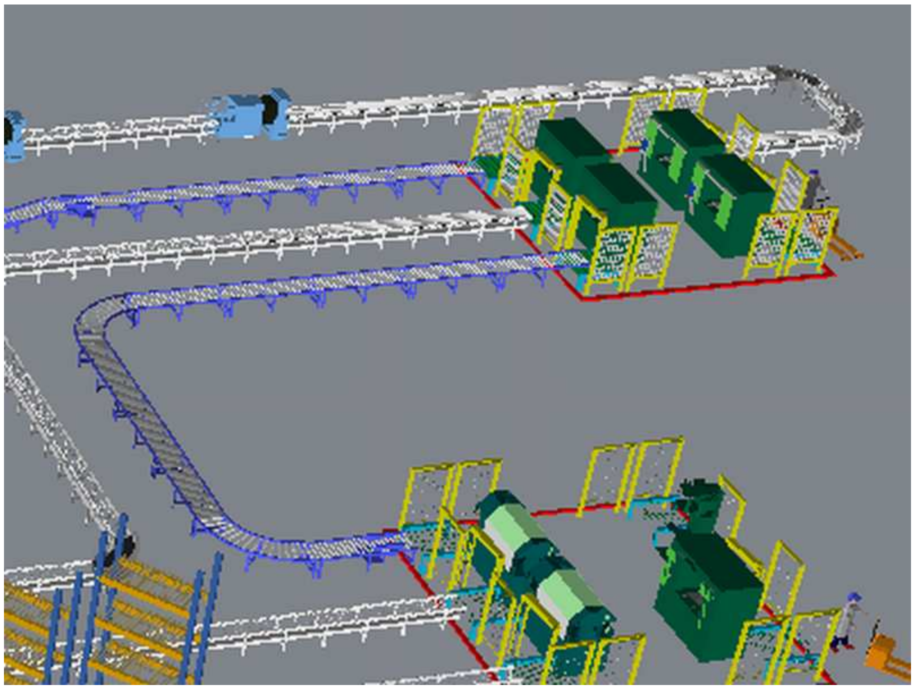

(b) Degree of details increased around the top machine cell, which has become significant

Figure 6.8: A closer look at the significant regions

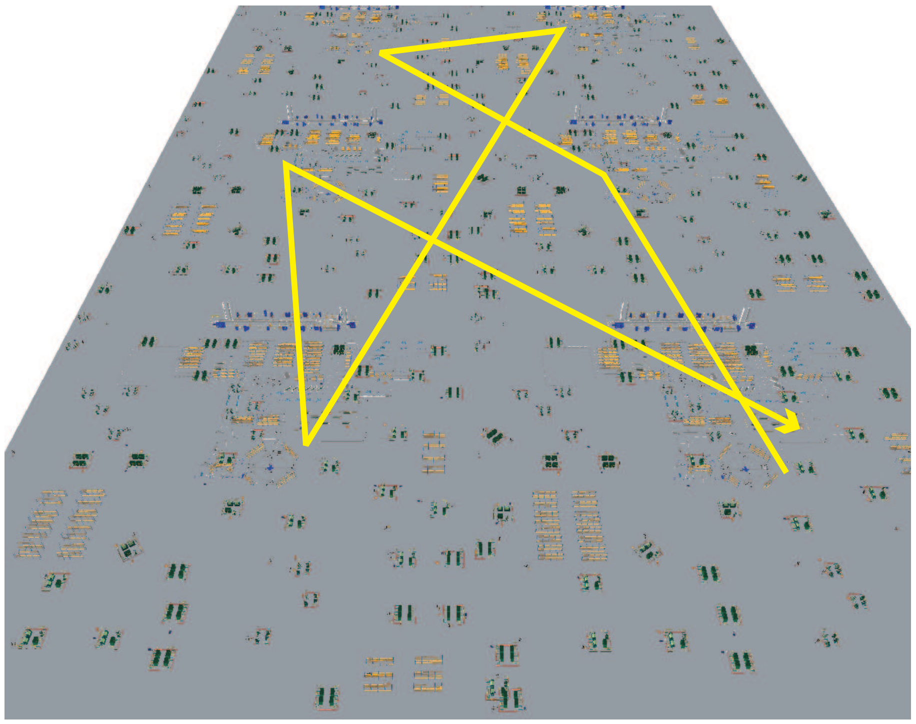
significant object itself became more detailed, as one can see when looking at the lattices surrounding the object, but in addition the adjacent conveyor belts received additional triangles as well.

Modifying the significance of objects did not have any influence on rendering performance. We looked at the frames per second on a sample path through Scene 2, which can be seen in Figure 6.9(a). 15 objects were significant, with different levels of significance. These objects are highlighted in Figure 6.9(b). None of the objects were moving to ensure that conditions were equal for every experiment.

Figure 6.10 shows the influence of significant objects on the rendering performance. Most of the time there have been no differences and therefore only the top line is visible. The red line indicates the frame rate when wewalked through the scene with no significant objects being present. The green and blue line show the results on the same path with significance considered. When getting the results for the blue line we allowed exceeding the triangle budget. Impact on the performance was so small, that it is not recognizable most of the time. Only between positions 700 and 800 there is a small drop of the frame rate, as we were near to multiple objects with a high significance value. The yellow line represents the results when object significance increased the degree of details of surrounding objects. All four curves show the same behavior. Rendering performance showed some fluctuation between six and 15 frames per second, depending on the position in the scene, but nearly no influence of significant objects on the performance can be recognized.

6.4 Out-of-Core Rendering

Our rendering algorithm allows displaying scenes exceeding memory capacities by loading only geometry needed within the next frames. We measured occupied memory when rendering the same three scenes as in section 6.2, following the same path in each of the scenes. For each frame we counted the size of nodes in main memory. Instantiation was not considered, every object was loaded independently of other representatives, even if their geometrical representation was identical. Global SEC-trees or the object list was always kept in main memory as its size is neglactable compared to the amount of geometry in the scenes. Further, our system requires about 124 MB for com-


(a) Measurements were made along this path

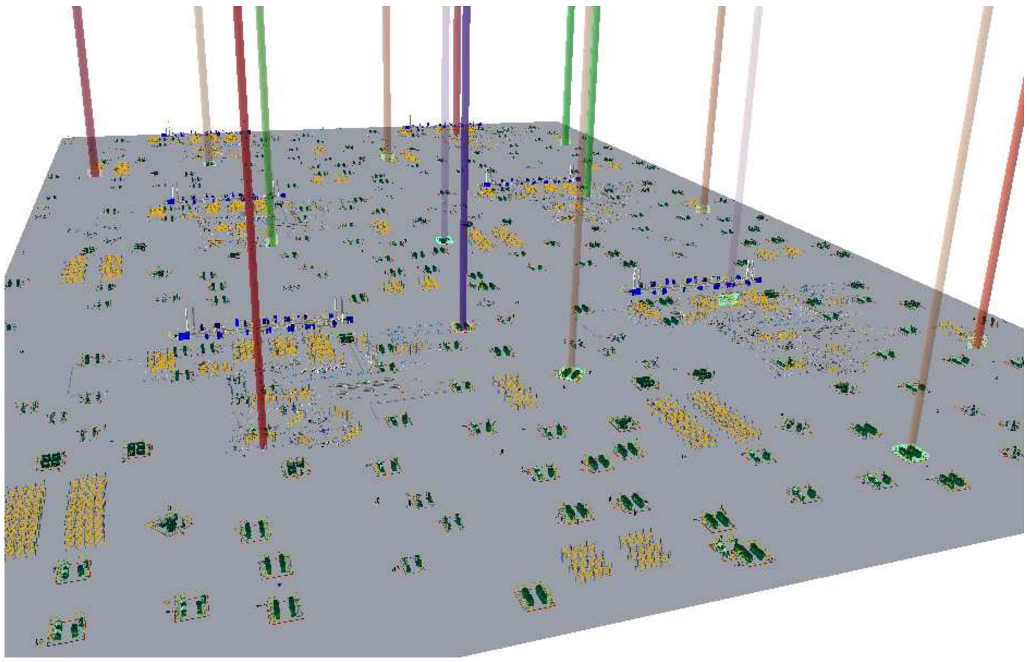

(b) These objects were significant

Figure 6.9: Sample scene containing significant objects

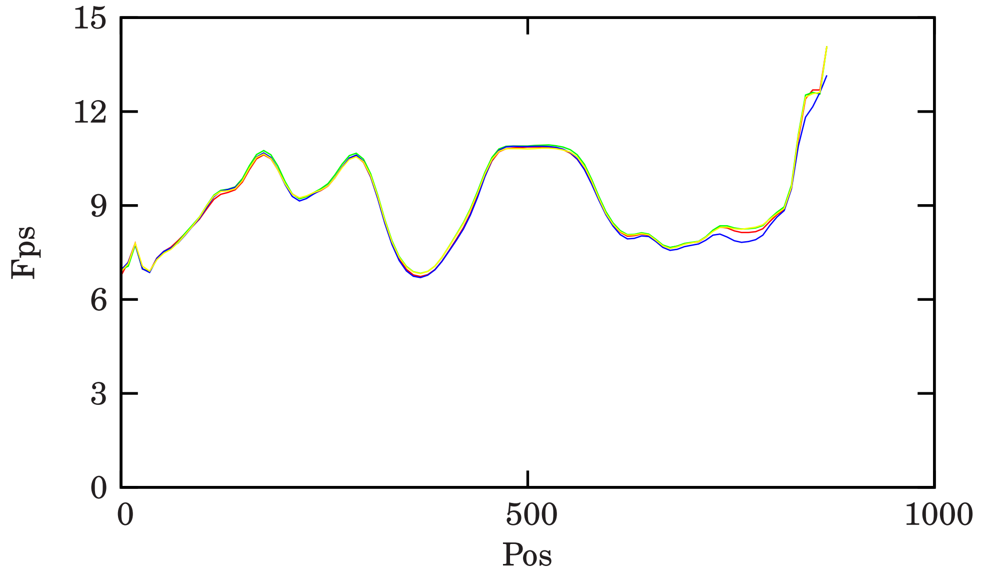


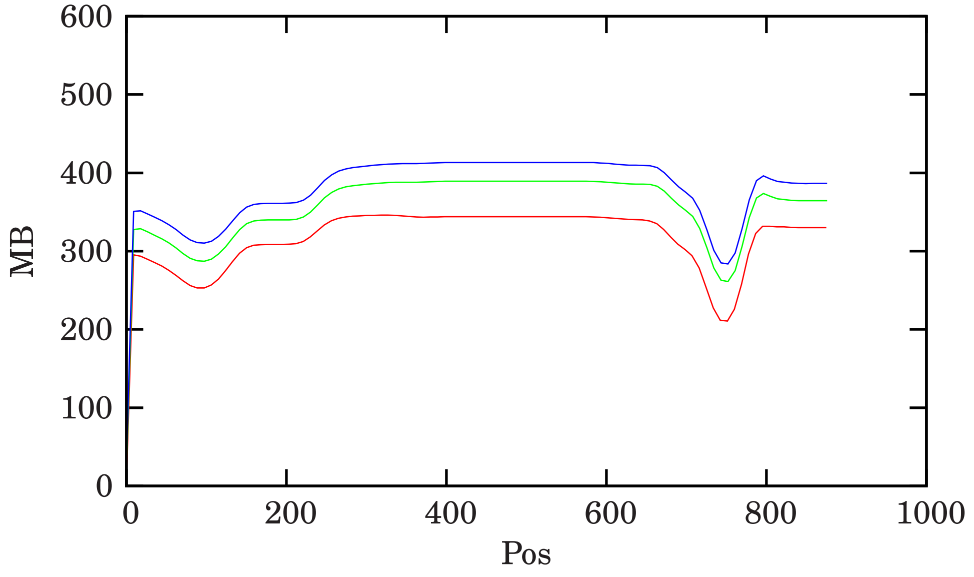
Figure 6.10: Fps while walking through the scene with no object being significant (red), objects becoming significant without budget exceeding (yellow), allowing budget exceeding (blue) and emphasizing regions (pink)

ponents of the rendering system, which is not considered within the following measurements.

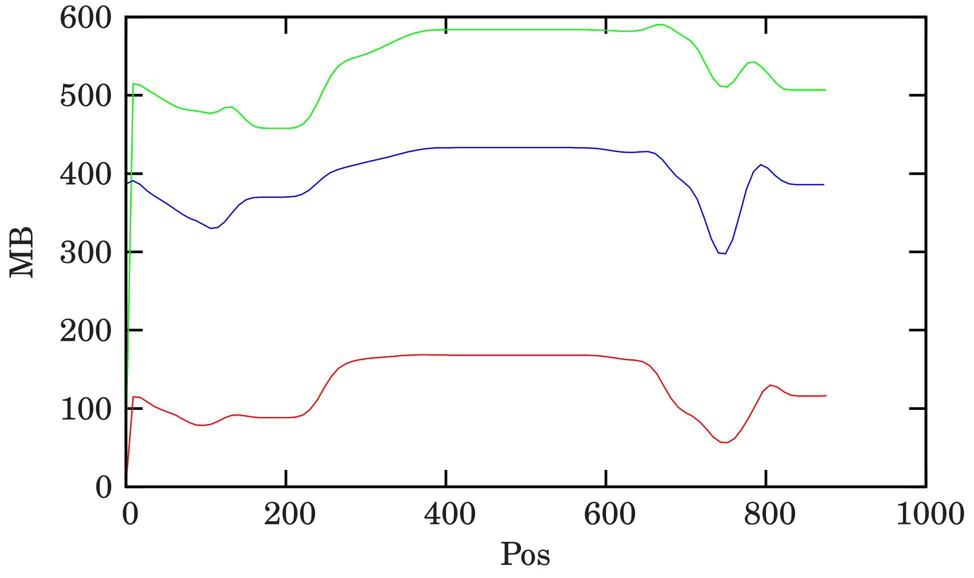
All objects of Scene 1 combined occupy about 660 MB if stored in main memory. A scene of this size does not require an out-of-core rendering approach, which is different for Scene 2. Memory requirements increase to about 3960 MB, which is more than a 32 bit windows XP machine can address. Scene 3 consists of 100 replications of Scene 1, therefore occupying even 66 GB.

Figures 6.11(a) and 6.11(b) show the influence of different parameters on size of nodes loaded into main memory. We followed the same path through Scenes 1, 2 and 3, and used different screen resolutions and movement speeds for Scene 2.

The red line in Figure 6.11(a) shows results for Scene 1 rendered at a resolution of 800×600. Memory consumption fluctuated between about 200 and 350 MB, which is about a third to half the scene size. The main influence on needed memory was the height of the viewer's position. At ground level, we needed about 300 MB for each position. When increasing the viewer's altitude, more triangles' spans were beneath our threshold for rendering.



(a) Consumed memory on the path through Scene 1 (red), Scene 2 (green) and Scene 3 (blue) . . .



(b) . . . and Scene 2 at lower screen resolution (red), higher screen resolution (green), and looking 50 steps ahead (blue) . . .

Figure 6.11: Memory consumption of SEC–Trees in different scenes

The results of increasing scene size can be seen at the green line in Figure 6.11(b). Scene 2 contains six times the original scene and therefore is six times as large. However, memory requirements only slightly increased to about 250 to 400 MB. The additional geometry was hardly loaded into main memory. This effect becomes even more visible at the blue line. Scene 3 is 100 times as large as Scene 1, but we never needed more than 425 MB of memory.

The red and green line in Figure 6.11(b) show Scene 2 rendered at a different screen resolution. Since we discard SEC–Tree nodes containing only triangles with a projected span below a threshold in pixel size, this influences the amount of memory that is loaded, as each pixel covers a smaller fraction of the image if the resolution is increased. The red line shows results measured at a screen resolution of 320×240, while the green line represents measurements for a larger resolution of 1200×900. With increasing screen resolution, memory consumption also goes up with 70 to 80 MB for the small resolution and 450 to nearly 600 MB for the high resolution. Thus, changing screen resolution has a larger influence than increasing scene sizes.

Up to now, we looked 10 steps ahead and loaded nodes, which might be rendered within the next 10 steps. The blue line in Figure 6.11(b) shows the results of considering 50 steps, this is equivalent to increasing the maximal speed of the user and objects to five times the previous value. We used Scene 2 again, at the resolution of 800×600. This lead to slightly increased memory requirements of about 50 MB in addition, compared to looking ten steps ahead, but was not as significant as altering screen resolutions.

6.5 A Comparison to Rendering Octrees

In addition to the SEC–Tree and rendering it with a triangle budget as described in 4, we have implemented an octree. We have chosen two ways of rendering the octree and compared them to the SEC–Tree. At first, we rendered the octree up to a certain depth and ignored small nodes. As an alternative, we applied our weighted rendering to the octree. In both cases, we discarded invisible nodes by applying frustum culling.

6.5.1 Discarding Small Octree Nodes

We achieve rendering at interactive frame rates by limiting the number of displayed primitives and thus allowing image errors. In contrast, other approaches like the Randomized Z-Buffer [WFP+01] focus on producing correct images of large scenes, but do not keep up a constant frame rate. The Randomized Z-Buffer is an octree based algorithm, which renders larger octree nodes completely, but only a representative chosen subset of smaller nodes. We have implemented an octree and a rendering approach between rendering SEC–Trees and the Randomized Z-Buffer, discarding nodes of projected size of at most one pixel completely, and therefore rendering even less primitives than the Randomized Z-Buffer would display.

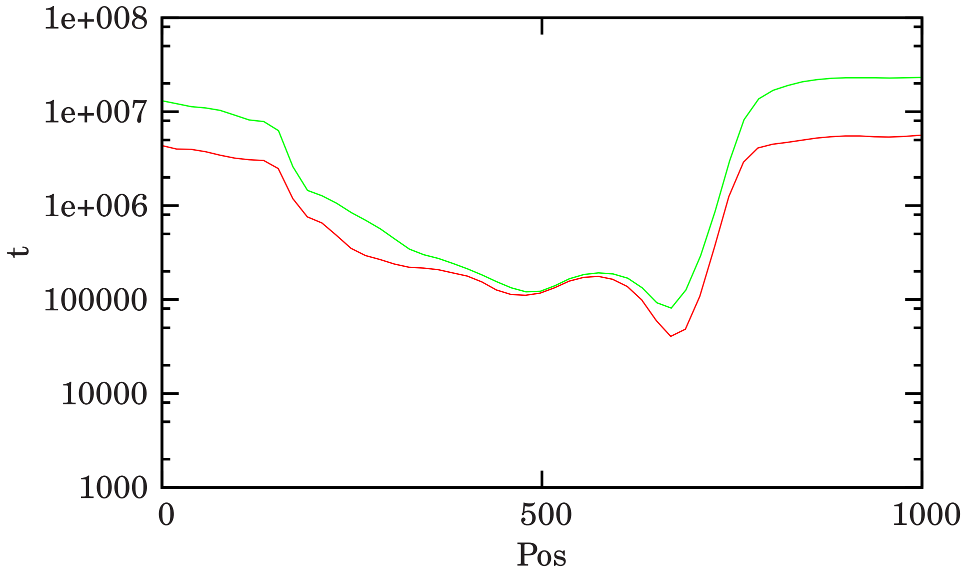


Figure 6.12: Number of rendered triangles t, when octree nodes of projected size of less than one pixel are not rendered, at resolutions of 480×360 (red line) and 1920×1080 (green line)

This rendering method was significantly slower than rendering the scene with a SEC-tree. Comparing actual rendering times is difficult, as this depends strongly on the implementation. E.g. our test of projected octree node size is quite expensive, but might be replaced by more efficient strategies. This resulted in rendering times of more than 10 seconds for most frames.

We have chosen a different measure for comparing these approaches, namely the number of rendered triangles. Our approach chooses a constant number of triangles in order to preserve the frame rate, while allowing an unlimited number of triangles to be rendered necessarily increases rendering times. Note, that the projected size of octree nodes depends on screen resolution. Therefore, more triangles will be rendered at higher resolutions if octree nodes of size less than one pixel are discarded.

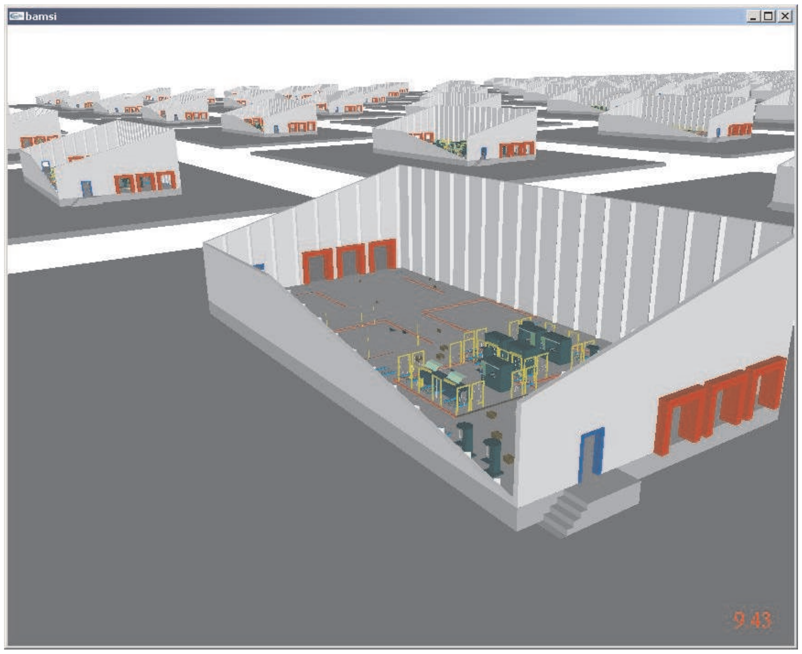
Following the same path in the same scene as for the measurements in static scenes which has been used in 6.2, we counted the number of rendered triangles at a resolution of 480×360 pixels (red line in Figure 6.5.1) and 1920×1080 pixels (green line in Figure 6.5.1). Both curves show the same behaviour as the curve displaying the number of triangles intersecting the view frustum 6.2(d), they only differ in scale. While most triangles have been discarded, at positions allowing the viewer to overlook a large portion of the scene, their number still is significantly larger than rendering capacities. In the first frame as an example, more than 4 million triangles have been rendered at lower resolution, and even more than 13 million at the higher resolution. While this discards more than 90% of the 131 million visible triangles, rendering at interactive frame rate is not possible, even if our implementation would be more efficient.

The shape of our models is one reason for this behavior. Objects are modeled, such that they contain many long and thin triangles, lattices or conveyer belts are examples for this. While contributing little to the image because of their small area, they are positioned at higher octree levels due to their large diameter. Such nodes are projected to areas than larger one pixel and therefore these small triangles are rendered. Other scenes containing triangles, which are equilateral or similar to that, would be represented much better by an octree.

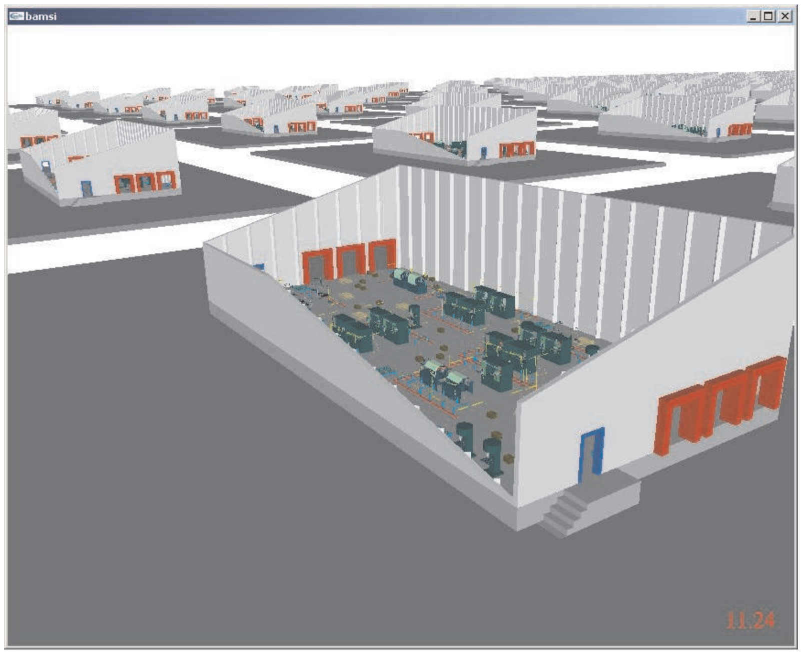
6.5.2 Rendering Octrees with Triangle Budgets

We combined a further rendering technique with the octree, similar to that presented in chapter 4. The number of rendered triangles was limited and each octree node was assigned a limit of triangles to render, corresponding to the area of triangles it contained and the distance of its center from the viewer.

The image quality produced by the octree was significantly lower compared



(a) Comparison of the SEC–Tree with an octree: the octree was used to render this



(b) The same view as in figure 6.13(a) rendered with the SEC-tree

Figure 6.13: Screenshots comparing weighted rendering of the SEC–Tree and octree

to the SEC–Tree. Figure 6.13(a) shows an image rendered with the octree approach described above. From the same position, the image in Figure 6.13(b) was drawn, but this time the SEC–Tree was used. Image quality has improved significantly. While some details like lattices around machines are missing in Figure 6.13(b), those are small details. Using the octree, models inside the front hall are highly detailed while other objects are completely missing. Therefore, the SEC–Tree gives a much better impression of the correct scene. This shows that even if our weighted rendering can be applied to other hierachical data strucures, it benefits from the way geometry in a scene is organized by a SEC–Tree.

The reason for this behavior is that objects receive triangle limits depending on nodes higher in the hierarchy. Objects stored at lower octree levels near to the viewer might get less weight than appropriate if centers of parent nodes are located at a larger distance. The SEC–Tree is much less problematic regarding such considerations. At the upper levels, position is not that important as the primary criteria for organizing triangles and objects is their size. The number of a node's successors can be arbitrary large, resulting in a breadth and shallow tree. Problems arising from data distribution over too many levels are reduced this way. Additionally, SEC–Tree nodes are a much better representation of stored data, since the tree structure does not rely on partitioning space at fixed positions, but is more flexible and adaptive to object locations compared to an octree.

CHAPTER 7

CONCLUSIONS

7.1 Contributions

The SEC–Tree proved to be an appropriate data structure to render massive industrial scenes containing hundreds of complex CAD models. By keeping models only partially in main memory and prefetching portions which might be needed in the future, the size of the scene could extend available memory capacities by several times over. Scenes were fully dynamic, in the sense that arbitrary insertions, deletions and movements of objects were possible, while static scenes allowed further optimizations by building up a second hierarchy on object level.

By selecting a number of triangles adjusted to hardware capabilities, fast rendering was ensured with only small dependency on the viewer's position in the scene. Organizing triangles in SEC–Trees, those triangles presenting large contributions to the final image could be selected, resulting in only few image errors, while only a fraction of the scene was rendered. Weights of objects in the scene could be adjusted to consider importance their and increase details at significant locations.

7.2 Future Work

While our approach of rendering SEC–Trees works well in practice, there are still possibilities for improvements. We will list some of them in the following section.

The weighting function considers size and distance of triangles from the viewer, while their actual influence on the correctly rendered image depends on more parameters than those. View depend weights considering occlusion or orientation of geometry might result in a better choice of rendered triangles.

The out-of-core approach considers scenes stored on hard drive and loaded on demand into main memory. However, considering graphics hardware, three levels of memory are available. Instead of sending geometry constantly to the graphics hardware, storing geometry needed immediately on the graphics card and reuse this geometry for some frames should speed up the rendering system.

We consider dynamic scenes, however an object hierarchy is only created for static scenes, since SEC–Trees do not support updates. A data structure allowing efficient insertions and deletions of objects would be desirable. A goal easier to accomplish could be animations, where movement is limited to local alterations or known during the preprocessing.

Objects are rendered by assigning weights hierarchically, depending on the distance from the viewer. Near to the viewer's location subnodes of the SEC–Tree might be located significantly closer than their parent nodes. This is not considered, when distributing triangles on these nodes. Increasing details in the foreground by temporarily moving close subtrees on a higher level might improve the overall image quality.

In theory, there is no limitation on image errors, while in practice we achieve satisfying results. It would be interesting to describe scenes or objects corresponding to our application and proving that these cases always lead to only small deviation to the correct image.

Image based rendering approaches might be an alternative to discarding geometry. In order to reduce image errors, nodes assigned only few triangles could render a simplified version determined as a preprocessing step. In addition to classical mesh simplification approaches, textured models might result in images visually closer to a correct one.

Occlusion culling approaches like [GKM93] utilize a precomputed octree to find appropriate triangles considered as occluders. This is an application the SEC–Tree might be better suited for in certain scenes. While octrees organize triangles based on their span, the SEC–Tree is based on actual area. Especially long but thin triangles are misrepresented by octrees. Because of their small area, they are a bad choice for occluders. However, exactly this small area leads to those triangles being stores at lower levels of the SEC–Tree, which should result in the SEC–Tree being the better suiting data structure in scenes containing many such triangles.

BIBLIOGRAPHY

[ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. *SIGMOD Rec.*, 28(2):49–60, 1999.

[ACW$^+$99] Daniel Aliaga, Jon Cohen, Andrew Wilson, Eric Baker, Hansong Zhang, Carl Erikson, Kenny Hoff, Tom Hudson, Wolfgang Stuerzlinger, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. Mmr: an interactive massive model rendering system using geometric and image-based acceleration. In *I3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 199–206, New York, NY, USA, 1999. ACM.

[AE99] P. Agarwal and J. Erickson. Geometric range searching and its relatives, 1999.

[ASVNB00] C. Andujar, C. Saona-Vazquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of details through hardly-visible sets, 2000.

[Ber02] Pavel Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[BSGM02] W. Baxter, A. Sud, N. Govindaraju, and D. Manocha. Gigawalk: Interactive walkthrough of complex environments. In *Rendering Techniques*, pages 203–214, 2002.

[BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624, sep 2004.

[Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974.

[CKS03] Wagner T. Correa, James T. Klosowski, and Claudio T. Silva. Visibility-based prefetching for interactive out-of-core rendering. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 2, Washington, DC, USA, 2003. IEEE Computer Society.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[COCSD03] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Trans. Vis. Comput. Graph.*, 9(3):412–431, 2003.

[CPK$^+$05] Jatin Chhugani, Budirijanto Purnomo, Shankar Krishnan, Jonathan Cohen, Suresh Venkatasubramanian, David S. Johnson, and Subodh Kumar. vlod: High-fidelity walkthrough of large virtual environments. *IEEE Trans. Vis. Comput. Graph.*, 11(1):35–47, 2005.

[CW02] Cem Cebenoyan and Matthias Wloka. Graphics performance: Balancing the rendering pipeline. Presented at Game Developers Conference 02, 2002.

[DHL$^+$06] Wilhelm Dangelmaier, Daniel Huber, Christoph Laroque, Mark Aufenanger, Matthias Fischer, Jens Krokowski, and Michael Kortenjan. d^3fact insight goes parallel – aggregation of multiple simulations. In *Simulation and Visualization 2006 (SimViS)*, pages 79–88. SCS European Publishing House, 2006.

[EGS05] David Eppstein, Michael T. Goodrich, and Jonathan Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 296–305, New York, NY, USA, 2005. ACM Press.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Evangelos Simoudis, Jiawei Han, and Usama Fayyad, editors, *Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, Portland, Oregon, 1996. AAAI Press.

[EM99] Carl Erikson and Dinesh Manocha. GAPS: general and automatic polygonal simplification. In *Symposium on Interactive 3D Graphics*, pages 79–88, 1999.

[EMWVB01] Carl Erikson, Dinesh Manocha, and III William V. Baxter. Hlods for faster display of large static and dynamic environments. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 111–120, New York, NY, USA, 2001. ACM Press.

[Eri96] Carl Erikson. Polygonal simplification: An overview. Technical report, Chapel Hill, NC, USA, 1996.

[ESSS01] Jihad El-Sana, Neta Sokolovsky, and Cláudio T. Silva. Integrating occlusion culling with view-dependent rendering. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 371–378, Washington, DC, USA, 2001. IEEE Computer Society.

[FKN80] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph.*, 14(3):124–133, 1980.

[FLH$^+$07] Matthias Fischer, Christoph Laroque, Daniel Huber, Jens Krokowski, Bengt Mueck, Michael Kortenjan, Mark Aufenanger, and Wilhelm Dangelmaier. Interactive refinement of a material flow simulation model by comparing multiple simulation runs in one 3d environment. In *European Simulation and Modelling Conference (ESM 2007)*, pages 499–505. EUROSIS, October 2007.

[FMM$^+$05] Matthias Fischer, Bengt Mueck, Kiran Mahajan, Michael Kortenjan, Christoph Laroque, and Wilhelm Dangelmaier. Multi-user support and motion planning of humans and humans driven vehicles in interactive 3d material flow simulations. In *WSC '05: Proceedings of the 37th conference on Winter simulation*, pages 1921–1930. Winter Simulation Conference, 2005.

[FS93] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *ACM SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 247–254, New York, NY, USA, 1993. ACM Press.

[FTSK96] Thomas Funkhouser, Seth Teller, Carlo Sequin, and Delnaz Khorramabadi. The UC berkeley system for interactive visualization of large architectural models. *Presence, the Journal of Virtual Reality and Teleoperators*, 5(1):13–44, 1996.

[Gar99] M. Garland. Multiresolution modeling: Survey & future opportunities. *Eurographics '99 – State of the Art Reports*, pages 111–131, 1999.

[GD98] J. P. Grossman and William J. Dally. Point sample rendering. In *9th Eurographics Workshop on Rendering*, pages 181–192, 1998.

[GH97] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *ACM SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 209–216, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[GKM93] Ned Green, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Computer Graphics Procieedings, Annual Conference Series*, pages 231–238, August 1993.

[GM05] Enrico Gobbetti and Fabio Marton. Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 878–885, New York, NY, USA, 2005. ACM.

[GS02] Michael Garland and Eric Shaffer. A multiphase approach to efficient surface simplification. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 117–124, Washington, DC, USA, 2002. IEEE Computer Society.

[Hop96] Hugues Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

[KKF$^+$02] Jan Klein, Jens Krokowski, Matthias Fischer, Michael Wand, Rolf Wanka, and Friedhelm Meyer auf der Heide. The randomized sample tree: a data structure for interactive walkthroughs in externally stored virtual environments. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 137–146, New York, NY, USA, 2002. ACM Press.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[KS00] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108–123, /2000.

[KS01] James T. Klosowski and Cláudio T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365–379, 2001.

[KS06] Michael Kortenjan and Gunnar Schomaker. Size equivalent cluster trees – realtime rendering of large industrial scenes. In *4th International Conference on Virtual Reality, Computer Graphics, Visualization and Interaction (Afrigraph 2006)*. African Graphics Association (AFRIGRAPH), 25. - 27. January 2006.

[KV07] Michael Kortenjan and Mario Vodisek. A note on throwing replicated balls into bins. Technical Report tr-ri-06-278, Heinz Nixdorf Institut, 2007.

[LRC$^+$03] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2003.

[Lue97] David Luebke. A survey of polygonal simplification algorithms. Technical Report TR97-045, 16, 1997.

[LW85] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical Report TR 85-022, University of North Carolina at Chapel Hill, 1985.

[Mat94] Jiri Matousek. Geometric range searching. *ACM Computing Surveys*, 26(4):421–461, 1994.

[MDL$^+$04] Bengt Mueck, Wilhelm Dangelmaier, Christoph Laroque, Matthias Fischer, and Michael Kortenjan. Guidance of users in interactive 3d-visualisations of material flow simulations. In Thomas Schulz, Stefan Schlechtweg, and Volkmar Hinz, editors, *Simulation and Visualisation 2004*, pages 73–83, Magdeburg, 4 - 5 March 2004. SCS European Publishing House.

[MLD$^+$05] Kiran Mahajan, Christoph Laroque, Wilhelm Dangelmaier, Christian Soltenborn, Michael Kortenjan, and Daniel Kuntze. d^{3}fact insight: A motion planning algorithm for material flow simulations in virtual environments. In Thomas Schulze, Graham Horton, Bernhard Preim, and Stefan Schlechtweg, editors, *Simulation and Visualization 2005 (SimViS)*, volume 1, pages 115–126. SCS European Publishing House, 3 - 4 March 2005.

[PS97] E. Puppo and R. Scopigno. Simplification, lod and multiresolution - principles and applications. *Eurographics '97 Tutorial Notes*, 1997.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *ACM SIGGRAPH 2000, Computer Graphics Proceedings*, pages 335–342. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[RH94] John Rohlf and James Helman. Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 381–394, New York, NY, USA, 1994. ACM.

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *sg00*, pages 343–352, 2000.

[Ros04] Jarek Rossignac. *Surface simplification and 3D geometry compression*, chapter 54, pages 1–32. CRC Press, 2004.

[Sed92] Robert Sedgewick. *Algorithms in C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[TS91] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics*, 25(4):61–68, 1991.

[WFP$^+$01] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In Eugene Fiume, editor, *ACM SIGGRAPH 2001, Computer Graphics Proceedings*, pages 361–370. ACM Press / ACM SIGGRAPH, 2001.

[XW05] Rui Xu and II Wunsch. Survey of clustering algorithms. *Neural Networks, IEEE Transactions on*, 16(3):645–678, 2005.

[ZGP00] Matthias Zwicker, Markus H. Gross, and Hanspeter Pfister. A survey and classification of real time rendering methods. Technical Report 2000-09, Mitsubishi Electric Research Laboratories, Cambridge Research Center, March 2000.

[ZMHH97] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. *Computer Graphics*, 31(Annual Conference Series):77–88, 1997.