UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty of Computer Science, Electrical Engineering and Mathematics
Heinz Nixdorf Institute & Department of Computer Science
Research Group Algorithms and Complexity

Dissertation

Spherical Visibility Sampling

Preprocessed Visibility for Occlusion Culling in Complex 3D Scenes

Benjamin Eikel

Paderborn, September 27, 2013

Reviewers:	Prof. Dr. Friedhelm Meyer auf der Heide Prof. Dr. Gitta Domik-Kienegger
Date of oral examination:	December 18, 2013
Contact:	Benjamin Eikel <benjamin@eikel.org>
Acknowledgment:	Benjamin Eikel was supported by a scholarship of the International Graduate School Dynamic Intelligent Systems, University of Paderborn.

Abstract

Many 3D scenes (e.g., generated from CAD data) are composed of a multitude of objects that are nested in each other. An industrial plant, for instance, may contain multiple machines and the machines may have an electric motor with many smaller parts like rotor and stator located inside. Since the objects occlude each other, only few are visible from outside. This work presents a new technique, Spherical Visibility Sampling (SVS), for real-time 3D rendering of such – often highly complex – scenes. SVS exploits the occlusion and annotates hierarchically structured objects with direction-dependent visibility information in a preprocessing step. For different directions, the direction-dependent visibility encodes which objects of a scene's region are visible from that direction from the outside of the regions' enclosing bounding sphere. Since there is no need to store a separate view space subdivision as in most techniques based on preprocessed visibility, a small memory footprint is achieved. Using the direction-dependent visibility information for an interactive walkthrough, the potentially visible objects can be retrieved very efficiently without the need for further visibility tests. The evaluation shows that using SVS allows to preprocess complex 3D scenes fast and to visualize them in real time (e.g., a Power Plant model and five animated Boeing 777 models with billions of triangles). The comparison with two state-of-the-art occlusion culling algorithms demonstrates the advantages and disadvantages of SVS. Because SVS does not require hardware support for occlusion culling during rendering, it is even applicable for rendering complex scenes on mobile devices.

Zusammenfassung

Viele 3-D-Szenen (z. B. aus CAD-Daten generierte) sind aus einer Vielzahl von ineinander verschachtelten Objekten aufgebaut. Eine Fabrik kann beispielsweise einige Maschinen beinhalten, wobei die Maschinen einen Elektromotor besitzen können, der wiederum kleinere Teile, wie Rotor und Stator, einschließt. Da sich die Objekte gegenseitig verdecken, sind nur wenige von außen sichtbar. Diese Arbeit präsentiert ein neues Verfahren, Spherical Visibility Sampling (SVS), für die Echtzeitdarstellung von solchen, häufig hochkomplexen, 3-D-Szenen. SVS nutzt die Verdeckung aus und reichert in einem Vorverarbeitungsschritt die hierarchische Objektstruktur um richtungsabhängige Sichtbarkeitsinformationen an. Diese Sichtbarkeitsinformationen beinhalten für verschiedene Richtungen, welche Objekte eines Szenenteils von außerhalb der umschließenden Kugel dieses Teils aus dieser Richtung sichtbar sind. Im Gegensatz zu den meisten auf vorausberechneter Sichtbarkeit basierenden Verfahren wird auf eine Unterteilung des Betrachterraums verzichtet, wodurch ein geringer Speicherplatzbedarf erreicht wird. Zur 3-D-Darstellung können die potenziell sichtbaren Objekte sehr effizient aus den richtungsabhängigen Sichtbarkeitsinformationen ohne zusätzliche Sichtbarkeitstests abgerufen werden. Die Evaluierung zeigt, dass SVS die effiziente Vorverarbeitung und Echtzeitdarstellung von komplexen 3-D-Szenen erlaubt (z. B. eines Kohlekraftwerks und fünf animierter Boeing 777-Modelle mit Milliarden von Dreiecken). Der Vergleich mit zwei aktuellen Verfahren zur Verdeckungsberechnung demonstriert die Vor- und Nachteile von SVS. Da SVS für die Verdeckungsberechnung zur Laufzeit keine Hardwareunterstützung benötigt, kann es auch zur Darstellung von komplexen Szenen auf Mobilgeräten benutzt werden.

Contents

Algorithms, Tables, and Figures

List of Algorithms

List of Tables

List of Figures

1 Introduction

Computer graphics are used in many areas of today's life. For example, the entertainment industry uses 3D rendering in video games and to produce computer-animated films. Commercial advertising uses computer graphics in television spots and illustrations in magazines. In video games, the 3D content is specifically designed to enable real-time rendering on different target platforms that are known in advance. No real-time rendering is required to produce the images for films or advertisements. Computer-aided design (CAD) software is used by the industry to design new products. For instance, a car is virtually designed before it is constructed. Another example is the architectural visualization that is used to present a virtual building before the real building is built. To present those products to people that are not accustomed to interpret CAD drawings, virtual design reviews are performed. In a virtual design review, an interactive 3D visualization of the product is created by using the CAD data. Often, virtual reality techniques like motion tracking and powerwalls are used to increase the immersion of the viewer. This is useful, because the real-time visualization of a complex CAD model can help a user to better understand the huge amount of information contained in that model [KBF05]. A *walkthrough* allows a user to move freely in the scene in real time by interactively changing the position and orientation of the camera that represents the view shown on the screen. This allows the user to spontaneously decide which parts of the scene are most interesting and shall be shown. In this setting, real-time 3D rendering is required to allow the interactive navigation. In contrast to other fields of computer graphics, like the ones mentioned before, the 3D data is not specifically created for the 3D visualization, which presents a challenge for the real-time rendering. Instead of graphics artists designing 3D content merely for the rendering, engineers create the CAD data to design a new product.

For *real-time rendering*, several frames per second have to be displayed [AHH08, Chapter 1]. The temporal sequence of images on the screen generates a fluent impression of the animations and allows the user to react timely while navigating. When the frame rate is very low, e.g., below 5 frames per second, an interactive navigation is hardly possible, because the latency from an user's input to the visual feedback is too high. With a rate of 5–10 frames per second, the navigation is possible, even though a stuttering in the animation can be noticed. The images are created by the graphics hardware that renders the data of a scene. The capabilities of graphics hardware have increased in the past years and new technical features like programmable shaders have opened up new fields of application (refer to Blythe [Bly08] for an overview of the development of graphics hardware over time). Still, the graphics hardware imposes an upper limit on the amount of data that can be processed in a fixed period of time: There is a linear relation between the scene size and the rendering time [HG94]. On the one hand, with the increasingly powerful graphics hardware, the complexity of scenes that can be rendered in real time becomes higher and higher. On the other hand, the data of complex scenes is larger than the amount that can be processed by modern graphics hardware in real time. Scenes with such a high geometric complexity often originate from CAD data or laser scans of real-world objects. The complexity results in a large amount of data that requires billions of bytes of memory. Therefore, the data handling and rendering of highly complex scenes in real time remains a fundamental problem in

computer graphics.

In order to render multiple frames per second, the data of a complex scene cannot be sent completely to the graphics hardware. The visibility in a scene is exploited to reduce the amount of data that will be sent to the graphics hardware at runtime. Culling algorithms detect scene parts that are occluded and cull them from the data that will be rendered. Online occlusion culling algorithms perform this procedure during the walkthrough. They are suited for densely occluded scenes, but suffer from additional overhead entailed by the visibility tests at runtime when too many objects are actually visible. Often, these algorithms rely on hardware-assisted occlusion queries (see Section 2.1.1) that are not available on all devices (like mobile phones or tablet PCs). Preprocessed visibility techniques analyze the scene data in a preprocessing step to determine and store the visibility information. Common techniques based on cell-based preprocessed visibility involve almost no runtime overhead leading to very efficient rendering. But, if the scene's space cannot be clearly split into a relatively low number of discrete view cells, those techniques suffer from high memory consumption and long preprocessing times, which effectively restricts the possible scene size.

The goal of this work is the development of a real-time rendering algorithm for complex 3D scenes that performs occlusion culling. Spherical Visibility Sampling (SVS), a novel rendering approach introduced in this work, is based on preprocessed visibility that overcomes several of the aforementioned limitations. In contrast to cell-based approaches, it does not require an explicit view space subdivision and allows the processing of large and spacious scenes. Contrary to online occlusion culling algorithms, SVS does not require occlusion queries during runtime. Nevertheless, it can be used to render highly complex 3D scenes.

The SVS algorithm achieves its acceleration mainly from the following observation. Objects in CAD data scenes are often nested: highly complex objects contain groups of smaller nested objects, which in turn contain even smaller nested objects. For example, the buildings of an industrial plant model enclose many chambers, which in turn enclose objects like tubings. Another example of a nested object is an aircraft model consisting of wings, turbines, and a passenger cabin, which in turn contains smaller nested objects like seats. Furthermore, the set of visible objects depends on the viewing direction (e.g., the left wing is not visible from most positions on the right). Hence, for outside positions of these object groups, usually only a fraction of nested objects is visible.

To take advantage of this observation, SVS identifies the nested objects that are visible from outside a group of objects in a preprocessing step. For the walkthrough, only these objects have to be displayed for outside positions. For each group of objects, a set of visible objects is computed for multiple viewing directions by a sampling method, and is hierarchically stored.

1.1 Outline of the Work

In the following, an overview of the structure of this work is given. Following this chapter, some fundamental definitions and an overview of related work is given (Chapter 2). Because SVS consists of two parts, subsequent, these parts are described in their own chapters:

SVS's preprocessing step (described in Chapter 3) requires a hierarchical data structure that stores the scene's objects. An inner node of the data structure is the root node of a subtree that represents a part of the scene. An illustration of the preprocessing for a scene part with eight objects belonging to the subtree of an inner node is shown in Figure 1.1. The *direction-dependent visibility* is computed for each inner node of the data structure. This visibility information will be

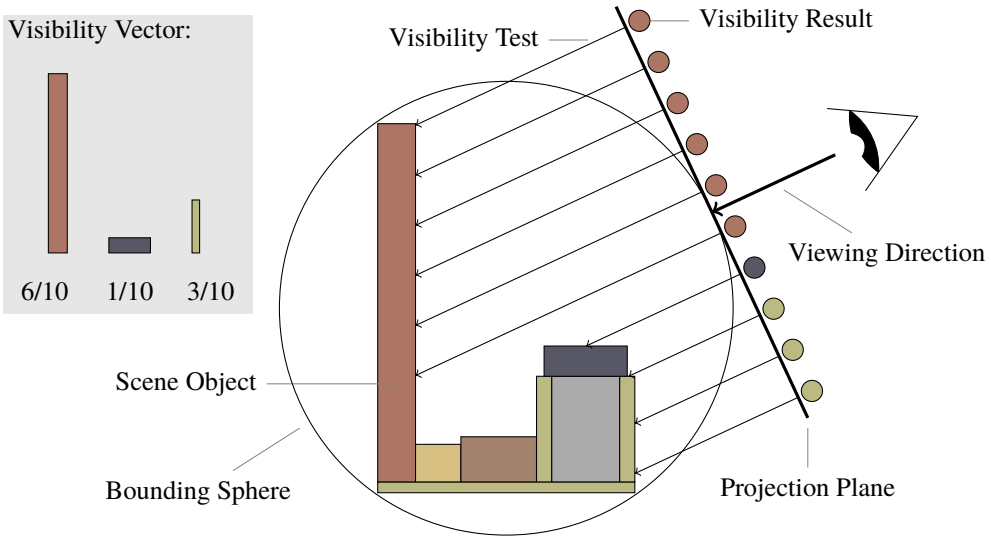


Figure 1.1: Illustration of visibility sampling for a single direction.

valid when the camera position is located outside of the node's bounding sphere. Therefore, at first, the node's bounding sphere has to be computed by taking the objects in the node's subtree into account. Depending on the viewing direction onto a node, the set of visible objects might change significantly. A viewing direction onto the objects corresponds to a point on the surface of the bounding sphere around these objects. To approximate the complete visibility information, SVS uses multiple sample points on the sphere surface for visibility testing. For each sample point, the set of visible objects is computed. For this visibility testing, a camera is positioned at the sample point on the bounding sphere's surface looking towards the center of the sphere. Then, the objects are rendered with an orthographic projection and the visibility is determined based on the pixels in the resulting image. Using this result, a visibility vector containing the information on how many pixels every object contributes to the image is created. The bounding sphere and, for all viewing directions, the corresponding visibility vectors are stored at the node in the hierarchical data structure. The preprocessing step finishes when all inner nodes have been processed.

During runtime, the hierarchical data structure is traversed beginning at the root node with SVS's rendering algorithm (Chapter 4). When the traversal reaches a node whose bounding sphere is currently seen from the outside, the node's stored visibility information can be used. The vector from the current camera position to the center of the bounding sphere defines the current viewing direction. As the stored data contains only visibility information for a fixed set of viewing directions, the neighboring directions of the current viewing direction are taken into account to create a set of potentially visible objects. This set of objects is rendered to create an image of the current subtree's objects for the current viewing direction.

The algorithms presented in the aforementioned two chapters are experimentally evaluated to analyze their function and performance (Chapter 5). As one result, a set of values for the parameters of SVS are provided. With these values, SVS can be used in practice with very little visibility errors and a high performance concerning running time and storage space. The evaluation shows that SVS is able to preprocess and render complex scenes with billions of

triangles. Another result is the analysis of areas in which SVS provides the best results. It gives an idea on where SVS is superior to other rendering algorithms and where it should thereby be applied for practical scenarios.

Finally, the work is concluded and an outlook on future work is given (Chapter 6).

1.2 Main Contribution

The main advantages of this work are summarized in the following:

Fast and memory-efficient preprocessing. Despite the fact that SVS is a preprocessed visibility algorithm, no subdivision of the view space is performed. Instead, the scene's objects are annotated with direction-dependent visibility information (further details in Chapter 3). Therefore, no view cells have to be constructed and stored. Even highly complex scenes can be processed in a short time and stored with a small memory overhead.

Real-time rendering on a variety of devices. During runtime, SVS does not need hardware support for occlusion culling and has hardly any overhead (Chapter 4). In addition, rendering fulfilling a budget constraint (see Section 4.3) is possible. Thereby, besides the rendering on powerful workstation PCs, SVS supports rendering with occlusion culling on mobile devices.

Rendering of complex moving objects. Another challenge for region-based methods are dynamic scenes with moving parts (if those are not treated separately). Every change requires rebuilding the data structure or at least a special updating process (e.g., [Bit+09]). SVS also does not allow fully dynamic scenes, but the movement of independent subtrees of arbitrary complexity is easily possible at runtime (see Section 4.4). The evaluation demonstrates that SVS is able to render highly complex plane models while they are flying through a scene.

Usability in practice. The algorithms that constitute SVS were implemented in PADrend[1] and are ready to use (see Section 5.1). A user can load and preprocess a scene with only little effort. After the preprocessing, SVS's renderer can easily be added to the rendering pipeline to enable an interactive walkthrough.

Spherical Visibility Sampling has been published in Computer Graphics Forum [Eik+13] and presented at the 24th Eurographics Symposium on Rendering. The published article contains the description of SVS together with an evaluation. Compared to the article, the description in this work is much more detailed. The evaluation was greatly enlarged and much more aspects are covered by the evaluation in this work. Additionally, SVS's budget rendering feature has been improved since the publication of the article. The version contained in this work is superior to the one used for the article.

[1] http://www.padrend.de/

2 Related Work

This chapter gives an overview of other works that are similar to this work and describe the state of the art. An introduction into rendering, as used in this work, and visibility culling is given in Section 2.1. Online occlusion culling algorithms, which perform the visibility tests at runtime, are discussed in Section 2.1.1. Techniques that use preprocessed visibility, sometimes called offline occlusion culling algorithms, are covered in Section 2.1.2. Rendering techniques that make use of samples on a sphere surface are mentioned in Section 2.2. Section 2.3 covers different ways of allowing a user to control the image quality by limiting the rendering time. Different rendering systems for mobile devices are shortly described in Section 2.4. At last, in Section 2.5, some characteristics of SVS are listed and SVS is classified based on the taxonomy of two surveys.

2.1 Visibility Culling

In this work, the term *primitive* denotes geometric primitives like points, lines, and polygons. The most important primitive for this work is the polygon with three corners: the triangle. The primitives are defined by *vertices* that are element of the three-dimensional Euclidean space $\mathbb{R}^3$. Rendering is performed by rasterization [Wat00, Chapter 6] using today's graphics hardware. Visibility is finally determined by the graphics hardware using the z-buffer algorithm. The z-buffer algorithm [Str74; Cat74; Mac01] decides by depth comparison which primitive will be used to fill a pixel on the screen. There is an upper bound for the number of primitives that the graphics hardware is capable of rendering in real time. If the number of primitives defined by a complex scene exceeds this bound, and all primitives are sent to the graphics hardware in a brute-force manner, the frame rate will be too low for an interactive walkthrough. Therefore, different kinds of visibility culling algorithms have been developed to reduce the amount of primitives that is sent to the graphics pipeline. The illustration in Figure 2.1 shows an example of these visibility culling algorithms that are described in the following. The survey of Cohen-Or et al. [Coh+03] gives an overview over visibility culling algorithms, and especially over occlusion culling algorithms.

Back-face culling Back-face culling detects the primitives that face away from the camera and are seen from behind (see the survey by Sutherland et al. [SSS74]). This is reasonable only for scenes with single-sided primitives. Back-face culling can be executed efficiently by examining the angle between the primitive's normal and the viewing direction. Therefore, it is a standard operation in the graphics pipeline and can be executed by the graphics hardware. The scenes used in this work use triangles to model single-sided surfaces. Hence, back-face culling is always applied in the scope of this work.

View-frustum culling View-frustum culling [Cla76] is executed to skip the primitives that are outside of the view frustum for rendering. When using a camera with perspective projection, the *view frustum* is a frustum of a rectangular pyramid. For an orthographic projection, the view frustum is a box. View-frustum culling can be performed by intersection tests between the primitives and the six planes of the view frustum. If a primitive is not fully

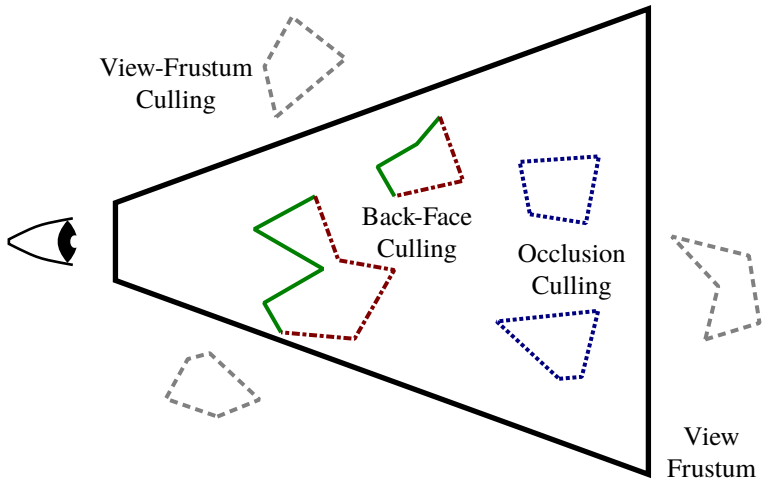


Figure 2.1: Different kinds of culling algorithms shown in a cut through an example scene: view frustum (thick black line), primitives that are visible (solid green lines), primitives that are removed by back-face culling (dashed-dotted red lines), by view-frustum culling (dashed gray lines), and by occlusion culling (dotted blue lines) [inspired by Coh+03].

contained in the frustum and does not intersect any plane, it can be skipped. Nowadays, view-frustum culling is usually not executed for single primitives, but for bounding volumes. The bounding volumes are stored for the scene's objects in the nodes of a scene graph to avoid costly computations over and over again. Furthermore, there are more efficient ways to execute the frustum test than the one that was just described. For this work, the basic intersection test by Assarsson and Möller [AM00] is used that tests only two corners of an axis-aligned bounding box against the frustum.

Occlusion culling Primitives that are inside the view frustum, but are occluded by other primitives, do not need to be sent to the graphics pipeline. The task of occlusion culling algorithms is to detect those primitives. As for the view-frustum culling, nowadays, the occlusion is not determined for single primitives, but for objects that consist of multiple primitives. This is done for efficiency reasons, because testing billions of primitives for occlusion or storing preprocessed visibility information for them is too costly. On the one hand, occlusion queries can be executed at runtime (see Section 2.1.1). On the other hand, occlusion culling can be performed by analyzing the scene in a preprocessing step to produce preprocessed visibility information (see Section 2.1.2).

The usage of occlusion culling is worthwhile only for scenes with high depth complexity. The term depth complexity is used in this work as defined by Sutherland et al. [SSS74]:

> "The *depth complexity* is a measure of how many front faces are pierced, on the average, by an arbitrary ray from the viewpoint. If the environment is composed of a large cube standing in front of a back-drop face, the depth complexity would be nearly 2. If the depth complexity of a scene is 1 throughout, the hidden-line or hidden-surface problem is trivial; all relevant faces and edges are visible. As the depth complexity increases, so does the difficulty of rendering the environment."

If occlusion culling is applied to render a 3D scan of a statue, the rendering performance will likely be decreased, because the depth complexity is very low and there are very few occluded parts. For rendering architectural models with high depth complexity, like a house or a production plant, occlusion culling will lead to a performance gain. Almost always, the decision about the depth complexity of a scene – and as a consequence about the usage of occlusion culling – is taken by an expert that manually analyzes the scene. However, there are methods [Jäh+13] to automatize this process, or at least give guidance to shorten the manual process.

Each of the following two sections characterizes a large class of culling algorithms, respectively: Online (Section 2.1.1) and offline (Section 2.1.2) occlusion culling algorithms.

2.1.1 Online Occlusion Culling

Online occlusion culling algorithms perform visibility tests during the walkthrough of the scene. Because they do not require the analysis of the visibility of the scene beforehand and do not have to store this data, they overcome the restrictions of limited dynamics, preprocessing time and memory overhead that are shared by many preprocessed visibility techniques. Many current techniques (e.g., CHC [Bit+04], NOHC [GBK06], CHC++ [MBW08]) mostly rely on hardware-assisted occlusion queries, and may thereby entail an additional runtime overhead for performing the queries. The overhead is negligible and high performance can be achieved when only few queries have to be performed and much geometry can be culled. But, the additional overhead will decrease the performance, if many objects are actually visible and still many queries have to be performed. Another problem is that currently, hardware-assisted occlusion queries are not available on mobile devices; making the corresponding techniques not applicable there (see Section 5.16 for further information). In contrast to this, older techniques create a hierarchy of images, for example the hierarchical z-buffer [GKM93] or hierarchical occlusion maps [Zha+97], that allow to perform occlusion queries on the CPU. Because these are occlusion queries like in the hardware-accelerated techniques, there is also an runtime overhead. Furthermore, an image hierarchy has to be built as a helper structure to allow the execution of occlusion queries. Therefore, the techniques using hardware-assisted occlusion queries are usually faster than the older techniques. Additionally, the implementation of the new techniques is often simpler, because the implementation of the occlusion queries is hidden in the graphics library. Since there are situations in which online occlusion culling algorithms suffer from a high runtime overhead, they cannot always be applied. Visibility techniques that have access to preprocessed visibility data can deliver superior performance in these cases.

2.1.2 Preprocessed Visibility

Using precomputed visibility information for rendering scenes with high depth complexity is a well known practice. The theoretical bounds for the size needed to store the complete visibility information have been given by the aspect graph [PD90]: the maximum size of the aspect graph for a nonconvex polyhedral scene with n faces under perspective projection is $\mathcal{O}(n^9)$ and its construction time is $\mathcal{O}(n^9 \cdot \log n)$. Clearly, this complexity is much too high for real-world scenes. Nevertheless, there are many techniques that make use of preprocessed visibility to accelerate the rendering process at runtime. The visibility information acquired by the techniques is an approximation of the exact visibility information. Most techniques use a subdivision of the view space into discrete regions. Such a region is often called *view cell*. Each view cell is associated with the parts of the scene which are potentially visible from within its region. This information

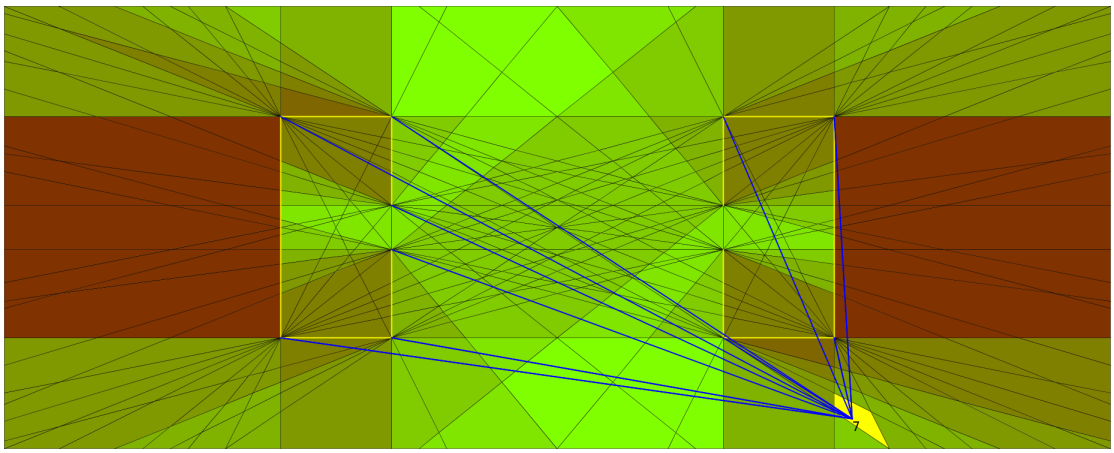


Figure 2.2: Arrangement of ten line segments (yellow) with 618 faces induced by visibility events (indicated by the black lines). The faces are colored depending on the number of visible segments: red means few visible segments, green means many visible segments. For the selected face (yellow), the blue lines point to the seven visible line segments for this face. The image has been created with a tool by Matthias Hilbig.

is stored as a *potentially visible set (PVS)*. A 2D example with a scene consisting of line segments is shown in Figure 2.2.

The visibility calculation can be performed analytically and the visibility information can be stored in a special data structure (e.g., viewpoint space partition (VSP) [PD90], view cells and their cell-to-cell visibility [TS91], visibility skeleton [DDP97]). Other techniques acquire their visibility information by sampling [Hua+02; SHT03; LSC03; NB04; Lai05; MBW06; Won+06; Mat+07; Bit+09]. Usually, the sampling-based techniques are faster and can be used for larger scenes than the analytical ones.

Most of the visibility techniques differ in the way the view space is subdivided into view cells. The sampling techniques use different methods for testing the visibility of the scene parts (geometric visibility using ray casting, or image-based visibility using the z-buffer algorithm of the graphics hardware). One important aspect, especially for the running time of the preprocessing, is the distribution and number of samples. Again, refer to the aforementioned survey [Coh+03] for an overview of different preprocessed visibility techniques (also see Section 2.5 for a classification of SVS).

Using preprocessed visibility in practice tries to approximate the visibility information that is stored by the aspect graph. As most preprocessed visibility techniques use view cells, the subdivision of the view cells inherently depends on the visibility that is dictated by the scene's geometry. This can be problematic, especially for spacious outdoor scenes. Imagine, for instance, a scene that consists of two houses that stand far away from each other. For this example, the view cells for the inside of the houses are ignored. Between the two houses, there is a large empty space. When a viewer moves through this space in between, the visibility of the scene parts might change significantly. For example, when the viewer's height above ground is increased, she can look through the windows of the top floor instead of the windows of the first floor. Now, there are two possibilities: Either, the space is subdivided into few large cells that store rather inaccurate visibility information. This would need only little memory, but would not accelerate the rendering much. Or, the space could be subdivided into many small cells, as dictated by the

visibility changes. The small cells would contain exact visibility information, but also need a lot of memory. This illustrates that there is always a trade-off between storage space and accuracy of the visibility information. As a preprocessed visibility technique, SVS is also subject to this trade-off. But, it has advantages for spacious scenes. When looking again at the example with the two houses, one would store each house in a separate subtree of the scene graph. By using each house's bounding sphere, SVS would compute and store the visibility information. With the help of the information stored in the two spheres, the rendering can access the visibility information for all positions outside of the houses. The two houses could even be moved away from each other, without the need to perform any updates to the stored visibility information.

Common to all region-based techniques is that the complexity of the calculated data structure as well as the needed preprocessing time does not only depend on the complexity of the scene's geometry, but also on the chosen or resulting number of regions. The need for regions thereby limits the complexity of scenes that can be practically preprocessed, and hinders the usage for spacious scenes. In contrast, SVS exploits the hierarchy of an existing spatial data structure that stores the scene (e.g., an octree [Hun78], a *k*d-tree [Ben75], or an R-tree [Gut84]). It does not calculate from-region visibility [Coh+03] for a cell, but direction-dependent visibility for the outside of a bounding sphere that encloses a node of the existing scene data structure. Therefore, it does not have to create view cells for empty regions of space and works well for spacious scenes.

An option to overcome the large storage space required by preprocessed visibility techniques are compression algorithms for visibility data. Lossy and lossless compression techniques for visibility data are presented by van de Panne and Stewart [vS99]. Applying them to a set of view cells results in much smaller storage space at the cost of additional operations needed to access the data. SVS does not use view cells, but also stores visibility information. Because the memory needed by SVS's visibility data is not that large (see Section 5.9), a compression of the data is not essential. If SVS's memory consumption should be reduced, an adjusted version of the compression technique could be applied to the visibility data as well.

2.2 Spherical Sampling in Computer Graphics

In the field of computer graphics, the distribution of random or structured samples on a sphere surface is used in different areas. Among other things, spherical sampling is applied for texture mapping and remeshing [PH03], shadowing for ray tracing [BRA06] and rasterization [LYX08], and for lighting calculations for ray tracing [Gai+10].

The idea of using sphere surfaces for visibility sampling has been used to determine the visibility of a single mesh's geometric patches [MS01; Mer02]. This technique groups the mesh's polygons into patches to increase the reliability of the image-based sampling and to decrease the required running time and storage space compared to direct usage of the polygons. The visibility of these patches is determined for different positions on a sphere around the object. The camera positions are vertices of an edge-subdivided sphere beginning with an octahedron. The camera looks in the direction of the object's center and is placed such that all parts of the object are visible. A unique identifier is assigned to each patch and a color value is used to encode this identifier. The colors are used for rendering the patches, and the resulting image is scanned for the colors to determine the identifiers of the patches that are visible. The visible patches are associated with the direction from which the object is seen. By using this directional visibility information at runtime, back-face culling and self-occlusion of a single object can be determined efficiently. The

visibility information relevant for the current camera position is found by traversing a hierarchy that is built over the faces of the sphere subdivision. The corners of the face that intersects the viewing ray are determined and their visibility information is fused.

SVS is based on the same general idea of exploiting the visibility of parts of a scene as seen from the outside. In contrast, it is not focused on the visualization of a single complex object, but on the visualization of complex scenes composed of many objects. It does not build new patches or objects from the polygons of the scene, but makes use of the objects that are usually already defined by the scene description. The approach to search for the neighboring viewing directions at runtime and unite their visibility data, is also applied by SVS.

2.3 Budget Rendering

In the following, several works are presented that allow the user to constrain the rendering algorithm by setting a time or resource budget. This feature is used to allow the user to trade running time off image quality. Sometimes, this adjustment is done automatically: While the user moves through the scene, the image quality is decreased to allow an interactive navigation with fast reaction to the user input. When the user stops, no fast reaction to the user input is required anymore, and the image quality is increased at the cost of a longer rendering time. Using a time budget to limit the time that a rendering algorithm is allowed to use is sometimes called time-critical computation [BJ96].

In the work of Funkhouser and Séquin [FS93], a target frame time can be specified by the user. The target frame time is used to bound the overall time that is available for rendering. They propose a cost heuristic to estimate the time needed to display level-of-detail representations of an object using different rendering algorithms. Together with a benefits heuristic, an optimization problem is formulated that is to be solved to select the representations and algorithms.

For a parallel rendering system that uses ray tracing, Reisman et al. [RGS97] use a user-defined frame rate together with a variable image quality. A load balancing algorithm is presented that manages the work of the parallel processors and computes the times that they are allowed to use. To be able to stop the rendering at a time computed by the load balancing, they use a progressive ray tracing scheme.

Klosowski and Silva [KS99] present a rendering system that allows the user to define a polygon budget to indirectly constrain the frame rate. The rendering system estimates if polygons are visible and tries to render the visible ones before rendering the occluded ones. It stops when the polygon budget is reached.

To render a scene using the HDoV tree [SHT03], a polygon budget is distributed inside the tree. Beginning with the full budget at the root node, the budget is distributed to the child nodes based on the estimated degree of visibility (DoV) computed beforehand and the number of polygons in the child's subtree. The budget is used to select a fitting level-of-detail representation for the objects during the tree traversal for rendering.

Refinement techniques (e.g., [Ber+86; LH91]) are also some kind of budget rendering techniques, because they trade running time for image quality. At first, a very coarse image is created and shown to the user as fast as possible. Then, while the user does not change the view, the image quality is progressively improved. The refinement can be achieved by using a cheaper algorithm first, e.g., flat shading, and using higher quality algorithms step by step, e.g., a complex material shader for the final image. Another way is to vary the sampling density, e.g., for progressive ray tracing: For the coarse image, a single sample per pixel is used, and the number of samples is

successively increased.

2.4 Rendering on Mobile Devices

In literature, several rendering systems for mobile devices have been presented. Some mobile rendering systems, especially those that want to display complex scenes or highly detailed 3D games on mobile devices, render remotely on a server and stream the images to the mobile device. Some of them use image-based rendering on the mobile device to display the remotely rendered images [YN00; CG02; BG04; BG06; BFA06; Jia+06]. Others create a video on the server from the rendered images, stream it to the client over the network, and use a simple video player to display it [NC03].

In contrast to the requirement of a server for rendering, there are mobile rendering systems that render the geometry directly on the mobile device, whereas the scenes are specifically designed and prepared for mobile rendering (e.g., preprocessing takes place on a workstation and rendering on the mobile device [SR09]). Tack et al. [Tac+04] use MPEG-4 compression to encode geometry information into a stream that can be decoded and displayed on a mobile device. Nurminen [Nur06] precomputes PVS-based visibility to allow interactive 3D rendering on a mobile device. Additionally to the limited hardware resources of the mobile device itself, the small bandwidth of mobile networks can be a problem. Special systems that improve the data transmission over the network have been developed (e.g., [Nur07]).

SVS provides rendering of complex scenes (compared to the capabilities of the mobile graphics hardware) with the support for occlusion culling on mobile devices. The preprocessing is performed on a workstation beforehand. The resulting data structure containing the visibility information can either be obtained via network, or it can be copied to a mobile device's internal storage together with the scene. In the second case, network transmissions become no longer necessary.

2.5 Classification of SVS

In the following, SVS is classified on the basis of the taxonomy created by two surveys. This is done by looking at different characteristics of SVS. The classification makes it possible to show similarities of SVS with other algorithms of the same class and to highlight differences.

A good summary of different visibility algorithms together with a classification into different categories and a brief description of the historical development is given by Cohen-Or et al. [Coh+03]. According to their taxonomy, SVS can be classified as an occlusion culling algorithm that is from-region, uses image precision, supports generic scenes, computes approximate visibility, uses structured sampling, uses all occluders, supports generic occluders, supports occluder-fusion, works in 3D, and requires preprocessing. The term "from-region" does not describe SVS precisely, because SVS does not compute visibility for regions the camera is currently contained in, but it computes direction-dependent visibility for regions that do not contain the camera position. SVS's visibility information computed for the geometry inside a sphere is valid for all viewpoints outside of the sphere. Therefore, according to the survey, it performs bulk computations that are valid anywhere outside the sphere volume, which makes it a from-region technique, but the term "to-region" is more suitable. In the current implementation, occlusion queries are required for the visibility tests during the preprocessing, but there are no special requirements at runtime. Occlusion queries do not require special hardware and are supported on

nearly all workstations with standard graphics hardware, but they are not supported on common mobile devices. Dynamic scenes are partly supported by SVS (see Section 4.4 for a description how animated objects are handled).

When looking at the different visualization techniques for massive models [GKY08], SVS can be classified as a data reduction technique that applies visibility culling. SVS uses rasterization with z-buffering and is a from-region algorithm that precomputes potentially visible sets. Gobbetti et al. [GKY08] state that preprocessing and rendering of scenes consisting of massive models is hard for preprocessed visibility techniques. They also mention the problem of deciding about the granularity of the view cell subdivision to trade the running time and storage space off the quality of the visibility information. Unlike other precomputed visibility algorithms, SVS does not use view cells and is able to process and render massive scenes consisting of billions of triangles and millions of objects.

3 Preprocessing

SVS performs a preprocessing step to determine and store the visibility inside a scene. For this purpose, a scene graph that describes the scene is required. If the input scene does not define such a scene graph, it is created from the input data (Section 3.1). Then, for each inner node, a tight bounding sphere is computed (Section 3.3). On the surface of the bounding sphere, multiple sample points are distributed (Section 3.4). A sample point corresponds to a viewing direction onto all the objects in the node's subtree, for which the visibility is determined. The direction-dependent visibility information for that node's subtree is stored in the scene graph as annotation of the node (Section 3.5).

3.1 Preparation of the Hierarchical Data Structure

The smallest entity that is considered by SVS is an *object*. An object consists of a triangle mesh and optional additional properties like textures, material definitions, shader programs, etc. SVS does not make further demands on the properties of a mesh. Even the restriction on triangle meshes is not directly demanded by SVS, but due to the requirements by the rendering pipeline (see Section 3.2).

SVS takes a 3D scene consisting of multiple objects arranged in a spatial hierarchical data structure as input. For SVS, this data structure is always a tree. The spatial hierarchical data structure will be called *scene graph* in the following.

If the input scene does not already consist of objects, but contains only a triangle soup, a spatial data structure is used to partition the scene's triangles. For this, the triangles are inserted into this spatial data structure. Depending on the input data, an expert has to decide which data structure should be used in particular (octree, kd-tree, R-tree, etc.) to create spatially compact objects with little overlap. When such a tree structure has been built and holds all triangles, a mesh can be built for each tree node by collecting the triangles stored in the node. For each mesh, an object will be created.

If the scene defines objects, but does not provide a spatial hierarchical tree structure, all objects are inserted into a loose octree [Ulr00]. In the resulting tree, the objects are contained in the leaf nodes. The inner nodes of the tree group multiple nodes together. The objects as well as the inner nodes store an axis-aligned bounding box. For an object, the bounding box is computed from the object's vertices. For an inner node, the bounding box is the union of the bounding boxes of the child nodes. These bounding boxes are used for frustum culling at runtime (see Section 4.1). The scene graph forms a bounding volume hierarchy (BVH), where a node's bounding volume contains all bounding volumes of the node's children.

3.2 Efficiency Considerations for Rendering

Owing to the peculiarities of modern graphics libraries, a mesh should have a reasonable size to be appropriate for a rendering system. In former times, it was a frequent practice to send

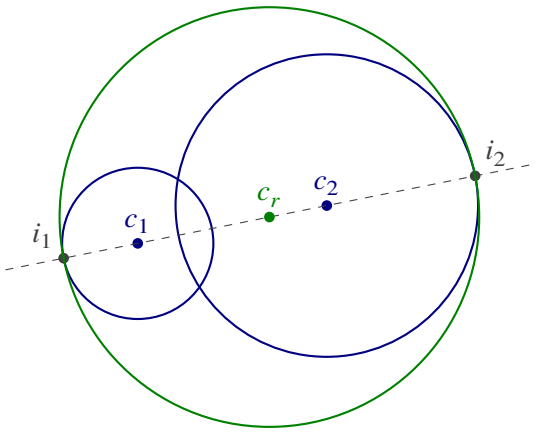


Figure 3.1: Resulting sphere with center c_r (shown in green) containing the spheres with centers c_1 and c_2 (shown in blue).

single vertices of a mesh to the graphics pipeline. The immediate mode, which allowed the direct processing of single vertices by multiple function calls per vertex, has been deprecated with OpenGL 3.0 [SA08] and removed from the OpenGL 3.2 core profile [SA09]. Since then, arrays containing multiple vertices have to be used to send geometry data to the graphics pipeline. Because there is an overhead in creating and switching these arrays, they should not be too small in order to saturate the graphics hardware while rendering. Concerning the efficiency, a reasonable size of a mesh depends on the graphics hardware. For modern workstation graphics cards, experience shows that the meshes should consist of a few thousand triangles.

The possibility to use quadrilateral and polygon primitives was also dropped in the course of renewing OpenGL with version 3. Therefore, when sending a mesh describing a surface to the graphics pipeline, it has to be composed of triangles. Meshes with polygons with more than three vertices have to be triangulated in advance. For that reason, SVS requires triangles meshes as input.

As for other culling techniques, a mesh should preferably be spatially compact. If a mesh contains triangles that spread over the whole scene, it will be visible from many locations and the probability that it can be culled is low.

3.3 Bounding Sphere Computation

The following computations require a scene graph that has been built before and that hierarchically organizes the scene's objects. For the inner nodes of the scene graph, a bounding sphere has to be computed. The tree traversal for this computation is performed bottom-up. A node's direction-dependent visibility basically corresponds to a mapping from the points of that sphere's surface to the set of objects visible from these points in orthographic projection. The visibility information is only valid for positions outside the sphere, so that a tighter bounding sphere results in more situations where the sphere can be used for rendering (compare Section 4.1), and thereby leads to a higher rendering performance. For the inner nodes, the bounding sphere encloses all the objects in the node's subtree. Two variants to compute the bounding sphere are implemented.

The first variant computes an inner node's sphere from the combination of the bounding spheres of its child nodes. This is done by iterating over the child nodes and successively combining two

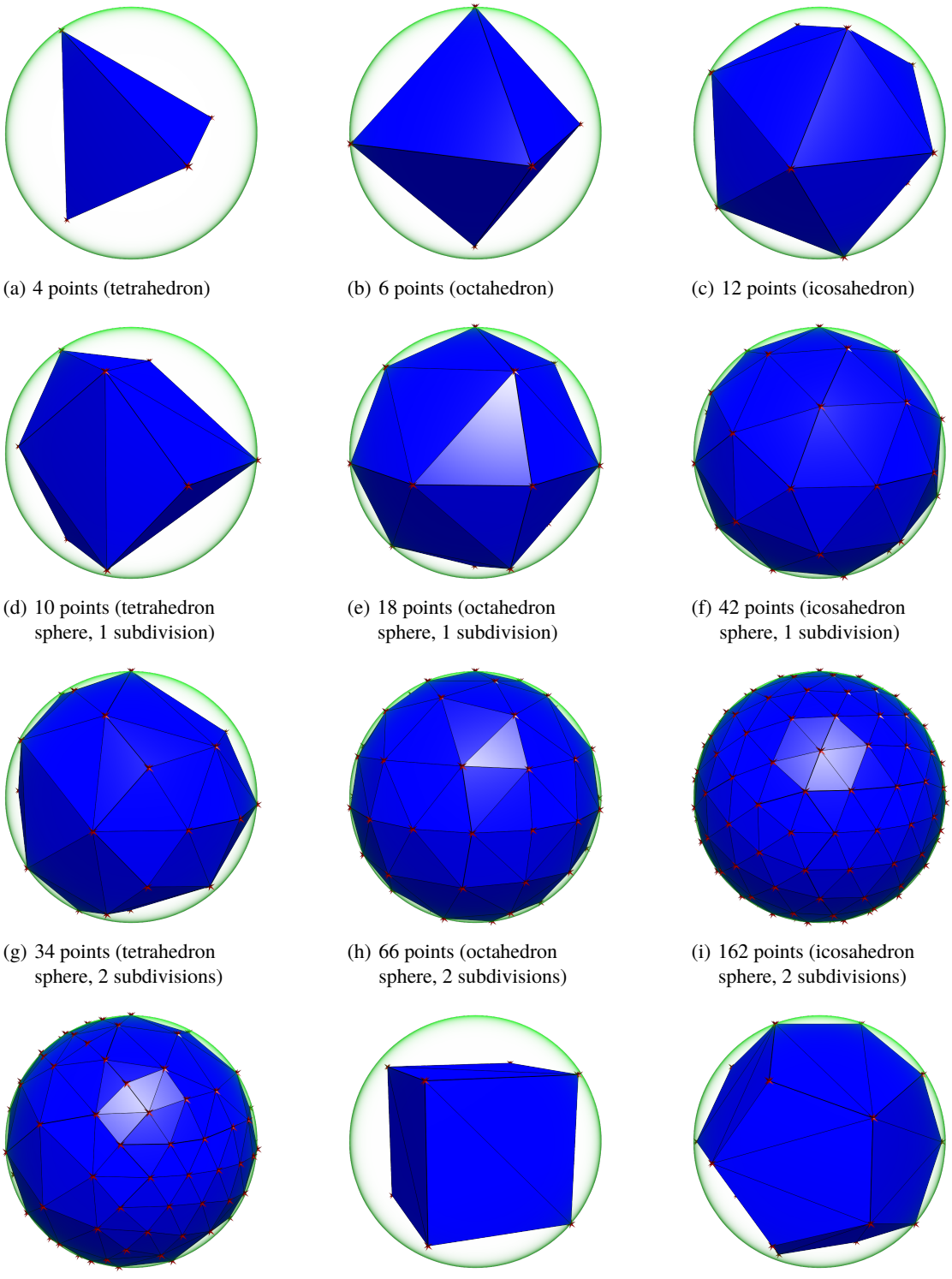
spheres. The spheres that are involved in the calculation are shown in Figure 3.1. To combine two spheres with centers c_1 and c_2, a straight line through those centers is created. The intersection of the straight line with the two spheres is calculated. Only the two intersection points i_1 and i_2 that do not lie on the line segment from c_1 to c_2 are of interest. The center c_r of the resulting sphere lies on the midpoint of i_1 and i_2. The new radius is chosen to include both input spheres and is equal the distance from c_r to i_1. The children of the current node that are inner nodes already have a bounding sphere, because the tree traversal is performed bottom-up. For the leaf nodes, to compute the bounding sphere that encloses the node's object, the vertex positions of the object's mesh are used. These vertex positions are given to the Extremal Points Optimal Sphere (EPOS) algorithm [Lar08] to efficiently compute the sphere that tightly encloses the objects' geometry. The EPOS algorithm identifies a set containing 98 extremal points, which is a subset of the input points, along 49 predefined directions. Then, it uses an exact solver, Miniball [Gär99], to compute the bounding sphere for the extremal points. At last, it iterates over the input points and enlarges the computed sphere to contain all points. If the supporting points of the optimal bounding sphere are elements of the set of extremal points, the EPOS algorithm has computed the minimum sphere. The authors report that EPOS-98 – the version that is used here – had a worst case radius increase of 0.05 % compared to the exact solver in their experimental evaluation. Additionally to the sphere that has been created by the by the repeated combination, SVS considers a second sphere: It creates a sphere with the corners of the inner node's bounding box as supporting points. If objects are tightly enclosed by their bounding box, e.g., a cube-shaped building, the bounding sphere created from the bounding box can be very small and sometimes even optimal. In such cases, this bounding sphere can be smaller than the non-optimal bounding sphere created by the repeated combination of spheres. SVS takes the smaller of the two spheres and stores it as bounding sphere for the inner node.

The second variant collects all objects from the subtree below the inner node. Then, it takes the union of the objects' vertex positions as input for the EPOS algorithm. The computed sphere is used as bounding sphere for the node in the end.

The first variant results in slightly oversized sphere volumes for inner nodes, but it greatly speeds up the overall bounding sphere computation step compared to computing the nearly exact bounding spheres, as in the second variant. The evaluation in Section 5.5 shows that the quality of the simple algorithm is not too bad, while requiring less time and memory. But, if enough preprocessing resources are available, the EPOS algorithm should be applied to deliver close to optimal bounding spheres.

3.4 Sample Point Distribution on the Sphere

The arrangement of the triangles stored in a node's subtree defines which of these triangles can be seen from outside of the node's bounding sphere. The sphere surface can be divided into areas, in which the set of visible triangles stays the same. For a sphere with n triangles inside, there can be $\mathcal{O}(n^6)$ many areas on the sphere surface [PD90, nonconvex case under orthographic projection]. Even if the visibility is determined for objects instead of triangles, the complexity is too high for large scenes. In order to approximate the exact visibility, SVS distributes sample points on the sphere surface that correspond to viewing directions. The goal of the distribution is to minimize the maximum distance of any point on the sphere surface to its closest sample point. Thereby, the sphere surface is covered uniformly with sample points, and an arbitrary point is as close as possible to a sample point. This heuristic tries to keep the average sampling error resulting in



(a) 4 points (tetrahedron)

(b) 6 points (octahedron)

(c) 12 points (icosahedron)

(d) 10 points (tetrahedron sphere, 1 subdivision)

(e) 18 points (octahedron sphere, 1 subdivision)

(f) 42 points (icosahedron sphere, 1 subdivision)

(g) 34 points (tetrahedron sphere, 2 subdivisions)

(h) 66 points (octahedron sphere, 2 subdivisions)

(i) 162 points (icosahedron sphere, 2 subdivisions)

(j) 130 points (tetrahedron sphere, 3 subdivisions)

(k) 8 points (cube)

(l) 20 points (dodecahedron)

Figure 3.2: Spheres (green) with different distributions of sample points (red) and their triangulations (blue).

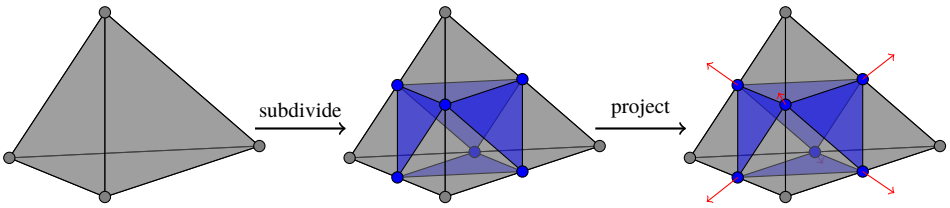


Figure 3.3: Edge-subdivision of a tetrahedron: First, a new vertex is created at the midpoint of each edge. The new vertices are connected to faces. Each face of the original polyhedron is replaced by four smaller faces. The new vertices are then projected and moved to the sphere surface.

inaccurate visible sets small. By using the vertices of a platonic solid as sample points, such a uniform distribution can be created. Unfortunately, the platonic solid with the maximum number of vertices is the dodecahedron with 20 vertices.

In general, there are many different possibilities to distribute many points on a sphere surface [SK97]. In this work, distributions that correspond to platonic solids and that are created by edge-subdivision of platonic solids are used. A subset of these sample point distributions can be seen in Figure 3.2. An illustration of the edge-subdivision of a tetrahedron is depicted in Figure 3.3. To subdivide the edges, new vertices are created on their midpoints. By this step, each original face is divided into four smaller faces. Then, the new vertices are projected from the center of the sphere to the considered sphere surface, and moved to this location on the surface. The subdivision scheme works evenly only for platonic solids with triangular faces, namely tetrahedron, octahedron, and icosahedron, because the resulting faces are triangles again. Instead of subdividing the edges, the faces could be subdivided by placing a new vertex at a face's center. For example, such constructions are used by Dutton [Dut84] for a data structure for global terrain data.

These subdivision schemes go back to the design of geodesic domes. According to Rothman [Rot89] and Hildebrandt and Richert [HR12], the first geodesic dome of this kind was designed by Bauersfeld [Bau25] for a planetarium[1] and patented in 1925. Fuller [Ful54] made the name *geodesic dome* popular and patented the design in 1954. Their construction also starts by subdividing a platonic solid: an icosahedron.

Although SVS works with arbitrary distributions, experimental evaluations (see Section 5.3) show that the vertices of a regular dodecahedron (corresponding to 20 viewing directions) provide a good balance between memory consumption and preprocessing time, on the one hand, and sampling accuracy, on the other hand. This sample point distribution (see Figure 3.2(l)) is used in the following.

An alternative to the static distribution of sample points on the sphere surface could be an adaptive scheme. An adaptive scheme can start with a fixed set of sample points in the beginning. To adapt to the visibility, it has to decide based on existing sample points, if a new sample point should be added. For this decision, it can check if visibility results in two sample points differ by a certain amount. If the difference is large enough, it can create a new sample point between the existing points. A subdivision scheme, similar to the edge-subdivision of a platonic solid, can be used for this. But, an adaptive scheme does not necessarily improve the distribution's quality. For instance, when the visibility results for two sample points are similar, an adaptive scheme

[1] `http://www.planetarium-jena.de/Geschichte.43.0.html`

would decide to stop the subdivision. But, there could be a discontinuity in between that has not been detected. Therefore, a fixed distribution as well as an adaptive distribution would not detect the visibility change when the sampling density is too low.

Another adaptive approach could start with a high density sampling. After the visibility has been determined for a large number of sample points, neighboring sample points could be merged, if their visibility was similar. Unfortunately, this would lead to long preprocessing times, which makes such an approach impractical.

Using randomized sampling does not yield an improvement as well. The random distribution has to be chosen very carefully to reach the aforementioned goals. By choosing such a random distribution, one has to make sure that the sphere surface is covered uniformly. Of course, such a random distribution can be found (e.g., [SB96]). A worst case scenario can be created more easily for a fixed distribution than for a random distribution of sample points. But, with the same number of samples, the randomized distribution is not guaranteed to deliver better results than the fixed one. Furthermore, if a random experiment would create a new distribution for each sphere, intermediate results from child nodes could not be reused in a parent node, because the positions would differ (see Section 3.5 for a utilization of existing results).

To conclude this step, a 3D Delaunay triangulation [Del34] of the sample points is computed and stored. Figure 3.2 shows the triangulations for the sample point distributions depicted there. The triangulation is used later on to determine the three next sample points of the viewing direction. To compute the Delaunay triangulation, the Detri library [Müc98] is used. The sample points together with the center of the bounding sphere are given as input to the Delaunay triangulation algorithm implemented there. The resulting triangulation contains a set of tetrahedrons, where every tetrahedron has the center of the sphere as one corner and one face of the tetrahedron recreates a part of the flattened sphere surface formed by three sample points.

3.5 Computation of the Visibility Spheres

For each inner node, a set of visible objects is determined for every sample point on the bounding sphere of the node. At first, all objects in the node's subtree are projected onto the sample point's tangential plane using orthographic projection (compare Figure 1.1). This is performed by using the standard z-buffer algorithm to fill the depth buffer. In a second step, a separate test for each object counts how many pixels contribute to the depth buffer and are thereby visible (using a hardware-accelerated occlusion query [CG07]). An object is visible, if at least one pixel has been visible during the test for that object. The number of pixels that have been visible for an object are the size of the visible area covered by the object. The set of visible objects, including those values for the visible area, is stored for that sample point as a *visibility vector* $VV = \{(o, v) \mid \text{object } o \text{ is visible and has } v \text{ visible pixels}\}$. The size of the covered visible area represents an object's importance value that is used to sort objects during rendering (for the optional budget rendering feature described in Section 4.3). The data of all sample points of one node's sphere is called the *visibility sphere* of that node.

For the construction of the visibility spheres as just described, the nodes of the tree are traversed bottom-up. When rendering a node's subtree to determine the visible objects, there are two approaches. The first approach collects all objects in the subtree and performs a visibility test for each object. Because the preprocessing for the child nodes has already been completed, the second approach can render only the potentially visible objects by using the information stored in the child nodes' visibility spheres. This is in most cases much faster than rendering all

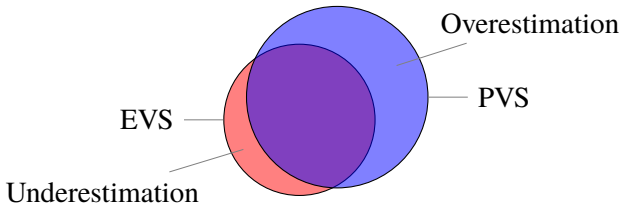


Figure 3.4: Set diagram of the exact visible set (EVS) and the potentially visible set (PVS).

objects in the subtree. For example, evaluating one sample for the Boeing 777 model by testing all objects can be much slower than by re-using the inner visibility spheres (see measurement results in Section 5.6). Re-using the already determined data is possible, because only samples taken from the exact same direction (under orthographic projection) are considered for the new sample. Therefore, no perspective error is introduced in this step. As a possible error, objects that were missing due to undersampling in the child nodes are now surely left out when re-using the previous results, although they might otherwise be included. This access pattern also supports data management routines, like out-of-core algorithms, which can swap out the data that is not needed anymore during the preprocessing.

3.6 Types of Sampling Errors

Due to the usage of a fixed set of viewing directions with orthographic projection in the preprocessing, for a viewing direction under perspective projection the visibility information contained in a visibility sphere might not be exact. The potentially visible set (PVS) that can be generated from the data in the visibility sphere may differ from the exact visible set (EVS) for the new viewing direction. The relationship between the two sets is shown exemplary in Figure 3.4.

The set of objects that are elements of the exact visible set, but are missing in the potentially visible set, is called *underestimation* $U = EVS \setminus PVS$. Sometimes, the term "false negatives" is used, because the objects are mistakenly classified as invisible. The missing objects will not be known to the rendering algorithm, therefore will not be rendered, as a consequence will be missing in the final image and lead to display errors.

The set of objects that are contained in the potentially visible set, but are missing in the exact visible set, is called *overestimation* $O = PVS \setminus EVS$. It is called "false positives" in some cases, since the objects are erroneously declared to be visible. The additional objects will not be visible in the resulting image, e.g., because they are occluded, and will lead to decreased rendering performance, because the renderer will send them to the graphics pipeline.

The goal of the sampling process is to approximate the EVS as good as possible. Because of the high complexity of the visibility information (refer to Section 3.4), an exact culling algorithm, where $U = \emptyset$ and $O = \emptyset$, cannot be used for complex scenes in practice. Since the underestimation leads to image errors, it is worse than the overestimation. Hence, a conservative culling algorithm with no underestimation, $U = \emptyset$, but little overestimation, meaning $|O|$ is small, is desirable. Because SVS's implementation uses the rasterization of the graphics hardware and only a small set of sample points for performance reasons, sampling errors are inevitable. For these reasons, SVS is an approximate culling algorithm, with $|U| \geq 0$ and $|O| \geq 0$.

4 Rendering

During runtime, the data acquired in the preprocessing step is utilized to display the scene for the current camera position. For an inner node of the tree that is traversed by the rendering algorithm (Section 4.1), the associated visibility sphere is retrieved. If the current camera position is outside of the sphere, the viewing direction onto that sphere is used to compute the potentially visible set of the node's subtree (Section 4.2). In order to achieve higher frame rates at the expense of image quality, one can optionally add a budget constraint to the rendering as described in Section 4.3. Although SVS does not allow fully dynamic scenes, animations are still possible more easily than in region-based methods (see Section 4.4).

4.1 Tree Traversal

Algorithm 1 RENDERINGTRAVERSAL: Recursive rendering algorithm executed during runtime.

```
procedure RenderingTraversal(node, cameraPosition)
    if node is a leaf then
        Render(node)
        return
    end if
    sphere ← GetVisibilitySphere(node)
    if cameraPosition is inside sphere then
        for all children child of node do
            RenderingTraversal(child, cameraPosition)
        end for
    else
        direction ← Normalize(center of sphere − cameraPosition)
        visVec ← QuerySphere(sphere, direction)
        for all potentially visible nodes pvNode in visVec do
            Render(pvNode)
        end for
    end if
end procedure
```

SVS's rendering algorithm uses the scene graph that contains the preprocessed visibility information. The scene graph is traversed beginning at the root node. The pseudo code for the rendering traversal is shown in Algorithm 1. Figure 4.1 shows the different types and states of nodes during the traversal. In all cases, frustum culling [AM00] is performed before rendering an object or traversing a node. The frustum culling is not shown in the pseudo code.

At first, the algorithm checks the kind of node it got as parameter. If the current node is a leaf node of the tree (a single object), the algorithm simply renders it. The rendering first configures

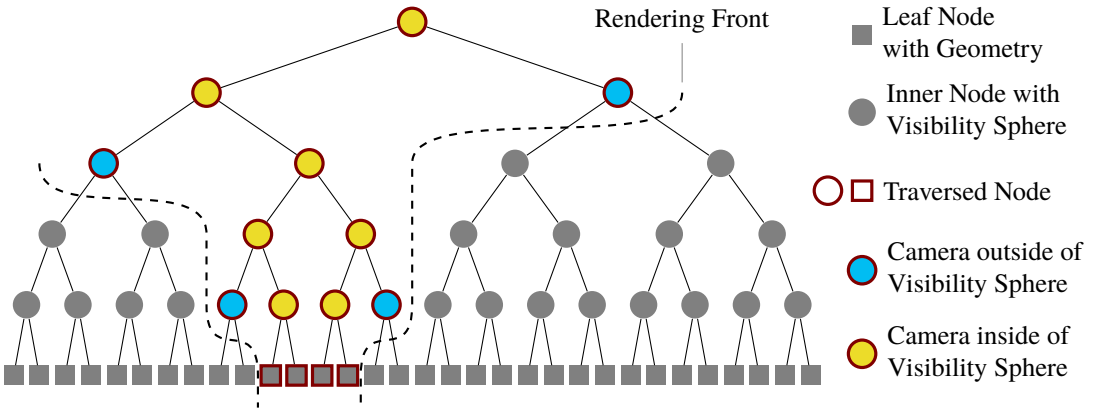


Figure 4.1: Illustration of an exemplary tree with nodes highlighted depending on their status during the tree traversal of the rendering algorithm.

the graphics pipeline depending on the properties of the node (setting textures, material definitions, shader programs, etc.). Then, the triangle mesh is sent to the graphics pipeline.

If the current node is an inner node, the position of the camera in relation to the node's bounding sphere is determined. If the camera is inside the bounding sphere, the node's visibility information is not valid for the camera position and the traversal continues with the child nodes. In the other case, where the camera is outside of the bounding sphere, the potentially visible objects are extracted from the node's visibility sphere for the current direction in which the sphere is seen. NORMALIZE computes a unit vector by dividing the vector that it gets as parameter by this vector's length. By calling QUERYSPHERE, an interpolated visibility vector for the queried direction (see Section 4.2) is received. The objects stored there are rendered, and the traversal of this subtree is finished.

The rendering front in Figure 4.1 illustrates why SVS performs very well for scenes with nested objects. The tree traversal stops early for inner nodes, if the camera is outside of their bounding sphere. These nodes are shown in blue. Their visibility data is used to render the objects contained in their subtrees. If such a subtree represents a hierarchy of nested objects with many of them not visible from outside, all of these occluded objects will not have to be touched by SVS's rendering algorithm.

4.2 Determining the Potentially Visible Objects

A visibility sphere stores a separate visibility vector for each of the sampled directions. These vectors alone are only valid as seen from those directions under orthographic projection. In order to acquire proper visibility information for an arbitrary direction, there are different possibilities.

The simplest technique, called NEAREST in the following, determines the sample point that is most similar to the query direction. In practice, SVS creates 20 sample points per visibility sphere, and a linear scan can be used to find the nearest sample point. If there was a large number of sample points, they could be maintained in a spatial data structure to speed up the search. The NEAREST interpolation returns the visibility vector of the nearest sample point as result.

The MAX3 interpolation utilizes the Delaunay triangulation of the sample points (see Section 3.4) to determine the tetrahedron that intersects the query direction near the sphere surface.

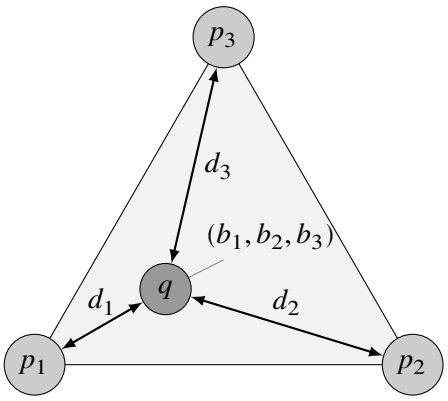


Figure 4.2: Interpolation for a query point q with barycentric coordinates b_1, b_2, b_3 inside the triangle of the sample points p_1, p_2, p_3 with distances d_1, d_2, d_3 to q.

Because the number of tetrahedrons is constant, this is also done in a linear scan. If the performance was too low here, a spatial data structure could be put to use. The tetrahedron intersecting the query direction has four corners: three sample points p_1, p_2, p_3 and the center of the sphere. As defined in Section 3.5, a visibility vector contains pairs (o, v) of objects o that are visible and that have an visible area of v pixels. The visibility vectors VV_1, VV_2, VV_3 are extracted from the three sample points. As result, a new visibility vector VV_r is created as the union of the three visible sets. By storing the visibility vectors as sorted arrays, the union can be computed in linear time. The visible area value of an object is the maximum of the visible area values of the sample points that contain the object:

$$VV_r = \left\{ (o, v) \in VV_1 \cup VV_2 \cup VV_3 \;\middle|\; v = \max_{(o,w) \in VV_1 \cup VV_2 \cup VV_3} \{w\} \right\}.$$

The third technique, the Weighted3 interpolation, uses the distance of the query direction to the sample points to interpolate the visibility information (see Figure 4.2). To perform the interpolation on the sphere surface, one possibility would be to use spherical barycentric coordinates [LBS06]. Since the deviation in results is negligible in practice [Car07], linear interpolation on planar coordinates can be used. Weighted3 works similar to Max3 in that it first identifies the tetrahedron that intersects with the query direction. For the triangle near the sphere surface that is formed by the three sample points p_1, p_2, p_3, the barycentric coordinates [Möb27] $b_1, b_2, b_3 \in [0, 1]$ with $b_1 + b_2 + b_3 = 1$ of the query direction q are calculated. An object's resulting visible size is calculated as weighted average of the stored visible sizes. As weights, the barycentric coordinate values corresponding to the sample points are used, respectively. Formally, in Weighted3 the resulting visibility vector VV_r is computed as follows:

$$VV_r = \left\{ (o, v) \;\middle|\; \exists i \in \{1, 2, 3\} : (o, v_i) \in VV_i \wedge v = \sum_{(o,v_j) \in VV_1 \cup VV_2 \cup VV_3} b_j \cdot v_j \right\}.$$

The evaluation of the interpolation schemes (see Section 5.4) proved Nearest and Weighted3 to be unsuitable (e.g., taking only the nearest stored sample results in much underestimation and thus many missing objects in the final image; weighting the visibility data with the distance of its sample point to the query position has no considerable benefit). Therefore, the Max3 interpolation scheme should be used. The underestimation introduced by using a perspective

camera during rendering (while the PVSs are based on orthographic projection), is also greatly reduced by MAX3 (see Section 5.8).

4.3 Optional Budget Rendering

For many scenes and viewing positions, the amount of visible geometry can easily exceed the amount that can be rendered in real time by the graphics hardware. This is especially true on mobile devices with weak graphics hardware. Even without using rendering techniques such as level-of-detail meshes or impostors, one can still limit the amount of rendered geometry by introducing a rendering budget and just skipping some objects if the budget is exceeded. SVS introduces an additional importance value to each object entry in a visibility vector (see Section 3.5) that allows an estimation on the visible influence of the object on the final image. Thereby, a budget can be distributed among the estimated most important objects; yielding a high image quality even when some parts of the scene are not rendered at all. Although the image quality can severely suffer in extreme situations, where no more insignificant objects can be left out to fulfill the budget constraint, this extension allows rendering of almost arbitrarily large scenes with a constant frame rate with a high image quality in most cases (see Section 5.15).

The next section discusses the problem that has to be solved when selecting the objects that shall be displayed when using budget rendering. It shows that solving the problem is equivalent to solving the NP-hard 0-1 knapsack problem. Therefore, the computation of an exact solution is not practical. The subsequent section explains the algorithm that is used in practice for SVS's budget rendering.

4.3.1 Computation of an Exact Solution

Assume a rendering algorithm knows the exact visible set EVS for an arbitrary view. Let EVS contain n objects $o_1, \ldots, o_n$. Additionally, let the algorithm know the number of visible pixels v_i that o_i will contribute to the final image when displayed. Obviously, it knows the number of triangles t_i of an object o_i. Technically, it would be possible to display only a subset of an object's triangles, but it cannot be determined in advance, how many pixels will be filled by that subset. Therefore, the rendering algorithm has to decide for every object if it shall be displayed fully, or not at all. Additionally, the user specifies a triangle budget T that indirectly limits the rendering time.

This problem can be transformed to a 0-1 knapsack problem. The 0-1 knapsack decision problem is known to be NP-complete [Kar72], therefore the combinatorial optimization problem is NP-hard. The n objects correspond to the items that are to be placed in the knapsack. The value of an object o_i is the number of visible pixels $v_i \in \mathbb{N}$. The weight of an object o_i is the number of its triangles $t_i \in \mathbb{N}$. The knapsack has an overall capacity of T. The goal is to maximize the overall gain $\sum_{i=1}^{n} v_i \cdot x_i$ such that $\sum_{i=1}^{n} t_i \cdot x_i \leq T, x_i \in \{0, 1\}$. As result, a binary decision variable x_i denotes whether an object o_i shall be displayed.

Because of the hardness of the problem, it is impractical to compute an exact solution. There is a dynamic programming approach to compute an exact solution to the problem using the functional equation approach [Bel54]. The running time of this approach is $\mathcal{O}(n \cdot T)$. The factor T in the running time makes the approach ineligibly for the problem stated here, because T is rather large when used as rendering budget (e.g., millions of triangles).

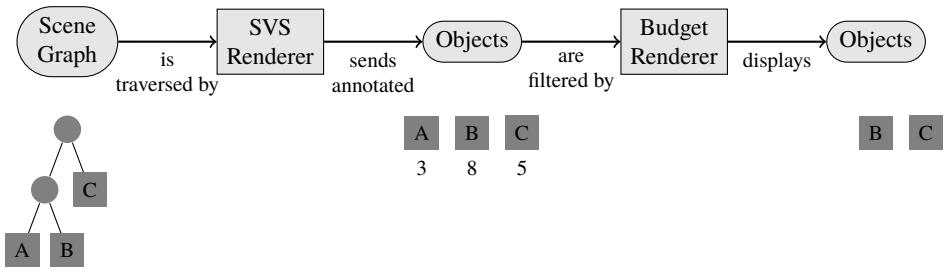


Figure 4.3: Illustration of the interaction of the two renderers used for budget rendering.

In general, a greedy algorithm [Dan57] can be used to solve the 0-1 knapsack problem. Unfortunately, the solution computed by such an algorithm might be far from optimal. For 3D rendering, Funkhouser and Séquin [FS93] use a greedy approximation algorithm to solve their special kind of knapsack problem. They compute a benefit-cost ratio for their items and iteratively select the item with highest ratio while the items fit into the knapsack. Pham Ngoc et al. [Pha+02] present a specialized approximation algorithm to solve a 0-1 knapsack problem inside their network rendering system. SVS also uses a greedy solution, because it provides good results in practice (see Section 5.15) and leads to a very simple implementation.

4.3.2 Practical Computation of a Solution

The budget rendering used together with SVS consists of two parts: the SVS renderer that was described before (see Section 4.1), and a new budget renderer that is described in the following. An illustration of the interaction of those two renderers is shown in Figure 4.3. In a first step, the SVS renderer traverses the tree the same way it is done when no budget rendering is used. It annotates every object that it wants to be displayed with the estimated visible size that is extracted from the visibility vector. But, instead of sending the objects that shall be displayed to the graphics pipeline, they are intercepted by the budget renderer. When the SVS renderer finishes, the budget renderer receives a collection of all nodes that shall be displayed for the current frame together with the annotated visible size.

For every object, the budget renderer determines the area of the object's bounding box when projected onto the screen using the current camera configuration. This projected size is additionally computed, because the estimated visible size, which is extracted and interpolated from the preprocessed visibility information, might be wrong for the current camera view. On the one hand, the object cannot be larger on the screen than the projected size of the bounding box. Therefore, the projected size is an upper bound for the object's screen area. On the other hand, using only the projected size is insufficient. The projected bounding box can be very large on the screen, but only little parts of the object might be visible on the screen due to occlusion. This is why the minimum of the estimated visible size and the bounding box's projected size is used.

As mentioned before, a greedy algorithm is used here to solve the 0-1 knapsack problem. Its description in pseudo code can be seen in Algorithm 2. As the value of an object, the minimum of its estimated visible size and the projected size of the object's bounding box is used. The weight is the number of triangles of the object. The collected objects are sorted by their value-weight ratio. After that, they are displayed one after another with decreasing ratio beginning with the highest. For every object that is displayed, the weight is accumulated. This is done as long as the accumulated weights do not exceed the budget.

Algorithm 2 Budget rendering with a triangle budget T using n collected objects $o_1, \dots, o_n$, their estimated visible sizes $v_1, \dots, v_n$, and their triangle counts $t_1, \dots, t_n$.

```
procedure BudgetRendering(T, o_1,...,o_n, v_1,...,v_n, t_1,...,t_n)
    for i ←1 to n do
        p_i ← ComputeProjectedSize(o_i)
        r_i ← min{v_i,p_i} / t_i                    ▷ Benefit-cost ratio for object o_i
    end for
    A ← [1,...,n]                                   ▷ Initialize array containing indices
    Sort(A such that ∀j ∈ 1,...,n−1 : r_A[j] ≥ r_A[j+1])   ▷ Sort indices by descending ratio
    W ← 0                                           ▷ Initialize weights accumulator
    for j ←1 to n do
        i ← A[j]
        if W + t_i > T then
            break                                   ▷ Stop if the next object exceeds the budget
        end if
        Render(o_i)
        W ← W + t_i
    end for
end procedure
```

4.4 Rendering Animated Objects

All methods based on precomputed visibility share their inability to support fully dynamic scenes (where each object can be moved freely) without the need of costly recomputations. For region-based techniques, the region that is subdivided into cells corresponds most of the times to the bounding box of the scene. Sometimes, this bounding box is enlarged to be able to look at the scene from outside (e.g., the bounding box is scaled by a constant factor). When an object should be moved inside the scene, nearly all cells have to be updated: Cells that contain visibility information about this object have to check if the object is hidden now. Cells that do not see objects, because the moved object occluded them before, have to check if new objects are visible now. Other cells have to check if they see the object now, because of its new position. If the object moves outside of the scene region, new cells have to be added, or – if this is not possible – the whole cell hierarchy has to be rebuilt.

Of course, one possibility is to leave out the moving objects for the visibility computations in the preprocessing phase. Then, there are no visibility information at all for the animated objects. Another possibility is to generate separate region subdivisions with view cells for each animated object. Then, at runtime, multiple view cells have to be determined to gather the visibility information. This corresponds to treating the animated objects like multiple scenes during the preprocessing, but rendering all of these scenes at once during runtime.

One benefit of SVS over region-based techniques is the possibility to arbitrarily animate independent subtrees during runtime. Subtrees are independent, if there is no visibility sphere in the scene graph above their root node. Hence, they do not share a common visibility sphere. The subtrees can be preprocessed by SVS separately and can be combined by a common root node without visibility sphere. Each animated subtree can be of arbitrary size and the rendering algorithm does not have to treat it specially. However, any animation inside such a subtree still requires an update of all visibility spheres from the animated node up to the root node, or until

no change in the visibility sphere is detected.

5 Evaluation

Different aspects of SVS will be evaluated in this chapter. On the one hand, this evaluation serves the purpose of experimentally proving SVS's described functionality. On the other hand, the influence of parameter choices on different performance factors is assessed. To obtain the data required for an evaluation, different measurements are performed.

Firstly, the software implementation and the test machine used for the measurements are presented (Section 5.1). Following, the test scenes used for the evaluation are described (Section 5.2). The evaluation answers the question on how the values for SVS's different parameters should be chosen. The quality of the visibility information, the running time, and the storage space are examined in relation to different choices of parameter values for these different parameters (Section 5.3 to Section 5.7). Additionally, the errors introduced by the usage of different projections are examined (Section 5.8). Subsequent, the space required to store the visibility data (Section 5.9) and the total running time of the preprocessing (Section 5.10) are considered. As SVS is designed as rendering algorithm for complex scenes, the evaluation ought to show how SVS performs in practice. For such an evaluation at runtime, a camera path (Section 5.11) is used for each test scene to simulate a walkthrough. In addition, two occlusion culling algorithms are presented that will be compared to SVS (Section 5.12). At first, the image quality during a walkthrough is tested (Section 5.13). Then, the 3D rendering on a workstation is examined (Section 5.14). After that, the budget rendering feature is evaluated (Section 5.15). Last but not least, the results for the 3D rendering on mobile devices are presented (Section 5.16).

As already stated, there are multiple parameters that influence SVS's behavior. A single evaluation with changes to all of these parameters would be difficult to analyze. Therefore, for every parameter, there is a separate evaluation, in which the values for that parameter are changed and all other parameters have fixed values. The evaluation of the parameters is ordered such that the values can be determined step by step. Where possible, the order is chosen so that an earlier evaluation does not depend on the value of a later evaluation. After an evaluation, a parameter value is chosen based on the acquired results. When such a reasonable parameter value has been

Table 5.1: Overview of the parameters that are part of the evaluation in this chapter. The last column shortly mentions the value that arises as result out of the evaluation of the respective parameter.

Parameter	Description	Evaluation	Parameter Value
Sample point distribution	Section 3.4	Section 5.3	20 sample points, dodecahedron
Interpolation technique	Section 4.2	Section 5.4	MAX3
Compute tight bounding spheres	Section 3.3	Section 5.5	yes, use EPOS
Use existing visibility results	Section 3.5	Section 5.6	no
Resolution in pixels	Section 3.5	Section 5.7	1024^2 pixels, screen resolution

found, it is used as a fixed value for the following evaluations.

Table 5.1 contains an overview of the parameters that are evaluated in this chapter. For each parameter, the section that describes the influence of the parameter and the section that contains its evaluation are referenced. Additionally, the parameter value that is chosen as result of the evaluation is very briefly summarized in the table.

The evaluation sections in this chapter are structured as follows: At first, the goal of the evaluation in the respective section is described. Then, the measurements that are executed to generate the required data are explained. Afterwards, the data is presented and analyzed. Finally, the results are summarized and the question posed by the section is answered.

5.1 Software Implementation and Hardware

In order to perform the experimental evaluation, the proposed methods were implemented in software. The implementation was done inside the *Platform for Algorithm Development and Rendering (PADrend)* [EJP11].

PADrend is a software framework for the visualization of complex three-dimensional scenes. Its main goal is to provide a common basis for the development of rendering algorithms. Because of its special functionality for evaluation (e.g., image quality measurements) and several standard rendering algorithms that are ready for use, it can be used to assess rendering algorithms in the scope of research and teaching. In effect, PADrend is used for the development and evaluation of new rendering algorithms, for support of students in writing their Bachelor's and Master's theses, and to perform design reviews of complex industrial plants on a cave-like HD visualizaton center. PADrend is developed since 2007 in the research group Algorithms and Complexity at the Heinz Nixdorf Institute, University of Paderborn. It is mainly developed by Benjamin Eikel, Claudius Jähn, and Ralf Petring. The basis of PADrend is a set of C++ libraries that were developed from scratch especially for their use in PADrend. Among others, there is a library for three-dimensional objects (e.g., boxes) and geometric computations, a library for abstracting the graphics pipeline, a library for the user interface, and a scene graph library. All performance-critical algorithms are implemented in C++. On top of these libraries, the EScript scripting language[1] is used for high-level implementation. In the EScript scripts, the whole functionality of the base libraries can be used. PADrend is fully modularized and on the top level, it consists of a plug-in system written in EScript.

In the scope of this work, PADrend was used during the development of SVS. SVS's preprocessing as well as the rendering algorithm were implemented in the scene graph library in C++. Some parts, e.g., the graphical user interface for controlling SVS's parameters, are written in EScript. After loading a scene, the user can select the parameters for the preprocessing and start it with a single click. When the preprocessing finishes, the user can add the SVS renderer, which is immediately used to render the scene. Also, SVS's budget renderer can be added by a single click and the user can select a triangle budget. When the triangle budget is changed, it is directly used by the rendering algorithm and can be observed by the user. Additionally, the preprocessed scene can be saved to disk and loaded again, in order to avoid the need to perform the preprocessing again.

The workstation PC that has been used to perform the measurements has an Intel Core i7-3770 CPU (4 cores, 8 threads, 3.4 GHz clock speed), 32 GiB DDR3 RAM (1600 MHz clock speed), and a NVIDIA GeForce GTX 660 GPU (2 GiB dedicated graphics memory). PADrend was

[1] http://escript.berlios.de/

Table 5.2: Number of triangles, number of objects, and number of inner nodes of the test scenes.

Scene	Number of triangles (in millions)	Number of objects	Number of inner nodes
PP	12.749	1,185	316
PP4	50.994	4,740	1,383
PP256	3,263.619	303,360	72,279
Pompeii	62.964	153,848	40,142
Boeing	337.143	228,756	65,535
PPBoeing5	1,698.464	1,144,965	327,959
SphereOfCubes	0.012	1,000	1

compiled with the GNU Compiler Collection[2] in version 4.8.1 on Debian GNU/Linux sid[3]. There is one exception regarding the measurements in this chapter: Obviously, the measurements on the mobile devices were not executed on this workstation PC, but on a tablet and smartphone (see Section 5.16).

5.2 Test Scenes

In the following, the test scenes – namely Scene PP, Scene PP4, Scene PP256, Scene Pompeii, Scene Boeing, Scene PPBoeing5, and Scene SphereOfCubes – are described shortly. The test scenes were prepared to be used by SVS (see Section 3.1): All test scenes already define objects and, thus, no triangle data structure had to be used. Since there was no hierarchy definition for the objects, a loose octree was used to create a hierarchical scene graph structure. For all test scenes, Table 5.2 contains an overview of the number of triangles, the number of objects, and the number of inner nodes.

SVS's preprocessing algorithm as well as the rendering algorithm have to traverse the scene graph, which is why the depth of the tree is of interest. The chart in Figure 5.1 gives an overview of the nodes' distribution in the scene graph. The chart shows of how many levels the scene graph structure consists for the different scenes. Furthermore, one can see how many nodes are stored in the different levels. For example, for all scenes, there is only a single node on the zeroth level – the root node of the scene graph.

Additionally to the description of the scenes in the following sections, Figure 5.2 shows the depth complexity of different viewpoints of the scenes. As defined in Section 2.1, the depth complexity is the number of triangles pierced by an arbitrary ray from the camera position, on the average. Instead of using rays, the depth complexity has been determined for the pixels in an image by rasterization. It has been counted, for each pixel, how many triangles filled that pixel with disabled depth test. If the pixel was not filled, it belongs to the background and is shown in white in the images. The minimum depth complexity of a filled pixel is therefore 1, which is shown in blue. The median and maximum depth complexity specified for an image are the median and maximum depth complexity of all filled pixels in this image. The images give an impression on where the geometric complexity is located inside the scenes.

[2] http://gcc.gnu.org/
[3] http://www.debian.org/

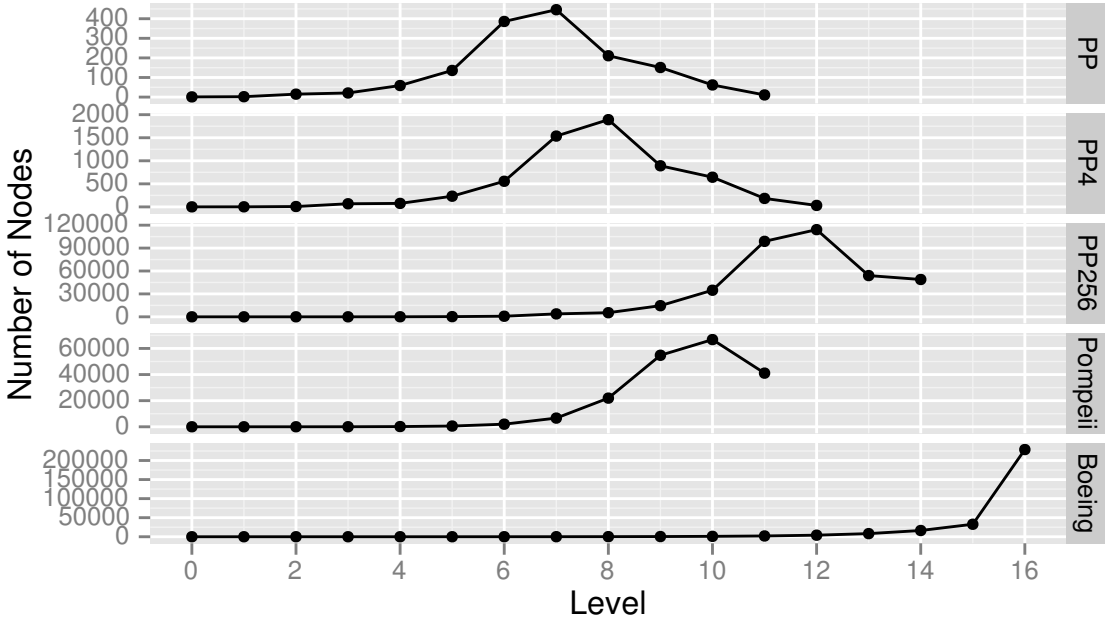


Figure 5.1: Number of nodes in the scene graph's levels for different scenes.

5.2.1 Scene PP

The Scene PP contains a single Power Plant model[4]. The Power Plant model is a standard model that is used as a benchmark in several computer graphics publications. The model has 12,748,510 triangles that are stored in 1,185 objects. The vertices of the model have colors and normals, but there are no textures. As stated above, the objects of the Power Plant model have been inserted into a loose octree to create a scene graph that consists of 316 inner nodes. Screen shots of the scene can be seen in Figure 5.3. The tubings inside the building constitute the majority of the model's triangles. A small part of these tubings can be seen in Figure 5.3(d).

5.2.2 Scene PP4

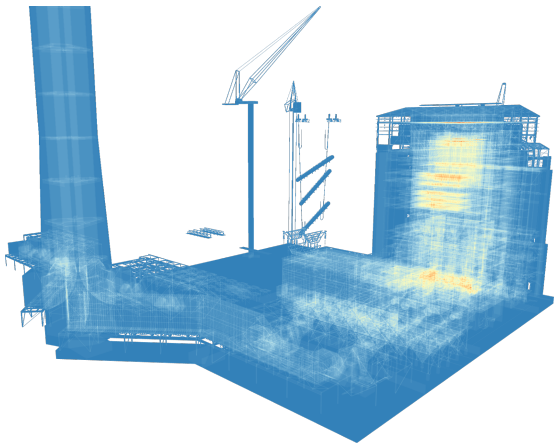
In the Scene PP4, four Power Plant models haven been placed aside in a 2×2 grid. This layout can be seen in the screen shots in Figure 5.4. As expected for a combination of four Power Plant models, there are 50,994,040 triangles in 4,740 leaf nodes. The scene graph that has been constructed to store those nodes has 1,383 inner nodes.

5.2.3 Scene PP256

The Scene PP256 consists of 256 Power Plant models that are arranged in a 16×16 grid. The 303,360 leaf nodes are stored in an octree-like scene graph with 72,279 inner nodes. The scene consists of 3.264 billion triangles. The Scene PP256 is the most complex test scene concerning the number of triangles. Figure 5.5 contains screen shots showing the arrangement of the 256 Power Plant models.

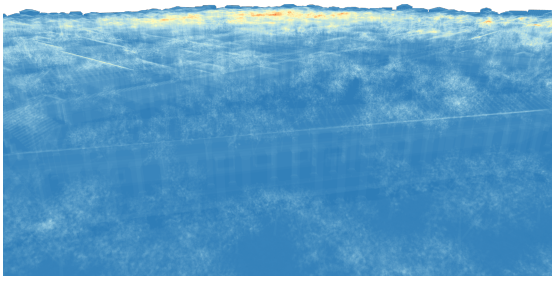
[4] http://gamma.cs.unc.edu/POWERPLANT/

(a) Color gradient to map depth complexity values to colors.

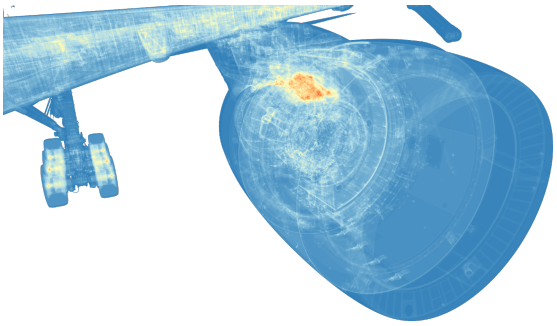


(b) Scene PP: median 4, maximum 67.

(c) Scene Boeing: median 15, maximum 127.

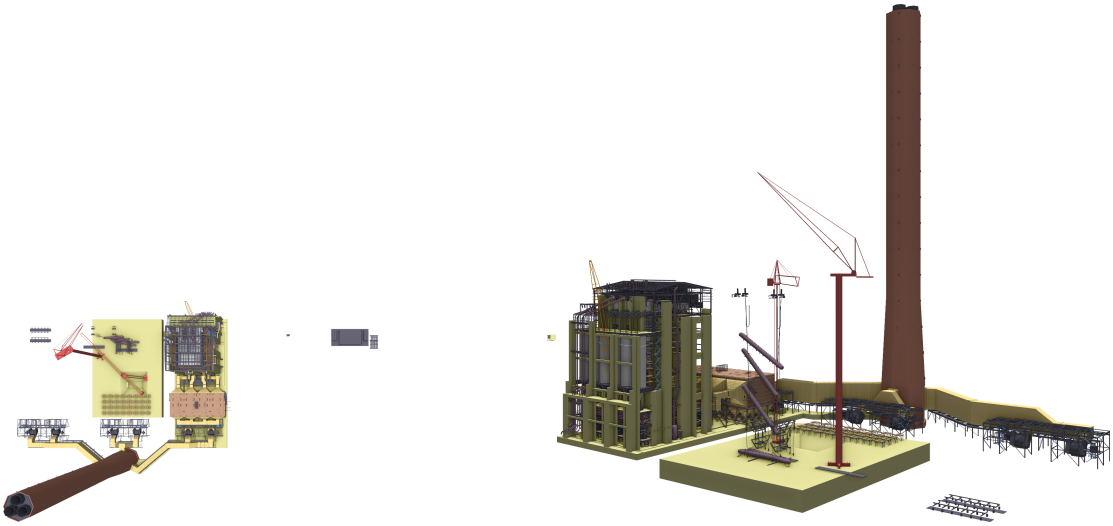


(d) Scene Pompeii: median 11, maximum 227.



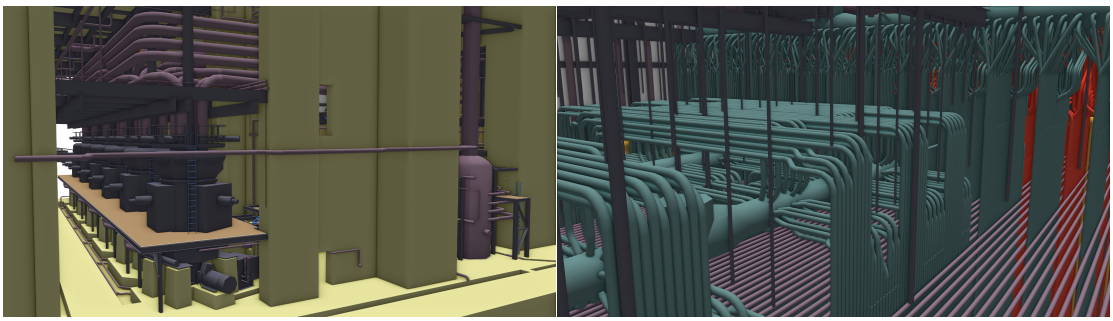
(e) Scene Boeing: median 12, maximum 120.

Figure 5.2: Visualization of the depth complexity for different views in different scenes.



(a) Top view.

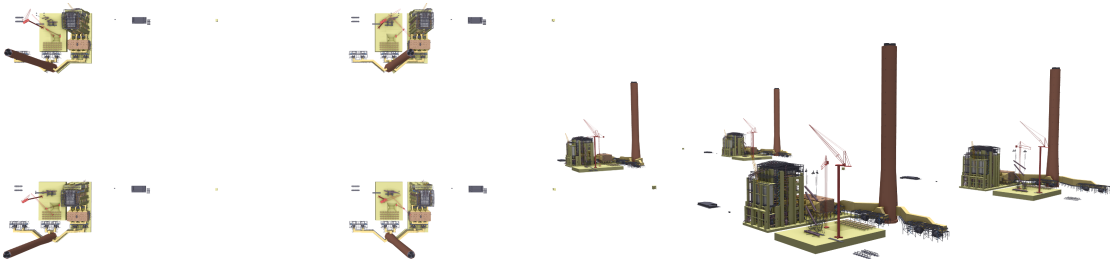
(b) Oblique view.



(c) Detailed view of a part of the building.

(d) Tubings inside the building.

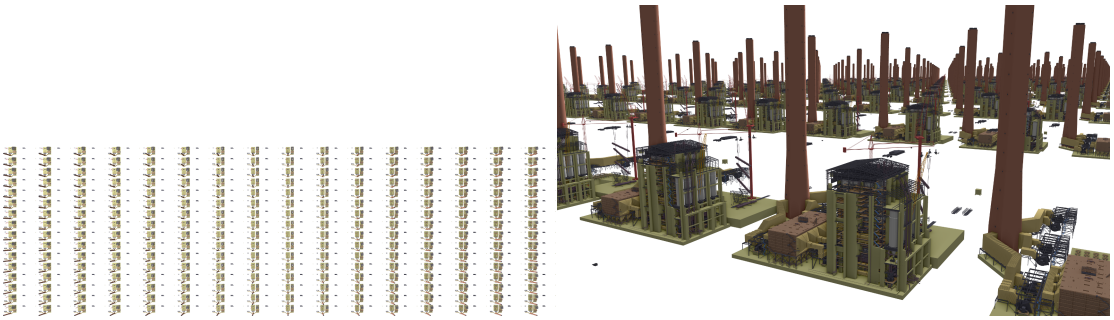
Figure 5.3: Screen shots of the Scene PP.



(a) Top view.

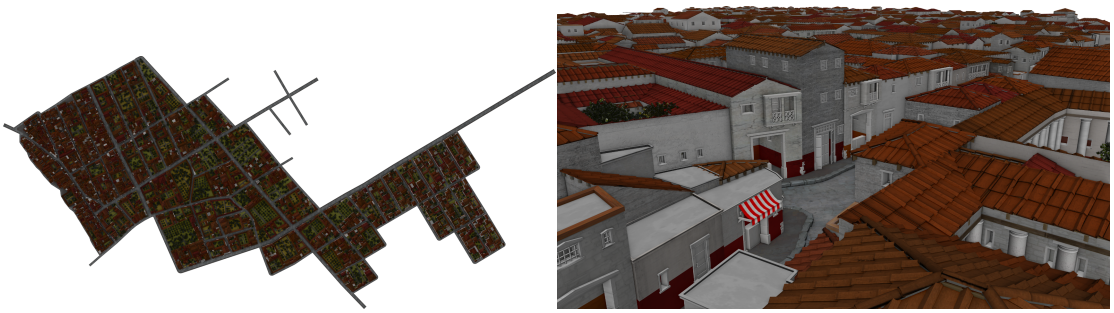
(b) Oblique view.

Figure 5.4: Screen shots of the Scene PP4.



(a) Top view. (b) Oblique view.

Figure 5.5: Screen shots of the Scene PP256.



(a) Top view. (b) Detailed view of a part of the city.

Figure 5.6: Screen shots of the Scene Pompeii.

5.2.4 Scene Pompeii

The Scene Pompeii is a model of ancient Pompeii exported by the CityEngine[5]. The city consists of a street network with houses densely standing next to each other at the roadside. The inside of the houses is empty (e.g., there are no objects like furniture). The scene is also organized in an octree-like scene graph and has 153,848 leaf nodes and 40,142 inner nodes. In contrast to the other test scenes, this scene's objects have materials and textures. The Scene Pompeii consists of 62.964 million triangles. Screen shots of the scene are shown in Figure 5.6.

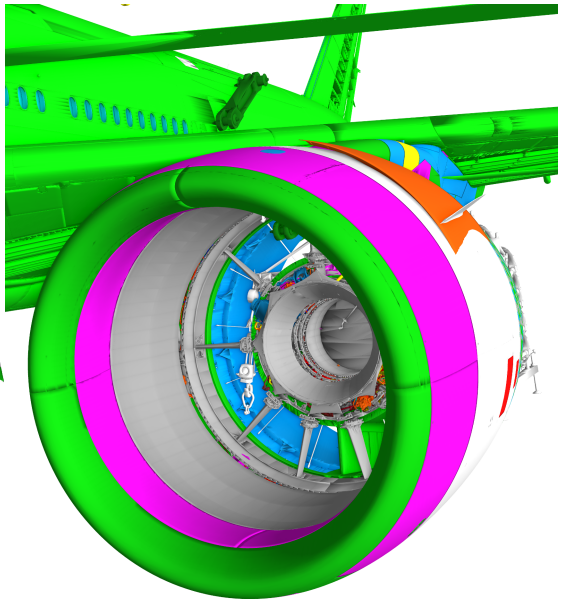
5.2.5 Scene Boeing

An aircraft model of a Boeing 777[6] constitutes the Scene Boeing. The model was released by The Boeing Company as benchmark for computer graphics, especially to test the visualization of CAD data. To protect their intellectual property, geometric errors were intentionally put into the model before the data was made public. The vertices of the model have colors and normals, but there are neither materials nor textures. The model consists of 228,756 leaf nodes that have 337.143 million triangles. A scene graph with an octree-like structure has been built over the nodes and has 65,535 inner nodes. Two overviews together with four detailed views of the scene

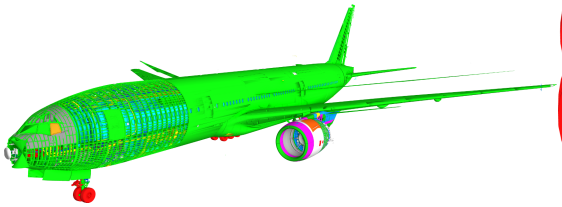
[5]The highest available level of detail of the scene, "LoD 3 – High Detail", was used. http://www.esri.com/software/cityengine/resources/casestudies/procedural-pompeii

[6]Source 3D data provided by and used with permission of The Boeing Company.

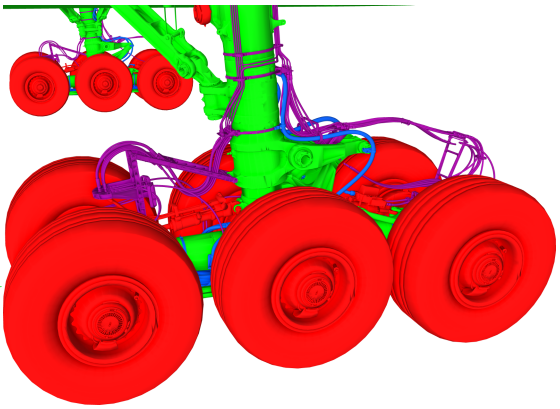
(a) Top view.



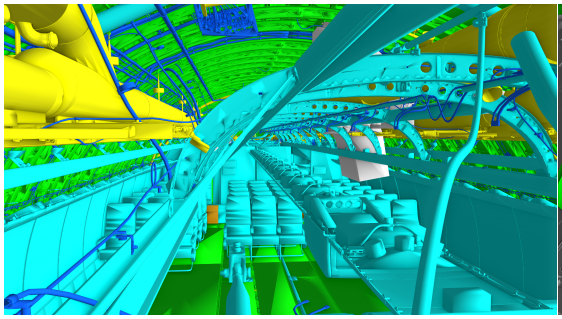
(b) Engine.



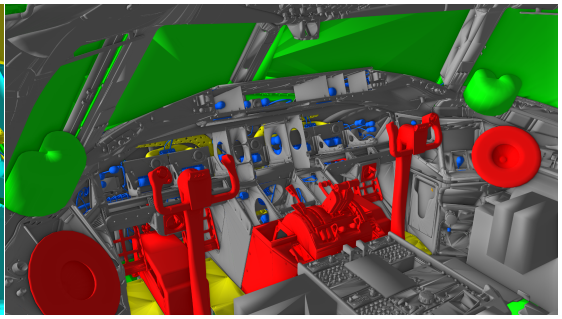
(c) Oblique view.



(d) Landing gear.

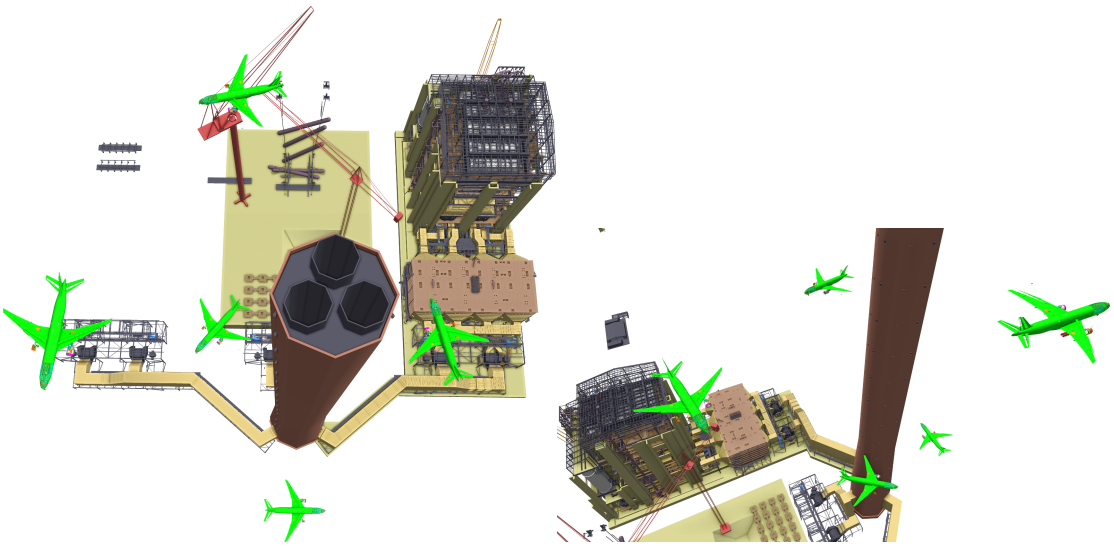


(e) Passenger cabin.



(f) Cockpit.

Figure 5.7: Screen shots of the Scene BOEING.



(a) Top view. (b) Oblique view.

Figure 5.8: Screen shots of the Scene PPBoeing5.

are depicted in Figure 5.7. The geometric detail of the model is very high. For instance, valves of the wheels (Figure 5.7(d)) and knobs in the cockpit (Figure 5.7(f)) are geometrically modeled.

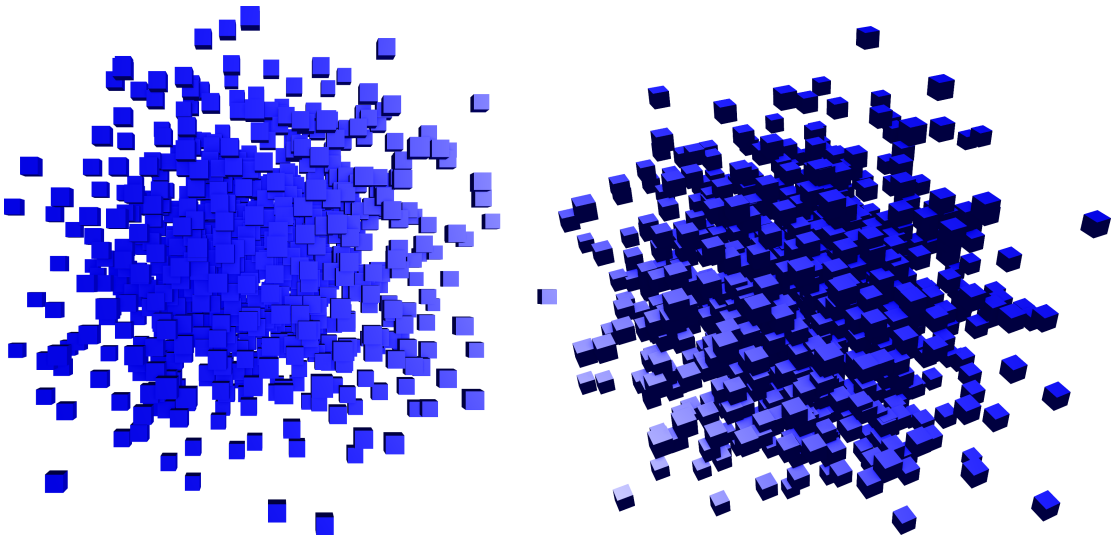
5.2.6 Scene PPBoeing5

The Scene PPBoeing5 (see Figure 5.8) is built of one Power Plant model and five Boeing 777 models. The Boeing 777 models are placed on a helix around the chimney of the Power Plant. Because it is used to test SVS's handling of animated objects, the root node has no visibility sphere. To create this scene, the Scene PP and the Scene Boeing have been preprocessed separately by SVS. Then, the preprocessed version of the Scene PP and five times the preprocessed version of the Scene Boeing have been added to a common root node to create the scene. The Scene PPBoeing5 is the most complex test scene concerning the number of nodes. It has 1.145 million leaf nodes, 327,959 inner nodes, and 1.698 billion triangles overall.

5.2.7 Scene SphereOfCubes

The Scene SphereOfCubes is a special test scene for SVS's preprocessing part. It consists of 1,000 cubes with edge length 1. Every cube consists of twelve triangles; the whole scene has 12,000 triangles. To create the scene, the three coordinate values of each cube's center have been normally distributed with mean 0 and standard deviation 5, respectively. Figure 5.9 shows two views of the scene.

Because the boxes have only axis-aligned faces, the visibility is special in this scene: When standing on one of the main axes, the viewer sees only one side of every cube under orthographic projection (see Figure 5.9(a) that shows a perspective view). This leads to many visible cubes, because the projected size of every cube is minimal. The projected size of every cube becomes maximal, when the viewer looks directly into the direction of one corner of all cubes (see Figure 5.9(b) that shows a perspective view). For these positions, three sides of every cube are visible, but, the number of cubes that are visible is smaller. Because of this specialty, the scene is



(a) Top view. (b) Oblique view.

Figure 5.9: Screen shots of the Scene SphereOfCubes.

used for testing different sample point distributions. There, the described effect largely influences the set of visible objects, when sample points are or are not available for the described viewing directions.

5.3 Sample Point Distribution

In principle, SVS works with an arbitrary distribution of sample points on the sphere surface, as described in Section 3.4. In practice, it is apparent that too few sample points yield very inaccurate visibility information whereas too many sample points need much memory. To see what it means to use different distributions of sample points, the influence on the visibility information's quality and storage space is examined in detail in this section. As a result, a sample point distribution will be identified that provides a good trade-off for the practical usage of SVS.

Twenty different distributions were used for the evaluation. The distributions are listed in Table 5.3. A visualization of twelve distributions – with four up to 162 samples – is shown in Figure 3.2, together with a visualization of the corresponding triangulation of the sample points.

The minimum pairwise angle for the different sample point distributions is given in the table. It is a measure for how dense the sphere surface is covered with sample points. The smaller the minimum pairwise angle, the denser are the sample points located together. The angle between two sample points on the sphere surface is the central angle between those points. The minimum pairwise angle of a sample point is the minimum of all angles between the sample point and all other sample points. For a distribution corresponding to a platonic solid, the value of the minimum pairwise angle is the same for all sample points. Since the other distributions do not place the sample points evenly on the sphere surface, the range over all sample points of the minimum pairwise angle is given in the table.

Table 5.3: List of sample distributions used for the evaluation. Angles are rounded to two decimal places. For the distributions not corresponding to a platonic solid, the range of angles is given.

Name	Number of samples	Minimum pairwise angle (in °)
Tetrahedron	4	109.47
Octahedron	6	90.00
Cube	8	70.53
Tetrahedron sphere, 1 subdivision	10	54.74–54.74
Icosahedron	12	63.43
Octahedron sphere, 1 subdivision	18	45.00–45.00
Dodecahedron	20	41.81
Tetrahedron sphere, 2 subdivisions	34	27.37–30.93
Icosahedron sphere, 1 subdivision	42	31.72–31.72
Octahedron sphere, 2 subdivisions	66	22.50–24.42
Tetrahedron sphere, 3 subdivisions	130	13.68–24.42
Icosahedron sphere, 2 subdivisions	162	15.86–16.41
Octahedron sphere, 3 subdivisions	258	11.25–15.46
Tetrahedron sphere, 4 subdivisions	514	6.84–15.46
Icosahedron sphere, 3 subdivisions	642	7.93–9.09
Octahedron sphere, 4 subdivisions	1,026	5.62–8.46
Tetrahedron sphere, 5 subdivisions	2,050	3.42–8.46
Icosahedron sphere, 4 subdivisions	2,562	3.96–4.69
Octahedron sphere, 5 subdivisions	4,098	2.81–4.35
Icosahedron sphere, 5 subdivisions	10,242	1.98–2.36

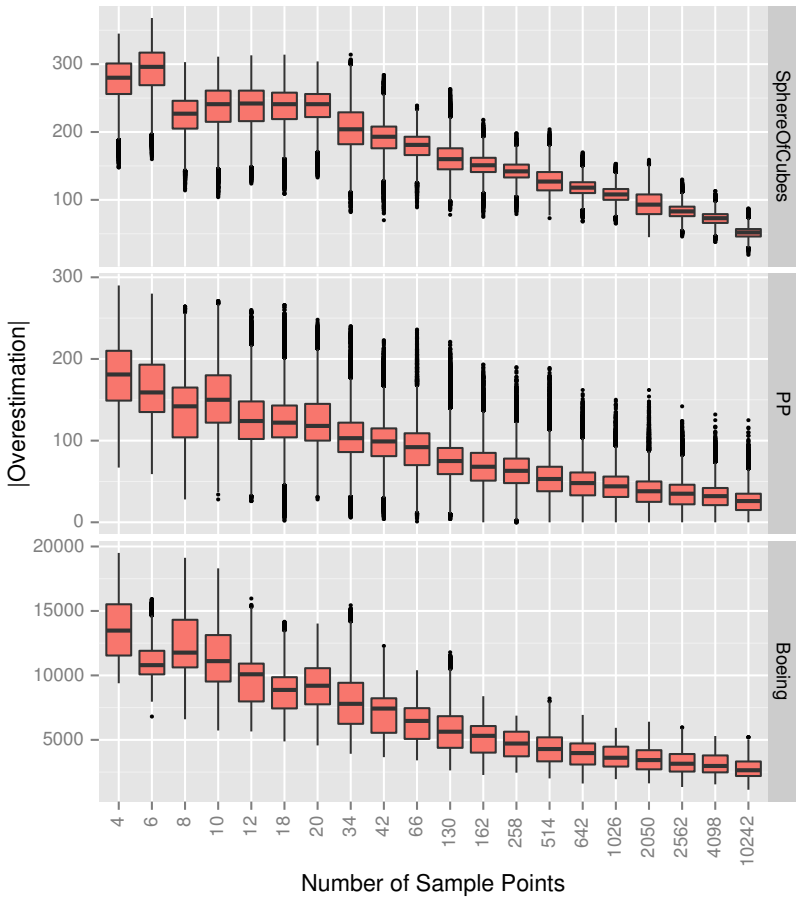


Figure 5.10: Distribution of the cardinality of the overestimation over the whole sphere surface when using different sample point distributions.

5.3.1 Quality of the Visibility Information

To assess the quality of a sample point distribution, the quality of the stored visibility information for different viewing directions is examined. Over the whole sphere surface, the visibility information stored in the sampling sphere is compared to the exact visibility. To determine the visibility information for an area such as the sphere surface, infinitely many measurements for points in this area would be needed in this setting. Obviously, infinitely many measurements cannot be performed in practice. Therefore, to approximate the visibility information of the sphere surface, 40,962 positions on the surface are chosen to represent the surface area. These positions are the vertices created by six edge-subdivision steps of an icosahedron. In the following, they will be called *reference positions*. The visibility is determined for the viewing directions represented by all reference positions. For each of these viewing directions, the resulting visibility information is the exact visible set (EVS).

To make sure that the regular structure of the subdivided icosahedron does not distort the results, 40,962 random positions were tested, too. The positions were chosen uniformly at random as spherical coordinates with inclination $\varphi \sim \arccos(U(-1,1))$ and azimuth $\theta \sim U(0,2\pi)$. $U(a,b)$ denotes the continuous uniform distribution giving values from the interval $[a,b]$ with probability $1/(b-a)$ and other values with probability zero. No significant difference to

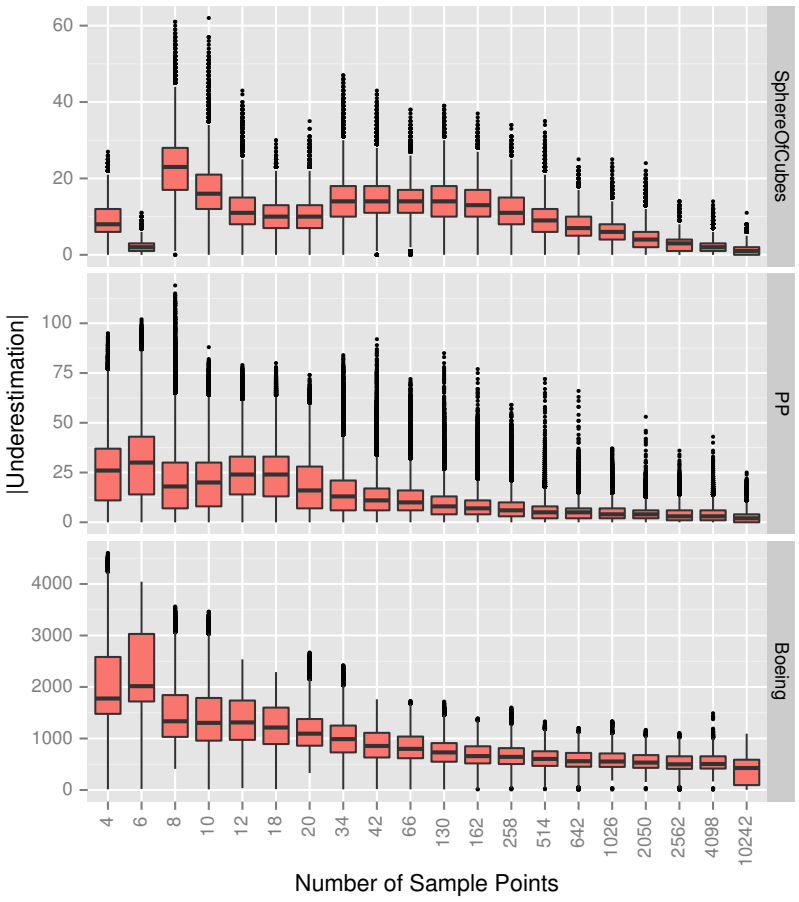


Figure 5.11: Distribution of the cardinality of the underestimation over the whole sphere surface when using different sample point distributions.

the subdivided icosahedron could be detected in the results when using the random positions as reference positions.

To assess the quality of the visibility information of a specific sample point distribution, the visibility sphere created by SVS using this sample point distribution is queried at the reference positions. For each reference position, the result is a potentially visible set (PVS) for the corresponding viewing direction that has been interpolated from the stored visibility information. The overestimation and underestimation are computed by using the EVS of the reference position (refer to Section 3.6). The cardinalities of these two sets are used to measure the difference between PVS and EVS. The term *missing objects* is used in the following to denote the objects in the underestimation set $U = EVS \setminus PVS$.

In anticipation of the results in the next section, the MAX3 interpolation is used here. The two other interpolation techniques have been tested, too, and have led to the same results concerning the quality of the different sample point distributions.

Since the quality for the whole sphere is evaluated, the results for a specific viewing direction are not particularly interesting here. Therefore, a box plot summarizing the distribution of the overestimation's and underestimation's cardinality over the whole sphere is created for every sample point distribution.

To keep the influence of the rasterization for visibility testing very small, a resolution of 4096×4096 pixels was used during the measurements. This means that for each viewing direction, each test can result in at most $4096^2 = 2^{24} = 16{,}777{,}216$ visible objects, which is far more than the number of objects present in the test scenes.

The overestimation and underestimation results are shown in the box plot charts in Figure 5.10 and in Figure 5.11, respectively. Each figure contains three charts, from top to bottom, for the Scene SphereOfCubes, for the Scene PP, and for the Scene Boeing. The results are analyzed in this order in the following.

A striking feature for the Scene SphereOfCubes is the very small underestimation for the six sample points (octahedron, compare Figure 3.2(b)). For this distribution of sample points, during the visibility testing the camera looks from positions on the main axes onto the scene containing axis-aligned cubes. As only one side of a cube will be projected orthogonally to the image plane, the projected size of the cubes is minimal over all viewing directions. This leads to very little occlusion and many cubes will be visible on a single image. Therefore, the PVSs stored for the sample points contain these many cubes. When querying the sphere, the data from the nearest three sample points will be returned, leading to a PVS containing a large subset of all the cubes. A large overestimation can be observed when looking at the overestimation results. The overestimation is maximal for the sample point distribution with six sample points. One may wonder why the results look differently for the subdivided octahedron distribution (18 sample points), because it contains the same six sample points, plus additional points created by the subdivision step. From these additional points, the camera does not look onto the cubes in the way as described above for the vertices of the octahedron. Hence, their PVSs contain less cubes. When interpolating the visibility data, these smaller PVSs lead to a resulting PVS with more missing objects. Therefore, there are more missing objects for the subdivided octahedron distribution (18 sample points) than for the octahedron distribution (six sample points).

For a sample point distribution, the maximum cardinality of the underestimation indicates how many objects are missing in the worst case. A small maximum cardinality of the underestimation is desirable, because missing objects lead to errors in the rendered image. When looking at the maximum cardinality of the underestimation for the Scene SphereOfCubes, the results for the distributions with 18 and with 20 sample points are a local minimum. The overall result, e.g., the median, of these two distributions is better than the result of the three distributions with a smaller number of sample points (8, 10, and 12) and of the six distributions with a larger number of sample points (from 34 to 258). The overestimation is quite high with all distributions up to and including the one with 20 sample points.

The two charts in the middle of the aforementioned figures show the results for the Scene PP. The dodecahedron distribution (20 sample points) has a low maximum underestimation compared to distributions with a similar number of sample points. The maximum underestimation is lower than the one of all distributions with a smaller number of sample points. The maximum underestimation is even lower than those of the distributions with 34, 42, 130, and 162 sample points. Additionally, the median overestimation and the median underestimation of the dodecahedron distribution, indicated by the black bars, are lower than the respective median values of the distributions with less sample points. Overall, the median underestimation for the distributions with more than 20 sample points decreases with increasing number of sample points. But, even when using 10,242 sample points, there are still missing objects, yet, they are present in the reference distribution with 40,962 sample points. Similar, the overestimation decreases with increasing number of samples.

The two charts at the bottom of each figure summarize the results for Scene Boeing. For

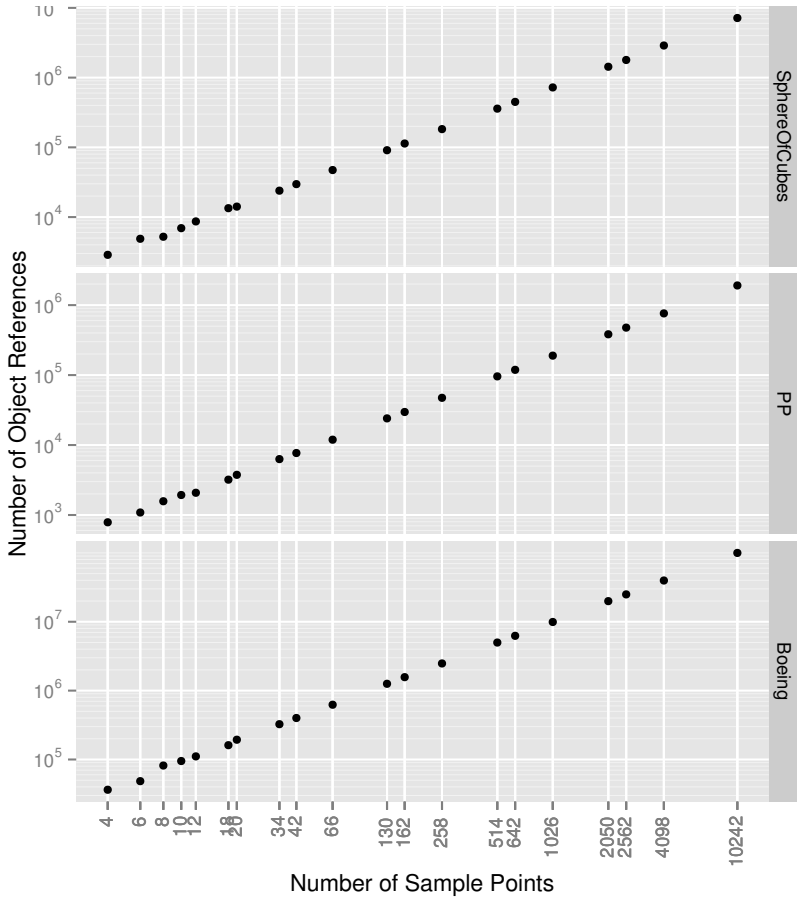


Figure 5.12: Storage space required for the sphere's visibility information when using different sample point distributions. The axes have a logarithmic scaling.

this scene, the sample point distribution corresponding to the dodecahedron is not very striking. Overall, the overestimation and the underestimation more or less decrease with increasing number of sample points beginning with 20 sample points.

5.3.2 Storage Space

To hold the visibility information of a potentially visible set in memory, a reference to every object that is element of this set has to be stored. Every sample point stores a potentially visible set, wherefore the storage space increases when increasing the number of sample points. To measure the storage space that is needed when using different distributions of sample points, the number of objects in the potentially visible sets are counted after SVS's preprocessing using the respective distribution. For each distribution, the sum over the number of visible objects of all sample points is calculated.

The results are depicted in the charts in Figure 5.12. The number of sample points as well as the number of object references are shown with a logarithmic scaling. The charts for all three scenes show a linear dependency between the number of sample points and the number of object references. Therefore, even with different visibility inside the different scenes, the storage space increases linearly when increasing the number of sample points. By keeping the number of

sample points small, the resulting memory that will be required will be small, too.

5.3.3 Conclusion

For the first two considered scenes, Scene SphereOfCubes and Scene PP, the quality of the sample points corresponding to the vertices of a dodecahedron is very good compared to the distributions with a similar number of sample points. To get a much better quality, the number of sample points has to be largely increased. The increase in the number of sample points would lead to a similar increase in storage space. For the Scene Boeing, taking more samples leads to a better quality. As the evaluation shows, using 20 sample positions yields a good trade-off between costs and benefits. Hence, this distribution is used for all the following measurements. As a side note, using the dodecahedron's 20 vertices corresponds to the maximum number of positions that can be distributed uniformly on the sphere (compare Table 5.3).

5.4 Interpolation Technique

Section 4.2 explains different ways to interpolate the visibility information stored in a visibility sphere for a queried viewing direction. The evaluation in this section shows that there is only one reasonable way to perform the interpolation in practice: Use the Max3 interpolation to take the union of the three PVSs stored in the sample points at the corners of a triangle containing the queried viewing direction. Although, this introduces an overestimation of the visible geometry, this technique effectively reduces errors due to missing objects induced by the approximation of the directional visibility with only few sample points.

To evaluate the quality of different interpolation techniques, a similar measurement as in Section 5.3 has been performed. Here, the 20 vertices of the dodecahedron have been used as sample point distribution. The visibility sphere containing the whole scene has been queried with different interpolation techniques. The queries have been performed at the 40,962 reference positions – vertices created by six edge-subdivision steps of an icosahedron – like in the previous section. At these positions, the EVS is known. The interpolation technique computes the PVS based on the data stored in the visibility sphere. Again, the cardinality of the underestimation and the overestimation are considered here to measure the quality. Mainly the underestimation is taken into account here, because it leads to missing object and therefore to image errors.

The chart in Figure 5.13 shows the underestimation results for three different test scenes. It shows that the Max3 technique always leads to the highest quality, meaning that the underestimation is smaller than it is when using other interpolation techniques. Nearest has the worst quality here. Where Nearest can only use the visibility data of a single sample point, Max3 always uses data of three sample points. Thus, for a query that lies in between the sample points, the Nearest technique produces inaccurate results. Max3 uses the data of the neighboring sample points and therefore produces a much larger PVS. The results of Weighted3 lie between those of Max3 and of Nearest. For Scene SphereOfCubes, it is nearly as good as Max3. But for Scene Boeing, it is nearly as bad as Nearest.

Using Max3 results in the best quality concerning the underestimation, but it cannot be ignored that it creates much overestimation most of the time. The overestimation results can be seen in Figure 5.14. The overestimation leads to decreased performance during rendering, because objects, which are invisible in the end, are sent to the graphics pipeline. For the sake of higher image quality, this performance decrease is accepted and Max3 is used in the following.

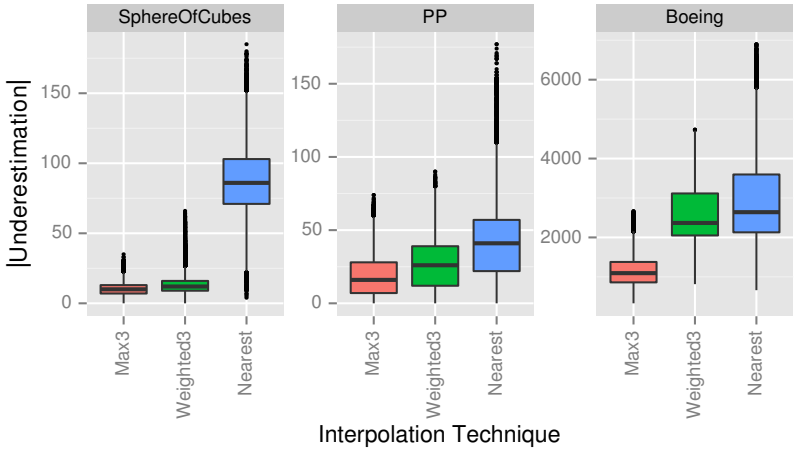


Figure 5.13: Distribution of the cardinality of the underestimation over the whole sphere surface when using 20 sample positions with different interpolation techniques.

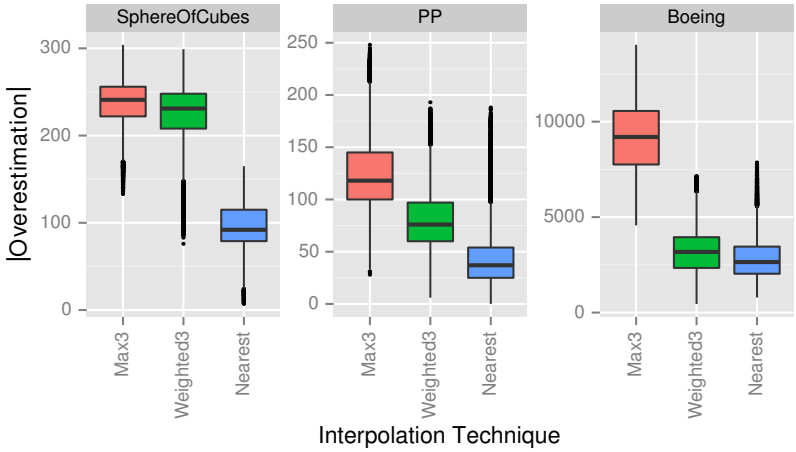


Figure 5.14: Distribution of the cardinality of the overestimation over the whole sphere surface when using 20 sample positions with different interpolation techniques.

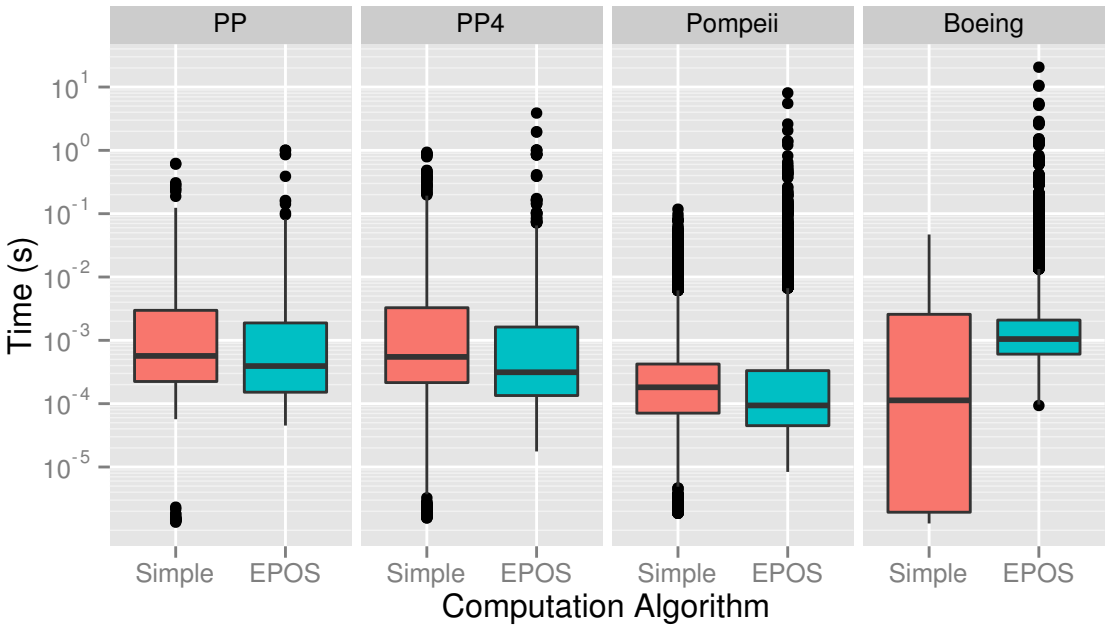


Figure 5.15: Running time distribution for the two different bounding sphere computation algorithms used in preprocessing different scenes. The vertical axis has a logarithmic scaling.

5.5 Computation of Bounding Spheres

Bounding spheres are computed for inner nodes using two different techniques, as described in Section 3.3: The first one, called *simple* in the following, takes the spheres of the child nodes and combines them to a sphere containing them. The second techniques, using the *EPOS* algorithm, collects all objects of the subtree below the inner node and computes a bounding sphere for the union of vertices of all meshes.

The following evaluation examines the costs and benefits of both techniques. The costs are the running time and memory consumption during the preprocessing. The benefits are the size of the spheres, because smaller spheres are beneficial for the rendering during runtime (see Section 4.1). In the end, advice is given on which technique should be used for SVS's preprocessing in practice.

5.5.1 Preprocessing Time and Memory Consumption

At first, the costs – namely running time and memory consumption – for the two different methods are examined.

The distribution of the running times of the preprocessing parts that compute the bounding spheres for inner nodes are depicted in Figure 5.15. Upon first look, the charts suggest that using the EPOS algorithm is faster than the simple combination of spheres, except for Scene Boeing. But, when summing up the running times of the individual steps to a total running time, the steps with a long duration gain a higher impact. These steps are the ones executed for the inner nodes in the upper levels of the scene graph. For preprocessing these nodes, a huge amount of vertices has to be collected as input for and has to be processed by the EPOS algorithm. When comparing the usage of EPOS to the simple combination of spheres, the total running time is 1.339 times

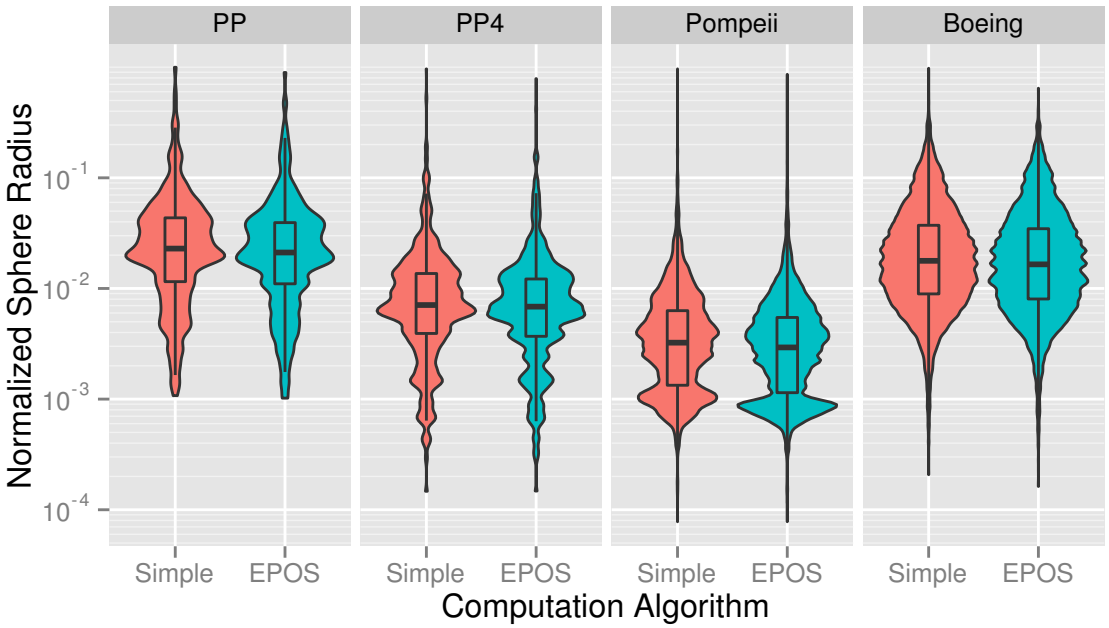


Figure 5.16: Radii of the inner nodes' visibility spheres for the two different bounding sphere computation algorithms. The vertical axis has a logarithmic scaling.

higher for Scene PP, 1.076 times higher for Scene PP4, 1.304 times higher for Scene Pompeii, and 3.003 higher for Scene Boeing. The results show that the simple combination of bounding spheres is much faster for complex scenes.

When using EPOS, preprocessing the Scene PP256 is not possible. When the preprocessing is nearly finished and the uppermost nodes in the scene graph are reached, the program crashes reproducibly. EPOS requires a set of points as input, which is the set of all of the scene's vertices for the root node of the scene graph. The program crashes when preprocessing the root node, because the amount of memory that is to be allocated is too large for the machine. For the Scene PP256, storing all vertex positions as three float values requires about 31.7 GiB of memory. The large memory consumption of the preprocessing while using EPOS is a large disadvantage. The simple combination of spheres needs much less memory, which makes it a valuable alternative for very complex scenes.

When using the simple combination, additional memory is only needed to store the vertices of a single object in a leaf node. Therefore, the memory requirements are bounded by the largest object inside the scene. In contrast, the EPOS algorithm needs memory for storing all the vertices in the scene when computing the bounding sphere for the root node. Hence, using the simple combination imposes much lower memory requirements for the preprocessing of a scene than using the EPOS algorithm.

5.5.2 Sphere Radii

Computing tight bounding spheres for inner nodes takes longer and needs more memory than just combining multiple spheres. The hypothesis is that investing these costs is worth it, because the resulting bounding spheres will be smaller. As stated above, smaller spheres can be used for more camera positions during rendering, resulting in higher performance.

The measured sphere radii for different scenes are depicted in Figure 5.16 as violin plots [HN98] together with box plots. A violin plot visualizes the density estimation of the measured data, similar to a density plot mirrored at the horizontal axis and then tilted by 90°. Such a visualization is useful to identify changes in the shape of the density estimation, which cannot be seen in the box plots. The sphere radius has been normalized, for better comparability between the different scenes, by dividing it by the maximum sphere radius inside a scene.

When using the simple combination of spheres instaed of the EPOS algorithm, the distribution of the radii moves slightly to larger values. Expressed in numbers, the sum of all radii increases by a factor of 1.115 for Scene PP, 1.125 for Scene PP4, 1.181 for Scene Pompeii, and 1.068 for Scene Boeing.

5.5.3 Conclusion

Since the increase in sphere radii is not very large when using the simple combination of spheres compared to the usage of the EPOS algorithm, it is advisable to use the simple variant. The simple variant is much faster for complex scenes. Its memory consumption is much lower and it allows the preprocessing of very large scenes, that do not fit into memory when using EPOS. Only when running time and memory consumption in the preprocessing phase is negligible and the best possible quality is to be achieved, the EPOS algorithm should be used.

To achieve the highest quality for the following measurements, the EPOS algorithm is used for all scenes except for Scene PP256. It cannot be used for Scene PP256 for the reasons already explained, which is why the simple combination is used for that scene.

5.6 Visibility Testing in Inner Nodes

When performing visibility tests to determine the visibility from the outside of each inner nodes, the existing visibility information from child nodes can be used. The first hypothesis is that using the existing visibility information speeds up the preprocessing process, because less tests have to be performed. The second hypothesis is that small objects will be missed when existing visibility information is used, because objects that were missed once due to rasterization errors cannot be hit in later tests. Both hypotheses are checked in the following. A preprocessing resolution of 4096^2 pixels has been used for the visibility tests.

5.6.1 Preprocessing Time

The running times that were measured for the visibility tests are shown in Figure 5.17 for preprocessing different scenes. Each preprocessing has been executed twice: On the one hand, the available visibility information has been used (“Yes”). On the other hand, this information has not been used and all objects in the subtree of the inner node have been tested (“No”).

The boxes in the charts show nearly no influence of the setting on the running time. For the smaller scenes – Scene PP and Scene PP4 in this case – even the outliers do not differ much (except the maximum value of Scene PP4). For the larger three scenes, the maximum values decrease significantly when using the existing visibility results. To compare the influence on the total running time, the running times for the individual visibility testing steps were summed up to a total number. When making use of the available visibility information, the total running time decreases to 95.5 % for Scene PP, to 93.5 % for Scene PP4, to 85.9 % for Scene PP256, to 90.7 % for Scene Pompeii, and to 88.1 % for Scene Boeing.

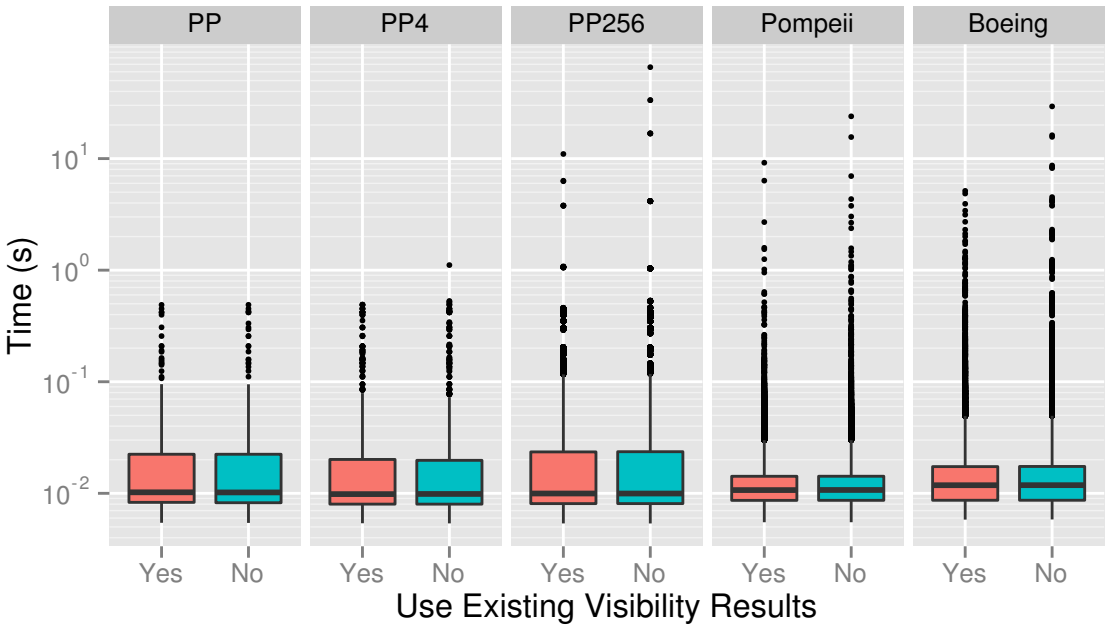


Figure 5.17: Distribution of running times of the visibility test steps during the preprocessing. The vertical axis has a logarithmic scaling.

Most probably, the difference between using and not using the existing visibility information is not larger, because the additional occlusion queries that are executed when not using the existing visibility information are not very costly. When testing the visibility of an object, the occlusion query is not immediately started for this object, which might have a high geometric complexity, but for its bounding box first. If the bounding box is invisible, no occlusion query for the object has to be started anymore. When not using the existing visibility information, the objects that have to be tested additionally are likely to be invisible, which is why their bounding boxes are likely to be invisible, too. Therefore, not many complex objects have to be tested, which explains the rather small running time differences between the two variants.

5.6.2 Quality of Visibility Information

The following measurements have been carried out with a resolution of 8192×8192 pixels $= 2^{26}$ pixels, which is intentionally higher than the resolution used for visibility testing. For each of the twenty sample positions, a measurement has been performed. For a single measurement, the camera has been positioned at the same position as it was positioned for the preprocessing (viewing direction defined by the sample position together with orthographic projection). The visibility data stored in the current sample provides the potentially visible set (PVS) for this position. The exact visible set (EVS) is determined by rendering the scene once and testing all scene objects with an occlusion query afterwards. The missing objects are determined by computing the underestimation $U = EVS \setminus PVS$. For all objects in U, the visible pixels provided by the occlusion query are added up. The sum represents the number of pixels of missing objects. To take the measurement resolution into account, the image part of missing objects is computed by dividing the sum by 2^{26}. The image part is expressed in parts per million (ppm). 1 ppm corresponds to a single pixel in an image with 1000×1000 pixels. The measurements that were

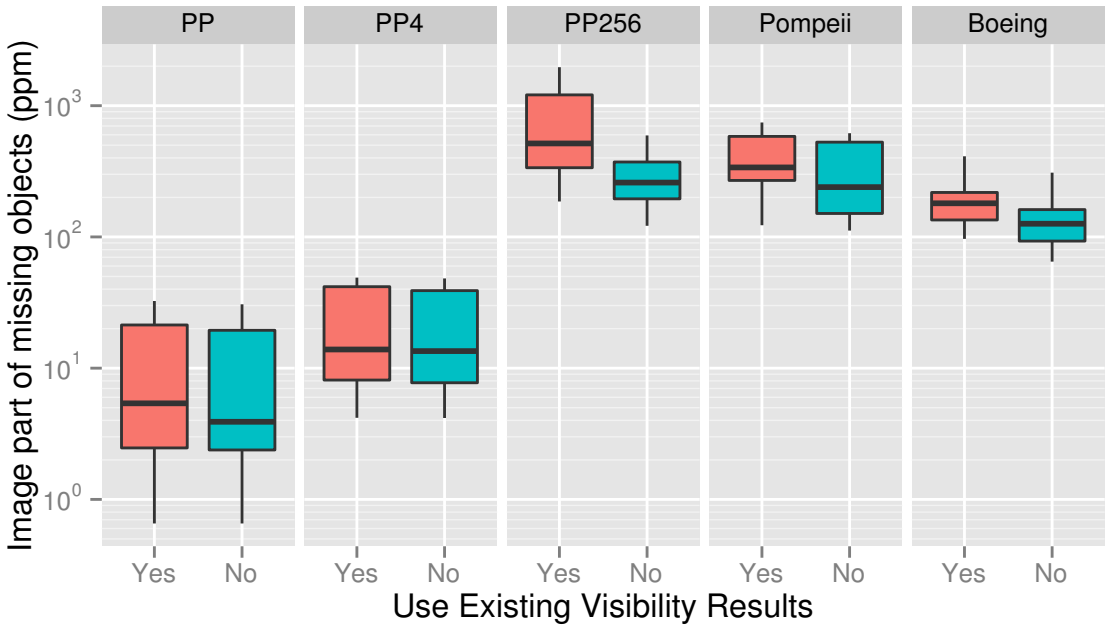


Figure 5.18: Quality of the visibility information for two different preprocessing settings. The vertical axis has a logarithmic scaling.

just described are carried out for different scenes and preprocessing settings (either using available visibility information, or not).

The distribution of the image part of missing objects is depicted in Figure 5.18. The errors that occur even when not using the existing visibility information are due to the higher resolution in the measurements. In the following, the two settings are compared to each other. For Scene PP4, there is only a very little improvement in quality when processing without using the existing visibility information. For Scene PP and Scene Pompeii, the median of the measured values drops clearly. For the very complex Scene PP256 and Scene Boeing, the whole distribution moves to smaller values. Especially for these two scenes, the running time decreased most when using the existing visibility information. Unfortunately, the quality also decreases most in these two cases. To sum up, using the existing visibility information for visibility testing leads to a lower quality of the resutling visibility information.

5.6.3 Conclusion

Using the available visibility information during preprocessing to reduce the number of visibility tests reduces the running time, which is especially beneficial for very complex scenes. For the complex scenes, the preprocessing time can be reduced by approximately 10 % to 15 %. But, due to the errors introduced by the rasterization, objects might be missed and the quality of the visibility information suffers. Therefore, it is advisable to test all objects of the subtrees without using the available visibility information. This leads to a longer duration for the preprocessing, but improves the quality of the information that is used for rendering at runtime.

In the following, the existing visibility results have not been used during the preprocessing and the tests have been executed for the whole subtree.

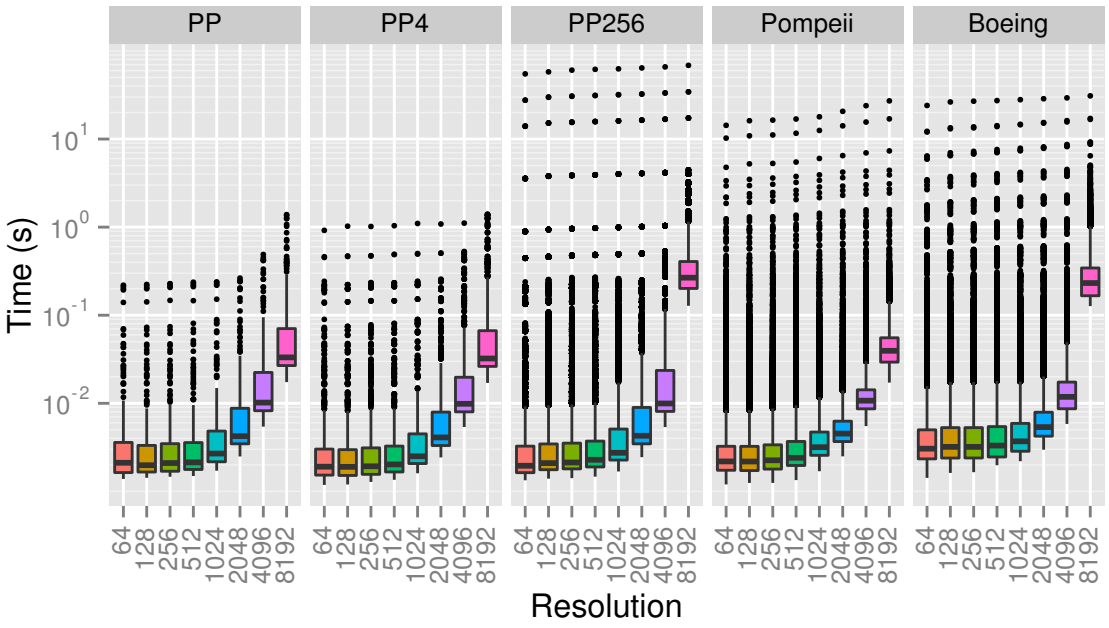


Figure 5.19: Running time of the visibility testing for different resolutions and scenes. The vertical axis has a logarithmic scaling.

5.7 Image Resolution

For the visibility tests during the preprocessing, the rasterization pipeline of the graphics hardware is used. Because the graphics hardware determines the visibility based on pixels in an image, the visibility signal that is present in the scene is discretized by this process. For every pixel, only a single object can be detected. If more than one object covers the same pixel at the same depth, it can be assumed that an arbitrary one of these object is detected. Furthermore, the graphics pipeline rasterizes a pixel only if the corresponding triangle of the object intersects the center of the pixel. During this discretization, the image resolution – measured in the the number of pixels that form the image – quantifies the sampling rate of the signal. Objects in the scene might be projected arbitrarily small on the image plane. If the resolution is too low, meaning the pixels are large, objects that are small will not be detected. The effects of the choice of the resolution on the running time and the quality of the resulting visibility information are examined in the following.

5.7.1 Preprocessing Time

The charts for different scenes in Figure 5.19 show a summary of the running time distribution for visibility testing when using different resolutions. Note the logarithmic scale on both axes. An enlargement step for the resolution quadruplicates the number of pixels.

The maximum running time for visibility testing is required for the inner nodes in upper levels of the tree and the root node. To determine the directional visibility for these nodes, (nearly) all of the scene's objects have to be tested. In the charts, the outliers for the maximum running time show that for a small scene, like Scene PP, the resolution has a high influence on the running time. The maximum increases significantly when increasing the resolution from 2048^2 pixels to 4096^2 pixels and then to 8192^2 pixels. When looking at the other scenes, the

increase of the maximum value is visible, but not as large as for the smaller scene. This leads to the conclusion that the bottleneck with very high resolutions for visibility testing in smaller scenes is the rasterization stage of the graphics pipeline. For larger scenes, however, the geometry stage of the pipeline dominates the time needed for the computations.

With increasing resolution, a clear increase of running times can be observed, when looking at the interquartile range of running times, represented by the boxes in the charts. The visibility testing steps, for which running times from this medium range have been measured, cannot have tested large subsets of the whole scene. Otherwise, the measured running times would have been larger. Therefore, the amount of geometry tested by them was only a small to medium subset of the whole scene. Thus, for these steps, the running time is not dominated by the geometry stage, and the resolution plays an important role, as can be seen from the increase in running times. When looking at the boxes, it follows that the running time for these visibility testing steps grows super-linear in the resolution.

To compare some numbers, the running times of all visibility testing steps have been summed up to generate an total running time for one specific preprocessing setting. For both Scene PP256 and Scene Boeing, increasing the resolution from 512^2 pixels to 1024^2 pixels increases the total running time for visibility testing only by a factor of approximately 1.1. But, when reaching the highest tested resolution of 8192^2 pixels, the increase of the running time is very high. Between the resolutions with 4096^2 pixels and 8192^2 pixels for the Scene PP256, the total running time increases by a factor of 11.5. For the Scene Boeing, the same increase of the resolution makes the visibility testing 15.0 times slower.

Additionally to the greater number of pixels that has to be filled during the rendering, the amount of memory that has to be handled by the graphics card increases. Images with an edge length of 4096 pixels, or especially 8192 pixels, are larger than usual values used for real-time rendering applications (e.g., 1080p with 1920×1080 pixels = 2,073,600 pixels is less than half of 4,194,304 pixels = 2048×2048 pixels). An uncompressed framebuffer with 32 bits color values, 24 bits depth values, 8 bits stencil values or padding, and a resolution of 8192^2 pixels is 512 MiB in size. Storing such a framebuffer in graphics memory reduces the amount of other data (meshes, textures, etc.) that fits into graphics memory. Accessing the other data from main memory during rendering is much slower, which explains the high running times for the preprocessing with high resolutions.

5.7.2 Quality of Visibility Information

The quality of the visibility information was measured as it was done before in Section 5.6 and the image part of missing objects was calculated in the same way. Since the measurements were performed with a resolution of 8192×8192 pixels, the measured results for the matching preprocessing resolution of 8192×8192 pixels might be distorted. The quality results can be seen in the charts in Figure 5.20. As in the previous figure, both axes have a logarithmic scaling. For the simpler Scene PP and Scene PP4, the quality increases linearly in the resolution. For the three complex ones – namely Scene PP256, Scene Pompeii, and Scene Boeing – the decrease of image error is even super-linear.

5.7.3 Conclusion

The quality of the visibility information increases super-linearly with the resolution. But, the costs, in this case the running time, also increase super-linearly. In practice, a resolution should be used

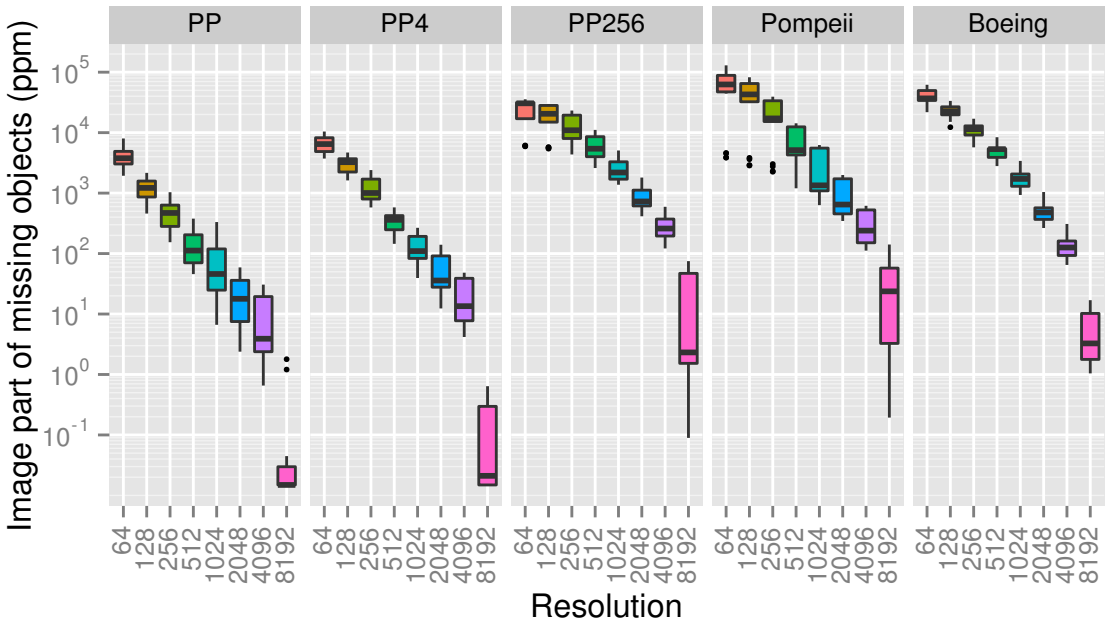


Figure 5.20: Quality of the visibility information for different scenes with changing preprocessing resolutions. The vertical axis has a logarithmic scaling. A resolution of 8192^2 pixels has been used for the measurements.

that is at least as high as the resolution that is used for rendering at runtime later on. If a smaller resolution would be used, many objects that should have been shown in the rendered image are not detected during the preprocessing, because of the undersampling by the rasterization. Because a resolution of 1280×720 pixels is going to be used for the runtime tests, the preprocessing resolution was set to 1024^2 pixels.

5.8 Errors introduced by Different Projections

During the preprocessing, an orthographic projection is used during the visibility tests (see the description of the process in Section 3.5). For the walkthrough at runtime, a perspective projection is applied. By using the perspective projection, objects might become visible that were hidden in the orthographic projection. In this section, the errors that are introduced by using the two different projections are evaluated.

To examine the errors, a visibility sphere is again queried for different viewing directions. Here, 10,000 random viewing directions were used for every measurement. As in Section 5.3.1, the random viewing directions were chosen uniformly at random as spherical coordinates with inclination $\varphi \sim \arccos(U(-1,1))$ and azimuth $\theta \sim U(0,2\pi)$. For each of these viewing directions, at first the underestimation with orthographic projection is determined. This is done by querying the sphere for the PVS and by computing the EVS with visibility testing under orthographic projection. The underestimation is then the EVS without the elements in the PVS, as defined before. This special underestimation set contains the objects that are missing due to non-projection errors (e.g., other sampling errors as discussed in this chapter). It is stored to be used in the following computations. For the following measurements, a perspective projection

with different aperture angles is used. Beginning with 15°, the angle is increased until 90° is reached. Since a quadratic viewport is used, the angle corresponds to the horizontal as well as to the vertical frustum angle. The camera is always placed such that the frustum fully contains the sphere. For the different frustum angles, the underestimation is determined. To get rid of errors from other sources, the special underestimation set that was stored for the orthographic projection is subtracted from the perspective underestimation set. The resulting set only contains the errors that were introduced by the different projections.

The results of these measurements can be seen in the charts in Figure 5.21 for two different scenes. Additionally to the description above, the measurements have been performed with different interpolation techniques, namely with NEAREST and MAX3.

When considering the Scene PP, for a majority of viewing directions, the errors introduced by using a perspective projection are very small. But, there are several outliers corresponding to viewing directions with a large image part of missing objects. For the NEAREST interpolation, the median is below 6 ppm. The amount of errors is further reduced when using the MAX3 interpolation: For this, the upper quartile is lower than 4 ppm for all frustum angles.

For the Scene BOEING, the errors introduced by the perspective are rather large. For NEAREST interpolation, the median is 123.2 ppm for 15° and 250.8 ppm for 90°. Again, the errors – especially the outliers – are significantly lower when using the MAX3 interpolation. The median for 15° is 112.5 ppm and 179.1 ppm for 90°. The differences between the interpolation techniques could be expected, when looking at the results in Section 5.4.

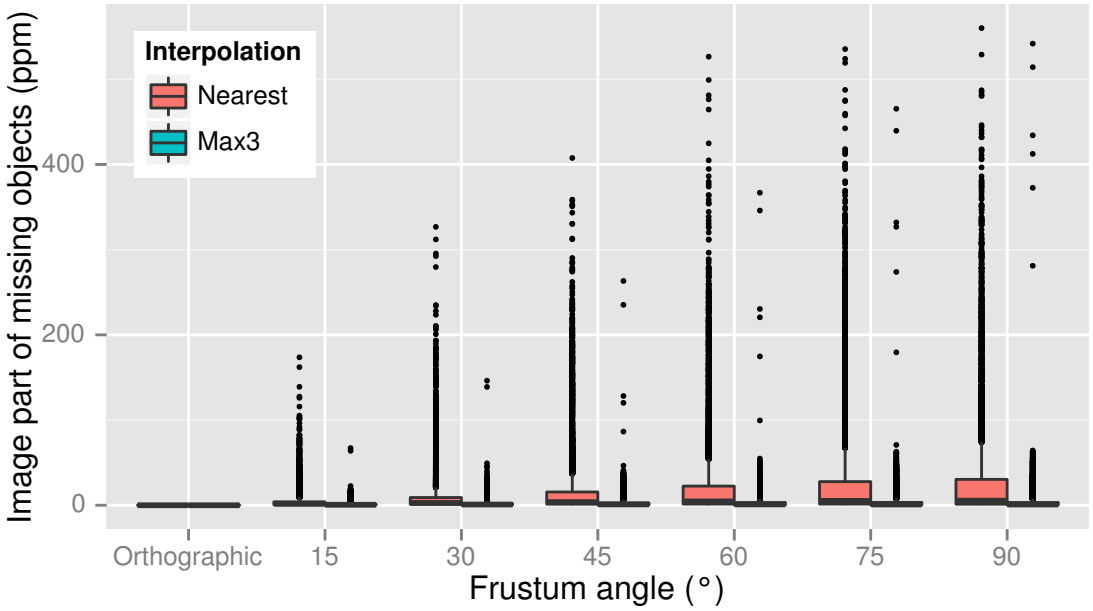
For both scenes and NEAREST interpolation, it can be seen that the size of the image part that is covered by the missing objects increases with increasing frustum angle. In the Scene PP when using MAX3, the same behavior can be seen when looking at the maximum value for every frustum angle. But, in the Scene BOEING when using MAX3, there is a decline in error values between frustum angles 60° and 75°, and between 75° and 90°. This is due to the fact that the cardinality of the EVS decreases with increasing frustum angle. Some objects disappear due to the high distortion introduced by such high aperture angles.

To sum up, using different projections results in errors. Using a perspective projection during the preprocessing would not solve the problem: If a different frustum angle was used in the preprocessing than during runtime, similar problems would arise. Furthermore, re-using visibility results from smaller spheres in the preprocessing step for a larger sphere would no longer be possible. It could be observed that the amount of errors is reduced when using the MAX3 interpolation instead of the NEAREST interpolation.

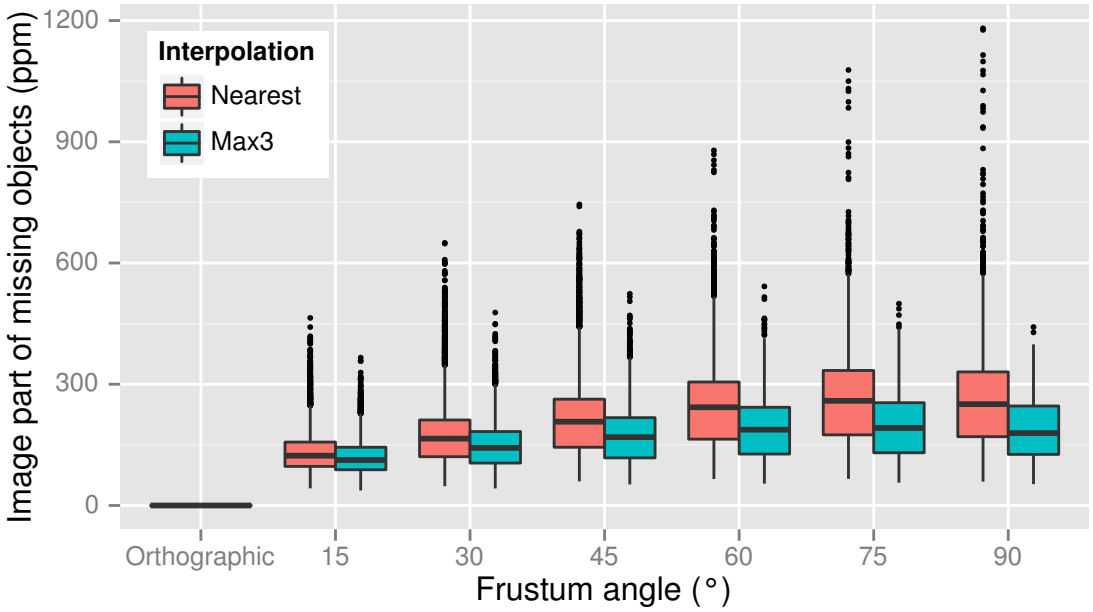
5.9 Storage Space

The storage space that is required by different parts of the scene description and by SVS's visibility data is examined in this section. This helps to get an impression about SVS's storage requirements in practice. In order to describe the scene, the nodes of the scene graph, the meshes, and the textures need to be stored. SVS stores the visibility spheres that contain a bounding sphere, a triangulation of the sample points, and the visibility data of the sample points. The storage space has been determined by counting the number of bytes that are allocated by different objects during runtime of the program. Table 5.4 summarizes the measured values. The last column shows the percentage of SVS's storage space in relation to the scene's storage space, which consists of the space needed for the scene graph, the meshes, and the textures.

For the Scene PP4 and the Scene PP256, the meshes of the Power Plant model are reused.



(a) Scene PP.



(b) Scene Boeing.

Figure 5.21: Perspective errors for two interpolation techniques: Image fraction of missing objects in parts per million for different frustum angles in different scenes (measured with 10,000 random views tightly enclosing the scene's outer sphere; 4096 × 4096 pixels resolution).

Table 5.4: Space needed to store different kinds of data for the test scenes.

Scene	Scene Graph (MiB)	Meshes (MiB)	Textures (MiB)	SVS (MiB)	Ratio $^{SVS}/_{Scene}$ (%)
PP	0.251	357.238	0.000	3.256	0.9
PP4	1.035	357.238	0.000	13.874	3.9
PP256	62.172	357.238	0.000	789.297	188.2
Pompeii	56.784	1335.366	319.457	455.554	26.6
Boeing	37.443	8175.625	0.000	987.390	12.0
PPBoeing5	187.468	8532.863	0.000	4940.208	56.7

Therefore, their storage space for the meshes is the same as for the Scene PP. Due to the additional inner nodes in the scene graph and multiple instances of the Power Plant model with potentially visible objects, SVS's data cannot be reused. Hence, it is larger for the combined scenes than it is for the single Power Plant model. When computing the ratio of SVS's data to the scene's data without mesh instances, the meshes' size grows by a factor of 4 and 256, respectively, and SVS needs only 1.0 % storage space of the Scene PP4 and 0.9 % of the Scene PP256, as it does for Scene PP. The situation is similar for the Scene PPBoeing5. There, the meshes for the Boeing 777 model are reused, but the visibility data for them is duplicated. When this shortcoming in the implementation would be fixed, the visibility data for the Scene PP and the Scene Boeing would be required only once (3.256 MiB + 987.390 MiB = 990.646 MiB). Then, the ratio would decrease to 11.4 %.

Without the usage of mesh instances, the highest ratio was measured for Scene Pompeii. There, SVS needs a little bit more than a quarter of the scene size.

To conclude, the memory required to store SVS's data is in general small compared to the scene size. Concerning the storage space, no issues arise when SVS is used in practice. A machine that is able to load and hold the scene data in memory should have no problems with SVS's additional data.

5.10 Total Preprocessing Time

If SVS shall be used in practice, it is important to know how long the preprocessing takes. Imagine an architect who wants to visualize a newly designed building and who does not want to wait several days for the first image to show up on the screen. For this reason, short preprocessing times are important for practical usage.

The total time that was required by SVS's preprocessing has been measured for the test scenes. For the preprocessing, the parameter values that were determined in the previous sections of this chapter were used. The resulting data of this preprocessing is used in the following for the measurements at runtime.

Table 5.5 contains the measured total times for the preprocessing of different scenes in minutes. For the smaller scenes, Scene PP and Scene PP4, the preprocessing needs less than a minute. Actually, it would be practicable to run the preprocessing on the fly every time after loading these scenes. For the Scene Pompeii the preprocessing runs a few minutes, and for Scene Boeing it needs less than 20 minutes. Only for the most complex Scene PP256, over an hour of preprocessing time is required.

Table 5.5: Time needed to preprocess the different test scenes with SVS.

Scene	Preprocessing Time (in minutes)
PP	0.152
PP4	0.702
PP256	74.855
POMPEII	6.614
BOEING	18.356

To sum up, the preprocessing of SVS is very fast, especially compared to other preprocessed visibility techniques (see Section 5.12.3). Even the most complex scene used here can be preprocessed in less than one and a quarter hours. Such short waiting times plead for SVS's usage in practice for rendering complex scenes.

5.11 Camera Paths

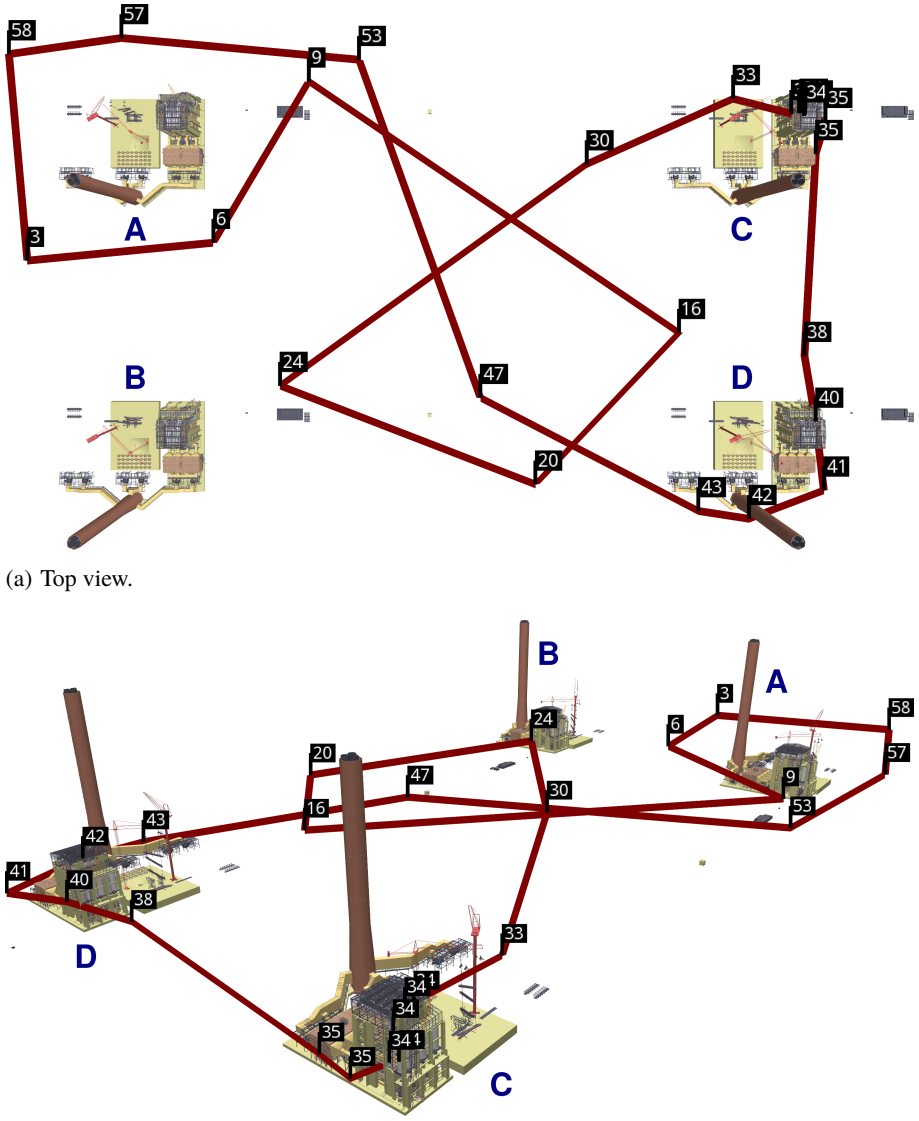
For the four scenes Scene PP4, Scene PP256, Scene POMPEII, and Scene PPBOEING5, measurements at runtime have been performed along camera paths on a workstation. The smallest scene Scene PP has been used for measurements on mobile devices. The camera paths for these scenes are introduced in this section. The positions on a camera path are specified in seconds with a walk over the path beginning at 0 s. For the runtime measurements, the camera path is sampled at distances of 0.1 s along the path. There is one exception: For Scene PP256, the distance between measuring points is reduced to 0.01 s, because the path has a shorter total length of time.

5.11.1 Camera Path in Scene PP

In order to perform the measurements on mobile devices, the Scene PP has been used. Here, the camera path is the same as the camera path in Scene PPBOEING5 (see Section 5.11.5 for a description and images of the camera path).

5.11.2 Camera Path in Scene PP4

The camera path in Scene PP4 is shown in Figure 5.22. The capital letters A, B, C, and D will be used in the description of the path to refer to the four Power Plant models as shown in the images. At the beginning of the camera path (top left of Figure 5.22(a), top right of Figure 5.22(b)), the camera looks at Power Plant model A in front with the three other models in the background. The camera circles around the building, still facing it, until only this building is inside the view frustum at 6 s. Then, the camera starts to turn right and at 9 s, it faces the two buildings C and D that were in the background before. Until reaching the waypoint at 16 s, the path leads towards D. While going to position 20 s, the camera turns right and looks at the two Power Plant buildings A and B. It keeps turning right, until it faces the buildings C and D again at 24 s. It runs towards the building C until 33 s. While looking in the direction of D, the camera strides sideways through building C. After leaving the building on the other side, it still goes sideways, now towards the building D, and looking at A and B. Beginning at approximately 39 s, the camera is directly looking at the building D, which occludes the rest of the scene. The path leads around the building



(a) Top view.

(b) Oblique view.

Figure 5.22: Camera path inside the Scene PP4. The numbers denote the waypoints' times in seconds on the path.

and is behind D's chimney at 42 s. At position 43 s, the camera looks in the direction of building C, which is hidden behind parts of building D that are in front. Then, the camera reaches the open space between the buildings again and starts to turn left. The two buildings A and B are inside the view frustum at the waypoint at 47 s. Turning left while moving, those two buildings are still inside the view frustum at 53 s. The camera keeps turning left, D enters the frustum at about 56 s, and C at about 57 s. The last waypoint of the path is reached at 58.4 s and closes the cycle.

5.11.3 Camera Path in Scene PP256

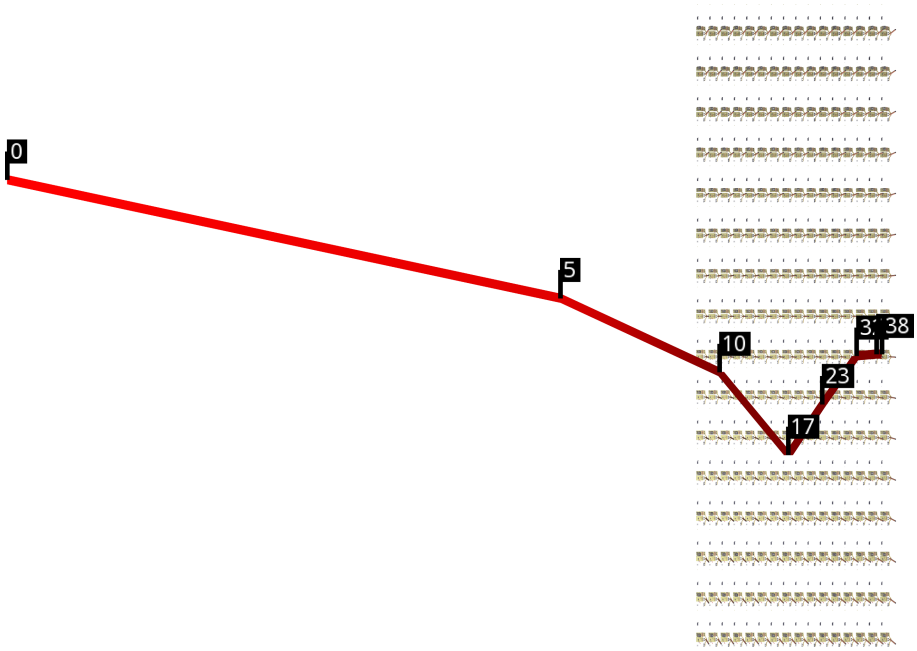
The camera path for the Scene PP256 is shown in Figure 5.23. It starts outside the scene at a large height. During the whole traversal of the path, the camera looses height. In the following description, a row means the arrangement of 16 Power Plant models that can be seen as horizontal line of Power Plant models in Figure 5.23(a). At the beginning, the camera looks towards the 16×16 grid of Power Plant models. The path leads towards the scene and the camera turns slowly right. At about 3 s, the Power Plant models on the left side of the screen begin to leave the view frustum. When reaching the waypoint at 10 s, seven rows of Power Plant models are, at least partly, inside the view frustum. Beginning from that waypoint, the camera turns left and moves slowly towards the waypoint at 17 s. Then, it moves forward facing the waypoint at 23 s. On the way to the waypoint at 32 s, it turns slowly right. There, the viewing direction is aligned with the row of Power Plant models and five models are, at least partly, inside the view frustum. Until the end of the path, the camera goes towards the last Power Plant model in the row. At the end, only this single Power Plant model is partly visible on the screen. The path has an overall length of 38 s.

5.11.4 Camera Path in Scene Pompeii

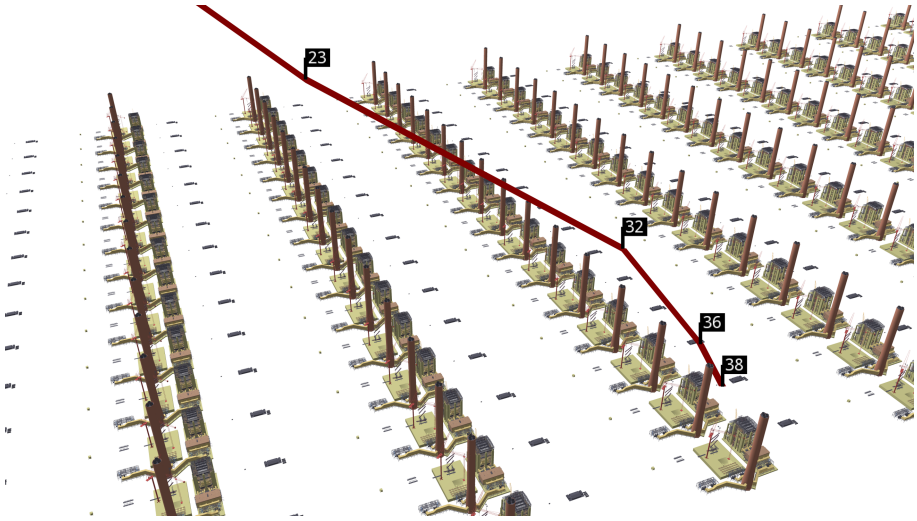
The path for Scene Pompeii represents a camera flight over the city. It is a large loop and is shown in Figure 5.24. The height over ground varies slightly over the path, but it is always above the roofs of the buildings. A walk over the path takes 145.8 s. At the beginning, only a part of the city is visible. The camera moves forwards and the amount of geometry inside the view frustum decreases. After 9 s, the camera turns left, and more and more parts of the city enter the view frustum. Until the waypoint at 81 s, the camera flies to the other side of the city with slight turns to the right and to the left. There, the amount of geometry reaches a local minimum, as the camera looks away from the city. It turns left until reaching the waypoint at 93 s, where large parts of the city are visible and the amount of geometry inside the view frustum reaches the global maximum. When moving towards the waypoint at 101 s, this amount decreases a little bit, but increases again on the way to the waypoint at 112 s. Then, the camera turns right and flies back to the part of the city where it started. During this flight, the visible part of the city gets smaller and smaller.

5.11.5 Camera Path in Scene PPBoeing5

The annotated red path in Figure 5.25 is used for the camera movement for measurements in the Scene PPBoeing5. The same camera path is also used for the Scene PP. Starting with a view that fully contains the Power Plant model, the camera moves forwards while turning to the right. At position 29 s, the camera looks into the direction of the chimney. From there, it moves behind the Power Plant building and circles it, while looking at it. Between 44 s and 53 s, the path gains height. It then leads straight towards the chimney until the waypoint at 66 s. Short



(a) Top view.



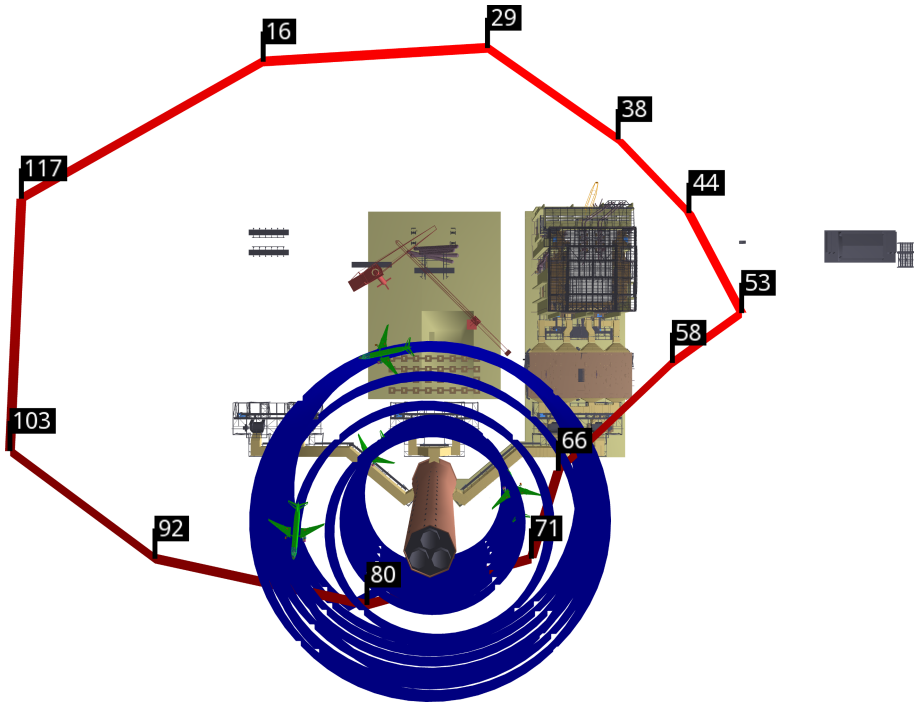
(b) Oblique view.

Figure 5.23: Camera path inside the Scene PP256. The numbers denote the waypoints' times in seconds on the path.

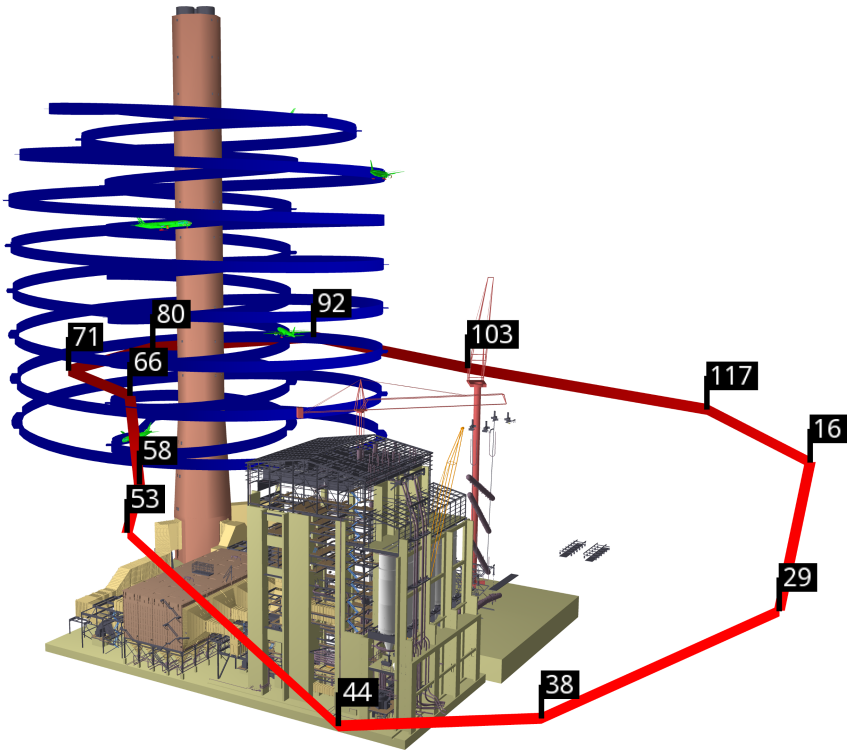
(a) Top view.

(b) Oblique view.

Figure 5.24: Camera path inside the Scene Pompeii. The numbers denote the waypoints' times in seconds on the path.



(a) Top view.



(b) Oblique view.

Figure 5.25: Camera path (red) and animation path (blue) inside the Scene PPBoeing5. The numbers denote the waypoints' times in seconds on the path.

before reaching it, it begins to turn right and passes the chimney on the left side. Because of the turning, the Power Plant building enters the view frustum at approximately 76 s again. At 92 s, the camera looks at the top of the chimney and moves backwards. From 103 s until the end at 117 s, the camera moves to the left, while looking at the Power Plant model.

The five animated Boeing 777 models are flying on a helix path, shown in blue in Figure 5.25, around the chimney of the Power Plant. The inner helix with smaller radius leads the planes upwards. When the topmost point of the path is reached, the planes fly downwards on the outer helix with larger radius. The path is a cycle, so that the planes start over when they reach the end of the outer helix at the bottom and fly upwards again. The images in Figure 5.25 show the starting positions for the five Boeing 777 models.

5.12 Other Occlusion Culling Algorithms for Comparison

To be able to compare SVS's results to the results of other algorithms, two state-of-the-art occlusion culling algorithms are used: The first is the *Coherent Hierarchical Culling Revisited (CHC++)* online occlusion culling algorithm [MBW08]. The second is the *Adaptive Global Visibility Sampling (AGVS)* [Bit+09] that is a preprocessed visibility algorithm. These algorithms were implemented in PADrend. For both algorithms, the same scene graph as for SVS is used. This means, all visibility tests, independent of the occlusion culling algorithm, are executed on the same set of objects.

In the following sections, the measured results of both algorithms are compared to SVS's results: For image quality measurements in Section 5.13, only AGVS is used, because CHC++ always renders a correct image. For performance measurements in Section 5.14, CHC++ as well as AGVS is used.

First, CHC++ (Section 5.12.1) and AGVS (Section 5.12.2) are described. Following, the preprocessing of a scene by AGVS is compared to SVS's preprocessing of that scene (Section 5.12.3).

5.12.1 Coherent Hierarchical Culling Revisited (CHC++)

The CHC++ algorithm is a conservative culling algorithm that does not suffer from missing objects in the rendered image like SVS does. It uses hardware-accelerated occlusion queries to determine the visibility of objects at runtime. These queries are not directly executed for the objects, but for the bounding boxes of the nodes in the scene graph. If a query result tells that an inner node is invisible, the subtree below this node does not need to be traversed, because all objects in this subtree will also be invisible. To achieve a high rendering performance when using occlusion queries in general, the main challenge is the execution of the occlusion queries without having to wait for their results. CHC++ applies many sophisticated measures to prevent idle waiting periods. As one important measure, it takes advantage of temporal coherence and remembers the previous visibility status of nodes. CHC++ is chosen, because it works well for many different scene types without the need for manual fine tuning of parameters. SVS is compared to an online occlusion culling algorithm to reveal the strengths and weaknesses of both types of algorithms. In the end, this allows the identification of regions in a scene where the usage of one of those two types of algorithms is advisable (see Section 5.14).

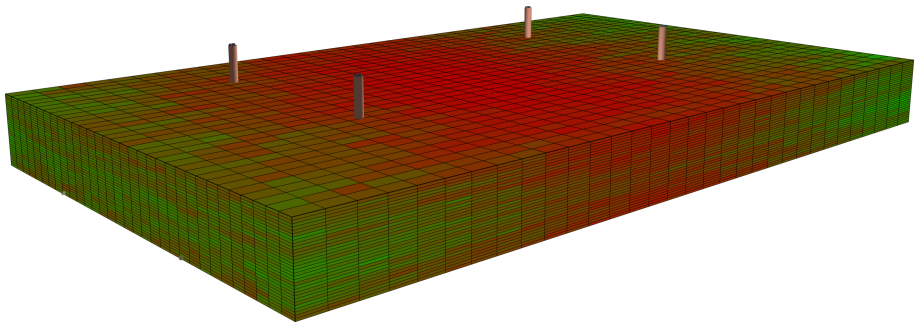


Figure 5.26: 32 × 32 × 32 grid of view cells that were used for AGVS in the Scene PP4. The chimneys of the four Power Plant models stick out of the view cells. Each view cell is colored depending on the number of triangles potentially visible from its region: green denotes few potentially visible triangles, red denotes many potentially visible triangles.

5.12.2 Adaptive Global Visibility Sampling (AGVS)

The AGVS algorithm is a preprocessed visibility algorithm that works on a given set of view cells. In contrast to SVS, the visibility is not determined by using rasterization, but by casting rays against the geometry of the scene. The rays are created from random samples. During the preprocessing, AGVS uses five different random sample distributions to generate these samples. The sample distribution that is used to generate a sample is itself also randomly selected. The probability to select a sample distribution depends on the amount of new visibility information that samples from this distribution contributed in the past. Every sample has an origin and a direction. For each sample, beginning at the sample's origin, a forward ray is cast into the sample's direction and a backward ray is cast into the sample's reverse direction. For each ray that hits an object, the given view cells are intersected by that ray. The object that was hit is added to the PVSs of the intersecting view cells. AGVS estimates the error of its current visibility data and stops the preprocessing when this error drops below a threshold.

The AGVS algorithm has been selected, because – according to its authors – it is able to process complex scenes in a short time. It is a preprocessed visibility algorithm as SVS, but uses view cells, whereas SVS uses direction-dependent visibility. The comparison of SVS and AGVS shall reveal advantages and disadvantages of this new kind of visibility data storage for complex scenes.

5.12.3 Preprocessing of Scene PP4 by AGVS

Since there is no source code publicly available for AGVS, it has been implemented in PA-Drend from scratch. This implementation is much slower compared to the numbers given by AGVS's authors. Due to long preprocessing times, only the Scene PP4 was preprocessed with this implementation of the AGVS algorithm. The preprocessing of this scene by AGVS took 1223.555 minutes, which is over 20 hours and 23 minutes. In contrast to the original algorithm, which uses a special technique [MBW06] to generate the view cells, 32,768 view cells arranged in a 32 × 32 × 32 grid were used. The region that is subdivided by the view cells was chosen to contain the camera path of this scene (defined in Section 5.11.2). These view cells can be seen in Figure 5.26. The preprocessing was stopped when the pixel error that is estimated by AGVS dropped below 100 pixels in a 1024 × 1024 pixels image. The naive, single-threaded ray

caster that is implemented in PADrend was able to shoot 3288–3553 rays per second. The authors of AGVS stated that their ray caster is able to shoot between 100,000 and 1 million rays per second depending on the scene on a machine with eight CPU cores. When using the average number of rays per second for both ray tracers, PADrend's ray caster is $^{550000}/_{3420.5} \approx 160.8$ times slower. When using this factor to divide the total preprocessing time of AGVS, it is reduced to 7.6 minutes. In comparison, SVS needed only 0.702 minutes to preprocess this scene. To store the view cells and the visibility data generated by AGVS, 1652.601 MiB of memory are required. SVS's data requires 13.874 MiB for the Scene PP4. Even if the implementation of AGVS is worthy of improvement, the comparison of both algorithms concerning the preprocessing time and the amount of memory required for the visibility information shows that SVS is superior in both aspects. In terms of the preprocessing, SVS's direction-dependent visibility information seems to be better suited for handling complex scenes than the traditional view cell approach.

5.13 Image Quality

To evaluate an approximate occlusion culling algorithm for practical usage scenarios, it is not enough to only measure the quality of the approximation of the exact visible set. It is more important to determine how severe missing objects appear in the rendered image. For measuring this severity, the human perception has to be taken into account.

In order to quantify the image quality for many positions on a camera path, an automatic technique is needed. Here, for the measurements the structural similarity (SSIM) [Wan+04] is calculated. It compares an image rendered by SVS to a correct image that is displayed by the z-buffer algorithm with frustum culling. The rendered images have a resolution of 1280×720 pixels. The SSIM technique detects changes in the structure of an image by comparing the pixel neighborhoods of all pixels in the two images. An SSIM value of one means that the images are equal. Relative ratings (e.g., image A looks better than image B), as produced by using SSIM, aligns better with human perception than non-structure-oriented techniques like PSNR [for a discussion, see WB09].

Sometimes, errors that are small compared to the dimensions of the image, like noise or aliasing artifacts, are given a rather large weight when using SSIM, because they look like structural differences in the image. To reduce the weight of such errors, additionally an image pyramid [Bur81] is used. SSIM is calculated for the full-size images and for multiple versions with halved resolution. Small artifacts are filtered out by the reduction of the resolution. The arithmetic mean of the multiple SSIM calculations for the different resolutions is used as final image quality value.

To relate the image quality that is quantified by SSIM with the quality of the visibility information, the underestimation of the exact visible set is measured for the positions on the camera path. As for the other measurements in this chapter, the visible size of the objects in the underestimation set is summed and expressed as image part of missing objects in parts per million.

5.13.1 Image Quality Results for Scene PP4

The charts in Figure 5.27 show the measurements for the Scene PP4. When comparing SVS's results in both charts, it is noticeable that the lows in the image quality do not always correspond to highs in the image part of missing objects. For SVS's low in image quality at approximately 2 s, there is a corresponding high in the image part. But, for the low shortly after 40 s in the

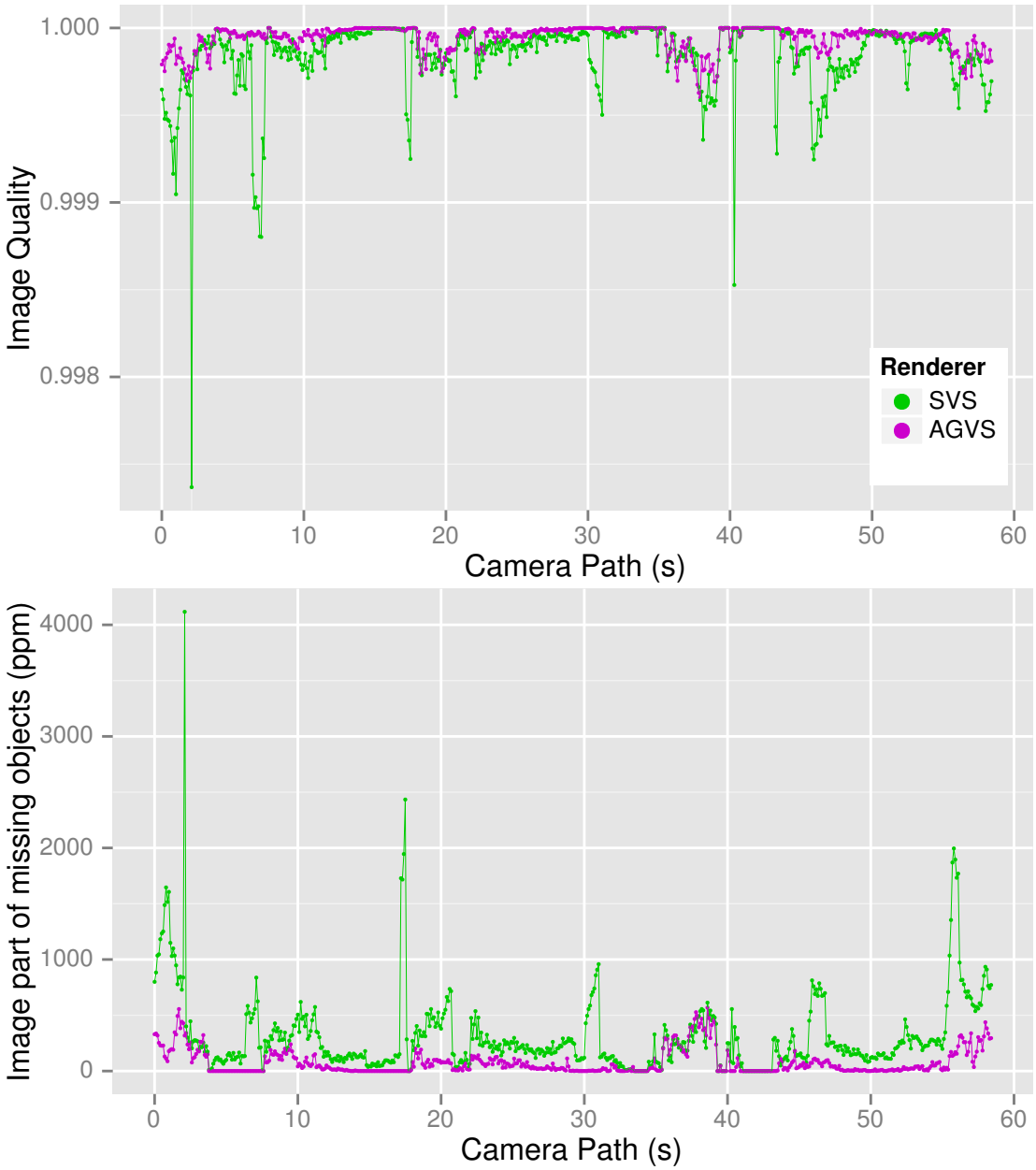


Figure 5.27: Image quality and quality of visibility information measured over the camera path in Scene PP4.

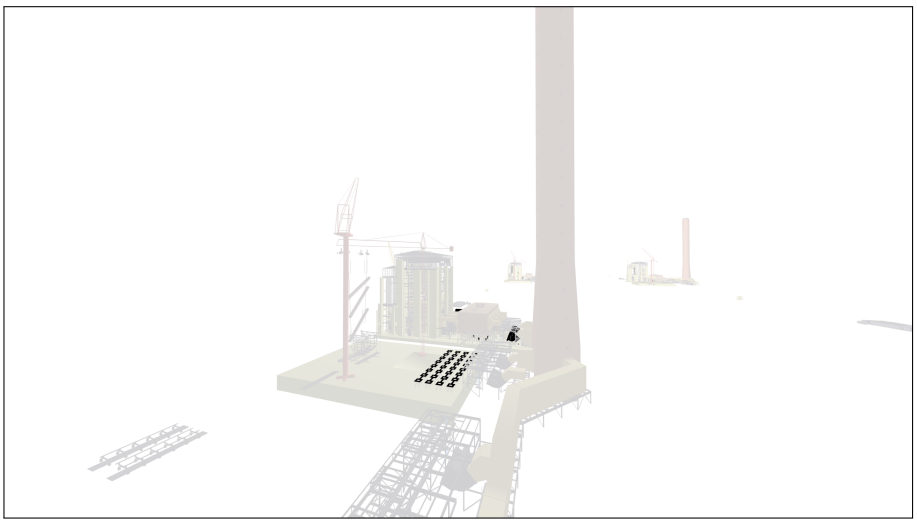


Figure 5.28: Image showing the objects in black that were missed by SVS. It has been produced from two screen shots that were taken at the position with worst image quality (position 2.1 s) on the camera path in Scene PP4. The rendered image has been brightened to increase the perceptibility of the black areas.

upper chart, there is no match in the lower chart. This emphasizes the importance of performing image quality measurements on the rendered images. Except for very few positions, SVS's image quality stays above 0.999. The experience shows that values as high as these indicate that there is nearly no error visible in the image.

AGVS provides an even better image quality than SVS. AGVS's image quality is always above 0.9995. Also, the image part of missing objects is smaller than the one of SVS for almost all positions.

Here, the quality of the visibility information, and as a result the image quality, could be improved by providing more preprocessing resources to the algorithm. If the threshold for terminating AGVS's preprocessing was lowered, the preprocessing would run longer, but more visible objects could be detected. If the number of sample points and the image resolution for visibility testing for SVS's preprocessing would be increased, the preprocessing time would also be increased, but the amount of underestimation would be decreased (see Section 5.3 and Section 5.7).

For the Scene PP4, a minimum image quality value of 0.997368 for SVS has been measured at position 2.1 s. For this position, Figure 5.28 shows the rendered image by SVS with the missing objects highlighted in black. Please note that by highlighting the objects in the images in this way, the errors appear much more prominent than they are perceived when looking at the rendered image. There are three larger clusters of black areas that arise from objects that are missing from the foremost Power Plant model. They can be perceived as distracting changes when comparing SVS's rendering to the original image created by the z-buffer algorithm. When looking only at SVS's image, they do not stand out and are noticed only when looking at the image very carefully. Further objects are missing from the two Power Plant models in the back, but their visible size is so small that they are hardly visible on the image. All in all, the image errors are rather small and the overall worst image quality that was measured on the camera path is acceptable.

5.13.2 Image Quality Results for Scene PP256

Figure 5.29 contains the charts depicting the measured results for Scene PP256. Here, a correlation between the image quality and the image part of missing objects becomes obvious when comparing both charts. Until approximately 3 s, the camera moves towards the scene and the outermost visibility sphere is queried for visibility data. The image quality decreases, because the visible size of missing objects increases with decreasing distance to the camera. Then, the camera enters the sphere and smaller visibility spheres are used that provide visibility data that is more exact. Between 12 s and 23 s, the camera pans and several Power Plant models are still far away. This leads to the lowest image quality, as explained below. During the end of the path, the camera focuses a single Power Plant building. Relatively small visibility spheres are used and the image quality is very high: most of the time at least 0.9995.

The minimum image quality with an value of 0.997684 occurred at 20.45 s on the camera path in Scene PP256. In Figure 5.30, an image at this position with the missing objects highlighted in black is shown. In the foreground, there are only very small black areas created by objects that are missing from the Power Plant buildings in the three rightmost rows of models. Since the camera is standing inside the large sphere that contains these rows, smaller spheres are used to provide the visibility data. In the upper left part of the image, several black areas are visible at the Power Plant models that stand in the rows that are further away. For these rows, a large visibility sphere is used, because the camera is standing outside of this sphere. Due to the large size of the sphere and the limited image resolution during preprocessing, small objects are missed during the visibility sampling. Overall, the erroneous areas are small, appear in the far zone of the scene, and do not hinder the perception of the silhouette of the model. Therefore, the maximum degradation of the image quality by SVS for this scene is fortunately quite small.

5.13.3 Image Quality Results for Scene Pompeii

The values measured for the camera path in Scene Pompeii are visualized in the charts in Figure 5.31. Here again, the two charts show similarities (e.g., at 1.5 s, 21 s, 29 s, 139 s). Over the camera path, the measured values fluctuate much. During the flight over the city scene, even small changes in the position of the camera can lead to large changes in the visibility. Furthermore, discontinuities emerge when the camera enters a visibility sphere. Most of the time, the measured image quality value is above 0.99.

At position 139.4 s, the minimum image quality value of 0.97963 was measured. This value is significantly smaller than the value measured for the previous two test scenes. There are three other positions where the image quality value drops below 0.985. In Figure 5.32, the missing objects for the worst position can be seen as black areas. There are only very little errors in the foreground. The largest errors occur in the middle left and top left parts of the image. The missing objects are parts of the road and lower parts of walls. Most likely, these parts were occluded by parts at a greater height, e.g., roofs, during the visibility sampling. When a visibility sphere for multiple houses is created and preprocessed, a situation with such an occlusion cannot be prevented. Unfortunately, the missing objects are quite distracting for a human viewer. However, situations with such a low image quality occur only seldom on the camera path.

5.13.4 Image Quality Results for Scene PPBoeing5

Figure 5.33 shows the charts for Scene PPBoeing5. The image quality is very high, with exceptions between 40 s and 50 s, 69 s, and 77.8 s. In the interval between 40 s and 50 s, the camera is near

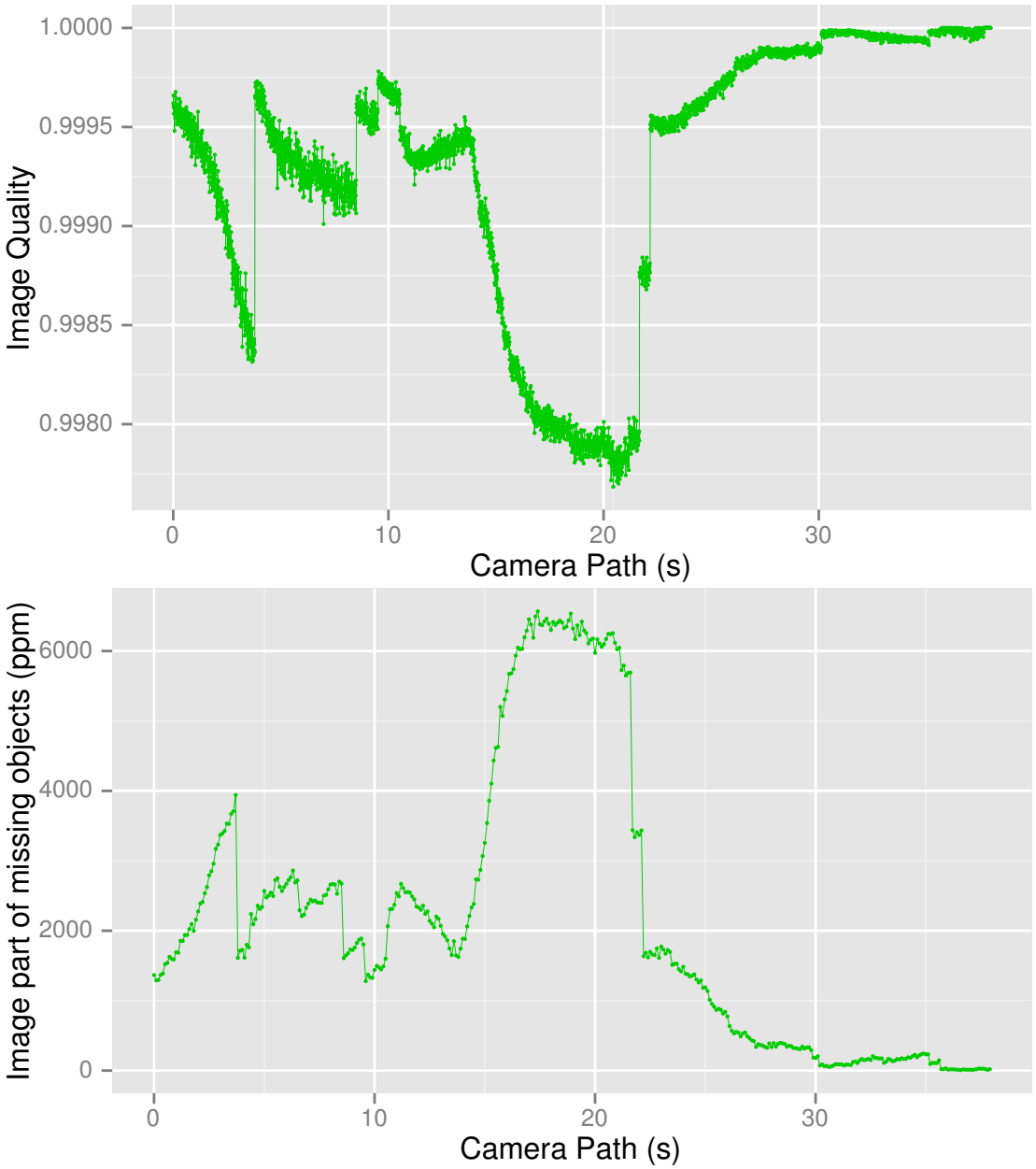


Figure 5.29: Image quality and quality of visibility information measured over the camera path in Scene PP256.

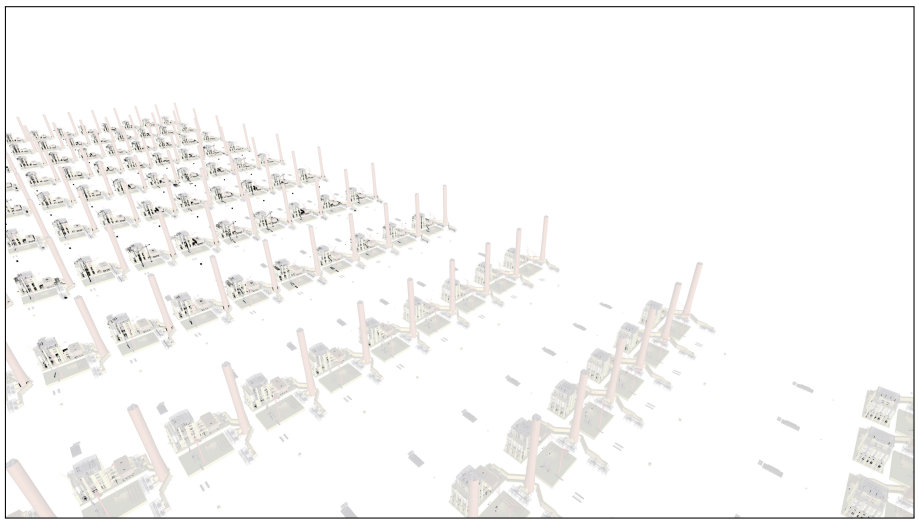


Figure 5.30: Image showing the objects in black that were missed by SVS. It has been produced from two screen shots that were taken at the position with worst image quality (position 20.45 s) on the camera path in Scene PP256. The rendered image has been brightened to increase the perceptibility of the black areas.

the building of the Power Plant model and faces it. If an object inside the building is missing, the image quality will drop, because the object is near the camera and therefore large in visible size. Beginning at 60 s, the camera is approaching the chimney that is circled by the Boeing 777 models. Sampling errors that are contained in their visibility spheres have a higher influence when the camera is near them. At position 77.8 s, the camera's distance to a Boeing 777 is small and therefore the image quality drops.

The worst image quality on the camera path in Scene PPBoeing5 with a SSIM value of 0.974627 has been measured at position 77.8 s. The view at this position is depicted in Figure 5.34. At this position, one of the Boeing 777 models is close the camera. A second model is visible, but it is located further away. Furthermore, parts of the Power Plant model can be seen. For example, the chimney occupies a large part of the image. The missing objects that are shown in black are located inside the hull or at the hull of the Boeing 777 models. Overall, many small details might be missing, but the overall model can be recognized. To sum up, notwithstanding the missing small objects of the highly detailed Boeing 777 models, the most important objects are rendered by SVS even in the worst case on the camera path.

5.13.5 Conclusion

By looking at the image quality of the different test scenes, it can be concluded that SVS's rendering provides a decent image quality. Even in the worst cases on the camera paths, SVS does not miss important objects. Sometimes, objects that are small on the screen are missing. They represent details of the models, or are in areas that are far away from the camera. The scene can always be recognized and no important structural parts are missing.

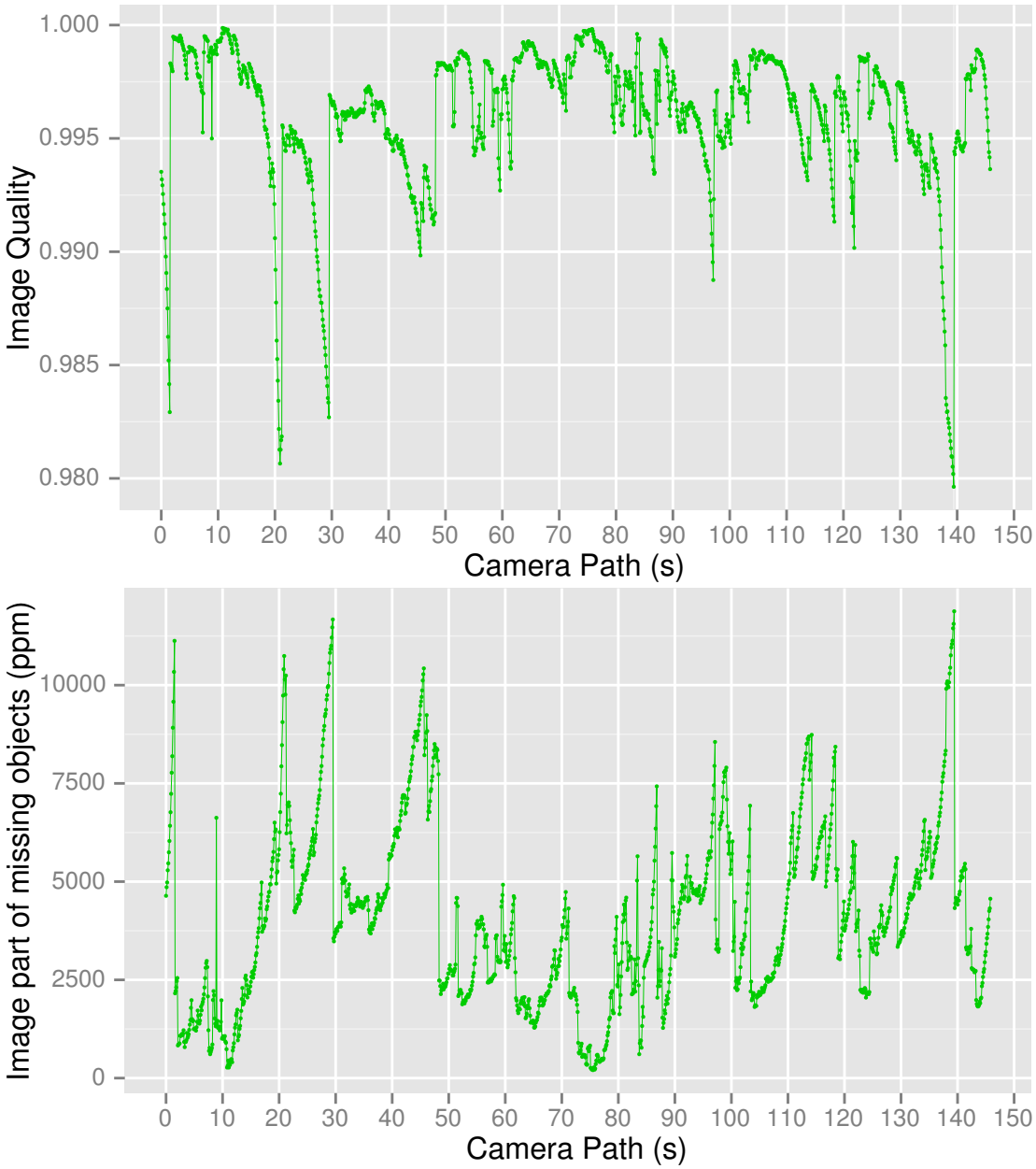


Figure 5.31: Image quality and quality of visibility information measured over the camera path in Scene Pompeii.

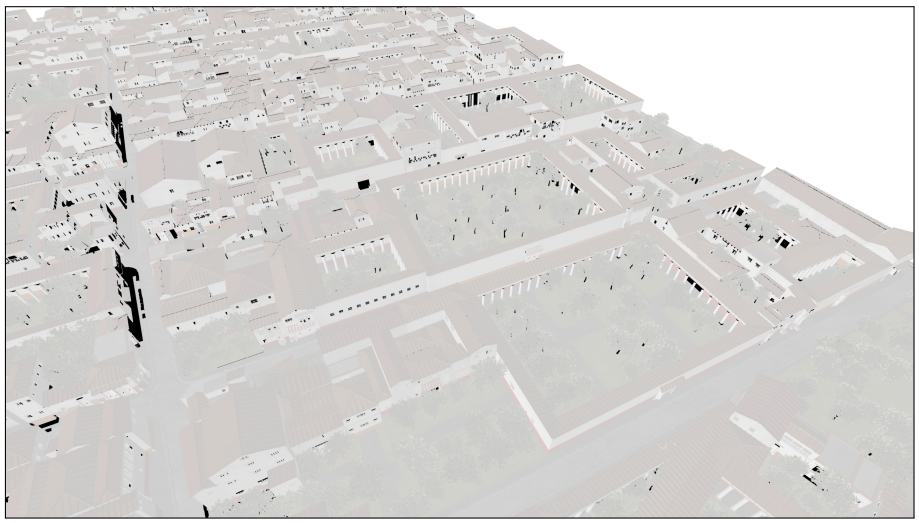


Figure 5.32: Image showing the objects in black that were missed by SVS. It has been produced from two screen shots that were taken at the position with worst image quality (position 139.4 s) on the camera path in Scene Pompeii. The rendered image has been brightened to increase the perceptibility of the black areas.

5.14 3D Rendering on Workstations

In this section, SVS is used for the 3D rendering in a walkthrough situation on a workstation. It is examined if SVS is capable of rendering the test scenes at interactive frame rates. Furthermore, measurements with other renderers are also executed to compare their performance to the one of SVS. For the comparison, the rendering times and the number of triangles that they send to the graphics pipeline are measured. This comparison makes the identification of scene regions possible, in which SVS is most beneficial concerning the rendering performance. The measurements are executed with a resolution of 1280×720 pixels.

The renderers used in this section are Spherical Visibility Sampling (*SVS*), Coherent Hierarchical Culling Revisited (*CHC++*), Adaptive Global Visibility Sampling (*AGVS*), and brute-force rendering with frustum culling (*FC*). The charts showing the number of rendered triangles additionally contain the number of triangles of the objects in the exact visible set (*EVS*). In the charts, the measured values for SVS are shown in green, for CHC++ in blue, for AGVS in purple, for FC in brown, and for EVS in gray, as also shown in the legends of the charts.

For the time duration measurements in this section, the traversal of the camera path is repeated ten times. The charts presented here show a dot for every measuring point. This dot represents the median of the values measured at the corresponding position on the camera path. A thin vertical line through every dot ranges from the lower quartile (0.25 quantile) of the measured values to the upper quartile (0.75 quantile). This line is not visible for most of the dots, because there is no large fluctuation in the measurements. The dots are connected by a thin line to improve the readability of the chart.

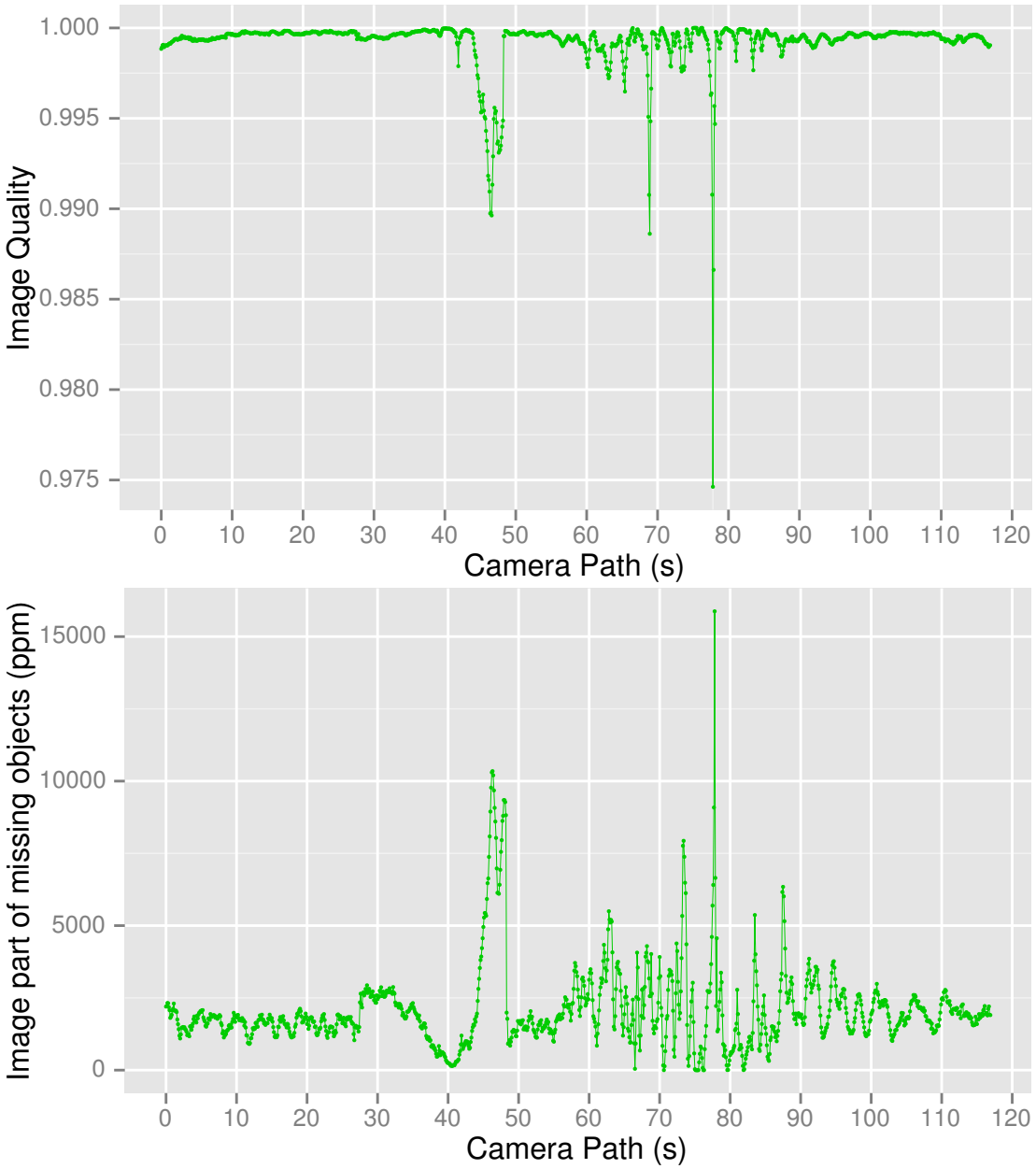


Figure 5.33: Image quality and quality of visibility information measured over the camera path in Scene PPBoeing5.

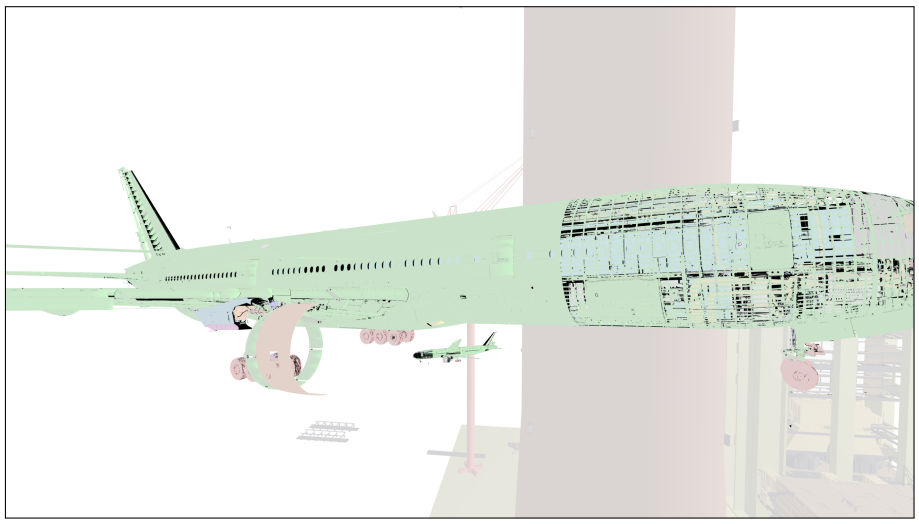


Figure 5.34: Image showing the objects in black that were missed by SVS. It has been produced from two screen shots that were taken at the position with worst image quality (position 77.8 s) on the camera path in Scene PPBoeing5. The rendered image has been brightened to increase the perceptibility of the black areas.

5.14.1 Performance Results for Scene PP4

The measurement results for Scene PP4 are depicted in the charts in Figure 5.35. The upper chart shows the running time of the different rendering algorithms, the lower chart shows the number of triangles that are sent to the graphics pipeline by the different rendering algorithms. FC renders all triangles of objects that intersect the view frustum. For that reason, it provides an upper bound on the number of rendered triangles. The steps in FC's graph are indicative of the number of Power Plant models that are in the view frustum for the respective positions. The EVS contains all objects that are at least partly visible, and therefore provides a lower bound for the number of rendered triangles for a conservative culling algorithm. Of course, this is true only when a binary decision to display or not to display an object is to be taken. If the decision was to be taken for a single triangle, the numbers would look differently.

When looking at the performance, as expected, FC needs the longest time for rendering. Following, the performance of AGVS is considered and compared to SVS's performance. Then, the same is done for CHC++.

The number of triangles that is sent to the graphics pipeline by AGVS is not much lower than the one of FC for large parts of the camera path. When compared to the values of the EVS, it becomes obvious that not much geometry is culled by using the data from the PVSs of the view cells. This means that the overestimation in the view cell's region is quite high. On the one hand, this leads to little underestimation and good image quality (see Section 5.13.1). On the other hand, the rendering performance is not very high. In some parts of the camera paths, AGVS is faster than CHC++ (e.g., in the beginning, and between 35 s and 40 s). In most other parts, it is slower than the CHC++. On this camera path, AGVS is never faster than SVS. In 75 % of the path, SVS is at least three times faster than AGVS.

There are two intervals where CHC++ is faster than SVS. Between 34 s and 35 s, the camera is inside of one of the Power Plant buildings. Much geometry is near the camera and CHC++ is

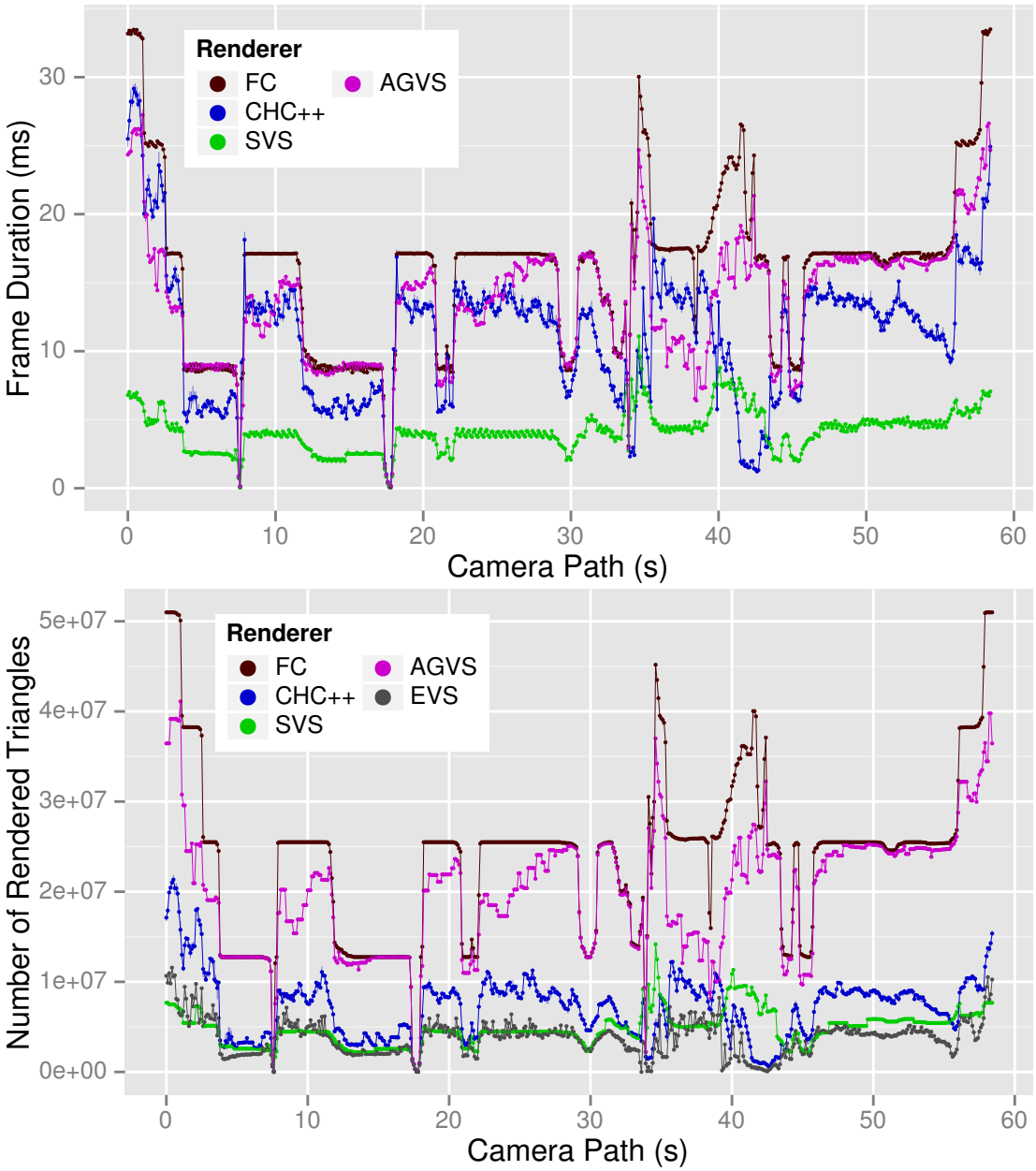


Figure 5.35: Frame duration (upper chart) and number of rendered triangles (lower chart) over the camera path in Scene PP4.

able to cull much of the hidden geometry with only few tests. In the interval between 41 s and 43 s, the camera looks towards a part of the Power Plant that hides nearly the whole scene. Here, again, CHC++ is able to issue only few tests to cull nearly the whole scene. These regions, with geometry near to the camera, are better suited for CHC++ than for SVS. The other part of the camera path, where the buildings are seen from the outside, SVS is faster than CHC++. In these parts, CHC++ renders more triangles and has the additional costs of executing the occlusion queries. SVS makes use of the precomputed visibility for the buildings and has nearly no runtime overhead. In more than 50 % of the camera path, SVS is 2.4 to 3.3 faster than CHC++. Due to of the powerful graphics hardware, an interactive walkthrough is possible with all rendering algorithms, even without occlusion culling.

5.14.2 Performance Results for Scene PP256

The charts in Figure 5.36 summarize the measured values for Scene PP256. Please note the logarithmic scaling of the vertical axis of both charts. In this scene, except at the end of the path, there are no occluders that are near to the camera. Therefore, CHC++ has to perform many tests (e.g., multiple tests for each of the 256 Power Plant buildings at the beginning of the path). The lower chart shows that CHC++ is able to cull many objects, but, when looking at the upper chart, it becomes obvious that the costs of the tests lead to a performance that is only slightly better than FC's performance.

Especially the beginning of the path is a prime example for SVS: Multiple complex, nested objects have to be rendered while standing far away. SVS can use its preprocessed visibility data and send only the potentially visible objects to the graphics pipeline very fast. When looking at the first 2 s of the camera path, the median rendering times are 2073.0 ms for FC, 1280.0 ms for CHC++, and 198.8 ms for SVS. SVS is an order of magnitude faster than FC, and over six times faster than CHC++. With approximately 5 frames per seconds, SVS is not fully interactive, but much better than the other two renderers. To reduce the frame duration at the cost of decreased image quality, SVS's budget rendering is applied to the Scene PP256 and the results are presented in Section 5.15.1.

The order of the renderers with respect to the frame duration stays the same over the whole camera path, but the advantages of SVS decrease to the end of the path. Beginning from 36 s, where only a single Power Plant building is located in the view frustum, FC's running time median is 8.631 ms, CHC++'s is 4.858 ms, and SVS's is 2.373 ms.

5.14.3 Performance Results for Scene Pompeii

For the Scene Pompeii, the measured values are shown in the charts in Figure 5.37. The frame duration chart shows that occlusion culling is reasonable even when flying over the city, because both occlusion culling algorithms have much lower frame durations compared to rendering with frustum culling only. One exception is the interval between 77 s and 82 s, were the CHC++ renderer requires more time than frustum culling. In this interval, only a little part of the scene is in the frustum and the tests executed by the CHC++ renderer at runtime take more time than the time saved by sending less geometry to the graphics card. When comparing SVS to CHC++, it can be seen that SVS is always faster than the CHC++, except for a very small part of the path at 101 s. In 75 % of the camera positions on the path, CHC++ needs at least 1.39 times the running time of SVS; in 50 % it needs at least 1.63 the time, and in 25 % it even needs at least 1.95 the running time of SVS. This is especially interesting when looking at the number of rendered

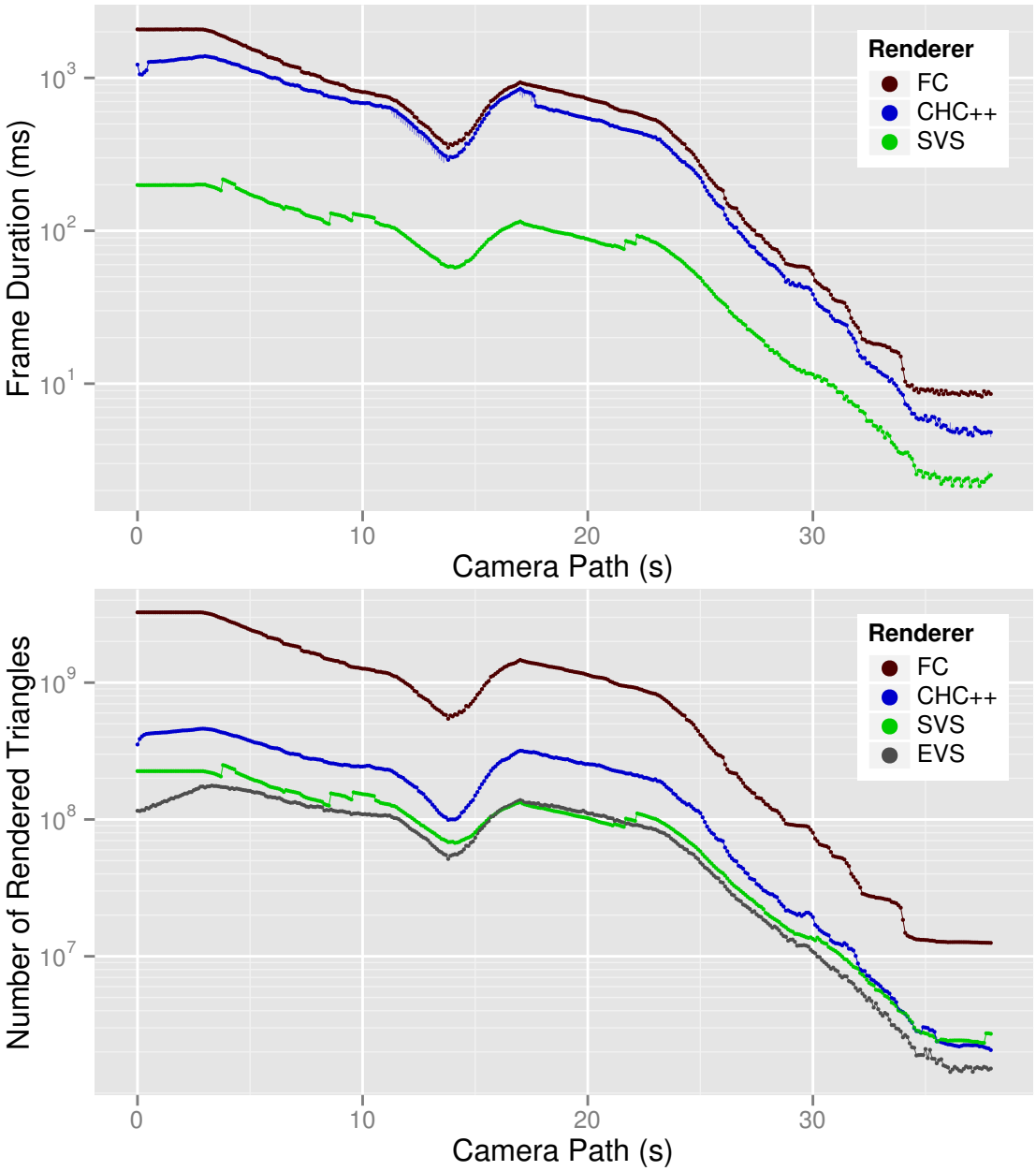


Figure 5.36: Frame duration (upper chart) and number of rendered triangles (lower chart) over the camera path in Scene PP256. Both vertical axes have logarithmic scaling.

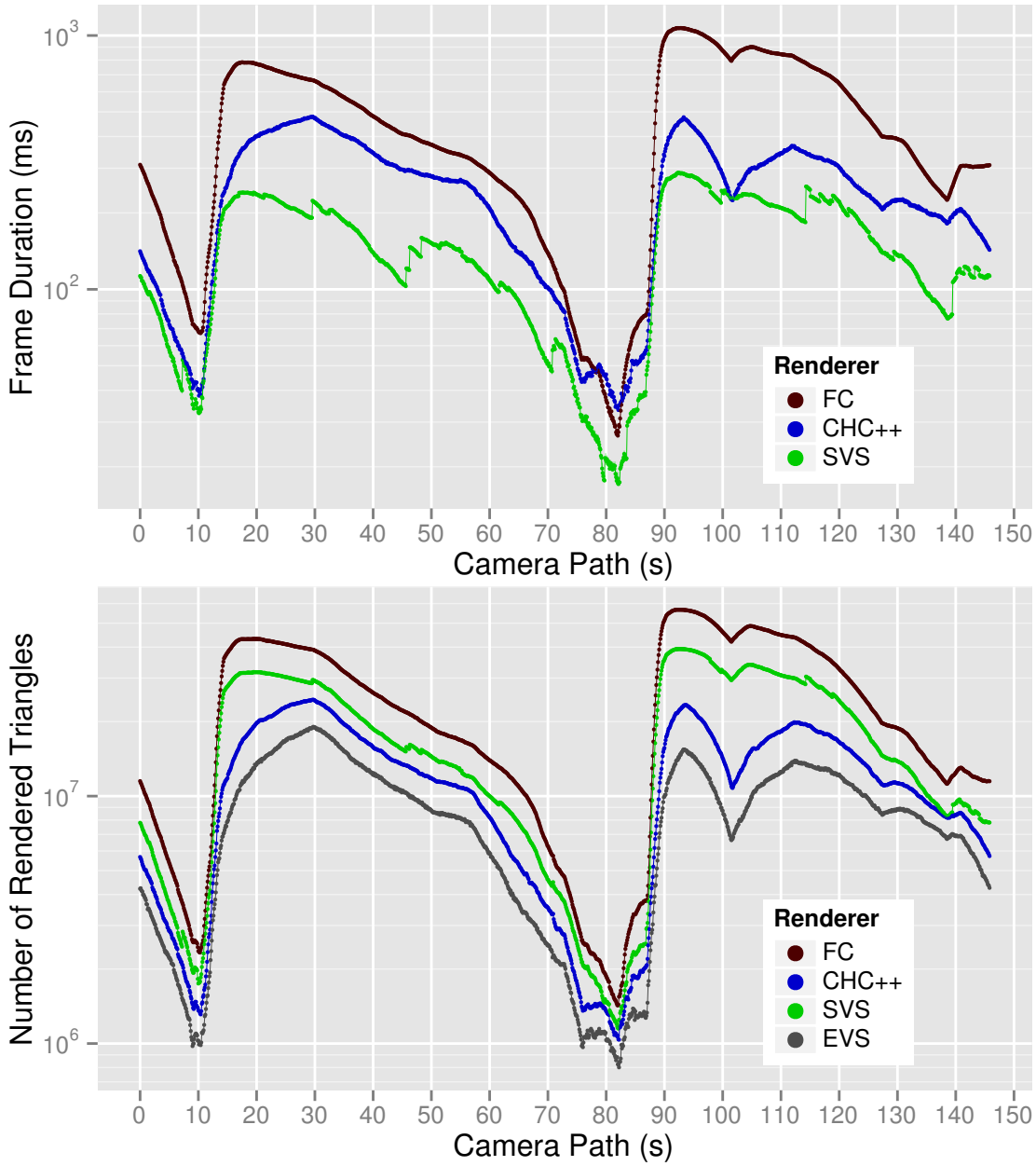


Figure 5.37: Frame duration (upper chart) and number of rendered triangles (lower chart) over the camera path in Scene Pompeii. Both vertical axes have logarithmic scaling.

triangles, because in this scene, SVS sends always more triangles to the graphics pipeline than CHC++. For SVS, the smaller runtime overhead due to the nonexistent need to perform occlusion queries compensates the larger overestimation of the exact visible set. Unfortunately, even when using SVS, the frame rate sometimes drops below 5 frames per seconds, which cannot be declared as interactive anymore. The large amount of visible objects and the extensive usage of textures brings the graphics hardware to its limits. To circumvent this problem, approximate rendering could be applied. One possibility is the usage of SVS's budget rendering feature, which is tested with the Scene Pompeii in Section 5.15.2.

5.14.4 Performance Results for Scene PPBoeing5

In contrast to the previous scenes, for the Scene PPBoeing5 the camera path has been traversed 30 times. This was done, because the fluctuation of the measured values was larger. Probably, this is because of the size of the scene's geometry that is much larger (> 8 GiB, see Table 5.4) than the graphics memory (the graphics card has 2048 MiB memory). Therefore, during a frame, the graphics library might decide to transfer additional data from the main memory to the graphics memory, which leads to a prolonged frame duration. When repeating the measurement, this may happen unpredictable for a frame, and the measured frame duration will sometimes be longer and sometimes be shorter.

The results of the measurements for Scene PPBoeing5 are depicted in the charts in Figure 5.38. The first part of the path is again very well suited for SVS: The Power Plant model and the five Boeing 777 models are visible from outside and are not near to the camera. In the first 30 s of the path, the median running time of SVS is 169.7 ms, of CHC++ it is 1814.0 ms, and of FC it is 3027.0 ms. The median of triangles displayed per frame by SVS is 159.3 million and by CHC++ 102.8 million. FC displays the whole scene with its 1.698 billion triangles. Here again, even when SVS displays more triangles than CHC++, the frame duration is much lower. SVS is over ten times faster then CHC++ in the first part of the path.

The situation changes completely when the camera moves behind the Power Plant building, where CHC++ becomes much faster, e.g., at 40 s. There is still much of the geometry in the frustum, but the models hide each other. CHC++ displays the building in the foreground and is able to detect quickly that the Boeing 777 models are hidden behind the building. SVS does not perform occlusion queries during runtime and has no visibility sphere above the planes or the building in the scene graph, because of the special scene structure due to the animations. Therefore, it has no possibility to detect the occlusion of the planes and displays them by using their visibility spheres.

Between 60 s and 90 s, the camera is near the Boeing 777 models. SVS and CHC++ render approximately the same amount of triangles. In a few cases, CHC++ is much faster than SVS, the rest of the time, SVS is faster. The end of the path looks like the beginning did. Looking at all traversals of the camera path, the frame rate achieved by SVS's renderer is higher than 5 frames per second in more than 98 % of the measurements.

5.14.5 Conclusion

The results have shown that SVS is able to efficiently perform occlusion culling in highly complex scenes. By using its preprocessed visibility information, it is able to significantly speed up the rendering compared to the CHC++ online occlusion culling algorithm. This works even when

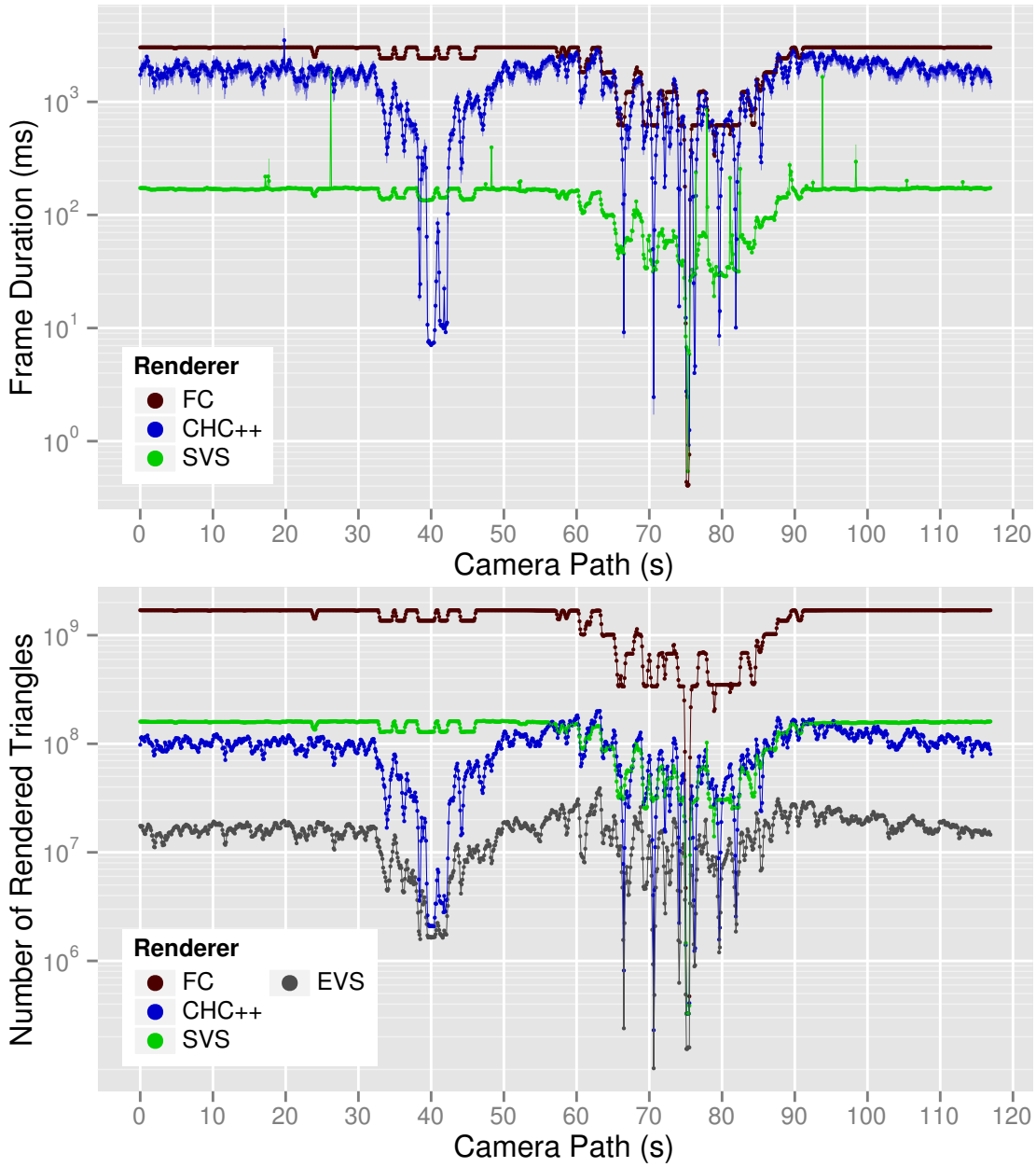


Figure 5.38: Frame duration (upper chart) and number of rendered triangles (lower chart) over the camera path in Scene PPBoeing5. Both vertical axes have logarithmic scaling.

SVS sends more triangles than CHC++ to the rendering pipeline. Due to the large overestimation of AGVS, SVS is also much faster than AGVS.

Regions that are most suitable for SVS were identified. As SVS exploits the occlusion of nested objects, it works best with those kinds of objects (e.g., the Power Plant model and the Boeing 777 model). SVS also works for scenes with high depth complexity, but with little or no nesting at all, like Scene Pompeii, but the performance gain is lower. In such a scene, the view onto the scene is important: For a flight over the scene, were many objects are visible, it works quite well. Inside the streets, with large occluders near the camera, the CHC++ is faster. A similar situation was observed in Scene PP4. Consequent, SVS works better if the viewer is viewing the objects from far away than if the viewer is standing inside or near the objects.

5.15 Budget Rendering

In the following, the budget rendering feature of SVS is evaluated. The evaluation will show that this feature can be used to trade rendering time off image quality. By limiting the number of triangles that the rendering algorithm is allowed to send to the graphics pipeline, the running time is indirectly limited at the cost of missing objects that lead to errors in the rendered image. In the following, the budget rendering feature is evaluated by using it in the Scene PP256 and in the Scene Pompeii, because interactive frame rates were not achieved over the whole camera path (see results in Section 5.14.2 and Section 5.14.3).

5.15.1 Performance and Quality Results for Scene PP256

The charts in Figure 5.39 show the running times, the number of rendered triangles, and the image quality for the Scene PP256. The measured data for the SVS renderer without budget is the same as in Figure 5.36. For the budget renderer, a budget of 100 million triangles and a budget of 10 million triangles was used. When looking at the number of triangles that were sent to the graphics pipeline (middle chart), it can be observed that SVS always adheres to the given triangle budget. A look at the chart showing the frame duration (top chart) reveals, as expected, that the reduced number of triangles results in a reduced frame duration. In the intervals of the camera path, in which a reduced number of triangles is rendered by the budget renderer, the frame duration is much lower. For the setting of 100 million triangles, the reduction in triangles is roughly the same as the reduction in frame duration. For the lower setting of 10 million triangles, the reduction in frame duration is not as large as the reduction in triangles. This is due to the fact that for such a low number of triangles, the bottleneck is no longer the triangle throughput of the graphics hardware, but probably the instructions executed by the CPU to prepare the objects to be sent to the graphics pipeline.

The image quality (bottom chart) is not changed by the budget rendering feature in the parts of the camera path without reduction of the number of rendered triangles (e.g., between 65 s and 85 s). In the other parts, where the number of rendered triangles is reduced, the image quality decreases. The minimum value 0.962995 was measured at 17.1 s. At this point, the missing objects can be seen in the rendered image. Nevertheless, the image quality is still acceptable, when keeping in mind that the number of rendered triangles is reduced to less than a tenth of SVS's standard rendering.

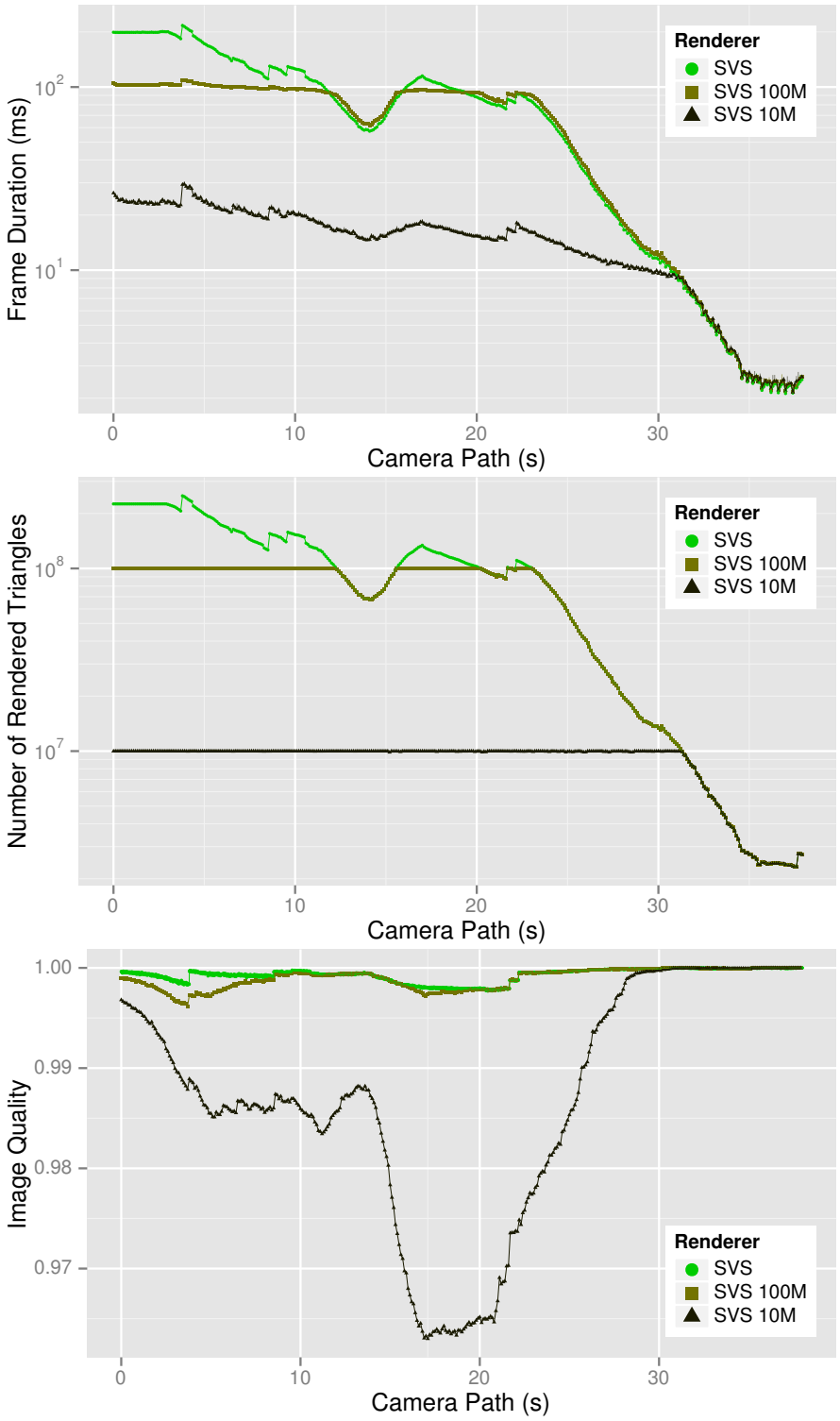


Figure 5.39: Frame duration, number of rendered triangles, and image quality over the camera path in Scene PP256. SVS renderer without budget rendering, with a budget of 100 million triangles, and a budget of 10 million triangles. Vertical axes in the first and second chart have logarithmic scaling.

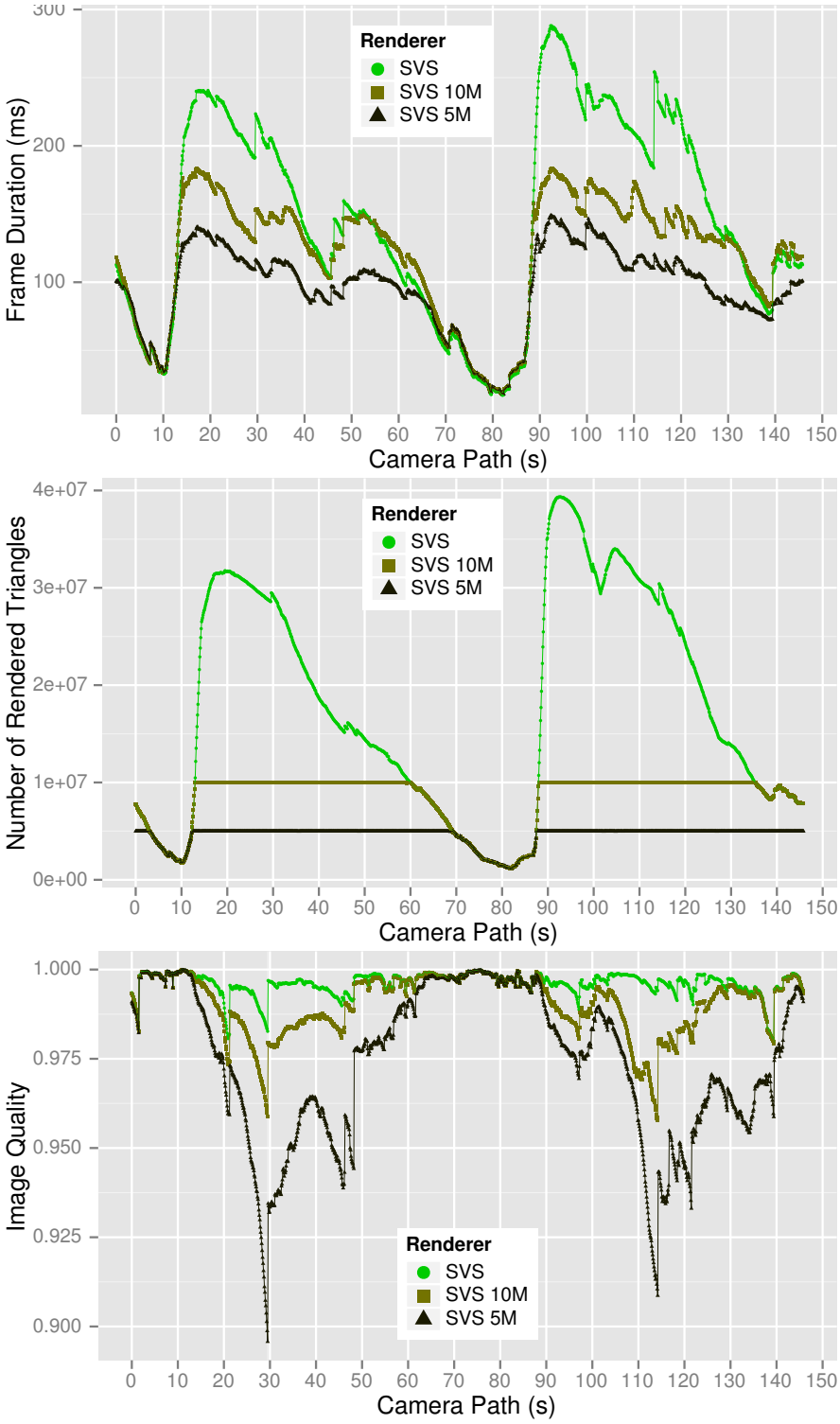


Figure 5.40: Frame duration, number of rendered triangles, and image quality over the camera path in Scene Pompeii. SVS renderer without budget rendering, with a budget of 10 million triangles, and a budget of 5 million triangles.

5.15.2 Performance and Quality Results for Scene Pompeii

The results for Scene Pompeii are depicted in the three charts in Figure 5.40. The data for the SVS standard renderer is the same as in Figure 5.37. Here again, it can be observed that the number of rendered triangles is successfully limited to the budgets of 10 million triangles and 5 million triangles, respectively. On the one hand, as a positive effect, the frame duration is also decreased. On the other hand, the image quality is decreased as a negative effect. On the camera path at 29.5 s, the minimum image quality value 0.895748 was observed, which is not very good anymore. Here, only a reduction with a budget of 10 million triangles should be used.

5.15.3 Conclusion

SVS's budget rendering feature works as expected: It is able to limit the number of rendered triangles successfully and, by this means, to decrease the time required for the rendering. Because potentially visible objects are left out in this process, a noticeable decrease of image quality is observed. The triangle budget can be interactively changed by the user and the effect is observable immediately. Therefore, the feature can be used easily to decide between high frame rates and good image quality at runtime. The user does not have to make a binary decision, but can set the triangle budget at will.

5.16 3D Rendering on Mobile Devices

For graphics programming on mobile devices, the standard software library is OpenGL ES ("Open Graphics Library for Embedded Systems"). OpenGL ES 2.0 [Khr07] was released in 2007 and many devices that run the Android operating system support it nowadays. As a problem in the scope of 3D rendering with occlusion culling, OpenGL ES 2.0 lacks support for hardware-accelerated occlusion queries. Due to this missing feature, many modern online occlusion culling algorithms – like the CHC++ – cannot be used on mobile devices that support only OpenGL ES 2.0. These occlusion queries were introduced with OpenGL ES 3.0 [Khr12] in 2012. Android 4.3, released in July 2013[7], provides support for OpenGL ES 3.0. At the time of this writing, there are only very few devices that support OpenGL ES 3.0.

Because currently online occlusion culling algorithms cannot be used on most mobile devices, rendering of complex scenes with high depth complexity is difficult on those devices. SVS provides a way out here, because it does not require hardware-accelerated occlusion queries at runtime. The preprocessing can be executed on a workstation and the generated data, which is small in size (see Section 5.9), can be copied to a mobile device. The following evaluation was performed to demonstrate that SVS provides occlusion culling on mobile devices supporting OpenGL ES 2.0.

5.16.1 Mobile Devices and Software Implementation

For the measurements in this section, two mobile devices were used. The first one is an ASUS Eee Pad Transformer Prime TF201 (NVIDIA Tegra 3 with 4 × 1.3 GHz, 1 GB RAM, display resolution 1280 × 800 pixels, Android 4.1.1). This device is called *tablet* in the following. The second mobile device, called *smartphone* from now on, is a Google Galaxy Nexus (GT-I9250, Texas

[7] http://android-developers.blogspot.de/2013/07/android-43-and-updated-developer-tools.html

Instruments OMAP 4460 with 2 × 1.2 GHz, 1 GB RAM, display resolution 1280 × 720 pixels, Android 4.3).

As part of this work, some of the main C++ libraries of PADrend have been ported to the Android operating system. The main work was adding an abstraction layer for the graphics library and adding support for OpenGL ES 2.0. A small Android project was created that provides the graphical user interface and wraps the calls to the C++ libraries.

When using the Android devices, the memory that can be used by an application is much lower than the overall 1 GB that are provided by the hardware. Much of the memory is reserved for the Linux kernel or used by the operating system. About at most half of the memory is free to be used by an application. Furthermore, the devices do not have dedicated graphics memory, which is available on graphics cards for PCs. Due to the restricted memory on the mobile devices, only the smallest test scene – Scene PP – can be used.

When loading the Scene PP on the smartphone with the initial implementation, out-of-memory errors occurred. To circumvent this problem, vertex buffer objects (VBOs) were deactivated on the smartphone. Since there is no dedicated graphics memory on the mobile devices, one might think that this does not lead to a big performance decrease. But, measurements on the tablet comparing activated and deactivated VBOs showed that the performance decreases by approximately a factor of ten. Therefore, the measurement results for the smartphone can be used to compare the different algorithms relative to each other, but the absolute running times are much too high. A side effect that was observed when deactivating the VBOs was the reduction of the fluctuation of the measured frame times.

5.16.2 Rendering Performance with and without Budget Rendering

Because the measured times fluctuate by a considerable amount on the mobile devices, the measurements have been repeated for every setup by traversing the camera path 50 times on the tablet and 30 times on the smartphone. The charts in Figure 5.41 show the running time for both devices and the amount of geometry that was sent to the graphics pipeline. Since the chart showing the number of rendered triangles looks the same for the tablet and the smartphone, only a single chart is shown here.

In the first part of the camera path between 0 s and 55 s, the building of the Power Plant model is inside the frustum. By using SVS instead of only frustum culling, occlusion culling is additionally applied and the amount of triangles that is sent to the graphics pipeline drops from over 12 million to around 3 million. For that reason, the frame duration drops alike from around 600 ms to below 200 ms on the tablet. On the smartphone, they drop from 7000–8000 ms when using frustum culling to 1400–2000 ms when using SVS. This shows that SVS can successfully be applied on a mobile device to enable occlusion culling rendering and to greatly improve the performance.

The budget rendering feature can be used to further increase the rendering performance at the cost of the image quality, as already examined before. The budget was set to 1.5 million triangles, 1 million triangles, and 500,000 triangles. The chart showing the number of rendered triangles shows that the given triangle budget is always kept. There is no position were more than the given number of triangles is rendered. Furthermore, when there is enough geometry inside the frustum, the number of rendered triangles is more or less the same as the given budget. In the camera path segments where the number of rendered triangles significantly drops by using the budget rendering, the frame duration is also decreased. In the segment with only little geometry inside the frustum, between 55 s and 75 s, the additional overhead introduced by the budget rendering increases the frame duration compared to SVS without budget rendering. In

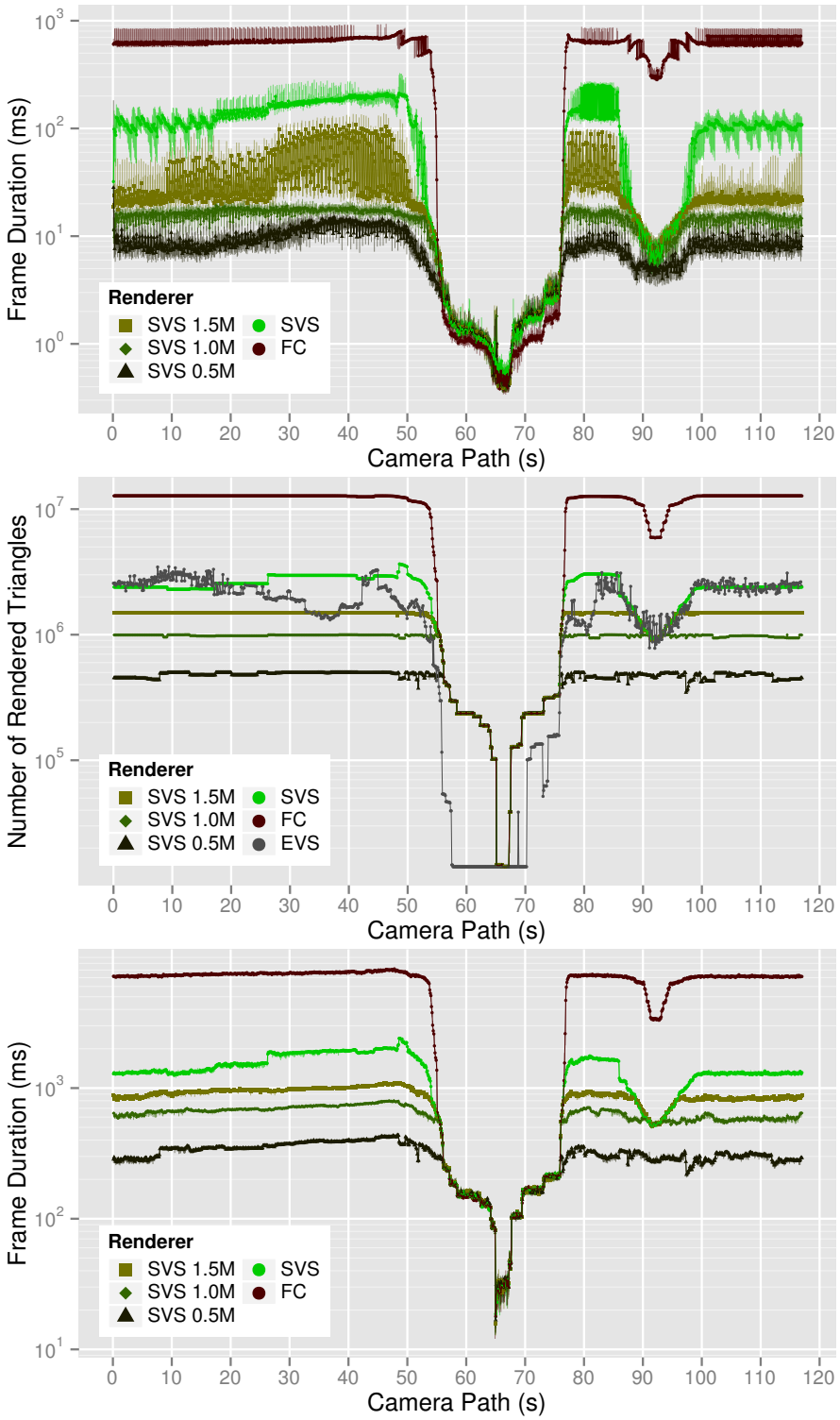


Figure 5.41: Frame duration measured on the tablet (top), number of rendered triangles (middle), and frame duration measured on the smartphone (bottom) over the camera path in Scene PP.

this segment, rendering using the SVS is a little bit slower than using frustum culling only. This is due to the additional overhead of accessing spheres and interpolation visibility information when there is only little geometry that is finally rendered. But the performance decrease is only small, and it is negligible compared to the performance gain in the path segments with much geometry inside the frustum.

5.16.3 Conclusion

In summary, SVS can be used on mobile devices to perform occlusion culling. When there is much occlusion for the current view, the performance gains are very high, e.g., more than a factor of five at the beginning and at the end of the camera path. The budget rendering feature can be used by the user to trade rendering performance off missing objects in the rendered image.

6 Conclusion

Spherical Visibility Sampling uses direction-dependent visibility to exploit the occlusion that is inherent in 3D scenes containing nested objects. SVS does not require view cells and, therefore, provides a very efficient way to use preprocessed visibility (see Section 5.10).

The rasterization used in SVS's preprocessing has the effect of filtering out unimportant objects. These unimportant objects are missing in the rendered image, but many objects can be culled, and therefore the rendering performance is much higher compared to using geometric visibility tests based on ray casting (see Section 5.14).

SVS works well for spacious scenes with many visible objects, which are particularly challenging for existing region-based preprocessed visibility techniques as well as online occlusion culling algorithms. The existing preprocessed visibility techniques have to decide on the granularity for their view cells to trade quality of the visibility information off memory consumption. The online occlusion culling algorithms suffer from a runtime overhead for executing many occlusion queries for the visible objects.

By using the direction-dependent visibility information, SVS allows real-time rendering of highly complex 3D scenes that contain billions of triangles and millions of objects. If the number of visible objects is still too high after applying occlusion culling, SVS's budget rendering feature can be used to leave out unimportant objects at the cost of decreased image quality (see Section 5.15).

SVS's lightweight runtime requirements enable the application of occlusion culling even on mobile devices (Section 5.16). A ready-to-use implementation of SVS is available in PADrend, and can easily be applied in practice.

The following sections describe SVS's flexibility to tune the accuracy of its visibility information (Section 6.1) and give an outlook on future work (Section 6.2).

6.1 Accuracy of the Visibility Information

The evaluation in Chapter 5 showed that there are several parameters that can be used to tune the accuracy of the visibility information generated by SVS. Parameter values have been presented that are reasonable in practice. Nonetheless, SVS is flexible enough to allow other parameter values to increase its accuracy for other use cases.

The visibility tests during the preprocessing (see Section 3.5) are based on the rasterization of pixels and thus small objects might be missed although they are geometrically visible. By using a higher image resolution, the accuracy can be increased at the expense of a longer preprocessing time.

In some cases, the union of three sample points used by the MAX3 interpolation (see Section 4.2) creates a potentially visible set that strongly overestimates the exact visible set. This happens, for instance, when the situation between the sample points completely changes from few visible objects (e.g., viewer inside the streets of a city) to many visible objects (e.g., viewer above a city). In other cases, the potentially visible set is underestimated, because visible objects are missed

during preprocessing (e.g., when the viewer cannot look through a window while standing at each sample point, but can look through it in between; see Section 3.4). The underestimation and overestimation are large, when there are large discontinuities in visibility between the three sample points that are used for the interpolation. A higher number of samples on the sphere surface leads to a higher accuracy, but again to higher storage space and running time (see the evaluation in Section 5.3). By using a distribution with more sample points, an increased accuracy is easily achievable.

Finally, the chosen rendering budget (see Section 4.3), which limits the size of the rendered visible set, directly influences the rendering time and image quality at runtime (Section 5.15).

6.2 Outlook on Future Work

An interesting approach is the combination of SVS with online occlusion culling to improve SVS's running time in situations, where occlusion arises from the interaction of objects inside different visibility spheres: occlusion queries for testing the visibility of these spheres themselves during runtime could be beneficial. Although, this might improve the rendering time in densely occluded scenes, it would not work on mobile devices without support for occlusion queries. Furthermore, it is difficult to keep the additional runtime overhead very small, when adding such a feature that makes use of occlusion queries.

In order to improve the image quality when using the budget rendering feature, level-of-detail techniques with multiple resolutions for single objects can be easily integrated into SVS as a reasonable extension. Instead of completely leaving out less important objects, they could be replaced by lower resolution versions chosen accordingly to their estimated influence on the final image.

Although the memory overhead introduced by SVS is already reasonably low in comparison to other preprocessed visibility approaches (see Section 5.12.3), one could exploit the similarity of the visible sets of neighboring sample points for compression. Existing compression techniques for visibility information (e.g., [vS99]) might be expandable to support SVS's visibility vectors.

Another interesting research direction is the combination of SVS with out-of-core techniques that store parts of the scene in external memory. SVS's visibility information could be used for guiding the loading mechanisms of the out-of-core technique. For example, the objects' estimated importance values could be used to prefer the most important objects for loading. Especially for the rendering on mobile devices, where one major challenge is the memory management, this could enable the loading of more complex scenes.

Bibliography

[AHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. 3rd ed. Wellesley, MA, USA: A K Peters, Ltd., 2008.

[AM00] Ulf Assarsson and Tomas Möller. "Optimized View Frustum Culling Algorithms for Bounding Boxes". In: *Journal of Graphics Tools* 5.1 (2000), pp. 9–22. DOI: 10.1080/10867651.2000.10487517.

[Bau25] Walter Bauersfeld. "Verfahren zur Herstellung von Kuppeln und ähnlichen gekrümmten Flächen aus Eisenbeton". Reichspatentamt Patentschrift Nr. 415395, Klasse 37a, Gruppe 2. 1925-06-19.

[Bel54] Richard Bellman. "Some Applications of the Theory of Dynamic Programming—A Review". In: *Journal of the Operations Research Society of America* 2.3 (1954-08), pp. 275–288. DOI: 10.1287/opre.2.3.275. JSTOR: 166640.

[Ben75] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975-09), pp. 509–517. DOI: 10.1145/361002.361007.

[Ber+86] Larry Bergman, Henry Fuchs, Eric Grant, and Susan Spach. "Image rendering by adaptive refinement". In: *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 29–37. DOI: 10.1145/15922.15889.

[BFA06] Azzedine Boukerche, Jing Feng, and Regina Borges de Araujo. "A 3D image-based rendering technique for mobile handheld devices". In: *International Symposium on a World of Wireless, Mobile and Multimedia Networks 2006*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 325–331. DOI: 10.1109/WOWMOM.2006.5.

[BG04] Paul Bao and Douglas Gourlay. "Remote walkthrough over mobile networks using 3-D image warping and streaming". In: *IEE Proceedings - Vision, Image and Signal Processing* 151.4 (2004-08). Proceedings of the IEE Conference on Visual Information Engineering (VIE 2003), pp. 329–336. DOI: 10.1049/ip-vis:20040749.

[BG06] Paul Bao and Douglas Gourlay. "A framework for remote rendering of 3-D scenes on limited mobile devices". In: *IEEE Transactions on Multimedia* 8.2 (2006-04), pp. 382–389. DOI: 10.1109/TMM.2005.864337.

[Bit+04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful". In: *Computer Graphics Forum* 23.3 (2004-09). Proceedings of Eurographics 2004, pp. 615–624. DOI: 10.1111/j.1467-8659.2004.00793.x.

[Bit+09] Jiří Bittner, Oliver Mattausch, Peter Wonka, Vlastimil Havran, and Michael Wimmer. "Adaptive global visibility sampling". In: *ACM Transactions on Graphics* 28.3 (2009-07), pp. 1–10. DOI: 10.1145/1531326.1531400.

[BJ96] Steve Bryson and Sandy Johan. “Time management, simultaneity and time-critical computation in interactive unsteady visualization environments”. In: *Proceedings of the conference on Visualization ’96*. VIS ’96. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996, pp. 255–261. DOI: 10.1109/VISUAL.1996.568117.

[Bly08] David Blythe. “Rise of the Graphics Processor”. In: *Proceedings of the IEEE* 96.5 (2008-05), pp. 761–778. DOI: 10.1109/JPROC.2008.917718.

[BRA06] Aner Ben-Artzi, Ravi Ramamoorthi, and Maneesh Agrawala. “Efficient Shadows for Sampled Environment Maps”. In: *Journal of Graphics, GPU, and Game Tools* 11.1 (2006), pp. 13–36. DOI: 10.1080/2151237X.2006.10129211.

[Bur81] Peter J. Burt. “Fast filter transform for image processing”. In: *Computer Graphics and Image Processing* 16.1 (1981-05), pp. 20–51. DOI: 10.1016/0146-664X(81)90092-7.

[Car07] Maria Francesca Carfora. “Interpolation on spherical geodesic grids: A comparative study”. In: *Journal of Computational and Applied Mathematics* 210.1-2 (2007-12). Proceedings of the Numerical Analysis Conference 2005, pp. 99–105. DOI: 10.1016/j.cam.2006.10.068.

[Cat74] Edwin Earl Catmull. “A subdivision algorithm for computer display of curved surfaces”. PhD thesis. Salt Lake City, UT, USA: Department of Computer Science, University of Utah, 1974-12.

[CG02] Chun-Fa Chang and Shyh-Haur Ger. “Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering”. In: *Advances in Multimedia Information Processing — PCM 2002*. Ed. by Yung-Chang Chen, Long-Wen Chang, and Chiou-Ting Hsu. Vol. 2532. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 2002, pp. 1105–1111. DOI: 10.1007/3-540-36228-2_137.

[CG07] Matt Craighead and Daniel Ginsburg. *OpenGL ARB_occlusion_query*. Online. Revision 7. 2007-04. URL: http://www.opengl.org/registry/specs/ARB/occlusion_query.txt (visited on 2009-10-16).

[Cla76] James H. Clark. “Hierarchical geometric models for visible surface algorithms”. In: *Communications of the ACM* 19.10 (1976-10), pp. 547–554. DOI: 10.1145/360349.360354.

[Coh+03] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. “A survey of visibility for walkthrough applications”. In: *IEEE Transactions on Visualization and Computer Graphics* 9.3 (2003), pp. 412–431. DOI: 10.1109/TVCG.2003.1207447.

[Dan57] George B. Dantzig. “Discrete-Variable Extremum Problems”. In: *Operations Research* 5.2 (1957-04), pp. 266–277. DOI: 10.1287/opre.5.2.266. JSTOR: 167356.

[DDP97] Frédo Durand, George Drettakis, and Claude Puech. “The visibility skeleton: a powerful and efficient multi-purpose global visibility tool”. In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. SIGGRAPH ’97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 89–100. DOI: 10.1145/258734.258785.

[Del34] Boris N. Delaunay. “Sur la sphère vide”. In: *Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk* 6 (1934-10), pp. 793–800.

[Dut84] Geoffrey Dutton. “Geodesic Modelling Of Planetary Relief”. In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 21.2 & 3 (1984). Monograph 32–33, pp. 188–207. DOI: 10.3138/R613-191U-7255-082N.

[Eik+13] Benjamin Eikel, Claudius Jähn, Matthias Fischer, and Friedhelm Meyer auf der Heide. “Spherical Visibility Sampling”. In: *Computer Graphics Forum* 32.4 (2013-07). Proceedings of the 24th Eurographics Symposium on Rendering, pp. 49–58. DOI: 10.1111/cgf.12150.

[EJP11] Benjamin Eikel, Claudius Jähn, and Ralf Petring. “PADrend: Platform for Algorithm Development and Rendering”. In: *Augmented & Virtual Reality in der Produktentstehung*. Ed. by Jürgen Gausemeier, Michael Grafe, and Friedhelm Meyer auf der Heide. Vol. 295. HNI-Verlagsschriftenreihe. Heinz Nixdorf Institut, Universität Paderborn, 2011-05, pp. 159–170.

[FS93] Thomas A. Funkhouser and Carlo H. Séquin. “Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments”. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. (Anaheim, CA). SIGGRAPH ’93. New York, NY, USA: ACM, 1993, pp. 247–254. DOI: 10.1145/166117.166149.

[Ful54] Richard Buckminster Fuller. “Building construction”. US2682235. 1954-06-29. URL: http://www.freepatentsonline.com/2682235.html.

[Gai+10] Athanasios Gaitatzes, Anthousis Andreadis, Georgios Papaioannou, and Yiorgos Chrysanthou. “Fast Approximate Visibility on the GPU Using Precomputed 4D Visibility Fields”. In: *Proceedings of the 18th International Conference on Computer Graphics, Visualization and Computer Vision*. WSCG 2010. 2010-02, pp. 131–138.

[Gär99] Bernd Gärtner. “Fast and Robust Smallest Enclosing Balls”. In: *Algorithms - ESA ’99*. Ed. by Jaroslav Nešetřil. Vol. 1643. Lecture Notes in Computer Science. Springer Berlin/Heidelberg, 1999, pp. 325–338. DOI: 10.1007/3-540-48481-7_29.

[GBK06] Michael Guthe, Ákos Balázs, and Reinhard Klein. “Near Optimal Hierarchical Culling: Performance Driven Use of Hardware Occlusion Queries”. In: *Proceedings of the 17th Eurographics Symposium on Rendering*. Ed. by Tomas Akenine-Möller and Wolfgang Heidrich. EGSR ’06. Eurographics Association, 2006-06, pp. 207–214. DOI: 10.2312/EGWR/EGSR06/207-214.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. “Hierarchical Z-Buffer Visibility”. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. SIGGRAPH ’93. New York, NY, USA: ACM, 1993, pp. 231–238. DOI: 10.1145/166117.166147.

[GKY08] Enrico Gobbetti, Dave Kasik, and Sung-eui Yoon. “Technical strategies for massive model visualization”. In: *Proceedings of the 2008 ACM symposium on Solid and physical modeling*. New York, NY, USA: ACM, 2008, pp. 405–415. DOI: 10.1145/1364901.1364960.

[Gut84] Antonin Guttman. “R-trees: a dynamic index structure for spatial searching”. In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. SIGMOD ’84. New York, NY, USA: ACM, 1984, pp. 47–57. DOI: 10.1145/602259.602266.

[HG94] Paul Heckbert and Michael Garland. “Multiresolution modeling for fast rendering”. In: *Proceedings of Graphics Interface 1994*. GI ’94. 1994-05, pp. 43–50.

[HN98] Jerry L. Hintze and Ray D. Nelson. “Violin Plots: A Box Plot-Density Trace Synergism”. In: *The American Statistician* 52.2 (1998), pp. 181–184. DOI: 10.1080/00031305.1998.10480559.

[HR12] Paul Hildebrandt and Clark Richert. “Domes, Zomes, and Drop City”. In: *Proceedings of Bridges 2012: Mathematics, Music, Art, Architecture, Culture*. Ed. by Robert Bosch, Douglas McKenna, and Reza Sarhangi. Phoenix, AZ, USA: Tessellations Publishing, 2012, pp. 545–548. URL: http://archive.bridgesmathart.org/2012/bridges2012-545.pdf.

[Hua+02] Wei Hua, Hujun Bao, Qunsheng Peng, and A. R. Forrest. “The global occlusion map: a new occlusion culling approach”. In: *Proceedings of the ACM symposium on Virtual reality software and technology*. (Hong Kong, China). VRST ’02. New York, NY, USA: ACM, 2002, pp. 155–162. DOI: 10.1145/585740.585766.

[Hun78] Gregory Michael Hunter. “Efficient computation and data structures for graphics”. PhD thesis. Princeton, NJ, USA: Department of Electrical Engineering and Computer Science, Princeton University, 1978.

[Jäh+13] Claudius Jähn, Benjamin Eikel, Matthias Fischer, Ralf Petring, and Friedhelm Meyer auf der Heide. “Evaluation of Rendering Algorithms using Position-Dependent Scene Properties”. In: *Advances in Visual Computing*. Ed. by George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Baoxin Li, Fatih Porikli, Victor Zordan, James Klosowski, Sabine Coquillart, Xun Luo, Min Chen, and David Gotz. Vol. 8033. Lecture Notes in Computer Science. Proceedings of the 9th International Symposium on Visual Computing (ISVC 2013). Springer Berlin Heidelberg, 2013, pp. 108–118. DOI: 10.1007/978-3-642-41914-0_12.

[Jia+06] Zhongding Jiang, Yandong Mao, Qi Jia, Nan Jiang, Junyi Tao, Xiaochun Fang, and Hujun Bao. “PanoWalk: A Remote Image-Based Rendering System for Mobile Devices”. In: *Advances in Multimedia Information Processing - PCM 2006* 4261 (2006), pp. 641–649. DOI: 10.1007/11922162_74. URL: http://www.springerlink.com/content/330u3054337pn767.

[Kar72] Richard M. Karp. “Reducibility Among Combinatorial Problems”. In: *Complexity of Computer Computations*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. New York: Plenum Press, 1972, pp. 85–103.

[KBF05] David J. Kasik, William Buxton, and David R. Ferguson. “Ten CAD challenges”. In: *IEEE Computer Graphics and Applications* 25.2 (2005), pp. 81–92. DOI: 10.1109/MCG.2005.48.

[Khr07] Khronos Group. *OpenGL ES version 2.0*. Specification. 2007-03-05. URL: http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf (visited on 2013-08-09).

[Khr12] Khronos Group. *OpenGL ES version 3.0*. Specification. 2012-08-06. URL: http://www.khronos.org/registry/gles/specs/3.0/es_spec_3.0.2.pdf (visited on 2013-08-09).

[KS99] James T. Klosowski and Cláudio T. Silva. "Rendering on a budget: a framework for time-critical rendering". In: *Proceedings of the conference on Visualization '99*. VIS '99. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 115–122. DOI: 10.1109/VISUAL.1999.809875.

[Lai05] Samuli Laine. "A general algorithm for output-sensitive visibility preprocessing". In: *Proceedings of the 2005 symposium on Interactive 3D graphics and games*. (Washington, District of Columbia). I3D '05. New York, NY, USA: ACM, 2005, pp. 31–40. DOI: 10.1145/1053427.1053433.

[Lar08] Thomas Larsson. "Fast and Tight Fitting Bounding Spheres". In: *Proceedings of SIGRAD 2008*. Linköping Electronic Conference Proceedings 34. SIGRAD, Swedish Chapter of Eurographics. Stockholm, Sweden: Linköping University Electronic Press, Linköpings universitet, 2008-11, pp. 27–30. URL: http://www.ep.liu.se/ecp/034/ecp08034.pdf.

[LBS06] Torsten Langer, Alexander Belyaev, and Hans-Peter Seidel. "Spherical Barycentric Coordinates". In: *Proceedings of the 4th Eurographics Symposium on Geometry Processing*. Ed. by Alla Sheffer and Konrad Polthier. SGP '06. Eurographics Association, 2006, pp. 81–88. DOI: 10.2312/SGP/SGP06/081-088.

[LH91] David Laur and Pat Hanrahan. "Hierarchical splatting: a progressive refinement algorithm for volume rendering". In: *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '91. New York, NY, USA: ACM, 1991, pp. 285–288. DOI: 10.1145/122718.122748.

[LSC03] Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. "Ray space factorization for from-region visibility". In: *ACM SIGGRAPH 2003 Papers*. (San Diego, California). SIGGRAPH '03. New York, NY, USA: ACM, 2003, pp. 595–604. DOI: 10.1145/1201775.882313.

[LYX08] Lingchun Li, Xubo Yang, and Shuangjiu Xiao. "Efficient visibility projection on spherical polar coordinates for shadow rendering using geometry shader". In: *IEEE International Conference on Multimedia and Expo 2008*. 2008-04, pp. 1005–1008. DOI: 10.1109/ICME.2008.4607607.

[Mac01] Carl Machover. "Computer graphics pioneers: the Giloi's school of computer graphics". In: *ACM SIGGRAPH Computer Graphics* 35.4 (2001-11), pp. 12–16. DOI: 10.1145/563710.563713.

[Mat+07] Oliver Mattausch, Jiří Bittner, Peter Wonka, and Michael Wimmer. "Optimized subdivisions for preprocessed visibility". In: *Proceedings of Graphics Interface 2007*. (Montreal, Canada). GI '07. New York, NY, USA: ACM, 2007, pp. 335–342. DOI: 10.1145/1268517.1268571.

[MBW06] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. "Adaptive Visibility-Driven View Cell Construction". In: *Rendering Techniques 2006*. Ed. by Wolfgang Heidrich and Tomas Akenine-Möller. Proceedings of the 17th Eurographics Symposium on Rendering. Eurographics Association, 2006, pp. 195–205. DOI: 10.2312/EGWR/

EGSR06/195-205. URL: http://www.cg.tuwien.ac.at/research/publications/2006/MATTAUSCH-2006-AVC/.

[MBW08] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. "CHC++: Coherent Hierarchical Culling Revisited". In: *Computer Graphics Forum* 27.2 (2008-04). Proceedings of Eurographics 2008, pp. 221–230. DOI: 10.1111/j.1467-8659.2008.01119.x. URL: http://www.cg.tuwien.ac.at/research/publications/2008/mattausch-2008-CHC/.

[Mer02] Oscar E. Meruvia-Pastor. "Visibility Preprocessing Using Spherical Sampling of Polygonal Patches". In: *Short Presentations of Eurographics 2002*. Ed. by Isabel Navazo Alvaro and Philipp Slusallek. 2002-09. URL: http://www.cs.mun.ca/~omeruvia/research/research.html#VisibilityPrecomputation.

[Möb27] August Ferdinand Möbius. *Der barycentrische Calcul - ein neues Hülfsmittel zur analytischen Behandlung der Geometrie*. Leipzig, Germany: Verlag von Johann Ambrosius Barth, 1827. URL: http://books.google.com/books?id=eFPluv_UqFEC.

[MS01] Oscar E. Meruvia-Pastor and Thomas Strothotte. "Approximated View Reconstruction Using Precomputed ID-Bitfields". In: *Short Presentations of Eurographics 2001*. Ed. by Jonathan C. Roberts. 2001-09. URL: http://www.cs.mun.ca/~omeruvia/research/research.html#IDBitfields.

[Müc98] Ernst P. Mücke. "A Robust Implementation for Three-Dimensional Delaunay Triangulations". In: *International Journal of Computational Geometry & Applications* 8.2 (1998), pp. 255–276. DOI: 10.1142/S0218195998000138.

[NB04] Shaun Nirenstein and Edwin H. Blake. "Hardware Accelerated Visibility Preprocessing using Adaptive Sampling". In: *Proceedings of the 15th Eurographics Symposium on Rendering*. Ed. by Alexander Keller and Henrik Wann Jensen. EGSR '04. Norrköping, Sweden: Eurographics Association, 2004, pp. 207–216. DOI: 10.2312/EGWR/EGSR04/207-216. URL: http://people.cs.uct.ac.za/~snirenst/Vis/nirenstein_aggressive.pdf.

[NC03] Yuval Noimark and Daniel Cohen-Or. "Streaming scenes to MPEG-4 video-enabled devices". In: *IEEE Computer Graphics and Applications* 23.1 (2003), pp. 58–64. DOI: 10.1109/MCG.2003.1159614.

[Nur06] Antti Nurminen. "m-LOMA - a mobile 3D city map". In: *Proceedings of the 11th International Conference on 3D Web Technology*. (Columbia, Maryland, USA). Web3D '06. New York, NY, USA: ACM, 2006, pp. 7–18. DOI: 10.1145/1122591.1122593.

[Nur07] Antti Nurminen. "Mobile, hardware-accelerated urban 3D maps in 3G networks". In: *Proceedings of the 12th International Conference on 3D Web Technology*. (Perugia, Italy). Web3D '07. New York, NY, USA: ACM, 2007, pp. 7–16. DOI: 10.1145/1229390.1229392.

[PD90] Harry Plantinga and Charles R. Dyer. "Visibility, occlusion, and the aspect graph". In: *International Journal of Computer Vision* 5.2 (1990-11), pp. 137–160. DOI: 10.1007/BF00054919.

[PH03] Emil Praun and Hugues Hoppe. “Spherical parametrization and remeshing”. In: *ACM Transactions on Graphics* 22.3 (2003-07), pp. 340–349. DOI: 10.1145/882262.882274.

[Pha+02] Nam Pham Ngoc, Wolfgang van Raemdonck, Gauthier Lafruit, Geert Deconinck, and Rudy Lauwereins. “A QoS Framework for Interactive 3D Applications”. In: *Proceedings of the 10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*. WSCG 2002. 2002-02, pp. 317–324. URL: http://wscg.zcu.cz/wscg2002/Papers_2002/D17.pdf.

[RGS97] Amit Reisman, Craig Gotsman, and Assaf Schuster. “Parallel progressive rendering of animation sequences at interactive rates on distributed-memory machines”. In: *Proceedings of the IEEE Symposium on Parallel Rendering*. PRS ’97. Los Alamitos, CA, USA: IEEE Computer Society Press, 1997, pp. 39–47. DOI: 10.1109/PRS.1997.628294.

[Rot89] Tony Rothman. *Science à la mode: physical fashions and fictions*. Princeton, NJ, USA: Princeton University Press, 1989. URL: http://www.physics.princeton.edu/~trothman/science_contents.html.

[SA08] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*. Version 3.0. The Khronos Group Inc. 2008-08-11. URL: http://www.opengl.org/registry/doc/glspec30.20080923.pdf (visited on 2013-09-02).

[SA09] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*. Version 3.2 (Core Profile). The Khronos Group Inc. 2009-08-03. URL: http://www.opengl.org/registry/doc/glspec32.core.20091207.pdf (visited on 2013-09-02).

[SB96] Min-Zhi Shao and Norman Badler. *Spherical Sampling by Archimedes’ Theorem*. Tech. rep. MS-CIS-96-02. University of Pennsylvania, Department of Computer and Information Science, 1996-01. URL: http://repository.upenn.edu/cis_reports/184/.

[SHT03] Lidan Shou, Zhiyong Huang, and Kian-Lee Tan. “Supporting real-time visualization with the HDoV tree”. In: *Proceedings of the 2003 ACM symposium on Applied computing*. SAC ’03. New York, NY, USA: ACM, 2003, pp. 966–971. DOI: 10.1145/952532.952721.

[SK97] Edward B. Saff and Arno B. J. Kuijlaars. “Distributing many points on a sphere”. In: *The Mathematical Intelligencer* 19.1 (1997), pp. 5–11. DOI: 10.1007/BF03024331.

[SR09] Wendel B. Silva and Maria Andréia Formico Rodrigues. “A lightweight 3D visualization and navigation system on handheld devices”. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. (Honolulu, Hawaii). SAC ’09. New York, NY, USA: ACM, 2009, pp. 162–166. DOI: 10.1145/1529282.1529318.

[SSS74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. “A Characterization of Ten Hidden-Surface Algorithms”. In: *ACM Computing Surveys* 6.1 (1974-03), pp. 1–55. DOI: 10.1145/356625.356626.

[Str74] Wolfgang Straßer. “Schnelle Kurven- und Flächendarstellung auf graphischen Sichtgeräten”. PhD thesis. Berlin, Germany: TU Berlin, 1974.

[Tac+04] Nicolaas Tack, Francisco Morán, Gauthier Lafruit, and Rudy Lauwereins. "3D graphics rendering time modeling and control for mobile terminals". In: *Proceedings of the 9th International Conference on 3D Web Technology*. Web3D '04. New York, NY, USA: ACM, 2004, pp. 109–117. DOI: 10.1145/985040.985056.

[TS91] Seth J. Teller and Carlo H. Séquin. "Visibility preprocessing for interactive walkthroughs". In: *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '91. New York, NY, USA: ACM, 1991, pp. 61–70. DOI: 10.1145/122718.122725.

[Ulr00] Thatcher Ulrich. "Loose Octrees". In: *Game Programming Gems*. Ed. by Mark DeLoura. Game Programming Gems. Boston, MA, USA: Charles River Media, 2000. Chap. 4.11, pp. 444–453.

[vS99] Michiel van de Panne and A. James Stewart. "Effective Compression Techniques for Precomputed Visibility". In: *Proceedings of the 10th Eurographics Workshop on Rendering*. Ed. by Dani Lischinski and Gregory Ward Larson. EGWR '99. Eurographics Association, 1999, pp. 305–316. DOI: 10.1007/978-3-7091-6809-7_27.

[Wan+04] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P. Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *Image Processing, IEEE Transactions on* 13.4 (2004-04), pp. 600–612. DOI: 10.1109/TIP.2003.819861.

[Wat00] Alan Watt. *3D Computer Graphics*. 3rd ed. Harlow, England: Pearson Education Limited, 2000.

[WB09] Zhou Wang and Alan Conrad Bovik. "Mean squared error: Love it or leave it? A new look at Signal Fidelity Measures". In: *IEEE Signal Processing Magazine* 26.1 (2009-01), pp. 98–117. DOI: 10.1109/MSP.2008.930649.

[Won+06] Peter Wonka, Michael Wimmer, Kaichi Zhou, Stefan Maierhofer, Gerd Hesina, and Alexander Reshetov. "Guided visibility sampling". In: *ACM SIGGRAPH 2006 Papers*. SIGGRAPH '06. New York, NY, USA: ACM, 2006, pp. 494–502. DOI: 10.1145/1179352.1141914.

[YN00] Ilmi Yoon and Ulrich Neumann. "Web-Based Remote Rendering with IBRAC (Image-Based Rendering Acceleration and Compression)". In: *Computer Graphics Forum* 19.3 (2000-09). Proceedings of Eurographics 2000, pp. 321–330. DOI: 10.1111/1467-8659.00424.

[Zha+97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff III. "Visibility culling using hierarchical occlusion maps". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 77–88. DOI: 10.1145/258734.258781.