Diplomarbeit

Occlusion Culling Using Hardware-Based Occlusion Queries

ausgeführt am
Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

unter Anleitung von
Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer
und
Univ.Ass. Dipl.-Ing. Dr.techn. Michael Wimmer
als verantwortlich mitwirkenden Universitätsassistenten

durch

Harald Piringer
Matr. Nr. 9826148
A-3400 Klosterneuburg, Holzgasse 186

Datum Unterschrift

Abstract

Today, a major challenge of computer graphics is the generation of realistic images at rates that arouse the impression of fluid motion in the viewer. Of central importance in this context is the application of specialized hardware, which has experienced an impressive evolution in recent years, increasing both speed and functionality in a significant fashion. However, displaying complex scenes like whole cities requires dealing with such an amount of data, that dedicated acceleration algorithms are still necessary in order to cope with the tight temporal constraints. In doing so, methods for detecting invisible parts of the scenery play a key role. A particular challenge is classifying those objects in an efficient way which are invisible due to being entirely occluded by other objects in front of them. Although this visibility problem belongs to the classical tasks of computer graphics, just recently introduced extensions to the graphics hardware permit the design of new algorithms with fascinating opportunities, but also requirements that have been of no concern to previous approaches.

This master thesis deals with the task of developing an efficient algorithm for solving the visibility problem for arbitrary scenes, which is tailored towards the *NV_occlusion_query* OpenGL extension, in different ways:

One method approximates the potential occlusion within a given scenery from a certain viewpoint by constructing a directed acyclic graph, which is in turn used to be able to issue as many occlusion queries as possible in a concurrent fashion. Several extensions amend and improve the core algorithm by employing a more equal load balancing and exploiting both temporal and spatial coherence. Finally, this approach is extended by incorporating an appropriate spatial hierarchy.

Although this technique is by and large significantly superior to rendering without occlusion culling, the obtained results are not satisfactory in every respect. Therefore, a second approach is proposed that does not rely on any graph. This hierarchical algorithm depends entirely on the visibility classification of previous frames and stresses thus the aspect of temporal coherence. Even though it is considerably simpler than the first approach, it yields superior and generally convincing results.

Both approaches are conservative (which means that they do not affect the correctness of the resulting image), but can easily be modified in a way that permits to trade off quality for speed by tolerating precisely definable mistakes.

Kurzfassung

Zu den größten Herausforderungen der heutigen Computergraphik zählt es, realitätsnahe Bilder mit solcher Geschwindigkeit zu erzeugen, dass beim Betrachter der Eindruck einer flüssigen Bewegung erweckt wird. Massive Bedeutung kommt dabei dem Einsatz spezieller Hardware zu, deren Entwicklung in den letzten Jahren in beeindruckender Weise sowohl zu bedeutend mehr Geschwindigkeit als auch einer ständigen Erweiterung der Funktionalität führte. Dennoch fällt bei der Darstellung komplexer Szenen wie ganzer Städte eine derartige Datenmenge an, dass Algorithmen zur Steigerung der Effizienz nötig sind, um den zeitlichen Anforderungen gerecht werden zu können. Einen wesentlichen Beitrag liefern dabei Methoden zur frühzeitigen Identifizierung unsichtbarer Teile der Geometrie, wobei es eine besondere Herausforderung darstellt, speziell jene Objekte auf effiziente Art und Weise als unsichtbar zu klassifizieren, die gänzlich durch davor liegende Objekte verdeckt werden. Wiewohl dieses Sichtbarkeitsproblem zu den klassischen Aufgabenstellungen der Computergraphik zählt, erlauben erst in jüngerer Zeit eingeführte Erweiterungen der Hardware das Design neuer Lösungen mit faszinierenden Möglichkeiten, aber auch bisher nicht auftretenden Anforderungen.

In dieser Diplomarbeit wird die Aufgabe, einen speziell auf die *NV_occlusion_query* Erweiterung von OpenGL zugeschneiderten, effizienten Algorithmus zur Sichtbarkeitsberechnung allgemeiner Szenen zu entwickeln, auf mehrere Arten angegangen:

Eine Methode approximiert potentielle Verdeckungen innerhalb einer Szene von einem gewissen Betrachtungspunkt aus mittels eines gerichteten azyklischen Graphen und verwendet diesen, um möglichst viele Sichtbarkeitstests parallel einsetzen zu können. Diverse Erweiterungen ergänzen und verbessern das Grundprinzip durch eine ausgewogenere Verteilung der Last, sowie durch Ausnützen von zeitlicher und räumlicher Kohärenz. Schließlich wird diese Methode durch den Einsatz einer geeigneten räumlichen Hierarchie ausgebaut.

Da dieser erste Ansatz zwar im Großen und Ganzen dem Rendern ohne Sichtbarkeitsberechnung deutlich überlegen ist, jedoch nicht in jeder Hinsicht zufriedenstellende Resultate liefert, wird eine zweite Methode vorgestellt, die ohne Graph auskommt. Dieser ebenfalls hierarchische Ansatz beruht zur Gänze auf der Sichtbarkeitsklassifikation vorheriger Frames und betont somit die Ausnutzung zeitlicher Kohärenz. Obwohl deutlich einfacher als der erste, erzielt er diesem durchwegs überlegene und generell sehr zufriedenstellende Ergebnisse.

Beide Ansätze sind vom Prinzip her konservativ (das heißt sie führen zu keinerlei Fehler im resultierenden Bild), können allerdings durch Betrachtung des Ergebnisses der Sichtbarkeitstests auf einfache Weise dazu verwendet werden, die Performance durch Tolerierung genau spezifizierbarerer Fehler weiter zu steigern.

Danksagung

Diese Diplomarbeit entstand im Rahmen des *Urban Visualization* Projekts am Instituts für Computergraphik der Technischen Universität Wien und wäre ohne die tatkräftige Unterstützung mehrerer nicht möglich gewesen.

Meinen Dank möchte ich zunächst an meinen Betreuer, Herrn Dr. Michael Wimmer richten, der mir einerseits das der Implementierung zugrunde liegende Framework zur Verfügung gestellt und mich andererseits in zahlreichen Treffen und Diskussionen wissenschaftlich in die eingeschlagene Richtung gelenkt hat. Weiters möchte ich mich bei Herrn Dr. Jiri Bittner bedanken, auf dessen Arbeit wesentliche Teile aller hierarchischen Ansätze beruhen und der mit mir in langen Diskussionen das Konzept des ‚hierarchical temporal-coherence based“ Ansatzes entwickelt hat. Ebenfalls gilt mein Dank Herrn Univ. Prof. Dr. Werner Purgathofer als zuständiger Professor.

Aber in aller erster Linie bin ich meinen Eltern zu Dank verpflichtet, ohne deren Unterstützung mir das Studium der Informatik nicht möglich gewesen wäre, zu dessen Abschluss diese Diplomarbeit verfasst worden ist.

Contents

1 Introduction

1.1 The Goals of Real-Time Rendering

Real-time rendering belongs to the area of computer graphics. While in former times, the ultimate goal has been to produce appealing still images by mimicking real photographs, the tremendous evolution of graphics hardware within the past few years has shifted the focus towards applications that generate several (dozens) images per second. In combination with interactivity, this allows a user to become immersed in an artificial world. Real-time rendering has numerous applications, with computer and video games being the most popular and economically most important.

The main goal of real-time rendering is speed, since the image must be updated about 60 times per second in order to avoid disturbing visual artefacts. Apart from speed, another important goal is visual quality: Both hardware and software constantly strive for an increase of the image quality referring to the physical resolution as well as the degree of realism determined by the scene complexity and the application of convincing shading techniques. When designing an application, one usually faces a tradeoff between these two contradicting goals, as improving the image quality normally slows down the rendering process. The challenge is to sustain sufficiently high frame rates while maintaining an image quality hardly discernable from reality.

1.2 Occlusion Culling

An imperative is that the generated images depict a correct visibility situation, which means that surfaces being closer to the viewer must not be hidden by more distant objects. Such errors are ruled out by modern graphics accelerators which employ a depth-buffer (Z-buffer) for this task and the applications usually do not have to heed this issue on a per-triangle, or even per-pixel level. However, before being rejected by the Z-buffer, much effort has already been wasted in transferring the geometry and according data like textures to the graphics card, transforming it and scan-line converting numerous triangles. Therefore, a significant speed up of the rendering process can be achieved when committing only such objects that will contribute to the resulting image.

The task of identifying visible geometry is referred to as visibility culling. On a per-object level (in this context, the term ‘object’ denotes a certain independent geometry – ranging from a single triangle up to several thousands – together with all data required for its depiction), visibility culling comprises view-frustum culling and occlusion culling. The intention of the first one is to remove those objects from the overall scene which are definitely invisible, being placed outside the space captured by the camera (referred to as view frustum), and is in practice not complicated to realize. Occlusion culling, on the other hand, further refines the set of objects passing view-frustum culling by seeking to detect those ones which need not be rendered due to being (entirely) occluded by others. This is a rather complex issue and a proper implementation must take many aspects into consideration in order to succeed in actually accelerating the rendering process.

1.3 Hardware Occlusion Queries

Many approaches have been presented for doing occlusion culling – see chapter 2.5. Most of them depend completely on the CPU by approximating the resulting image in one way or the other and rejecting those objects that fail the usually more conservative CPU-test. The drawback is the observation that the bottleneck of a considerable number of applications already lies on the CPU side – thus additional work by doing occlusion culling could easily have the contrary effect as intended. Now, the idea behind employing the graphics hardware is to distribute the load of occlusion culling more equally between CPU and GPU as well as to exploit the specialized hardware for a task very similar to its actual purpose.

Hardware occlusion queries follow a simple pattern: A typically conservatively simplified geometry of an object is sent to the graphics hardware as usual. Yet instead of affecting any buffers, a result is returned to the application providing the information if anything would have been drawn. If not, rendering the original object can safely be skipped. The main problem is the non-negligible latency between issuing a test and waiting for the result to become available, hence a major challenge when designing a respective algorithm is to fill this time with meaningful work for the CPU.

1.4 Main Contributions

All research done for this master thesis bases on the application of hardware occlusion queries with certain properties, which are already realized in the NV_occlusion_query OpenGL-extension. Therefore, all algorithms are designed towards its very properties. This master thesis contributes to the field of occlusion culling for real-time rendering three algorithms that strive for an optimal application of this extension. The cornerstones of my work are as follows:

- A method for approximating occlusion within a scene by establishing a directed acyclic graph – the so-called occlusion graph – and utilizing the constraints imposed by it in order to maximize the parallelism between CPU and GPU.
- Ways to increase efficiency by exploiting both temporal and spatial coherence within the occlusion graph and the obtained test results in a non-hierarchical setting.
- A second approach with the goal to reduce the average effort by incorporating a hierarchy into the occlusion-graph based approach and demonstrating how temporal coherence can then be exploited in an even more distinct way.
- A separate hierarchical approach which stresses temporal coherence and manages to overcome flaws of prior approaches almost entirely.

The pros and cons of the various approaches are extensively discussed and compared to each other, providing evidence that remarkable speed ups are feasible and have actually been achieved in a concrete implementation.

Furthermore, this master thesis contains an overview of real-time rendering in general and other approaches for occlusion culling in particular.

1.5 Structure of this Thesis

This master thesis is organized as follows:

- After giving a brief introduction to the field of real-time rendering, chapter 2 focuses on work related to the topic of occlusion culling. It especially centers on a thorough description and comparison of available hardware occlusion queries, yet it deals with other acceleration techniques as well.
- Chapter 3 motivates the idea of approximating occlusion within a scene by a directed acyclic graph and uses it for a first approach. After analyzing major flaws, several improvements are presented and discussed to mitigate them.
- Chapter 4 deals with the application of spatial hierarchies in the context of occlusion culling. First, it motivates this step and points out general facts concerning their properties and construction. Then it incorporates a kd-tree into the graph-based approach of chapter 3. Finally it introduces a second hierarchical approach which exploits temporal coherence in a very distinct fashion.
- While the approaches have already individually been discussed subsequent to the respective descriptions, chapter 5 provides a detailed overall comparison and assesses the practical relevance.
- Chapter 6 concludes the thesis by summarizing the most important insights and proposes possible future work.

2 Related Work

2.1 Real-time Rendering

Real-time rendering belongs to the area of computer graphics and as such it is concerned with generating images. What distinguishes real-time rendering from other fields like photo-realistic rendering is the requirement to update individual images at a rate that does not permit the viewer to tell apart separate frames any more. Unlike movies, the user is usually able to control the process, and his actions – or reactions – take effect immediately. This way, the user becomes immersed in a dynamic process, a feedback loop. Whilst a certain feeling of interactivity is already aroused when updating the displayed image about six times per second – usually referred to as *frames-per-second*, abbreviated as *fps* – a refresh rate of the monitor of at least 60 fps must be exceeded in order to get rid of any disturbing visual artefacts [Helm94].

Furthermore, rendering in real time normally means depicting three-dimensional scenes. While interactivity with three-dimensional worlds at frame rates that convey the impression of fluid motion could be – and has also been – realized entirely by the CPU, the constant demand for an increase of scene complexity and detail has been the driving force for an amazing development: Beginning with the introduction of the 3Dfx Voodoo 1 in 1996 [Eccl00], more and more work has been shifted from the CPU to dedicated specialized graphics accelerators, so called *GPUs*, thus making the CPU free for other tasks (more precisely, graphics accelerators like the workstations of Silicon Graphics have also been available before 1996, but the Voodoo 1 was the first serious respective product that was successful in the consumer market).

Among important applications of real-time rendering, like the interactive visualization of scientific data or the ability for architects to study not yet realized buildings in a previously unthinkable way by walkthroughs, the killer applications nowadays are without doubt computer games. Their importance manifests itself in the economical dimension of the computer-gaming industry and the fact that enhancing realism even further has become a major argument for the development and sale of still faster CPUs, which must keep up with the higher speed of the graphics hardware in order to supply the graphics accelerator with meaningful work.

Summarizing, real-time rendering in our context refers to an interactive process of generating and depicting satisfactorily realistic images of mathematically defined three-dimensional scenes utilizing specific hardware at frame rates that allow the user to become convincingly immersed. An excellent introduction to the field of real-time rendering which also served as an important source of information for this master thesis can be found in [Möll02].

2.2 The Graphics Pipeline

As emphasized in the previous chapter, a major criterion for real-time rendering is speed. In the context of work to be done, speed means that a certain task is accomplished in as little time as possible. When the work can be broken down to several small jobs, we achieve a short overall duration either by carrying out each job on its own in almost no time, or by dealing with several jobs simultaneously without (remarkably) slowing down the progress of an individual one.

The scene is usually broken down to dozens or even hundreds of smaller parts (for instance one wall of a house) which are easier to handle and are sent to the graphics hardware one by one. Although the individual parts obviously may differ completely from each other with regards to the optical result, the steps of the processing as such – the involved 'jobs' – are more or less the same for each. The totality of all stages necessary to achieve the desired optical result is called *graphics pipeline*. The idea behind subdividing the overall work this way is the same why modern CPUs are organized in the same manner: parallelism [Patt98]. Generally speaking, dividing work into *n* pipeline stages should ideally give a speed-up factor of the whole work of *n* by increasing the throughput *n*-fold. Note however that the time it takes to complete a single part of the work – the latency of this part – remains roughly identical. Concerning the graphics pipeline, several jobs can be processed in a concurrent fashion when each stage deals with one job. This implies that the speed of the pipeline is determined by its slowest stage, called *bottleneck*, no matter how fast the other stages may be [Möll02]. Therefore, when optimising performance, it is crucial to locate that bottleneck and implement measures that sustain a well balanced load throughout the whole time.

On a coarse level, the application generates drawing commands and commits them to the GPU. There, the geometry gets first transformed and then rasterized before being written to the buffer which serves as source of the displayed image. Each stage can further be subdivided by distinguishing various steps. While modern GPUs may comprise several dozens of internal stages, Fig. 2.1 illustrates the (traditional) graphics pipeline by means of eleven stages, which shall briefly be outlined in this chapter. Since this graphics pipeline is the core of real-time rendering, just a brief glance on each stage can be given here. Most aspects are highlighted in much greater detail in [Möll02] which itself references much work providing focussed discussions on specific topics. Furthermore, note that applying vertex shaders and pixel shaders within very recent GPUs replaces parts of the traditional graphics pipeline. However, since this has no further impact on the discussion about occlusion culling, this aspect is omitted here.

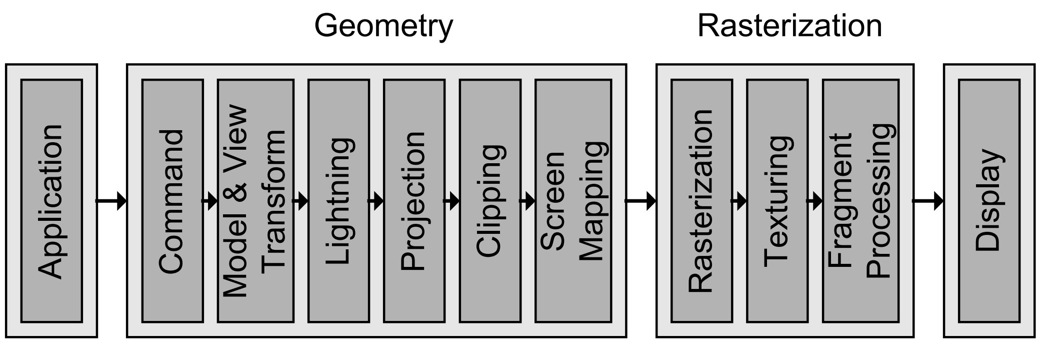


Fig. 2.1: The traditional graphics pipeline

- **Application**: One major task of an application generating real-time graphics is to maintain a description of the scene in some form (for instance a scene graph, as described in the next chapter) and to issue appropriate drawing commands – usually by calling functions of a certain API like OpenGL [Woo99] or Direct3D – that allows the subsequent stages to produce the desired results. However, an application typically has to deal with lots of other tasks as well: Handling (maybe user-generated) events, doing computations like the AI of characters in a computer game and maintaining an overall state, just to mention a few.

- **Command**: Since the rates at which the application issues drawing commands and at which the GPU can actually handle them will typically not coincide, buffering commands is a crucial issue in order to balance the load and to maximize parallelism. Besides, the driver must maintain a certain graphics state and the (mostly platform-independent) commands sent by the application must be interpreted and unpacked in order to obtain instructions which are meaningful to the GPU.
- **Model & View Transform**: Each geometry must be transformed to a global coordinate system (known as world space) that must itself be transformed in order to simulate an arbitrarily position-able camera. Among other literature, refer to [Woo99], [Hear94] or [Watt93] for a more detailed description.
- **Lighting**: While modern GPUs also permit lighting to be done on a per-pixel basis, typically the influence of the various light sources is computed by evaluating a lighting equation for each vertex and the result is interpolated over the triangles (in subsequent stages). This technique is referred to as *Gouraud shading* [Gour71].
- **Projection**: The view volume must be transformed into a unit cube, which is called *canonical view volume*. Common transformations are the *orthographic* (or parallel) *projection* and the *perspective projection*. The latter takes depth into consideration by bringing two points the closer to each other the greater their distance is to the viewer. This simulates more accurately how the human eye works (see for instance [Woo99] for a detailed description).
- **Clipping**: Primitives that are partially inside the viewing volume require *clipping*, which means cutting away those parts that are outside (possibly yielding new vertices).
- **Screen Mapping**: Before entering the rasterization stage, the normalized x- and y-coordinates must be scaled in order to match the resolution of the output device (known as *screen coordinates*).
- **Rasterization**: This step refers to converting continuous triangles to a set of discrete fragments by sampling each triangle in a non-ambiguous way (this process is called *scan conversion* and is described for instance in [Akel88]). Furthermore, properly interpolating all data specified on a per-vertex basis (mainly colour, light and depth) for each fragment plays an important role as well.
- **Texturing**: In order to increase the degree of realism, details are added by providing additional information on an interpolated per pixel basis. Formerly, this simply meant ‘gluing’ an image onto the object. Nowadays, several textures can be combined in various ways (called *multi-texturing*) and in addition to being displayed directly, the information provided by the textures can be used as input for further calculations (like bump-mapping).
- **Fragment Processing**: Before becoming actually visible, tests decide if a certain fragment should actually affect the final image, the *stencil test* (confining the area where modifications are admissible) and the *depth test* – or *Z-test* – (rejecting fragments lying behind already drawn geometry) probably being the most famous. Moreover, instead of overwriting previous pixels, using *alpha blending* permits to simulate transparency by combining the current colour with the previous one.
- **Display**: In the final stage, the digital image is converted to an analogue signal for the display device (typically a monitor).

2.3 Scene Graphs

As briefly mentioned in the previous chapter, one task of an application dealing with real-time rendering is maintaining a sort of database that describes the scene and serves as a basis for generating drawing commands. A *scene graph* is a higher-level tree structure that comprises all necessary information in order to render images. It is discussed here because the system which served as basis for the implementation of the approaches of this master thesis employs a scene-graph technique.

Scene graphs organize the scene in a hierarchical fashion using trees. However, unlike other hierarchical spatial data structures like ordinary bounding-volume hierarchies or BSP trees (see chapter 4.1 and 4.2), they contain more than just geometry: The structure is augmented with textures, transformations, material properties, light sources and other settings relevant for the rendering process. A typical way of enabling a certain feature (e.g. setting a certain colour) is to generate an appropriate node and place everything that should be affected in form of a subtree underneath this node.

One often distinguishes between nodes bearing some actual content like some kind of geometry or light source (these nodes are normally leaves of the tree) and so called *group nodes* that can define a common property for the respective subtree or activate only a certain part of its subtree for rendering. Therefore, typically many specialized group nodes for various tasks exist. The idea is that restricting all effects on the subtree permits a modular construction of the scene because all other parts of the scene graph stay unaffected. Moreover, nodes frequently require further *components* for their work – for instance, a shape node may comprise both the geometry and a certain colour.

Performing some action typically involves traversing the scene graph. Possible actions are saving the scene in its current state or finding the first object that intersects a certain ray, but probably the most common action is rendering the scene graph. Traversals are performed in depth-first order and an important issue for rendering is maintaining a state in order to realize the restricted effects as explained above. In order to allow fast culling techniques, each node has usually assigned a bounding volume. Thus scene graphs are somewhat related to bounding-volume hierarchies.

A common practice is that certain parts of the scene graph are referenced by multiple parents. For instance when modelling a city, much memory can be saved if a car object is stored only once and used various times – each time being transformed differently. However, when sharing nodes, the graph degrades from a tree to a more general *directed acyclic graph* (see [Corm90]) which may tremendously complicate some algorithms working on it, as discussed in [Eber00].

Among others, well known scene graph APIs are Open Inventor [Wern94] and Java3D [Nade98], which also the graphics engine of this master thesis was modelled on.

2.3.1 Engine

The implementation to this master thesis is based on the *YARE* graphics engine (YARE is the abbreviation for Yet Another Rendering Engine), which has been developed at the institute for computer graphics of the technical university of Vienna mainly by Michael Wimmer. Although it is part of the *Urban Visualization* project that aims at the creation of an integrated solution for modelling and real-time visualization of large and medium-scale urban environments (see: http://www.cg.tuwien.ac.at/research/vr/urbanviz/), its capabilities are by no means restricted to this specific kind of scene, but it is a generally applicable graphics engine that also serves as platform for the implementation, testing and assessment of new techniques concerning image-based rendering and occlusion culling.

As mentioned above, it organizes the scene in form of a scene graph, which shows many similarities to Java3D. Internally, OpenGL is used as lower-level graphics API. Most facts concerning scene graphs pointed out in the previous chapter apply to it as well: It distinguishes between various kinds of group nodes (e.g. transforming the contents, selecting one child for rendering and allowing the shared usage of a subtree), it knows several sorts of leaves (e.g. three-dimensional shapes, many kinds of lights and nodes affecting others) and many components. Traversals follow the *visitor design pattern* [Gamm94] and implementing the approaches for occlusion culling presented in this master thesis could be done by deriving the class implementing rendering the scene graph without occlusion culling.

2.4 Overview about Acceleration Techniques

As pointed out in chapter 2.1, real-time rendering means striving for steady frame rates being at least the update frequency of the monitor. Although modern systems (this includes both CPUs and GPUs) are already amazingly fast, the scene complexity has been continually rising at a speed the computers could not keep pace with: They are still entirely overwhelmed with generating images at the desired rate that come at least close to reality. Combining the contradicting goals of maintaining high image quality and constantly high frame rates requires intelligent acceleration algorithms: By achieving the final result as efficiently as possible, both higher frame rates as well as a more detailed scene geometry become possible using the same hardware.

Acceleration techniques are a huge topic and a vivid area of research. There is an abundance of approaches that try to enhance performance in many different ways (refer to [Möll02] for an overview). Because this master thesis deals with an approach that belongs to the category of *occlusion culling*, only the work related to this subject is discussed in more detail (in the next chapter), while this chapter gives but a brief glance on other topics concerning acceleration techniques.

There is one aspect, all acceleration techniques have in common: As the name suggests, the ultimate goal is to increase performance. As reasoned in chapter 2.2, this performance is determined by the bottleneck of the graphics pipeline. On a coarse level, this can either be the application-, the geometry- the rasterization stage or the graphics bus connecting the CPU with the GPU. Many approaches – among them those presented in this master thesis – aim at reducing the load of one particular stage which often incurs increasing the load of another stage (at least slightly). Consequently, acceleration techniques can only succeed if they reduce the load of that stage that actually turns out to be the bottleneck, which requires careful examinations and measurements. However, the location of this bottleneck is likely to change several times within a frame, which usually renders accurate predictions about the effect of acceleration algorithms very difficult.

While some approaches can reasonably be combined (for instance employing spatial data structures for occlusion culling), others are unaffected by each other as far as the implementation is concerned, but may be sensible complements in order to achieve the desired frame rates. For example, if all culling techniques fail to reduce the geometry to an amount the GPU can cope with – maybe because almost the whole scene is actually visible – a further option is to compromise quality with speed by decreasing the complexity of the individual objects using for instance LODs.

LODs (which is the commonly used abbreviation for *Levels of Detail*) refer to using a more and more simplified version of a certain geometry if it contributes less and less to the rendered image. They have been used first by Clark [Clar76]: When being looked at from a distance of two meters, a car model may require thousands of triangles to convey a realistic look, but with gradually increasing distance, a few hundred triangles may suffice and once the car is kilometers away, about 50 triangles may be enough. Doing so is usually feasible without seriously affecting the image quality, since the area covered by the model has become so tiny that using the original complex model for an object being kilometers away will typically not significantly increase its degree of realism – if the simplification is done properly. In general, using LODs decreases the load mainly for the geometry stage, while the number of fragments remains roughly identical. In the context of occlusion culling, LODs are a reasonable complement as they decrease the complexity of the visible geometry.

Issues related to LODs are the generation, their selection and the way how the transition between the levels is realized. Generating LODs means simplifying a certain geometry by removing vertices and is usually done as a pre-processing step (see for instance [Garl97]). Typically, some kind of metric is employed to identify the vertex that causes the least distortions when being removed. Throughout the rendering process, a concrete level is chosen based on some static criterion like the distance or – interesting in the context of the NV_occlusion_query extension – the amount of covered space. Further possibilities are reacting to recent frame times or evaluating some cost/benefit model [Funk93]. Switching between various levels can either be done in an abrupt fashion (which often yields ugly popping artifacts), or by some sort of blending [Gieg02] or morphing between the levels of interest.

Another acceleration technique is *image-based rendering*. According to Lengyel [Leng98], rendering geometric models is a physically based way to obtain the visual result, while permitting images as another primitive is an appearance based way to achieve the same goal. Polygons represent an object in a reasonable fashion from any view, but images have the advantage of being independent of the scene complexity they depict. *Textures* are the most common way to incorporate images into the rendering process. Other simple methods are *sprites* which can arbitrarily be moved around the screen (for instance the mouse cursor) [McCu00] and *billboards* [McRe99] referring to rotation-symmetric objects comprising a single polygon which always faces to the viewer. In combination with partial transparency, many ‘special effects’ like *lens flares* [King00] and *particle systems* (see [Möll02] for an overview of literature) simulating fire, smoke, falling water and so on can be modelled this way.

In the context of acceleration algorithms, primarily *impostors,* introduced in a statically pre-computed version by Maciel and Shirley in 1995 [Maci95] and extended for dynamic generation by Schaufler [Scha95], play an important role. Simply put, the idea is to replace a complex three dimensional object by an image depicting it. These images are obviously view dependent and thus only valid as long as the viewpoint stays within a certain region. Generating an impostor requires rendering the according geometry into a texture (see [Wimm01]). Impostors can further be augmented with a depth component which is usually called *depth sprite* or *nailboard* [Scha97]. Furthermore, impostors can be used in a hierarchical fashion, known as *hierarchical image caching* [Shad96]. Moreover, instead of being a single polygon, they can be used as textures of simple meshes which permits a better adaptation to complex structures, called *multimesh impostors* [Deco99].

Rather than being an acceleration technique on their own, *spatial data structures* refer to organizing the scene in a hierarchical fashion and serve as the basis for many algorithms. They usually allow to increase efficiency by exploiting spatial coherence within the scene and often succeed in reducing the average effort from *O(N)* to *O(log N)*. This can be used for instance when computing ray intersections, doing collision detection and not least in the context of occlusion culling. After all, two of the approaches presented in this master thesis employ spatial hierarchies as well; that is why a whole chapter (chapter 4) was dedicated to them where they are discussed in more detail.

Another tool for many acceleration techniques is the usage of *bounding volumes* (BV). A BV entirely encloses one or multiple objects and is typically a much simpler geometrical shape than the contained objects. This permits tests to be done much faster using the BV than with the original shapes. Common types of BVs are spheres, oriented bounding boxes and axis-aligned bounding boxes. An extension is organizing them in a hierarchical fashion, called *bounding-volume hierarchies*, in order to exploit spatial coherence, as mentioned in the previous paragraph. The approaches of this master thesis make heavy use of bounding volumes (especially axis-aligned bounding boxes) as well as their organization within a hierarchy.

Finally, various *culling techniques* exist in order to remove all those portions from a scene that are not considered to contribute to the final image. In the context of computer graphics, also the term *visibility culling* is common to refer to this task. This term is mentioned within this overview as it is of course another (and very important) acceleration technique, yet since the topic of this master thesis falls into the category of occlusion culling, being part of visibility culling, it is discussed in more detail in the next chapter.

2.5 Visibility Culling

In the following discussions, objects refer to geometric elements that make up the scene. An object may be a collection of graphics primitives or a single primitive.

Visibility culling is essentially based on the observation, that most of the time, large parts of the scene are not visible and thus do not contribute to the visual result. This invisibility basically may be due to three reasons (as illustrated in Fig. 2.2):

- The camera only covers a certain space, known as view frustum. All objects being outside this view frustum are definitely invisible; detecting them is referred to as *view-frustum culling*. Analogically, the human eye can not perceive anything behind our back (neglecting the effect of mirrors).
- Objects being (at least partially) inside the view frustum can still be invisible because they are behind other (opaque) objects – they are occluded by them. Detecting them is called *occlusion culling*.
- Even for non occluded objects within the view frustum, usually not the whole surface is visible at the same time, but only those parts facing to the viewer. Removing parts of the surface facing away is known as *backface culling*.

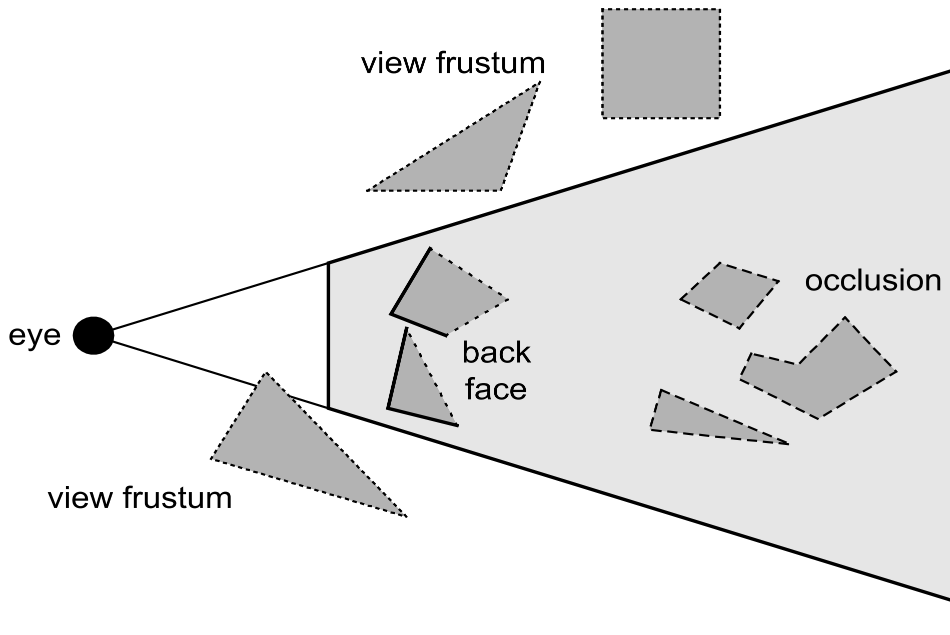


Fig. 2.2: Different techniques of visibility culling. (Illustration after Cohen-Or et al. [Cohe02])

The remaining parts are those actually depicted in the final image. They are known as the *Exact Visible Set* (EVS), being defined as all primitives that are partially or fully visible. *Visibility Culling* refers to removing objects from the scene before rendering it (in this context 'remove' means that they are simply not sent to the GPU, of course they are not actually deleted) and comprises backface culling, view-frustum culling and occlusion culling as listed above. Ideally, the set of objects passing visibility culling exactly coincides with the EVS. In practice, only a certain approximation can be determined with reasonable effort, which is called *Potentially Visible Set* (PVS). The more closely this PVS matches the EVS, the higher is its quality. The PVS can be distinguished as follows:

- *Conservative*: PVS $\supseteq$ EVS. We do not lose image quality but tolerate to render some additional objects in vain. Conservative approaches are usually preferred.
- *Approximate*: PVS $\sim$ EVS. Some objects are rendered in vain, while some minor mistakes are tolerated.
- *Aggressive*: PVS $\subseteq$ EVS. All objects being actually rendered contribute to the image. However, this often leads to perceivable errors.

Rendering invisible objects does not affect the image, though: Parts outside the view frustum are clipped in the geometry stage of the graphics pipeline (see chapter 2.2) and occlusion is resolved on a per-pixel basis using the Z-buffer. The reason why we still bother to compute the PVS is speed: The fastest polygon to render is the one never sent down the graphics pipeline. Rejecting great parts of the scene already at the application stage may save much traffic on the graphics bus (which can be a bottleneck as well), it may prevent the transformation, lighting and clipping of thousands of vertices, it may avoid the rasterization of countless fragments and – last but not least – it may avert myriads of expensive memory accesses on the graphics board for both textures and buffers. However, it must be emphasized once more that all these savings are pointless if the bottleneck already is on the CPU: Every kind of visibility culling – and this especially applies to all approaches dealing with occlusion culling – can only pay off if it manages to take away load from the bottleneck, otherwise, it may actually deteriorate performance.

Nowadays, *backface culling* is implemented within the GPU and can be turned on and off using appropriate functions of the graphics API. As explained above it detects those triangles facing away from the viewer within the geometry stage and does not pass them on to subsequent stages, thus reducing work for the rasterization stage. The sign of the scalar product of the normal vector of a triangle with the vector of the viewing direction determines whether the triangle is facing away from the viewer or not. If – after appropriate transformations – the camera looks in the direction of the negative z-axis (as it is usually the case), this computation boils down to checking the sign of the z-component of the normal vector itself. Shirman and Abi-Ezzi [Shir93] extend this technique to groups of back-facing polygons by calculating a cone out of all normal vectors and prove that the geometry can be culled away if the viewpoint is located in some region with respect to the cone. This technique has further been extended by Kumar and Manocha [Kuma96].

View-frustum culling [Clar76] is implemented in software, but due to its simplicity and general applicability, it is part of almost all real-time rendering systems. Generally speaking, some kind of bounding volume is tested in world space against all six planes comprising the view frustum of the virtual camera and is thereby classified as entirely inside, partially inside or entirely outside – the latter objects are culled away, thus reducing the load for the GPU and traffic on the graphics bus. View-frustum culling is often done in a hierarchical fashion, where the classification is refined for bounding volumes being partially inside the view frustum. Assarsson and Möller propose several improvements [Assa00] that are partly implemented in this thesis and are discussed in more detail in chapter 4.3.1.

2.5.1 Basics of Visibility and Occlusion

Determining visibility shows many similarities with computing shadows and many terms are used in both contexts alike. Various approaches for solving the visibility problem can roughly be classified as follows:

- *From-point approaches*
- *From-cell approaches*

Generally speaking, occlusion culling is usually done on a per-object level, instead of dealing with single polygons. Firstly, each occlusion test done at runtime incurs some non-negligible overhead and the geometry must be complex enough that skipping it can actually pay off, while approaches pre-computing visibility may face a memory problem. Secondly, triangles are usually sent to the GPU in triangle strips – a per-triangle classification would involve a costly rearrangement of data structures at runtime.

From-point approaches seek to determine all visible objects from a single viewing location in a certain viewing direction. Consequently, they are view dependent and (assuming that either the viewer or the scene is in motion) valid merely for a single frame. The computations are comparable with calculating shadows for a single point light source representing the viewpoint: In both cases, all objects of interest can be classified as either *occluders* (hiding other objects or receiving light, respectively) or *occludees* (hidden by other objects or lying in the shadow of other objects, resp.). The invisible space behind occluders (the shadow cast by them) is called their *shadow volume* or *umbra*. For convenience, let us assume in the following discussion that in order to become an occludee, an object must be entirely invisible, otherwise it is an occluder.

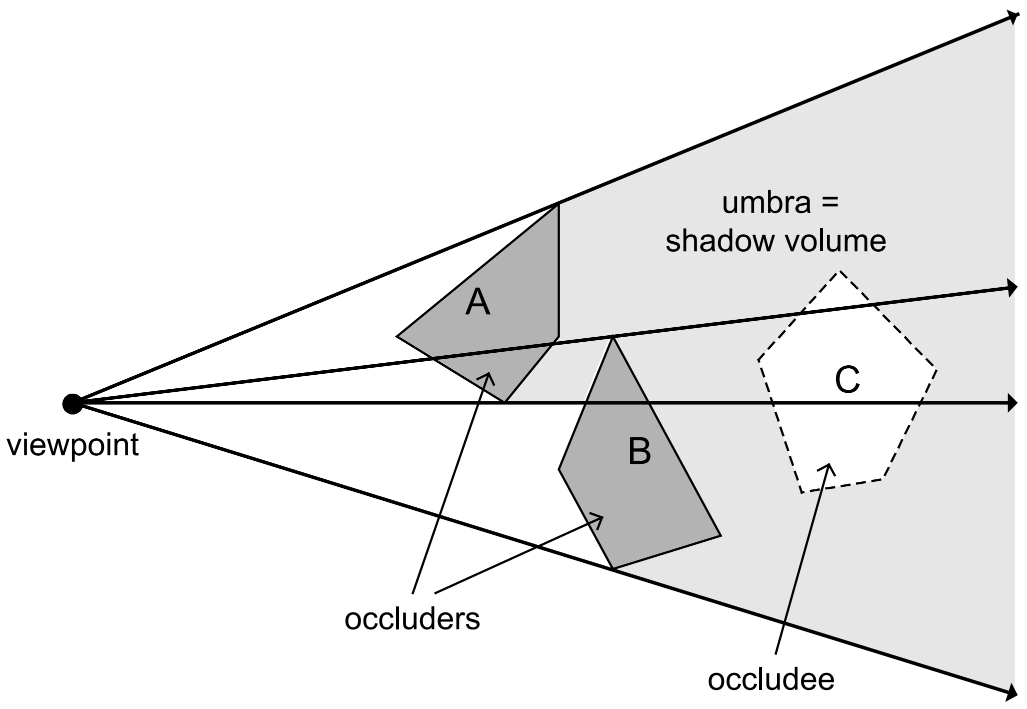


Fig. 2.3: Visibility from a point: The cumulative occlusion of A and B hides C.

An essential fact about visibility is that the *cumulative occlusion* of multiple objects can be far greater than the sum of what they are able to occlude separately. In order to take this fact into consideration, the union of all individual shadow volumes must be computed before testing if an object is within. Merging the umbrae of several occluders is referred to as *occluder fusion*. This is illustrated in Fig. 2.3: A and B are occluders while C is an occludee. However, neither A nor B can entirely occlude C on their own, thus C would be considered visible without occluder fusion. Therefore, dealing with cumulative occlusion properly emerges as important requirement for any occlusion culling approach.

From-cell approaches determine which objects are invisible from all points within a given cell. Analogous to point light sources, this is similar to computing the shadow of an area light source: In addition to the umbra, objects also have a *penumbra*, which is the space that is visible from some, yet not all parts of the cell. One distinguishes between *bounding planes* and *separating planes* being the border of the umbra and the penumbra region, respectively. For bounding planes of a certain occluder, the viewing region is located on the same side as the occluder itself, while it is on the other side for the separating planes. Visibility from a cell with a single occluder is illustrated in Fig. 2.4.

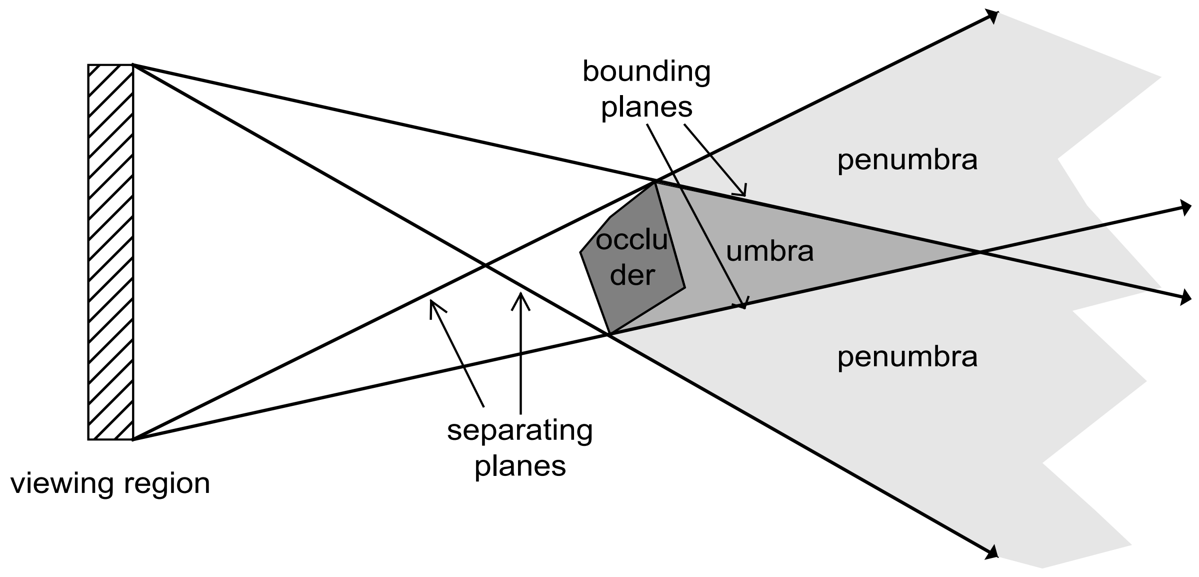


Fig. 2.4: Visibility from a cell: Only the umbra is invisible from all points within the viewing region.

Basically, those objects overlapping the penumbra must be considered visible, as they are not invisible within the whole viewing region. However, in contrast to visibility from a point, determining the cumulative occlusion of multiple objects is more than computing the union of the individual umbrae, but those regions must be added where the penumbrae merge to an umbra. Generally speaking, determining visibility from a region is more time consuming than from a point and is thus often done as a pre-processing step.

A 'natural' set of regions can be obtained on a global basis by partitioning the whole space in such a way that the classification for each object remains the same within each cell. The boundaries separating these regions are called *visual events* and characterize changes of the classification of one or multiple objects. This has been extensively investigated using so-called *aspect graphs* [Egge92, Dura97]. While being important for the study of visibility, a major problem is that they do not scale well to large scenes.

Another important observation is the fact that not all objects serve equally well as occluders: The occlusion power of little or loosely connected objects will on average tend to be smaller than that of large compact ones like houses. Therefore, many approaches employ a heuristic to calculate the occlusion potential of an object, based on attributes like the distance to the viewpoint, the area of the occluder and the angle between some average normal and the viewing direction. This process is referred to as *occluder selection* and is crucial especially where computational requirements severely restrict the set of objects which are taken into consideration for occlusion culling.

2.5.2 From-Point Approaches for Occlusion Culling

From-point approaches for occlusion culling are done at runtime for every new frame. The challenge is to design approaches that:

- Scale well to arbitrarily complex scenes.
- Are general in the way that they do not make assumptions about the scene.
- Always yield a conservative PVS that almost coincides with the EVS.
- Cause little overhead so that employing them leads to significant speed-ups where occlusion is present and does not affect the frame rate in cases of no occlusion.

This is an extremely complex issue. For instance, speed gains due to a tighter PVS are often foiled by a higher overhead and painfully often, 'improvements' turn out to have the contrary effect most of the time.

A generic occlusion culling algorithm explicitly considering accumulative occlusion was presented by Zhang [Zhan98] (see listing 2.1). It distinguishes between an occlusion representation O_R and a set of potential occluders P_O. All objects must be traversed in roughly front-to-back order to ensure that later objects take advantage of the occlusion of prior ones. If an object is found visible, it automatically becomes an occluder itself. However, since updating O_R is expensive, new occluders are first moved to P_O which is in turn used to update O_R if it exceeds a certain complexity. This is referred to as *multi-pass occlusion culling*. The rate at which O_R is updated by P_O is a significant criterion in which various approaches may differ: One extreme case is selecting a few large occluders in advance without performing any update at all. The other end of the spectrum is actually updating O_R with each newly found occluder (this is called *progressive occlusion culling*).

***Listing 2.1**: Generic occlusion culling algorithm after Zhang [Zhan98]:*

```
O_R = empty;
P_O = empty;

for each object g ∈ Scene
{
  if (isOccluded(g, O_R)) Skip(g);
  else {
    Render(g);
    Add(g, P_O);

    if (isComplexEnough(P_O)) {
      Update(O_R, P_O);
      P_O = empty;
    }
  }
}
```

From-point approaches can be categorized whether they operate in *object space* or in *image space*. Maintaining a proper representation of the fused umbrae of all occluders is generally more complicated in object space. Therefore, many approaches operating in object space select only a few objects as occluders, which are considered to contribute a high occlusion power due to some heuristics. All other objects are tested against the shadow volume defined this way. According approaches have been proposed by Coorg and Teller [Coor96], Bittner et al. [Bitt98] and Hudson et al. [Huds97].

In order to simplify the problem, some algorithms assume scenes to be *2½D*. This means that the depth complexity is never greater than 1 along some axis, or simply put: all occluders are connected to the ground (like buildings). This can be used to do occlusion culling in urban environments, where primarily buildings serve as source of occlusion (although arbitrary other three-dimensional objects may be contained in the scene as well). Downs et al. [Down01] present an according algorithm operating in object space that stores occlusion information as an *occlusion horizon*, which is a conservative approximation of a cross section through the occlusion shadow defined by a plane. The main idea is to sweep a plane parallel to the near plane away from the viewer, thereby creating an occlusion horizon as a piecewise constant function on the fly, which accumulates the occluding power of all occluders and can be used to cull objects conservatively. Fig. 2.5 demonstrates the principle of occlusion horizons.

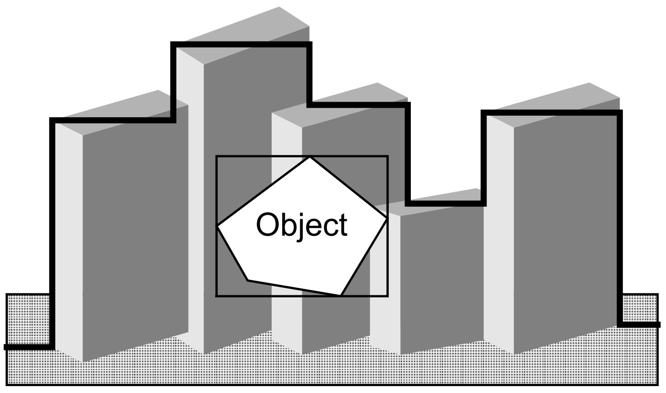


Fig. 2.5: Approximating occlusion with an occlusion horizon: The symbolized buildings are considered as occluders, while the black piecewise constant function is the approximated horizon. An object (like the depicted one) can be culled if it is entirely below this horizon.

Most approaches operating in *image space* attempt to approximate some aspects of the final image in one way or the other. Wonka and Schmalstieg [Wonk99] propose an algorithm working in image space that approximates the occlusion within urban environments: An orthographic projection is employed for drawing the fused shadows cast by some selected occluders when viewed from above. Since all occluders must be 2½D in this approach, the content of the Z-buffer gives at each pixel the height below which all objects are definitely hidden.

Rather conceptually important than practicable for actual implementations is the *hierarchical Z-buffer* (HZB) introduced by Greene et al. [Gree93]. It organizes the Z-buffer as an image pyramid, where the standard Z-buffer is its finest (highest-resolution) level. At all other levels, each z-value is the farthest in the corresponding 2 x 2 window of the adjacent finer level. Consequently, the topmost level contains a single value, which is the farthest value of the overall scene. Inserting an object is performed in a hierarchical fashion, starting at the coarsest level: If the nearest depth of an object is farther, the object must entirely be occluded and can be skipped. Otherwise, testing continues recursively down the HZB until an area is found to be occluded, or until the bottom level of the pyramid is reached. Updates to the buffer, on the other hand, must be propagated in a bottom-up fashion which can be much effort. Instead of implementing this in software (which would be rather pointless, since the Z-buffer is one of he final stages of the graphics pipeline, see chapter 2.3), Greene suggests modifying the hardware Z-buffer accordingly, which has also to some extend been actually realized – for instance the so called Hyper Z II technology in the Radeon 8500 GPU by ATI organizes the Z-buffer in a hierarchical fashion.

Another way of enabling hierarchical image space culling is the *hierarchical occlusion map* (HOM) proposed by Zhang [Zhan98]. It is characterized by the decomposition of visibility determination into a two-dimensional overlap test and a depth test. The former is realized by occlusion maps which are organized as an image pyramid to perform the test hierarchically. It basically serves to classify all objects as definitely visible which are not entirely inside the cumulative occlusion of all considered occluders. The occlusion maps are created by reading back the frame buffer after rendering some chosen occluders and repeatedly filtering the retrieved image. The depth test decides whether a potential occludee is actually behind the occluders and differs from other algorithms like the Z-buffer in so far as it does not by itself determine an the visibility of an object. Zhang proposes several possible realizations. One implements a software Z-buffer with a coarser resolution than the screen that stores the farthest Z-value for each region (unlike standard Z-buffers which store the nearest Z-value). For an object to be occluded, the rectangle of its projected bounding volume must pass both the overlap and the depth test. Furthermore, Aila and Miettingen propose the Umbra system [Aila00], which implements what they call *incremental occlusion maps* (IOM). Since they are created in software, the system does not depend on rapid read operations of the frame buffer. This approach combines several existing algorithms with new techniques and is currently considered state-of-the-art in occlusion culling.

Finally, *hardware occlusion queries* belong to the category of from-point approaches as well. However, since they are the core part of this master thesis, they are discussed separately in chapter 2.6.

2.5.3 From-Cell Approaches for Occlusion Culling

This kind of approaches attempts to solve the visibility problem for a whole region of possible viewing space. In order to classify an object as occluded, it must be occluded from all points within the region of interest. Due to reasons stated in chapter 2.5.1, this is even more complex than doing occlusion culling from a point and thus usually performed within a preprocessing step. At runtime, the pre-computed results are utilized, thus incurring a negligible overhead compared to from-point approaches. However, the drawback is that any modification to the geometry (at least the part considered for the visibility calculations) invalidates (at least some of) these results and requires a new computation, which can take a long time.

A well-known approach for pre-computing visibility within architectural models like buildings is *portal culling* (confusingly also often referred to as *PVS*) introduced by Airey [Aire90] and extended by Teller and Séquin [Tell91]. This approach is based on the observation that walls are the main source of occlusion in indoor scenes, while open spots within these walls like windows and doors, called *portals*, are the main reason why one room can be visible from another one. Consequently, the scene is partitioned, whereby cells correspond roughly to rooms and hallways of a building and portals connect adjacent rooms. One possibility is to obtain the visibility information from a preprocessing step which can be done either automatically or manually, and stored as an adjacency graph that tells which cells are potentially visible from a given cell and which are definitely occluded. At runtime, we have to locate the cell where the viewpoint is positioned and render all visible cells according to the graph (alternatively, more fine-grained techniques may be employed after having resolved visibility at this coarse level). Another possibility is to preprocess the scene only in order to partition it and to detect walls and portals, while the actual visibility is determined at runtime by diminishing the view frustum to fit closely around each portal. This way, the algorithm resembles view-frustum culling with several small view frustums. This is illustrated in Fig. 2.6.

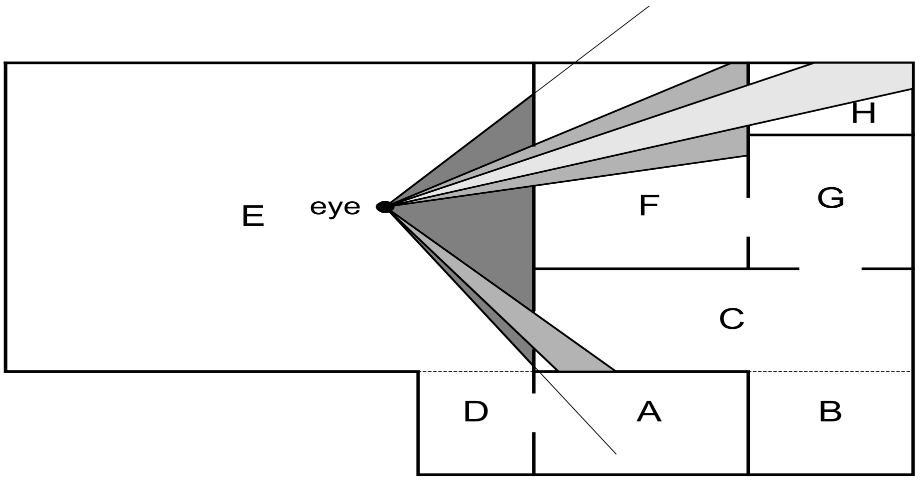


Fig. 2.6 : Portal culling by diminishing the view frustum (after [Möll02]).

As reasoned above, calculating visibility from a point is in general less complex than visibility from a cell. Wonka et al.[Wonk00] present a technique called *occluder shrinking* that permits to compute visibility valid for a whole cell using an algorithm that determines visibility from a point. The central idea is as follows: If visibility can be computed for a certain point *P*, then a conservative classification of a sphere around *P* with radius *R* can be obtained if all occluders are shrunk by the same amount *R*. The shrinking operation can be realized in a comparatively simple way within a 2½D setting as presumed by the approach, or if regular volumetric data structures (like octrees) are used to store the occluders. Since the occlusion power diminishes fast with increasing values for R, he extends this approach for bigger view cells by point sampling the cell space in a fashion that many spheres cover the boundary of the view cell. The visibility from the cell is then given by the union of the PVSs of all sampling spheres and can be stored with the cell.

Koltun et al. [Kolt00] introduce the term '*virtual occluder*', being a simple convex object used to replace the fusion of several much more complex occluders.

Another approach operating from a cell using shafts is presented by Schaufler et al. [Scha00]. View cells and occluders are axis-aligned bounding boxes and the occlusion of the scene is stored as an octree, where each leaf is classified as opaque (e.g. fully inside a definitely invisible space), boundary or empty. To begin with, the algorithm attempts to find an opaque leaf node. This is then extended as much as possible by combining adjacent opaque nodes. Afterwards, the occluder is extended by its own shadow volume, i.e. leaf nodes in hidden space are considered opaque regardless of their actual classification. A semi-infinite shaft is constructed from the view cell to the occluder, where all objects inside can be culled.

Durand et al. [Dura00] introduce *extended projections*, which are based on a similar algorithm as the hierarchical occlusion maps, as described above, but modify the way how occluders and occludees are projected onto the occlusion map: Durand et al. define how the projection of occluders and occludees should look like so that an image-space algorithm can be applied to occlusion culling from a region.

While all approaches so far have operated either in image space or in object space, Bittner et al. [Bitt01a] introduce a powerful algorithm working in *ray space* (or line space), being a dual space where each possible ray within object space corresponds to a certain point. In ray space, an occluder can be seen as a polygon that describes which rays are occluded. The algorithm calculates a conservative PVS for a view cell within a 2½ D scene by subdividing the two-dimensional ray space into areas of 'similar' visibility: Several occluder polygons in ray space are merged using BSP trees. Bittner also presents an according visibility test that provides an exact analytic solution to from-region visibility in 2½D scenes.

Wonka et al. propose an approach called *Instant Visibility* [Wonk01], which is characterized by computing visibility on-the-fly on a distinct visibility server in parallel to the rendering pipeline executed on the actual display host. The visibility server might communicate with the display host over a network. The idea is that the display host continues rendering new frames while the visibility server calculates visibility for future frames, thus seeking to eliminate any latency. The validity of the results is ensured by restricting the movements and rotations of the viewpoint to some extent which permits the application of *occluder shrinking* (as described above) with a certain radius.

2.6 Hardware Occlusion Queries

So far, basically two different categories of occlusion-culling approaches have been presented: Techniques determining visibility from a cell and from a point and rely primarily on the CPU. Both suffer from shortcomings, though: While the former are generally quite inflexible concerning any modifications of the scene due to the pre-processing, the latter usually mean a significant overhead for the CPU that sometimes even exceeds the benefits – especially if the bottleneck already lies on the CPU.

Now, the idea behind hardware occlusion queries is to exploit the graphics hardware not only for the rendering itself, but also for supporting occlusion culling, intending to make use of the incredible speed of the rasterizer unit. Besides, the load for occlusion culling might be partitioned more equally this way. As mentioned in chapter 2.5.2, Greene proposed the *hierarchical Z-buffer* [Gree93], which is supposed to be implemented in hardware. There has been much discussion relating to this approach, for instance by Hong [Hong97], and it has to some degree also been applied within recent commercial products. However, although the HZB is relevant, because it is a way to employ special hardware support for more sophisticated visibility algorithms than the ordinary Z-buffer, it differs substantially from the kind of occlusion queries this work is based on.

Bartz et al. [Bart98] propose an OpenGL extension for occlusion queries along with a discussion concerning a potential realization in hardware, that comes in some aspects remarkably close to actual implementations. A first concrete realization of occlusion queries was implemented in 1998 by Hewlett Packard in the VISUALIZE fx graphics hardware [Scot98]. It is usually referred to as HP-occlusion test, HP-occlusion bit and similar terms. The according OpenGL extension is titled *HP_occlusion_test*. This extension is the direct predecessor of the *NV_occlusion_query* extension (often simply called *NV query*) introduced by NVidia with the Geforce3 graphics accelerator. It is also accessible via an OpenGL extension and is the topic of this master thesis. Therefore, these extensions are described, discussed and compared to each other in detail in the next three chapters.

Apart from these extensions offering hardware functionality to the application, there are purely hardware based methods like ATI's Hyper-Z and NVidia's Z-Cull. These are meant to reduce demands on fill, but can not decrease the traffic on the graphics bus.

Instant visibility [Wonk01], as described in chapter 2.5.3, is relevant to occlusion queries in so far, as it also has to deal with aspects of parallelism: The display hosts continues rendering new frames while the visibility server calculates visibility for future frames. However, since a different hardware is employed for the visibility computations and the actual rendering, it differs essentially from hardware occlusion where both is done by the same GPU.

2.6.1 Description

Hardware occlusion queries operate in image space. Although it would eventually be possible to use them for preprocessing purposes of from-cell approaches as well, all algorithms employing them so far (including those proposed by this master thesis) determine visibility from a point. Their intention is to determine whether a certain object is occluded and thus they can be seen as one possible realization of the '`isOccluded`' function in the pseudo code of the generic algorithm presented in listing 2.1 (chapter 2.5.2).

The principle of hardware occlusion queries follows a simple pattern: In order to find out whether a certain geometry is visible, a conservatively simplified version (usually a bounding box) of it is sent to the graphics hardware in a special mode that does not affect any buffer and thus generates no visual results, but considers visible pixels by setting a flag (HP test) or incrementing a counter (NV query). Afterwards, the application can query if anything of that geometry would have been visible (which means if any fragment has passed both the depth test and the stencil test). If nothing had been visible, the actual geometry can safely be culled, otherwise the original geometry must be rendered as usual. In order to minimize the effort for the tests, every feature (like lighting, texturing,…) should be turned off as it has obviously no impact on the result but may reduce performance.

The advantage is that between issuing an occlusion test and querying the result (which can take a considerable amount of time), the CPU is free to do other tasks – in contrast to purely CPU-based occlusion culling. However, asking for the result too early (before it is actually available) will stall the CPU, as discussed in the next chapter. Therefore, applications employing hardware occlusion queries must be designed accordingly in order to make use of this newly gained parallelism – simply replacing CPU-based tests (for instance relying on software buffers like the *hierarchical occlusion map*, see section 2.5.2) by hardware queries would probably work, yet is unlikely to be a suitable design.

A nice aspect of hardware occlusion queries is that they do not require a fundamental modification in the architecture of existing graphics hardware: The only difference to normal rendering is that the write-enable signal of the buffers must be caught and evaluated differently, while the rest of the graphics pipeline stays exactly the same. Since modern GPUs perform tasks like transforming vertices and rasterizing triangles much faster than any CPU, this explains why the graphics hardware is actually adept for testing the visibility of a certain geometry.

As already mentioned, this new functionality is offered to the application by extending existing graphics APIs like OpenGL (in fact, OpenGL is up till now the only possibility to make use of these features, which is due to its flexible extension mechanism). The specification of both HP_occlusion_test and NV_occlusion_query can be found in [Crai02]. Since the HP test has been released earlier, some approaches utilizing it have been published, e.g. [Meiß99, Stan02], while there has been very little work tailored towards the NV query. Recently, Salomon et al. [Salo02] have proposed an algorithm which employs the NV query: They subdivide a scene using a uniform grid. Then the cubes are traversed in slabs roughly perpendicular to the viewer. Tests are issued for all cubes of a slab at once, before any result is fetched and the visible geometry of this slab is rendered. Another implementation of them uses nested grids: Instead of containing geometry, a cell contains further cubes that are traversed in the same way if the overall cube is proven visible. In the highly complex model (a power plant consisting of more than 12 million triangles) used for measurements, they are able to achieve speed ups of four to ten times.

2.6.2 Benefits and Drawbacks

Employing hardware occlusion queries has pros and cons. The main advantages are as follows:

- As pointed out above, occlusion queries hardly differ from normal rendering, thus they take full advantage of high fill rates of modern GPUs. At the same time, the CPU is free to do arbitrary other tasks as long as they do not depend on the result of the query. Therefore, if applied properly, significant performance increases are possible by stressing the aspect of parallelism.
- They are very generally applicable and do not depend on any specific type of scene or geometry. Furthermore, the result of NV Queries can also be used for other purposes than occlusion culling, for instance selecting a LOD-level.
- The usage is straightforward and almost identical to rendering as usual (note that this only refers to the way how a query is issued, it does definitely not apply to the overall design of the algorithm).

However, hardware occlusion queries have disadvantages compared to purely CPU-based approaches as well:

- The latency between issuing a test and being able to ask for its result without stalling the CPU is significant.
- Measurements have shown that it takes a considerabe amount of CPU time within the driver to issue an occlusion query – independent of the state of the command buffer, the actual rasterization effort or anything else. This prohibits an indiscriminate application.
- Typically even more fragments must be rasterized for a conservative bounding volume – hence for an occlusion query – than for the geometry itself.
- The functionality is still vendor specific, although the NV_occlusion_query extension is – apart from recent graphics cards by NVidia, creator of the extension – meanwhile also supported by graphics cards from ATI. For this reason, commercial software like games should not completely rely on the availability of respective extensions, but should be prepared to do occlusion culling differently, if the extension is not available.
- OpenGL is the only API supporting the functionality through extensions. There is no way to use it for instance within current versions of DirectX.

The first point needs further discussion: Generally speaking, the term latency refers to the time between making a query and receiving its result. In the concrete case of hardware occlusion queries, the latency basically comprises the time for processing the geometry itself – with an effort being approximately linear to the number of pixels being rasterized, as shown in [Stan02] – as well as accomplishing all tasks waiting in the command buffer at the time the query is issued. This has a serious impact on the overall load balancing within an application: Typical applications have parts where they commit more drawing jobs than the GPU can handle in the same time (procedures dealing with rendering the scene), but these parts are usually followed by purely CPU-based tasks (like calculating the AI of game characters) where the GPU can finish its work meanwhile. However, this principle is only valid when the CPU does not depend on any information from the GPU for further rendering. With hardware occlusion queries, the GPU finishes more or less at the same time when the CPU leaves the part for rendering the scene and is usually idle for the rest of the frame. In order to outweigh this flaw, a scene must have a considerable amount of occlusion.

As pointed out, the CPU is free for other tasks while the occlusion query is being executed. However, this is based on the assumption that there actually is enough meaningful work. This is often not the case, because the only reasonable task would be to render further objects which is not feasible as this would require the result of the current test. While asking for a result is always possible – even immediately after issuing a test – it incurs stalling the CPU (which means doing nothing but waiting) if the test has not been finished yet and this in turn foils the parallelism which is the main reason for hardware occlusion queries. Consequently, care must be taken when designing according approaches, and reducing the work for the CPU will not always reflect in better performance as it often simply increases the time wasted by stalls.

Summarizing, hardware occlusion queries are definitely not for free. Therefore, rendering the original geometry must be much more expensive than performing an occlusion test in order to justify this action. Besides, the bounding volumes should be as tight as possible to avoid a significant overestimation of visibility: It must not be forgotten that visible objects require the effort of both the occlusion test as well as the normal rendering and are thus inherently slowed down by any attempt of occlusion culling.

2.6.3 Comparison HP_occlusion_test – NV_occlusion_query

Although the HP test is the direct predecessor of the NV query, they differ in the details of the functionality which has a tremendous impact on the design of potential algorithms employing them.

The first distinction is the exact result: The HP test returns a simple *yes* or *no* (*yes* means that at least one pixel of the geometry used for testing actually passed all tests), while the NV query provides the precise number of visible pixels. This latter version permits a much greater flexibility in the evaluation: For instance, (as we will do in chapter 3.4.2.1) it enables compromising quality with speed (a certain threshold can classify objects as invisible where only very few pixels have passed the test). Furthermore, the LOD-selection may be based on this result. Craighead [Crai02] proposes using it for determining the contribution of light sources which can in turn be used to adapt the brightness of lens flare effects accordingly. The only potential advantage of the HP test is that, unlike with NV queries, the hardware can theoretically return the result as soon as the first visible pixel is found which could save some rasterization effort. However, I have no information if this is actually considered by the implementation.

A further disadvantage of the HP test is that the extension gives us no hint when the query has actually been accomplished. Since the latency includes the effort for rendering all geometry currently waiting in the command buffer – and the application has no idea how much this is – it is impossible to predict how long a query might take. This flaw has been tackled with NV queries: This extension essentially contains a mechanism similar to the *NV_fence* extension which permits to ask whether the GPU has already executed a certain job – without stalling the CPU if not (`glFinish()` guarantees that all work committed before is complete afterwards but it is no option for it causes the same stalls). Therefore, we can quickly (measurements have shown that the effort for this is negligible and thus it can be done as often as needed) ask if any new results are available (without actually retrieving them if they are) and base the further execution on this information.

Finally and probably most important for the design of sufficiently parallel applications is the fact that multiple NV queries can be issued before asking for the result of any one (arriving in the same order as they have been sent), while only one HP test is allowed at a time. As argued in subsequent chapters, this latter restriction permits only one kind of algorithm, which incurs the full latency each time. On the other hand, issuing multiple NV queries which are independent of each other is crucial for the design of algorithms that are actually capable to exploit the full parallelism, which is the main reason for hardware occlusion queries.

	HP_occlusion_test	**NV_occlusion_query**
Result	yes/no	Number of visible pixels
Completion	No information	Can be checked
Parallel queries	No, only one at a time	Yes

Table 2.1: The main differences between HP tests and NV queries.

Concluding, the NV_occlusion_query extension is superior in all respects and permits much greater flexibility in the design of algorithms, therefore, all research done in the course of this master thesis has been modelled after this very extension. However, since the approaches presented in this master thesis are designed to be applicable to any future extensions and implementations in other APIs, all subsequent chapters use the general term ‘occlusion query’, instead of calling them explicitly ‘NV query’.

3 Occlusion-Graph Based Approach

As it was outlined in the previous chapter, an efficient approach using hardware-accelerated occlusion queries must take several factors into account that are of little or no importance in pure CPU-based occlusion tests. This chapter introduces a way of modeling approximated occlusion within a scene from a given viewpoint, being the basis for the development of an algorithm that is designed to meet the requirements of occlusion queries. After describing the core algorithm, it points out imperfections and presents several improvements to overcome some of them.

3.1 Motivations and Considerations

When designing an algorithm for the efficient application of occlusion queries, one of the main issues that must be dealt with is the order in which individual parts of the overall geometry that make up the scene can be tested and rendered. This order should meet multiple criteria:

- The loss of occlusion, which happens when actually occluded objects are tested before their occluders have been rendered, should be minimized or – in the best case – ruled out completely. It is worth mentioning explicitly that it does not suffice when occlusion queries have already been issued for these occluders – they must have been actually drawn (this is comprehensible when keeping in mind that occlusion queries never affect any buffer).
- The penalty incurred by CPU stalls due to requesting occlusion-query results that are not yet present ought to be minimized as well – this is intuitively achieved by maximizing the time span of each object between issuing the test and asking for its result and this in turn means that the order should seek to issue tests as soon as possible.
- Determining the order should not be too computationally expensive. Otherwise, potential benefits are likely to be foiled by the additional costs.

Since it is a basic property of occlusion that (opaque) objects next to the viewer tend to occlude objects which are farther away, the order should sort the objects (at least roughly) by their distance from the viewpoint, starting with the closest ones. However, as two different actions – testing and rendering – are done for each object at different times, this still leaves an abundance of possibilities, which are shown by means of two extreme cases, both ordering the objects in a strict front-to-back way. Formally, assume a scene S containing n objects is specified for a certain viewpoint as follows:

$$S = \langle O_1, O_2, .. O_n \rangle, \text{ with } dist(O_i) \geq dist(O_j) \text{ for } i,j \in [1,..,n] \text{ and } j \leq i$$

$dist(O_i)$ gives the distance of the i^{th} object to the current viewpoint (this implies that the order can change whenever the viewpoint moves).

One possible (extreme) sequence is to issue the occlusion query for the i^{th} object after rendering all objects with indices smaller than i, then wait for the result and render it if the test proved the object visible. The advantage is the guarantee that no occlusion will be lost and that the overhead for determining the order is rather moderate; However, concerning CPU stalls, no other approach could be worse since the CPU is idle for the complete duration of an occlusion test and this applies to all objects of the scene, hence we incur the full latency. Despite of all inconveniences, this algorithm has practical relevance indeed and is from now on referred to as *stop-and-wait approach*. Since the HP_occlusion_test does not support multiple queries to be issued in parallel, every application using the HP_occlusion_test more or less boils down to implementing this algorithm.

Another possibility is to approximate the correct visibility situation very roughly by dividing the scene into two parts: As mentioned in chapter 2.5.1, certain objects can be assumed to be occluders due to some heuristic, while the rest is considered to be occludees. The first set would then be rendered without test. Afterwards, tests are issued for all objects of the second set before querying their results in the same order the tests were issued, and immediately rendering eventually visible objects. In the context of occluder selection, the distance is a commonly used criterion: The first m objects ($m \leq n$) of S are classified as occluders regardless of their actual visibility.

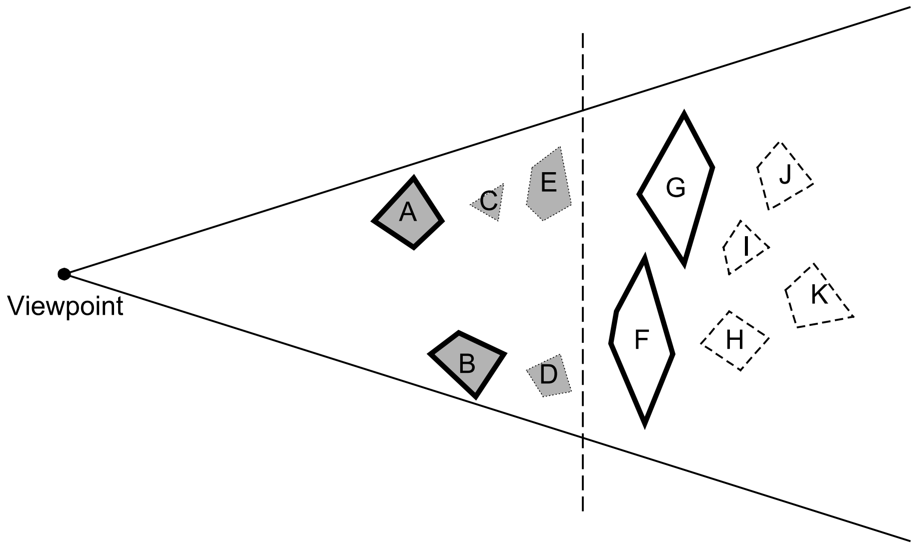


Fig. 3.1: An example for a scene where the occlusion is very badly estimated by employing a distance-based criterion. Object with bold outlines are actually visible, dotted outlines indicate occlusion. Filled objects (to the left of the vertical line) are assumed to be occluders and are thus rendered whereas empty ones (to the right) are thought to be occludees.

This approach will stall the CPU much less than the first one, yet if the geometry is not evenly distributed within S, it is likely to lose a lot of occlusion – thus wasting rendering time – in both parts: Even near objects (assumed to be occluders) can actually be occluded while even distant objects (thought to be occludees) can be visible and in turn would occlude objects that are even farther away. An example is given in Fig. 3.1: The objects A and B are the only ones which are classified correctly, while C, D and E are rendered in vain. The situation is even worse to the right of the vertical line indicating the border between those objects considered as visible due to their proximity and those assumed to be occluded: Because all objects (F to K) will be tested before any one gets actually rendered, the test will classify all objects as visible, although this is in fact merely correct for F and G. Note that the constellation presented in Fig. 3.1 is by all means of practical relevance: For instance when looking along a street, buildings to the left and to the right will be quite close to the viewer (corresponding to the objects A to E), whereas the centre will permit free sight to distant objects (F to K).

The two approaches can be combined by breaking down the scene into more than two slices (as the second order does), but still gathering more than one object per segment (as in the first approach). Each set of objects is then tested against all geometry in prior parts, but neglects occlusion within itself. This might yield satisfactory results if the size of the individual slices is appropriately chosen - an issue being closely related to the discussion if occlusion culling for a scene should be done in a progressive or multi-pass way, as elucidated in [Zang98] and also mentioned in chapter 2.5.

However, major shortcomings remain: On one hand, the occlusion of objects within one part (independent of its size) is assumed to depend on all objects in all prior parts, which is a massive overestimation. Consider once more Fig. 3.1: Object D for example can be identified as occluded as soon as object B is rendered and is in fact completely independent of the time when the objects A and C get rendered, although both are closer to the viewer. On the other hand, members of the same segment actually can occlude each other, as shown for the objects F to K. Assuming them as independent is an underestimation of occlusion that might cause unnecessary rendering.

Another imperfection might be less obvious and is specific for hardware occlusion queries: Once the next step would be to fetch the result of a test, there is no alternative work to do and one is forced to wait. As described in the previous chapter, the NV_occlusion_query permits to ask in almost no time if a result is already present. Consequently, it would be beneficial if the tasks at hand could be rearranged in order to continue with other meaningful work in the meantime and postpone the actual query of the test result until it can be achieved without penalty.

The perfect solution would be to know for each object which other objects contribute to its occlusion and to which extent. Answering this question accurately would mean solving the visibility problem itself (this is essentially the *EVS*, as introduced in chapter 2.5) and no further tests would be needed at all, but doing so is costly which contradicts the third criterion as stated above. Hence, what we attempt to find is an order for testing and rendering the various objects that minimizes the loss of occlusion – unlike heuristic occluder selection as described above –, and rules out the false classification of visible objects (thus it should be conservative). Furthermore, it must be much cheaper to determine than the EVS and allow interleaving the classification of many objects – unlike the stop-and-wait approach.

3.2 The Occlusion Graph

After having outlined some criteria a rendering order should meet, this chapter introduces a graph which will turn out to be suitable for guiding the interleaved process of testing and rendering.

3.2.1 Description and Properties

One approach proposed by this master theses seeks to find out for each object which other objects may contribute to its occlusion. The calculations are not done for the actual geometry, but for a very rough approximation which both greatly simplifies the computations and leads to convenient properties that even the exact solution does not show.

Basically, what we need is a set of constraints of the form:

$$A \to B$$

which means that A must be rendered before B can be tested in order to avoid any loss of occlusion. This induces a certain relationship R between the objects of a scene from a given viewpoint: $R(A, B)$ – or written differently as $A \to B$ – is valid if and only if the approximation of A directly contributes to the complete occlusion of the approximation of B. Aspects worth emphasizing are:

- *A* must contribute *directly* to the occlusion of *B*: It is not sufficient for *R(A, B)*, if *A* occludes an arbitrary object *X* which in turn directly occludes *B*. For this reason, *R* is not transitive.
- *B* must be *entirely occluded* to occur in *R* as an occludee. This need not be done by *A* alone: *R(A, B)* is also valid if *A* only partially covers *B* directly, as long as the uncovered parts of *B* are occluded by some other objects.
- Apart from being *non-transitive*, *R* is definitely *non-reflexive* (*R(X, X)* can never occur as no object can entirely occlude itself) and must be *non-symmetric* (if *R(X, Y)* is valid, *R(Y, X)* must never occur). Being non-symmetric is not fulfilled for arbitrary shapes; therefore, all objects must be approximated in a way when determining *R* that rules out symmetric cases.
- *R* is *unambiguous*, provided that all objects are approximated in the manner demanded to ensure the property of non-symmetry.

R can be visualized as a directed acyclic graph, summing up all constraints within a scene from a given viewpoint (and adding isolated objects being not constrained in any way). Nodes correspond to objects and edges represent the constraints that are part of *R*. This graph is named *Occlusion Graph* (from now on abbreviated as *OG*) and is illustrated in Fig. 3.2:

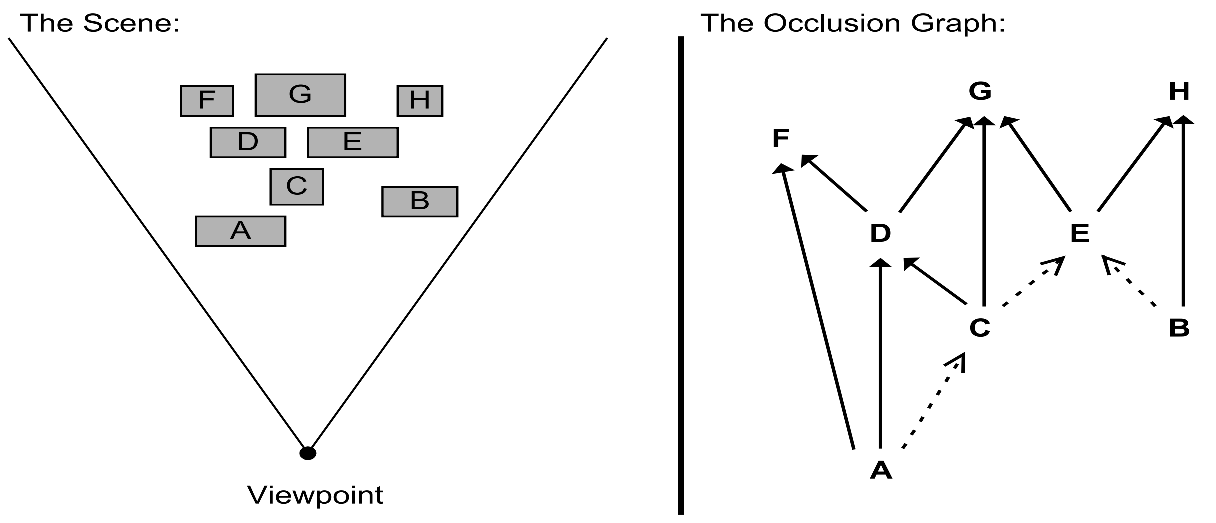


Fig. 3.2: A scene consisting of eight object approximations and the according Occlusion Graph. Note that the dotted arrows lead to partially uncovered objects and are thus not part of the actual OG.

Even more precise would be the name ‘Potential Occlusion Graph’: Some constraints may be due to the usage of conservatively overestimated geometry within the construction and could be removed for the original geometry. Some remarks may help to clarify things:

- The reason why a direct contribution of the occlusion is required for *R* is that this greatly simplifies the construction of the OG (as subsequent chapters will explain). Besides, if $A \rightarrow B$ and $B \rightarrow C$, *C* is implicitly constrained by *A* anyway, even if $A \rightarrow C$ is not explicitly contained in the OG – thus we save many abundant constraints this way. On the other hand, relationships like $A \rightarrow F$ in Fig. 3.2 (which is also implicitly given via *D*) can not be removed, since improvements introduced later assume each object to be aware of all other objects it occludes.
- A complete coverage is required for any potential occludee because objects where the approximation is not entirely hidden by other objects must be considered visible and

thus rendered anyway. Therefore, constraining the moment when they are rendered makes no sense. The dotted arrows in Fig. 3.2 are present just in order to illustrate this explicitly once, and will consequently be omitted from now on, as they do not belong to the actual OG.

- It is worth realizing how cumulative occlusion (i.e., occluder fusion) is explicitly modelled – an important condition as discussed by [Zang98] and stated in chapter 2.5. For all nodes being pointed to by more than one arrow, multiple objects are necessary for complete occlusion. Consequently, all predecessors must be rendered before the object itself can be tested.
- Finally, one might wonder why arrows originating from non-root nodes (nodes being constrained themselves) are of any relevance at all: If the according objects are invisible, they do not get rendered and thus induce no constraint. However, since different geometries are used for rendering (namely the original ones) and testing (approximations), such objects have in fact good chances to be visible and can therefore occlude others. Generally speaking, when establishing the constraints for an arbitrary node, it is treated as if it was a starting node, no matter how much it might actually depend on others.

3.2.2 Creation

As the meaning and the basic properties of the OG have been pointed out now, let us turn to the way how this graph gets constructed: According to the fact that we are about to design a occlusion-culling approach from a point and thus the OG is only valid for one specific pair of position and viewing direction, this construction has to be done each frame. Therefore it must be so fast that frame rates of 60fps and more are still feasible (after all, occlusion culling is supposed to speed things up, not to slow them down). Furthermore, all objects must have been correctly transformed to their actual position and orientation in world space before approximating them. Because the implementation is integrated into a scene-graph based environment, putting all objects to the intended places means traversing the scene graph before starting to create the OG. Another important step preceding the actual construction of the OG is view-frustum culling, which should be listed separately, but comes in our case as a by-product of the scene-graph traversal.

The construction is done entirely by the CPU, which implies that nothing can get tested or rendered before the OG is complete – hence the GPU is idle at that time (at least in the basic approach without improvements). As another consequence, the overall performance depends to a great extent on the CPU speed – a slow CPU will decrease the frame rate drastically even if the GPU is very fast.

The basic idea is to project a simplified version of each object to screen space. There, the objects get sorted by their distance to the viewpoint and drawn to a software buffer with a much coarser resolution than the actual output device. This software buffer always contains the index of the object with the greatest distance at each point. Collecting these indices within the area occupied by an object before adding the object itself yields its predecessors within the OG.

In more detail, the algorithm for creating the OG can be outlined as follows:

- The *axis-aligned bounding boxes* (being the chosen conservative approximation) of all objects passing view-frustum culling are projected to *screen space* in the very same way as this transformation will be done by the GPU for the actual geometry. This yields a set of two-dimensional points.

- *Screen-space bounding boxes* (in the following text abbreviated with *SSBB*; a SSBB is essentially the smallest possible rectangle containing the entire projection of an object to screen space) are constructed out of these points overestimating the area where the regarded objects will be drawn.
- The smallest distance of the respective approximated geometry from the near plane of the camera is assigned as depth value to these rectangles.
- Strictly sorted by this depth value, all rectangles are drawn into a *software buffer* with a much lower resolution than the screen, hereby roughly approximating the actual scene.
- Before overwriting other rectangles within the software buffer, the algorithm collects the indices of the objects associated with those rectangles. These objects will become predecessors of the currently processed object in the OG.

Before explaining the involved steps in more detail, the choice of approximation for the objects (i.e., the type of bounding volumes) shall be reasoned: As mentioned, all objects are approximated by *axis-aligned bounding boxes* (from now on abbreviated as *AABBs*) for they seemed to be a good compromise between simplicity and an excess of conservative behaviour. While *oriented bounding boxes* usually approximate the original objects more accurately, they would make several algorithms more complicated and thus slower. *Bounding spheres* on the other hand seem to offer a reasonable complexity for all needed purposes, but they tend to overestimate the geometry too much – especially when considering that they get simplified by SSBBs themselves. For more information about bounding volumes, refer to [Möll02].

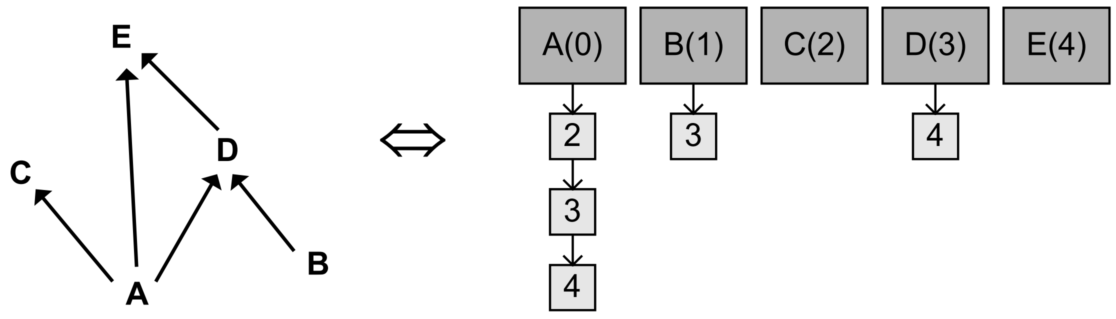


Fig. 3.3: A simple occlusion graph and its internal representation as adjacency list. The numbers in brackets express that each object is identified by its index within the array

Finally, it must be stated that the OG is internally stored as an adjacency list, being a well known data structure suitable for the representation of graphs of all kinds: Each node corresponds to an instance of a certain data structure. These instances are kept within an array so that the index suffices to uniquely identify a node. Apart from other members like the represented geometry, this data structure contains a list of indices of the nodes the according geometry might occlude – basically the 'arrows' in the OG originating from the respective node. Fig. 3.3 demonstrates the principle of adjacency lists by comparing a small OG to the resulting adjacency list.

3.2.2.1 Projection of AABBs to Screen Space

The following description is based on the assumption that an arbitrary number of objects is given and that each object permits easy access to its AABB given in world space. The first issue that must be dealt with is how a SSBB can be computed from an AABB. Basically, this involves two steps:

- Transforming of all eight vertices to screen space.
- Determining the smallest rectangle that contains all transformed vertices.

The first step requires that we emulate exactly how each vertex would be transformed if it were defined at the very same place within the code using the OpenGL command `glVertex()`. Briefly summarizing, three matrices usually participate in the transformation: The *modelling* matrix transforms an object from its local coordinate system in which it was loaded to a global one (the already mentioned *world space*). The *viewing* matrix arranges this world space in order to simulate an arbitrarily position-able camera, and the *projection* matrix finally calculates the position of each vertex within a (normalized) two-dimensional coordinate system with x and y values ranging from –1 to +1, known as screen space. Many books about computer graphics cover this topic much more extensively, among them [Woo99], [Hear94] and [Watt93]. In OpenGL, the first two matrices are combined to a single matrix consequently named *modelview* matrix. This one and the projection matrix can be queried in OpenGL and combined, resulting in one overall matrix (denoted as *M*) which transforms vertices directly from their local space to screen space. However, while the projection matrix and the viewing matrix will typically remain unchanged throughout the whole frame, the modelling matrix might vary with each object, so that *M* would have to be re-computed for each object. But as we request the AABBs to be already given in world space, the same *M* can be applied to the AABBs of all objects and will successfully transform the vertices to two-dimensional points in screen space.

Nonetheless the implementation of this projection is more complex than simply multiplying 8 points with *M* and computing their bounding rectangle: Efficiency can be increased by exploiting the fact that we are dealing with AABBs (as described in appendix A of [Zang98]), and obtaining correct results requires clipping against the near plane of the camera.

Finally, computing the bounding rectangle (the SSBB) is straightforward: Simply the minimum and the maximum of the x and y coordinates of the transformations of all considered points (which can actually be more than eight due to clipping) have to be found. By employing SSBBs for the construction of the OG, which have assigned a single depth value for the whole occupied area, it is ensured that symmetric relationships between two objects will never occur as requested in chapter 3.2.1.

3.2.2.2 Inverse Z-Buffer

What we have done so far is conservatively overestimating the place where a given object will appear on the screen (provided that is located within the view frustum). This approximation is a rectangle in screen space and we assume the same depth value for its whole area (namely the distance of the closest vertex to the near plane). The next step is to establish relationships between these objects as described above. The algorithm used to accomplish this task can be outlined as follows:

- Assign a unique index to each object.
- Ensure that all objects are dealt with in front-to-back order.
- Scale the coordinates of the rectangle to the resolution of a two-dimensional software buffer.
- Collect all indices encountered within the area enclosed by the rectangle. If still empty space is covered as well, the respective object is considered visible and no constraints are added to the OG for this object.
- Write the index of the object to the respective rectangle within the software buffer.

The first step is already done when preparing the 'bare' adjacency list (without any relationships) by setting up the data structure.

The second step means sorting the objects by their distance (which should have been computed by then) in front-to-back order. The precise algorithm used to achieve this is of no significance as long as it is fast (I first used a `multimap` of the Standard Template Library and later replaced this by a quicksort due to speed reasons).

For the next step, the already mentioned software buffer must be introduced: Sorted by the depth, the SSBB of each object is written to its appropriate place there after collecting the indices of all encountered objects. '*Inverse Z-Buffer*' (from now on abbreviated as *IZB*) seems to be an appropriate name since the way it works shows much resemblance to a usual Z-buffer, except for the fact that points get overwritten by objects behind them instead of ones lying in front of them. Another way to think about it is to imagine a view from infinitely far behind with inverted viewing direction. However, unlike a normal Z-buffer coping with arbitrarily shaped objects, we do not need to check if an object is behind another – not even on a per object basis – since owing to the fact that only flat rectangles are inserted with increasing distance, it is for certain that each new object will have a greater depth value on each point than all objects that have previously been there.

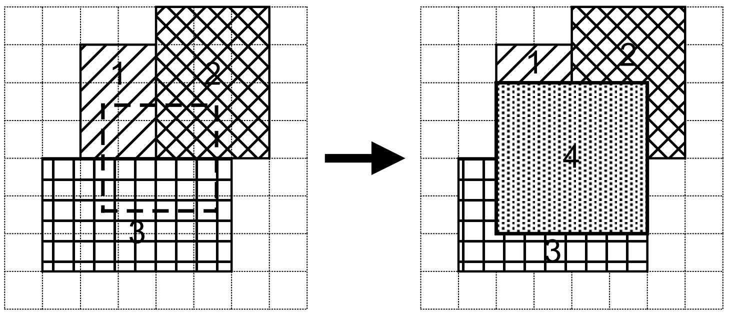


Fig. 3.4: A new rectangle gets added to the inverse Z-buffer.

Before scan-converting one rectangle with corner coordinates ranging from –1 to +1 after the transformation, it must be scaled and aligned to match the resolution of the IZB which is much coarser than the actual output target (a discussion about the resolution is given below). Note that the rectangle is enlarged in order to snap to the grid given by the resolution. Afterwards, the rectangle is scan-converted (a trivial task with SSBBs and another reason why this kind of representation has been chosen). The index of each covered point is added to the indices of the predecessors (unless it is already there) before being replaced by the own index. This is illustrated in Fig. 3.4 which shows an IZB containing three rectangles and a fourth one to be added. The left image depicts the previous situation and outlines the new rectangle after scaling but before snapping to the grid. The right image demonstrates the situation after the update: large parts of the objects 1, 2 and 3 have been overwritten and – since no empty space was met – they become the predecessors of the new object 4.

It is worth emphasizing that especially the steps done for each point are extremely time-critical. Assuming an average number of 100 IZB-pixels per object and 1000 objects per frame (actual numbers vary of course and depend on many factors, but these assumptions are by all means realistic for a scene of average complexity), the according code has to be executed 6 million times per second in order to achieve a frame rate of 60fps. Thus it is crucial that the implementation is designed to meet the goal of high performance. The most effort is needed to gather all found indices, but to count each just once. My implementation uses a very simple hash table with a hash function masking the n least significant bits (a simple logical AND). Besides, allocating memory the usual way using `new` and `delete` each time slows down the code dramatically so that pre-allocation is necessary. Such implementation specific details are covered in more detail in Appendix A.

After adding the object to the IZB, a check must be done if also yet uncovered space has been discovered. If so, the object is regarded as a starting node (as reasoned in chapter 3.2.1) and no dependencies are created. Otherwise, the new object is added as successor to the lists of all encountered objects.

A few comments about the resolution of the IZB are in order: We face a tradeoff between accuracy and speed. Since all rectangles are conservatively aligned to the grid which always leads to an increase of the occupied area, a lower resolution will tend to yield larger rectangles which in turn means more unnecessary constraints. On the other hand, a high resolution will significantly reduce performance, since index look-ups are done on a per-pixel basis and they are costly when considering the overwhelming number needed. My choice was a resolution of 64 x 64, which both proved high enough to avoid over proportional increases in the size of the rectangles and still managed to do the look-ups in reasonable time.

Note that the idea of using a software buffer for approximating cumulative occlusion is not new: For example, the approach presented here shows similarities to one level of hierarchical occlusion maps as described in [Zhan98], which also serves the purpose of finding out if an object can possibly be occluded by the cumulated projection of others. The most important difference is that while Zhang accelerates the test by making it hierarchical, the IZB provides more information by exactly identifying the objects that contribute to the occlusion.

3.2.3 Traversal

Once the OG has been created, it is used to determine the order in which objects are tested and drawn as described in this chapter. This process is called traversal and it must account for the constraints given by the OG as well as select among all possibilities (e.g. nodes that could be tested next) in a fashion that minimizes the overall duration.

While up till now, the execution time was (without improvements) solely determined by the CPU, the bottleneck of this part typically lies on the GPU since it must both render and test the objects while the work for the CPU is comparatively simple. Thus, additional computations leading to less occlusion queries will normally pay off (a fact utilised by enhancements discussed in later chapters).

3.2.3.1 Structure of the Pipeline

In each moment of the traversal, each node falls into one of four possible categories representing the stage of progress of the node:

A: Yet untested nodes.

B: Nodes waiting for the result of their occlusion queries to become available.

C: Nodes with determined visibility (the result is present or no occlusion query was needed) but before a potential rendering.

D: Nodes that have been rendered or skipped due to occlusion.

These classifications can be regarded as a pipeline for a single node and they are sorted by the time a node passes them. Without improvements, each object (apart from the starting nodes) starts as part of A and makes its way to D in a step-by-step manner. Later, we will introduce some kind of shortcut, allowing a node to move directly from A to C (therefore saving the occlusion query by skipping B). Note that all successors within the OG of an arbitrary object will always be within the same or a former stage, but can never be ahead. Since this concept of a pipeline comprising four stages is important for the further discussion and in a distinct fashion reflected by the implementation, Fig. 3.5 represents the according state machine and titles the transitions.

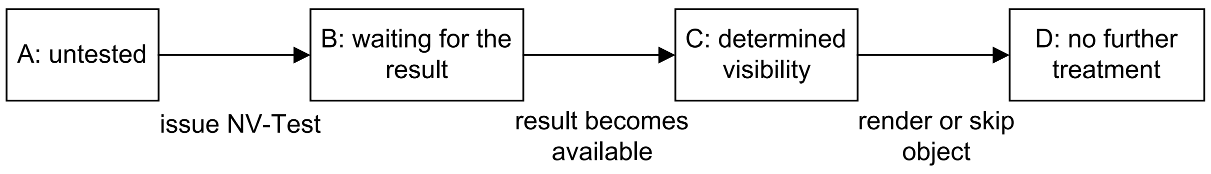


Fig. 3.5: The testing/rendering pipeline of a single node.

The initial situation is that all nodes identified as starting nodes are assigned to C while all other nodes still belong to A; B and D are empty. The traversal is complete when all nodes have reached D which is equivalent to B and C being empty (each node of A must have at least one predecessor in B or C which will issue its occlusion query).

The sets B and C are explicitly modeled by data structures in the implementation (a queue and a heap, respectively). A is implicitly given by the successors of the objects of B and C within the OG – since each node is aware of its successors, it can propagate them from A to B before it moves to D itself. The nodes within D need not be remembered as they are of no further importance anyway. Details concerning the mentioned data structures are discussed in a later chapter.

Note that the structure of this pipeline applies to most occlusion culling algorithms - for those not employing hardware tests, B might be missing. Yet unlike straightforward approaches simply starting to handle the next object when another one has reached D (characterizing the basic *stop-and-wait approach* presented in chapter 3.1), we overlap the pipelines of many objects, seeking to interleave the execution of the various steps as much as possible.

3.2.3.2 Node Selection

An important aspect of the OG is that it defines a set of constraints rather than a unique order. There are still many different sequences of tests and renderings feasible, some being much better than others with respect to the time of the incurred CPU stalls. Choosing randomly may lead to situations where quite isolated nodes are dealt with first, while other nodes occluding many objects are postponed causing all successors to wait for them. What we need is a way to prioritize nodes, thus assigning a *priority value* to each node that represents its importance. This allows us to simply select the one with the highest priority among all possibilities.

```
    $32:
A   mul      $14, $4, 8
B   addu     $6, $5, $14
C   mul      $15, $7, 8
D   addu     $24, $15, $8
E   l.d      $f4, 0($24)
F   mul.d    $f6, $f4, $f12
G   l.d      $f8, 0($6)
H   add.d    $f10, $f6, $f8
I   s.d      $f10, 0($6)
J   addu     $7, $7, $9
K   addu     $4, $4, $10
L   addu     $2, $2, $1
M   bne      $11, $2, $32
```

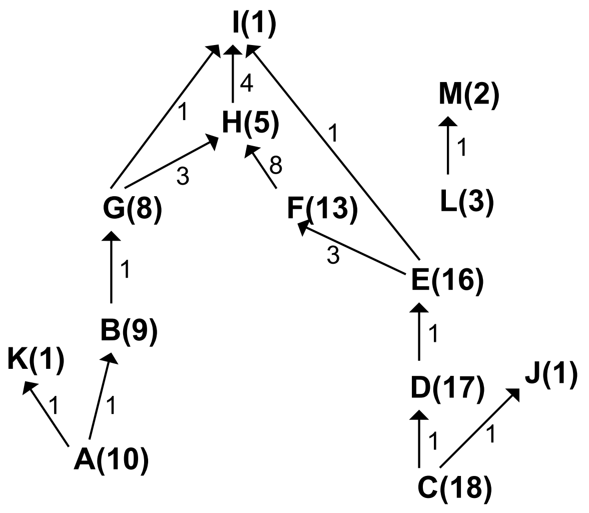


Fig. 3.6 : A program written in MIPS assembly language and the according data dependence graph with execution times and priorities.

A similar problem arises within a completely different branch of computer science: Compilers for modern CPUs face the challenge of ordering the individual assembler instructions being the result of former steps in a way that both takes all dependencies into account and maximizes the parallelism within the CPU by minimizing the time wasted by pipeline bubbles [Broc02]. In this context, dependency refers to a situation where for example the result of one instruction is used as an operand by an immediately following one. The second instruction must wait until the result is actually present, yielding unwanted pipeline stalls that could have been avoided if independent instructions had been put between. These dependencies are modelled by an acyclic graph very similar to the OG – called *Data Dependence Graph* –, as shown in Fig. 3.6, where a small program written in MIPS-assembly language is compared to the according graph.

However, there is a difference: While the execution time of each instruction is known in advance (each edge is titled with the number of clock cycles needed for the originating instruction in Fig. 3.6), we typically do not know how long an occlusion query will take. Although the time for the test itself is proportional to the number of fragments, as shown for HP tests by Staneker [Stan02], several other factors render useful predictions impossible – the state of the command buffer being the most significant. Therefore we assume the same duration for all occlusion queries.

According to [Broc02], all algorithms for finding the best order for the instructions (the exact solution) show NP-complete behaviour. Consequently a heuristic algorithm called *List Scheduling* is introduced, which is worth describing here because it will turn out to be conceptually equal to the algorithm traversing the OG. It repeatedly selects the instruction which should be scheduled next by evaluating the following two criterions:

- Among all potential instructions (where all predecessors have been scheduled), select the one that will wait the least or – if possible – not at all.
- If there are more candidates, select the one with the longest path to the end of the graph.

The second criterion is a proposal how to obtain the required priority value, as mentioned in the first paragraph of this chapter: For each instruction we know the best-case duration until all other instructions being directly or indirectly dependent have completely been executed. When each edge is assigned the duration of the execution of the originating instruction, this value coincides with the maximal path length (given by the numbers in brackets beside the letters of the instructions in Fig. 3.6).

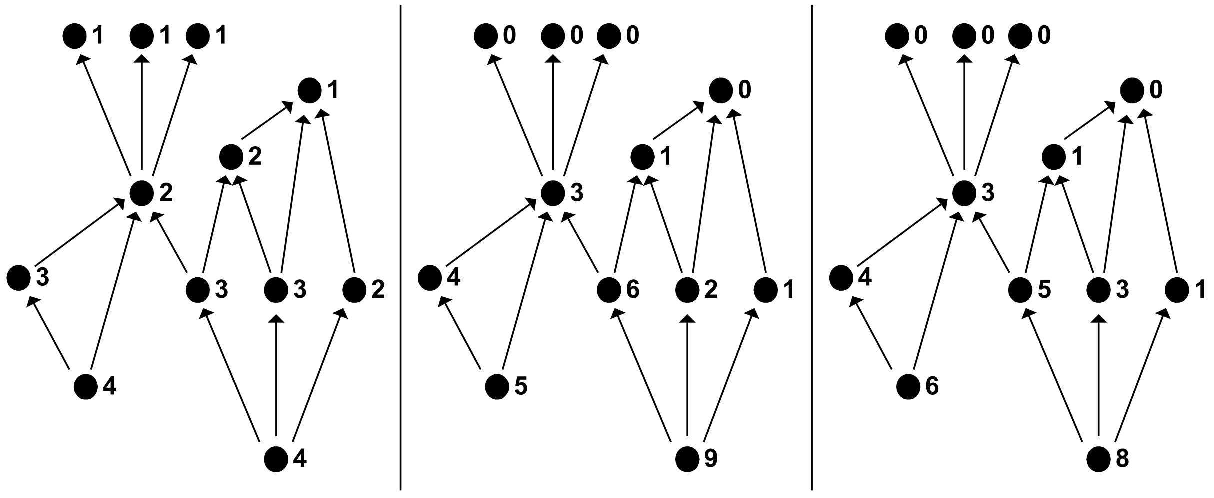


Fig. 3.7: Three different ways to compute priorities for the same graph: The List-Scheduling heuristic (left), exactly counting all dependent nodes (middle) and the introduced approximation (right).

This concept is directly applicable to the OG: Apart from the difference that each node is assumed to take equally long (taking unity as duration) as reasoned above, we successfully attain a meaningful priority for each node as shown in the left third of Fig. 3.7. This is still not the optimal solution though: Fig. 3.8 shows a situation where the node A and B would be assigned the same priority value. In reality, rendering node A is more urgent than rendering B which should be reflected by their priorities. This problem is caused by the fact that we basically count nodes, while the original heuristic counts execution times. In the latter approach, the number of dependent instructions is not significant as long as these instructions can be executed fast. So, how can we overcome this flaw?

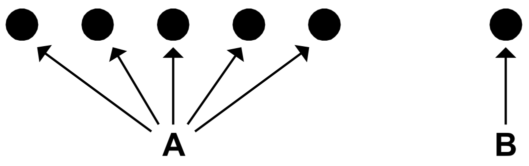


Fig. 3.8: A situation, where the List-Scheduling heuristic yields bad results.

Assuming the same time for all occlusion queries, an optimal priority would be the number of nodes being direct or indirect successors, yet counting each node only once (the resulting priorities are depicted in the middle third of Fig. 3.7.). While this is a trivial task for tree-like data structures, the challenge with general acyclic graphs are diamond shaped constellations: nodes merging multiple branches originating from the same node should be counted only once. Since I did not find an algorithm solving this problem in linear time, I decided to approximate the results by the following heuristic which can be computed with O(N) effort and generally yields convincing results:

$$P(A) = \mathrm{Max}(P(S_i(A))) + C(A),\ i = 0 \,..\, C(A) - 1$$

P(A) is the priority of the node *A*, *$S_i(A)$* is the *i*-th successor of *A* and *C(A)* is the number of direct successors of *A*. In other words: The priority of each node is equivalent to the number of its direct successors plus their maximum priority. This way, each node will definitely be counted not more than once and this formula can easily be computed in a recursive function. The result is shown in the right third of Fig. 3.7.

3.2.3.3 Traversal Algorithm

Apart from the formula used to compute the priority, the actual traversal algorithm strongly resembles List Scheduling as outlined above. However, for the sake of completeness and since this is a core piece of the whole OG-based approach, a more detailed description follows in terms of the pipeline introduced in chapter 3.2.3.1.

In the initial situation, all starting nodes are within C while all other nodes are part of A; B and D are empty. Listing 3.1 illustrates the algorithm in pseudo code.

Note the comments indicating when an object advances from one stage to another. Furthermore, the idea behind the *parent counter* needs some explanation: Since a node is not directly aware of its predecessors (in contrast to its successors), the initialization at least tells it how many other nodes are referencing it – the parent counter. This value is needed to determine when an object is free to be tested (being the time when all predecessors have reached D).

Listing 3.1: *The traversal algorithm for the OG:*

```
while ((B not empty) AND (C not empty))
{
  // B -> C
  Move all nodes where test result is present from B to C

  // B -> D, resp. C -> D
  if (C is empty) N = node from B with oldest test
  else            N = node from C with highest priority

  Visible = occlusion query result(N)
  if (Visible = TRUE) Render(N)

  Toggle GLState to testing

  for each Successor S of N
  {
    S.ParentCounter = S.ParentCounter - 1
    if (S.ParentCounter = 0)
    {
      Issue occlusion query(S)

      // A -> B
      Add S to B
    }
  }

  Toggle GLState to rendering
}
```

3.2.3.4 Implementation Issues

The algorithm as presented in listing 3.1 can be improved in several ways which will be subject of later chapters, yet one basic optimization is already introduced now, for it does not affect the concept as such: In this version, we switch back and forth between the OpenGL states for rendering and testing too often. Frequent GL-state changes become costly, so we must seek to reduce them. Objects issuing no occlusion query for their successors require no toggling at all which can be considered by changing the state lazily (referring to changes only done when actually needed for a subsequent action). But even then, once we are in testing mode, we will probably return to rendering mode for a single object before restoring testing mode again.

The idea is to reduce the number of loop executions by gathering multiple objects from C and rendering all of them before testing their children. This has no severe impacts on the pseudo code of listing 3.1, *N* simply becomes a set of nodes instead of a single one. However, an issue is how many objects should be gathered at one time. Very small numbers result in many state changes, yet collecting lots of objects might interfere with the priority-based selection (if all objects within C would be dealt with immediately, any tests for successors could be issued possibly only after all objects currently stored within C have been rendered, which would contradict the requirement to issue tests as early as possible and would foil the benefits of the priority-based selection). In practice, gathering five objects has proved to be a reasonable compromise. In any case, no additional nodes should be taken from B as this would incur CPU-stalls. Hence if C is empty, still only the oldest object of B is considered and if C contains less than five nodes, only these are handled without adding further objects from B.

For the sake of brevity and since presenting every little detail would probably rather cause confusion than to make things clear, no source code is shown here. However, a few remarks concerning the implementation may prove helpful to highlight practical aspects of the concepts that have been dealt with in a theoretical manner so far.

Perhaps the most important question is which data structures are appropriate for the classifications that have been called B and C. The requirements for C was to:

- Allow quick access to the element (which is an OG-node in our context) bearing the highest priority.
- Support fast insertion and removal of elements – this means in logarithmic time.
- Make no assumptions about the elements (e.g. multiple nodes can have the same priority).

A data structure matching all requirements is a *heap*. A heap can be characterized as an array constraining the order of its elements. Let P_i be the priority of the *i*-th element, then

$$P_i \geq P_{2i} \text{ and } P_i \geq P_{2i+1}$$

must evaluate true for all elements. It is immediately obvious that the element with the highest priority must be the first one which grants access in constant time. As required, insertion and removal can be done in logarithmic time, so the heap is an efficient data structure for our needs. Note that heaps are also sometimes referred to as a priority queues. Further information about heaps can be found in several books about standard algorithms. I want to add that I implemented the heap as a generic C++ template permitting flexible reuse.

The purpose of B is to hold back tested nodes, where the result is still outstanding. According to the specification of the NV_occlusion_query [Crai02], tests are guaranteed to arrive in the same order as they have been issued, so the result of the oldest test will be available first. Therefore a *queue* is the most appropriate data structure for this task (this means an ordinary queue not to be confused with the priority queue mentioned in the previous paragraph). While this choice is quite evident, a more interesting question is if the queue should be bound in size. Doing so implies an assumption about the maximum duration of a test (in terms of how many other tests can be issued and objects rendered until the result is ready) and I actually attempted at first to find out a proper value for its size.

However, this turned out to be problematic: Having a small queue, objects are likely to proceed to C just because their space is needed for further tests – hence C contains objects with unfinished tests causing unnecessary CPU stalls when waiting for the results. On the other hand, a large queue might delay objects too long, interfering with the priority-based selection. Even worse, very long queues lead to situations where almost no node proceeds to C at all and each node must already be taken from B due to a lack of alternative work. Yet worst of all is the observation that no optimal value for the size of such a queue exists: Mainly the state of the command buffer, but also other factors beyond the scope of predictability like interrupts by other tasks make an 'optimal' value vary to an extent that makes it reasonable to look for an alternative solution.

Fortunately, such a solution exists: As described in chapter 2.6.3, the NV_occlusion_query permits to ask if a result of an outstanding test is present without incurring any stall. Furthermore, measurements have shown that such requests consume almost no time even when using them excessively. Thus, a better way to implement B is as an unlimited queue, where the oldest tests within the queue are checked for availability at each pass of the loop and moved to C if the according tests are ready. The best place for this check is right before nodes get chosen from C (as shown in Listing 3.1). This way, all nodes of C are guaranteed to cause no delay as well as nodes are only taken from B if there is really nothing else to do (and because the test of the first node always dates back longest, exactly that node will be fetched that incurs the shortest CPU stall).

3.3 Problems

So far, theory and practice of the OG-based approach (without extensions and major improvements) have been discussed in detail, but the most important question has not been dealt with: Does the effort actually pay off?

Generally speaking, the disillusioning answer is: It does not. Scenes with little occlusion are slowed down dramatically and even in cases where much occlusion can be found, the performance is hardly better – if not worse – than with the straightforward *stop-and-wait* model as described in chapter 3.1. Concrete results are presented in chapter 3.5 and for a comparison with other approaches, refer to chapter 5, so this chapter basically confines itself to highlighting main insights and flaws informally.

The most striking issue is: Why does the theoretical superiority compared to the simple methods of chapter 3.1 – and the approach actually is superior with regards to its flexibility, generality and the quality of the obtained order – hardly reflect in better frame times? The reasons range from general problems of occlusion culling (if no occlusion is present, doing any sort of occlusion culling will inevitably slow down the whole rendering process), to problems specific for hardware occlusion queries as well as – and this is the main point – design faults of the algorithm. These are the major shortcomings of the approach as presented so far:

- Since the scene-graph traversal and the generation of the OG are pure CPU tasks, the GPU is idle very long, while it becomes the bottleneck afterwards; hence the load is badly balanced.
- The pure CPU tasks take too long – basically this is equivalent with the insight that we have to deal with too many nodes.

- The occlusion queries themselves are expensive (see the discussion below) and can only pay off if the geometry to be culled away exceeds a certain complexity which is most often not the case.
- The time wasted by CPU stalls could considerably be reduced, but is still far from being negligible.

How can these problems be solved? The following list outlines potential solutions regardless of a concrete realization. How these improvements can actually be achieved will be the subject of the following chapters.

- Increase the parallelism between CPU and GPU by rendering visible objects before constructing the OG.
- Decrease the number of objects for which the OG gets constructed.
- Increase the average complexity of an object thus making an occlusion query relatively cheaper compared to rendering it.
- Reduce the number of occlusion queries to a minimum.

Before showing how these improvements can be attained, let us answer the question why occlusion queries are so expensive:

- As discussed in chapter 2.6, these tests differ from ordinary rendering operations only in so far as no buffers are affected, yet the rest of the work basically stays the same: While the tested geometry is trivial and hardly means an additional strain for the geometry stage of the GPU, and no textures or other non-geometry data consume AGP-bandwidth, the number of fragments to be rasterized usually even exceeds the according original geometry (although lighting, texturing and other fancy effects ought to be turned off).
- As already discussed extensively, asking for the result of a test can lead to CPU stalls.
- The task of issuing the tests turned out to be much more expensive than expected. Apart from the OpenGL commands for specifying the geometry, especially the procedures `BeginOcclusionQueryNV()` and `EndOcclusionQueryNV()` emerged as substantial speed killers: On the computer where the implementation for this master thesis was developed (refer to chapter 3.5 for technical details), only executing these two commands 1000 times (and some scenes may require much more) takes at an average 2,4 milliseconds, being more than 14 percent of the desirable overall frame time for rendering at 60fps. These times are independent of the state of the command buffer and other circumstances as can easily be found out by calling `glFinish()` before.
- Finally, one must not overlook that a great part of the preparations (above all the construction of the OG) merely serves the purpose of allowing a sensible application of occlusion queries and is not needed for any other reason.

These points mean an additional effort compared to a straightforward rendering which is definitely not negligible. Therefore the original geometry of an object must be enriched to such a great amount that an occlusion query has any chance to pay off at all and it is an absolute imperative to maintain a reasonable ratio between the effort spent on testing and the rendering costs avoided due to occlusion culling.

3.4 Improvements

As reasoned in the previous chapter, the approach is not practicable so far since speed gains are foiled by an over proportionate effort for the occlusion culling itself. Fortunately it can be enhanced in several ways, realizing a better load balancing as well as reducing the amount of occlusion queries and the overall number of objects. As it will turn out, even the implementation itself can be made faster when sacrificing a little bit of flexibility.

Many improvements have in common that they exploit coherence between successive frames in one way or the other: Up till now, each frame was completely independent of what has been rendered before and did not care if its classifications could be of any interest to the following frames. The same way, it did not matter if an object was close to an already classified one, or not. However, in reality, both temporal and spatial coherence can be observed and exploiting them appropriately is crucial to cope with the problems as described above.

One remark to clarify the structure of this thesis: A fundamental modification will be to do the approach hierarchically. Since a separate chapter (chapter 4) was dedicated to hierarchical approaches in order to separate hierarchy-related discussions from the basics of the OG and to permit a better comparison between the hierarchical OG-based approach and an alternative, also hierarchical approach doing without OG, this topic will be subject of chapter 4.

3.4.1 Reduction of Occlusion Queries

As argued in chapter 3.3, occlusion queries are inherently expensive, so we must seek to reduce them as much as possible. This can be achieved in two ways:

- Having less objects will automatically lead to fewer tests.
- Tests can be saved when we are able to predict their result.

While the reduction of objects will be one effect of having a hierarchy, the measures presented here deal with the prediction of test results. Generally speaking, care must be taken when classifying an object without test: False classifications always come at a cost. While regarding an actually invisible object as visible simply means superfluous work but does not do any harm to the visual result due to the Z-buffer, assuming an actually visible object as invisible deteriorates the image quality and ought to be avoided (still, one modification will do right that). Consequently, especially rejects are tricky. On the other hand, assuming an object as visible can sometimes even make sense if we are not sure at all: Due to the considerable overhead of occlusion queries, objects being simple enough (e.g. containing very few triangles) could be rendered without wasting any time on tests. However, this principle is not applicable to the non-hierarchical version we are currently discussing as will be pointed out in chapter 4.1.

3.4.1.1 Using Temporal Coherence

So far, the approach assumes that no frame resembles the previous one in any way. In practice, subsequent frames will hardly differ: For example when moving with walking speed, each step will alter the position by about half a meter and will take about half a second, hence (assuming 60fps) the viewpoints of approximately 60 frames are within one meter. With a very high probability, all of these frames (and presumably many further ones) will show almost the same objects, which in turn means a very similar classification of occluders and occludees. When standing still, no changes will probably occur at all, but this principle even holds when moving faster than with walking speed or when performing rotations.

This phenomenon is referred to as temporal coherence and can be observed in several contexts: Movies like MPEGs for instance do not store each frame independently, but reuse former (and also later) frames. Furthermore, many caching strategies are based on temporal coherence: Once a certain server has been contacted, a further request to the same server is usually much more likely than to an arbitrary other server. In the context of occlusion culling, temporal coherence can be exploited in several ways. Here, we are about to reuse one test result for several frames.

As argued above, doing occlusion culling conservatively requires that only positive classifications (such indicating visibility) are reused since drawing objects for too many frames is wasted effort, yet will not affect the final image. Falsely classifying an object as invisible on the other hand may lead to ugly distortions. Since the state of occlusion can not be guaranteed to hold for subsequent frames as well, this principle can not be applied to occluded objects. From now on, this modification will be referred to as *assumed visibility*.

The decisive parameter is, for how many frames a test should be skipped once an object was found to be visible (from now on, this number is referred to as *VF*, abbreviating visible frames). A serious discussion requires that we anticipate a few results: Fig. 3.9 plots the average frame times of three different walkthroughs against varying values for *VF*. The scene of the first walkthrough is a box which is densely populated with teapots – small yet quite complex objects with much cumulative occlusion. The scene of the second and the third walkthrough is a model of an urban environment; the difference is that one time (titled as 'Simple City'), we move through streets where much occlusion occurs, the other time ('City - No Occlusion'), we look at an open area from above, causing hardly any occlusion. For more information, refer to the description of the test walkthroughs in chapter 3.5.

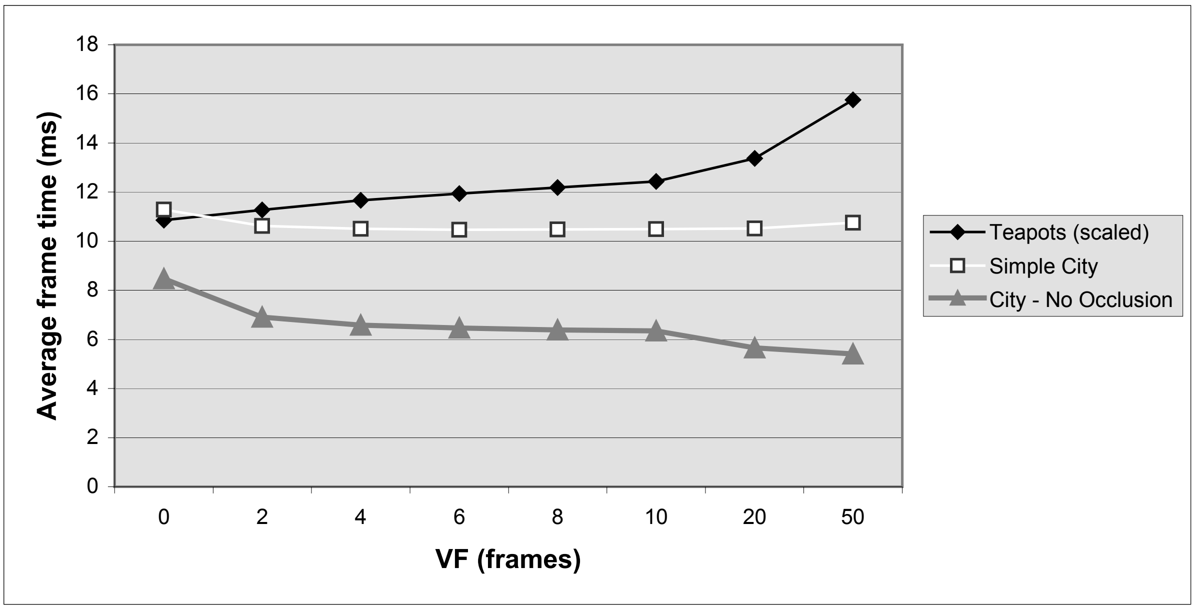


Fig. 3.9 : The effect of *Assumed Visibility* (using a fixed number for *VF*) in three different scenes for varying values of *VF*. The frame times of the teapot scene were divided by 10 to permit a better comparison by drawing all three plots into the same graph.

While the teapot scene was considerably slowed down, the first city walkthrough basically remained quite unaffected and the second one could even be remarkably accelerated. For an interpretation of these results, one has to become aware of the costs and benefits of a*ssumed visibility*:

- Costs are objects rendered due to false classification.
- Benefits are skipped tests for correctly classified objects.

Note that the costs differ tremendously with the complexity of the objects; hence if a teapot consisting of approximately 2000 triangles is rendered in vain, the penalty is much greater than for a flat wall of a house with maybe less than 10 triangles. Apart from the number of triangles, the occupied number of fragments, the textures and the presence of expensive effects like pixel shaders contribute much to the complexity of an object.

In the teapot scene, just a small part is actually visible for many objects, peering behind some tiny hole. Even slight changes of the viewpoint are likely to close these holes and open new ones, showing mostly different objects. Therefore, when assuming visibility for several frames, a significant part of the objects considered to be visible is actually hidden. Besides, especially the teapot scene would require a precise classification due to the high complexity of the objects as explicated above. In conclusion, comparatively moderate benefits of a few correctly skipped tests oppose huge costs – this explains the bad performance.

In the first walkthrough of the city scene, slight improvements can be observed. Here, only a small fraction of the overall geometry is visible – yet for these few objects, the principle of temporal coherence is applicable. However, since the number of visible objects is so small and these objects are moreover very simple, the costs are almost negligible just like skipping a few occlusion queries does not mean a big gain.

In the second walkthrough of the city scene, almost all objects passing view-frustum culling are actually visible, hence each occlusion query is wasted effort. The benefits exceed the costs by far and this explains the good results there.

But can we actually call this modification an improvement, if it can also have the contrary effect as intended? It must be revised so that *VF* depends on how likely an object is to stay visible for several frames. This can be achieved by using the result of the occlusion queries: Knowing both the total size and the number of visible fragments allows us to compute the percentage p to which the object could be seen. Furthermore, let VF_{min} and VF_{max} denote the number of frames for which an object is assumed to stay visible at least and at most, respectively, then a reasonable actual value for *VF* can be obtained by linear interpolation:

$$\mathrm{VF} = \mathrm{VF}_{\mathrm{min}} + \mathrm{p} * (\mathrm{VF}_{\mathrm{max}} - \mathrm{VF}_{\mathrm{min}})$$

So far, we discussed a special case where VF_{min} coincided with VF_{max}. As illustrated in Fig. 3.10, calculating *VF* by linear interpolation slightly compromises the benefits where temporal coherence has already worked well, yet no considerable deterioration can be observed any more for small values of *VF* in any scene. Fig. 3.10 shows the same walkthroughs as before with varying VF_{max} and keeping VF_{min} set to zero.

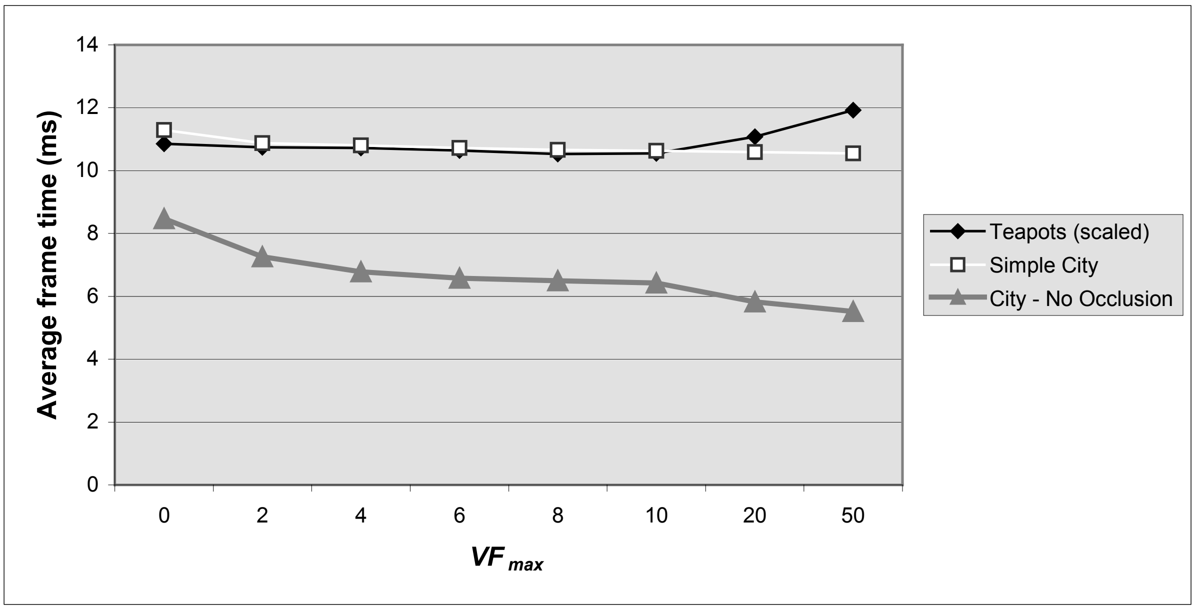


Fig. 3.10: The effect of *Assumed Visibility* in three different cases (using a linearly interpolated number for *VF*). Again, the times of the teapot scene have been divided by 10.

After all, no generally optimal values for VF_{max} and VF_{min} exist, since the actual amount of temporal coherence differs from scene to scene and from view to view. Generally speaking, VF_{min} should not be much higher than one or two in order to avoid the same troubles as with the constant value. Increasing VF_{max} amplifies both the benefits and the costs. Usually, for low values, the benefits rise faster than the costs while this situation is reversed at some point being the optimal value. But this values possibly ranges from zero to several hundred, depending on the scene and of course on the movement within that scene. However, in practice, setting VF_{max} to 10 has been found to be a reasonable compromise as it is conservative enough to avoid a massive over-classification and still high enough to permit a perceptible effect. Therefore, this value will be used from now on in full awareness that for some scenes, a lower or higher value would be more appropriate.

One might object that the performance gains are still far from being impressive. This is true as long as the number of visible objects is a minority with regards to the overall number of objects. However, *assumed visibility* is still important since it is a prerequisite for a better load balancing as discussed in chapter 3.4.2.2 which can lead to more substantial performance gains.

The idea behind this approach is not new: Bittner [Bitt01b] updates the visibility of visible objects (that are nodes of a hierarchy) with a certain probability, otherwise they are considered to stay visible, which is basically the same idea, only he lacks the information about how many percent have actually been visible.

An idea for a potential improvement is to incorporate the current movement into the computation of *VF*: For instance, when standing still, much higher values might be reasonable than when racing through a town. The idea of making the validity of a set of visible objects dependent on the movement speed is related to *Instant Visibility* as presented by Wonka [Wonk01]. However, no further research has been done on the concrete application within our setting and it is up to future work to do so.

3.4.1.2 Using Spatial Coherence of the Occlusion Graph

As demonstrated in the previous chapter, saving occlusion queries only for the visible part of all objects does not suffice. This chapter presents a method to skip tests for occluded objects as well. More precisely, we want to be able to classify objects as invisible without testing and we want to do this conservatively.

The core of the approach we have been discussing for several pages now is the OG. So far, however, its only purpose is to reduce the time of CPU stalls by stating which objects are independent of each other. It manages to do this task successfully, yet the performance gains can never pay off the costs of its construction. Fortunately we can also use its information in another way.

A very primitive OG comprising only two objects is A $\rightarrow$ B. It tells us that B can only be tested after A has been rendered, as has been thoroughly discussed. Remembering the construction, it tells us even more: B's SSBB must be completely covered by A's SSBB. Otherwise, A would not constrain B. If A is invisible (for instance if this is only a part of a greater overall OG), it is very likely that B is invisible too. Since 'very likely' is not enough for a conservative approach, let us examine under which circumstances B can still be visible, if A has been proven invisible:

- Parts of B lie in front of A.
- B's approximated geometry used for the occlusion query (e.g. its AABB) is not completely covered by A's approximated geometry, although this is true for their SSBBs.

As it was discussed in chapter 3.2.2.2, all objects are approximated by planar SSBBs bearing the distance of the vertex of the original geometry which is nearest to the near plane. These SSBBs are in turn sorted by that distance before being written to the IZB. Therefore, situations are rendered impossible where a 'nearer' object could be penetrated by a 'farther' one – so the first point can never happen.

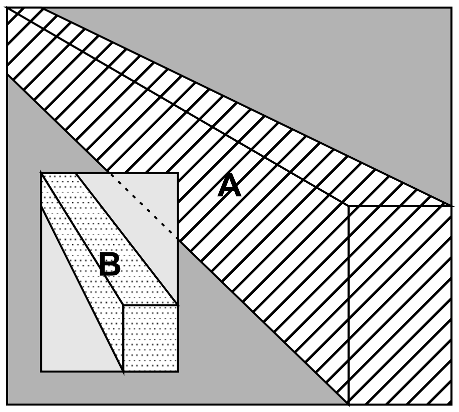


Fig. 3.11 : A case where the SSBBs indicate occlusion, although none is actually present: Object A is assumed to be in front of object B. The (perspectively projected) AABBs do not overlap, while the SSBB of A entirely covers the SSBB of B.

Fig. 3.11 illustrates a situation as described by the second point: Although A's SSBB covers B's SSBB completely, B's projected AABB can still be seen (even entirely in this example). However, tackling this problem is not difficult either: It simply must be guaranteed that we use exactly the same geometry for the occlusion queries and for constructing the OG. Since scan converting arbitrarily projected AABBs into the IZB is no option due to complexity and performance reasons, from now on, we must use SSBBs for the occlusion queries. The disadvantage is that a SSBB is an even more conservative approximation than an AABB, thus we will lose some occlusion (as it will turn out, this loss can actually be significant). On the other hand, in most cases we can skip a vast part of the occlusion queries for the occludees.

Generally speaking, provided that all objects are sorted by their minimal distance from the near plane and that precisely the same geometry is used for the occlusion query and for the construction of the OG, an arbitrary object is definitely invisible if all predecessors have been proven invisible. From now on, this technique will be referred to as *Early Rejection*. Since exploiting it incurs no further costs (apart from the more conservative approximation), it means a substantial improvement (see chapter 3.5 for graphs comparing rendering with and without *Early Rejection*).

However, the implementation issues are not completely trivial and should briefly be discussed: Propagating occlusion from one node to another is not difficult to implement: All nodes but the starting nodes are at first assumed to be invisible. Once a node has been proven visible, it sets all of its successors to potentially visible, which means that an occlusion query is required for each to make sure. If a node is fetched from A (using the terms for the testing pipeline of chapter 3.2.3.1 again) which is still classified as invisible, it can immediately be forwarded to C in order to 'free' its successors and then moved to D without any rendering and testing. In turn, the same applies to its successors.

The tricky part of the implementation is deriving the geometry for the occlusion query: As the name suggests, SSBBs are given in screen space, not in world space where we usually define geometry. What we basically have to do is:

- Compute the device coordinates of the SSBB.
- Find a way to specify it in OpenGL.

Even the first step is not as straightforward as it may seem, since we must take into account that all SSBBs get aligned to the grid of the IZB. The resulting enlargement must be considered before scaling the values to match the resolution of the output device.

Afterwards, we know the area which has to be covered by the geometry in device coordinates as well as its depth value, which is the distance from the near plane, and must take care to specify this correctly. The clean way to do this is to switch to orthographic projection, which allows the usage of the device coordinates without further computations. The actual challenge is defining the correct depth value: Let *d* denote the distance from the viewpoint, let *P* be the current perspective transformation applied to the ordinary geometry and let *O* be the orthographic projection. When specifying geometry as usual, *d* is not used for the Z-buffer test as it is, but after being transformed by *P*, referred to as *P(d)*. The same way, *O* transforms depth values before using them in the Z-buffer test, yet by another formula than *P*. What we need is a transformation *T* with

$$O(T(d)) = P(d)$$

The equation is true for $T(d) = O^{-1}(P(d))$ with O^{-1} being the inverse of *O*. In OpenGL, a perspective transformation *P* is generated by calling `glFrustum(l, r, b, t, n, f)` and an orthographic projection *O* is generated by `glOrtho(l, r, b, t, n, f)` – in both cases, `l` means the left border of the viewport, `r` the right one, `b` its bottom and `t` its top while `n` stands for the distance of near plane and `f` for that of the far plane. The matrices for *P* and O^{-1} can be found in [Woo99]; considering only their third and forth row (we are only interested in the depth value), and negating the result of the intermediate steps since distances are assumed to be negative, we get:

$$T(d) = \frac{fn - d(f+n)}{d}$$

This result can be passed as third parameter to the `glVertex3f()` command.

Early Rejection can be seen as one way to exploit spatial coherence: Objects tend to be classified like nearby objects. As with temporal coherence, taking advantage of spatial coherence is not a new idea in occlusion culling algorithms: Basically the main intention behind all sorts of spatial hierarchies is to make use of it (just like we will do in chapter 4). An approach remotely related to *Early Rejection* is presented in [Bitt01b], titled *Visibility Propagation*: It also attempts to determine the visibility of a node by examining its neighbours. While the idea is similar, the realization differs fundamentally from *Early Rejection*: Visibility Propagation is done in a hierarchical setting (using kd-trees) and seeks to determine the visibility of the bounding box of a certain node by regarding the classification of neighbouring nodes. Unlike *Early Rejection*, no graph is used and the involved computations are not trivial.

3.4.2 Implementation-Related Improvements

The previous chapter presented two techniques to reduce the number of occlusion queries: As explained, *Assumed Visibility* is beneficial when we have a large set of objects staying visible for several frames and *Early Rejection* is effective when we have to deal with many occludees. However, since the other shortcomings mentioned in chapter 3.3 must also be addressed, further improvements are necessary.

3.4.2.1 Introducing a Visibility Threshold

The intention was to design a conservative approach, which means that an object being culled away must not contribute to the final image at all, and this holds for all improvements presented so far. Nevertheless, it can sometimes be desirable to trade off quality for speed. The NV_occlusion_query extension permits to do this very easily: Unlike with the HP_occlusion_test extension, we get the exact number of pixels passing both Z-buffer and stencil-buffer test. Thus we can simply introduce a threshold T_V rejecting objects with less or equal visible pixels: Instead of a single value (zero), occlusion becomes an interval $[0, T_V]$.

Small values for T_V can even make sense if we want to preserve the conservative behaviour on the whole: Due to the over estimation of the geometry itself, it is likely that only a small band at the edge or a part of a corner of the SSBB is visible which does not cover the actual geometry anyway. Such assumptions are justifiable yet need not be true – if T_V is greater than zero, we inevitably lose the conservative behaviour.

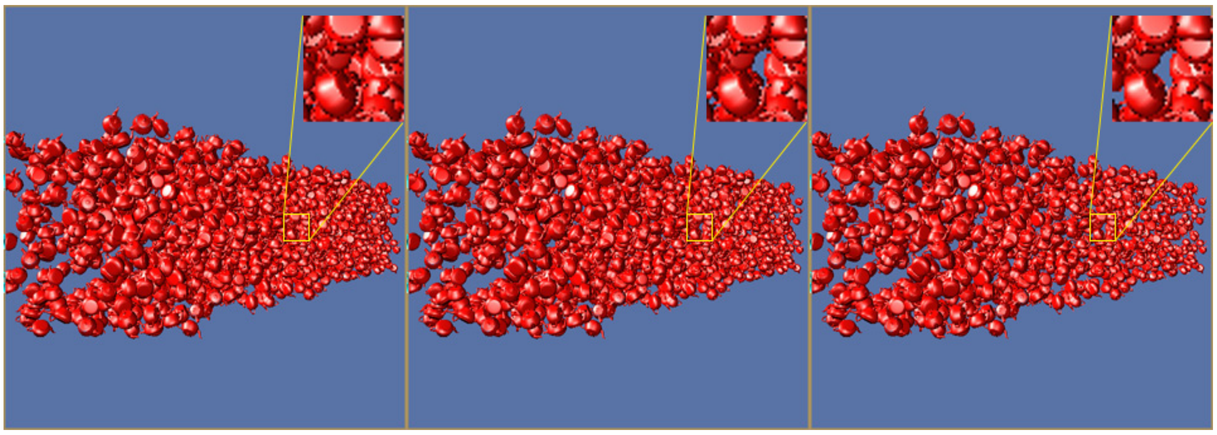


Fig. 3.12: The teapot scene, rendered with $T_V = 0$ (left), $T_V = 100$ (middle) and $T_V = 500$ (right).

Once more, an optimal value for T_V mostly depends on the properties of the scene: As subjective observations show, an urban environment with buildings as big occluders predominantly being not too far away permits much greater values for T_V than the teapot scene where the majority of the visible objects appears behind some small hole and even modest values for T_V may deteriorate the image quality perceivably (see Fig. 3.12 for a comparison of the image quality with $T_V = 0$, $T_V = 100$ and $T_V = 500$).

Another effect should be mentioned: Small objects being far away can suddenly vanish even if they are completely uncovered and still within the view frustum. If the SSBB shrouds less than T_V pixels, an object has no chance to be proven visible. At first sight, this fault resembles view-frustum culling with a far plane too close to the viewpoint.

Impacts on the performance are discussed in chapter 3.5. Further considerations can be found in [Zang98] where similar thresholds are used.

3.4.2.2 Better Load Balancing between CPU and GPU

A major design fault of the OG-approach as presented so far is the badly balanced load between the CPU and the GPU: Scene-graph traversal along with view-frustum culling, sorting the objects by their distance, computing the SSBBs and generating the OG by projecting them into the IZB – all these tasks are completely done by the CPU before the first triangle of this frame is committed to the GPU. Afterwards, work is lavished on the GPU while the CPU – despite the OG – idles a considerable amount of time waiting for test results.

As pointed out in chapter 2.6.2, such a discrepancy can also be observed with other applications where one task is to render the scene while other tasks are pure CPU work (for example calculating the AI). The difference is, though, that the command buffer usually manages to balance the load when the rendering procedure does not have to wait for any feedback from the GPU. Since our approach inherently relies on the results of the occlusion queries, this balancing does not work, so when returning from the rendering procedure, the work for the GPU is (almost) done as well.

This problem can hardly be solved entirely, yet it would mitigate the situation if certain objects could be rendered before doing the CPU-intensive tasks. On one hand, we have no OG then that could guide our selection, yet on the other hand, we do not want to lose occlusion by rendering objects which finally turn out to be occluded. The only objects that will definitely be rendered independently of the OG are those assumed to stay visible (see chapter 3.4.1.1) – that is why *Assumed Visibility* was said to be a prerequisite for this improvement.

Summarizing, all objects being classified as visible without occlusion query and passing view-frustum culling are rendered before computing SSBBs and creating the OG. From now on, this modification is referred to as *Pre-Rendering*. Since we still need to traverse the scene graph and must do view-frustum culling before, such an object can immediately be rendered after it has been proven to lie inside the view frustum (this means before doing view-frustum culling for other objects). The rest of the approach remains unchanged, particularly the OG still gets constructed out of all objects passing view-frustum culling – including the pre-rendered ones. Of course, care must be taken that an object does not get rendered twice per frame which can be ensured by applying a frame counter.

The effect of *Pre-Rendering* is tightly interwoven with the command buffer: Any drawing command is enqueued within the driver and its execution is delayed until all previously committed work has entirely been pulled from the GPU. Consequently, the overall latency L of a drawing command in general (and an occlusion query in particular) comprises both the actual execution time within the GPU (T_{own}) as well as the execution times of all commands (T_{other}) which are stored in the command buffer at the time when the respective command is added. L can be long, which is unpleasant if the application relies on feedback from the GPU – as it is the case with occlusion queries – since longer latencies of occlusion tests will on average mean that more time is wasted on the CPU side waiting for results.

Pre-Rendering tackles this problem: Although the execution times themselves of the various commands (T_{own}) are determined by the speed of the GPU and can not be influenced, it seeks to reduce T_{other} for each occlusion query by keeping the command buffer as empty as possible within the OG traversal. This is achieved by shifting much of the rendering effort for drawing visible geometry away from the GPU-intensive part of the OG-traversal: With *Pre-Rendering*, the GPU can accomplish the rendering of considerable parts of the visible scenery, while the CPU is still busy with traversing the scene graph and creating the OG. This is illustrated in Fig. 3.13: It shows the fill level of the command buffer and the state of the CPU versus time. With *Pre-Rendering*, the scene traversal takes longer (since rendering commands must be issued), but during the traversal of the OG, less work accumulates within the command buffer, reducing the time wasted by stalls. Besides, the figure highlights that the GPU is idle for a long time without *Pre-Rendering*. Note that – for the sake of simplicity – Fig. 3.13 assumes that all further work depends on the result of an occlusion query, which is not the case with the OG-based approach, as objects can be handled independently of each other, but the overall effect remains correct.

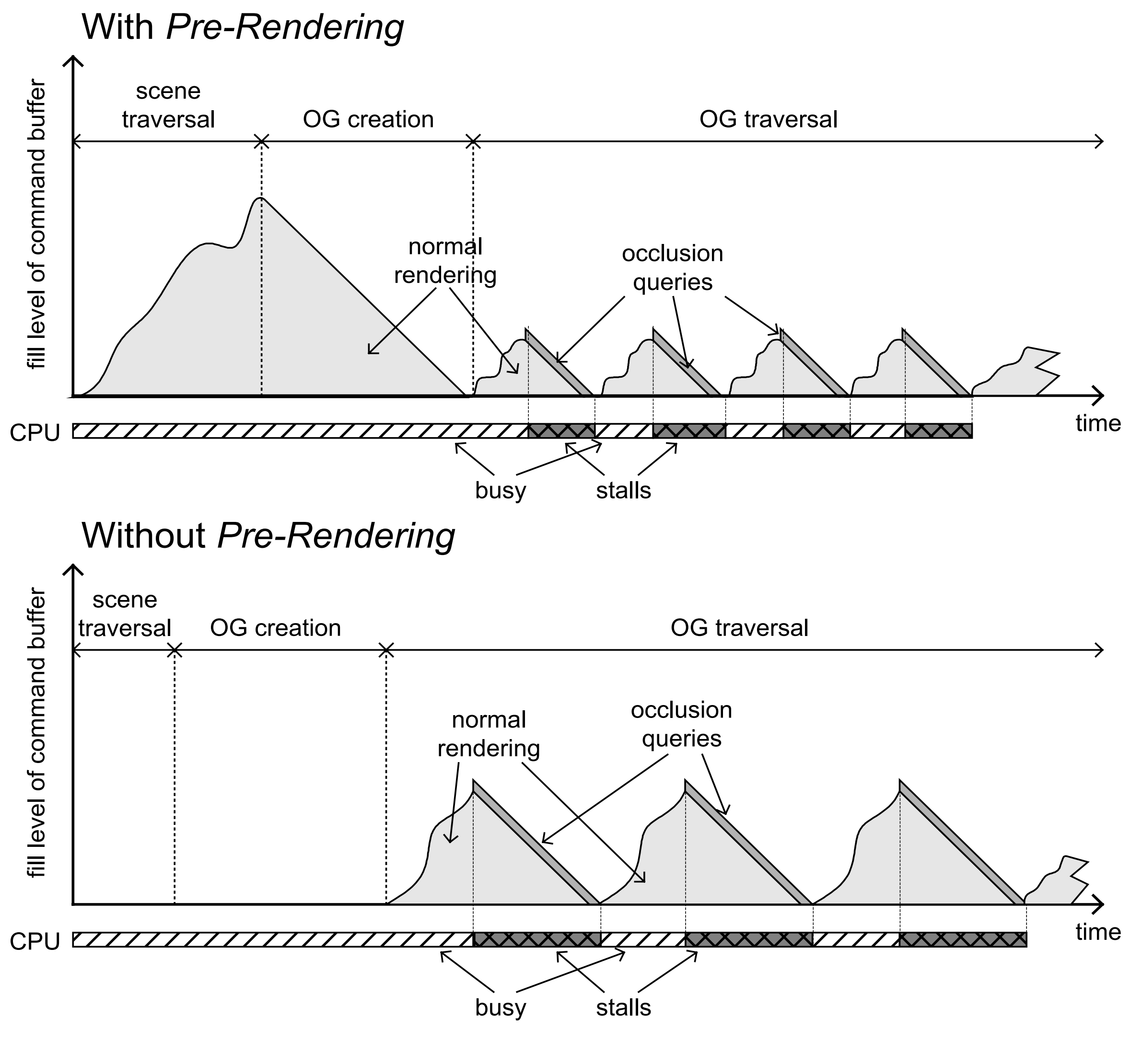


Fig. 3.13: The effect of *Pre-Rendering* on the fill level of the command buffer and the duration of CPU-stalls.

3.5 Results and Discussion

Chapter 3 introduced an approach for occlusion culling using hardware occlusion queries based on a directed, acylic graph – the OG –, pointed out several shortcomings and presented some improvements. While most of the previous discussion has been held theoretically, some figures should finally highlight practical aspects. However, this examination focuses on the OG-based approach as such – for comparison with other approaches, more detailed assessments and conclusions, refer to the chapters 5 and 6.

All measurements have been taken on an Intel Pentium IV system with 2,26GHz and 512Mb DDR-RAM. The graphics card was an NVidia GeForce 4, Ti-4400. All times have been measured when rendering into a window with a size of 640 x 640.

From now on, some test scenes and walkthroughs are needed that should possibly cover a great variety of different cases while their number should stay small for the sake of clarity. The following test scenes have been chosen:

- A scene containing 2500 teapots (total number of triangles: 5.760.000) being randomly distributed and oriented within an (invisible) box – see Fig. 3.12 for an example. This scene can be characterized by its high complexity, much cumulative occlusion, rather little temporal coherence (as reasoned in chapter 3.4.1.1), many objects being almost occluded (yet small parts still visible) and a high number of triangles per object (which makes an exact classification crucial).
- A part of a Vienna modelled with a very simplified geometry (e.g. the walls of buildings are flat) and an overall triangle count of 240.388. While the occlusion and temporal coherence are usually good when walking through the city, viewing from above tends to be problematic because this leads to a rapid increase in visible objects. In this scene, the average number of triangles per object is quite small.

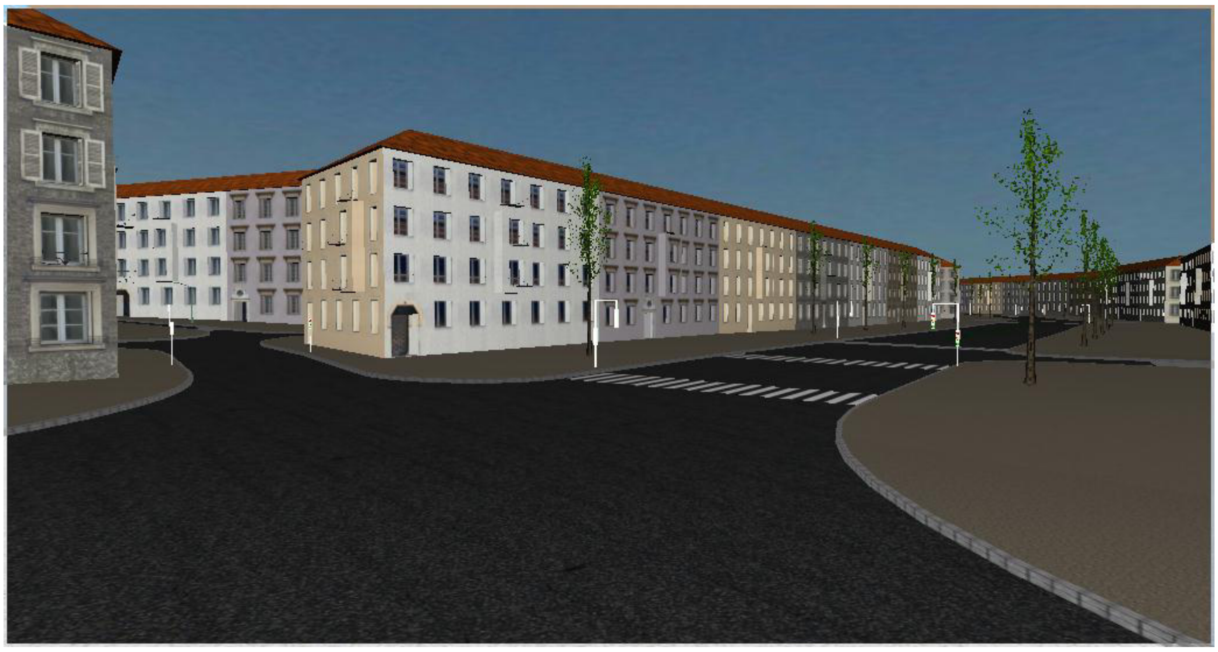


Fig. 3.14: A typical view of the city model.

- The same part of Vienna but modelled with a much more detailed geometry (1.042.378 triangles). While most properties apply to this scene as well, doing unnecessary rendering tends to be more expensive due to a higher average number of triangles per object. Fig. 3.14 shows a typical view.

- A terrain-scene (modelling the hills surrounding the town of Klosterneuburg). Since this landscape basically consists of two valleys, mentionable occlusion can only be encountered when looking from inside one valley to the direction of the other one. However, the overall number of triangles (87.424) as well as the average number of triangles per object are quite low which makes it quite difficult to attain significant performance increases by doing occlusion culling.

Within these four models, the following walkthroughs have been chosen:

- **Teapots**: Moving around in the teapot scene: Partly, the box is viewed from the side (as shown in Fig. 3.12) and partly standing in front of it (with much occlusion).
- **Simple City**: Walking through the simple city: Partly walking (about 2 meters above the ground) along some broad and some narrow streets, partly moving across an open square and – decreasing performance tremendously – partly making a 360° panorama rotation about 50 meters above the ground (higher than the average roof level). This kind of walkthrough shows dramatic peaks (in the detailed figures in chapter 5), because the amount of visibility is subject to very strong variations.
- **Normal City**: Exactly the same walkthrough in the city, but within the detailed city scene. For many applications, this is probably the most representative walkthrough.
- **City – No Occlusion**: Making the view wander around a place within the complex city model where (almost) no occlusion is present. This basically demonstrates the overhead incurred by the various approaches. Here, performance increases are almost impossible to achieve by occlusion culling itself without compromising the image quality. However, other means like implementing a more efficient traversal may lead to speed ups.
- **Terrain**: Flying along one valley in the terrain scene (below the tops of the hills), making a turn around one ridge into the other dale. At the end, the height is increased until we peek over some summits to some other hills – where only very little occlusion is present any more.

All measurements dealing with temporal values (e.g. frame times, times of CPU-stalls, etc.) have been taken using an according profiling mechanism of the YARE-engine, which is based on reading the current clock cycle counter of the CPU using the Pentium instruction RDTSC. In order to determine the time of CPU stalls, the execution time of the NV_occlusion_query extension function `glGetOcclusionQueryivNV()` has been measured when asking for the result. The reason for this is that stalls occur within this function if the result is not yet available, and the overhead of this function apart from stalls is negligible.

Generally speaking, for the sake of brevity, only the most striking aspects of the following figures are highlighted in the text – they essentially reflect what has been argued in the text. The following measurements will be shown in this chapter: The first figure (Fig. 3.15) compares the basic OG-based approach without improvements to the stop-and-wait approach and to rendering without occlusion culling. The figures 3.16 to 3.20 illustrate the separate effects of the various modifications to the basic OG-based approach, and Fig. 3.21 finally provides an overall performance comparison when applying all improvements simultaneously.

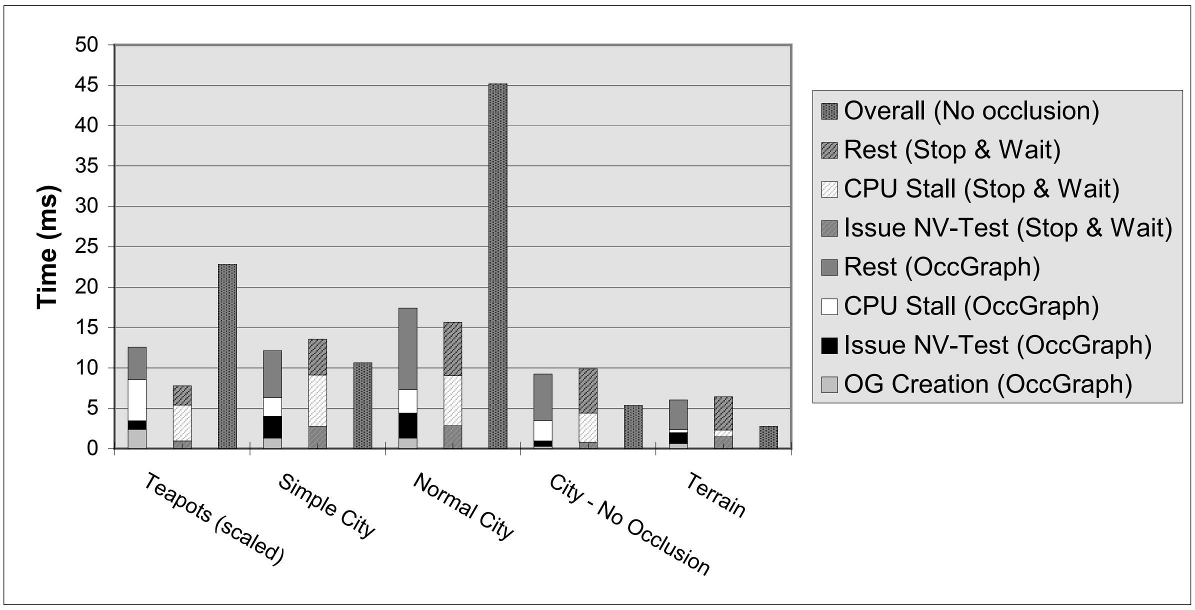


Fig. 3.15: Comparison of rendering without occlusion culling, applying the simple stop-and-wait approach and the basic OG-based approach (without any improvements) by decomposing the average frame times of all five walkthroughs (the times for the teapot scene had to be divided by 10 in order to make the values lie approximately within the same range).

The main aspect of Fig. 3.15 is to show that without improvements, the OG-based approach is sometimes worse than the simple stop-and-wait model: The performance gains by reducing the time of stalls can hardly outweigh all other costs. Besides, it demonstrates that speed ups by doing occlusion culling in general are more likely where in fact occlusion is present and the geometry exhibits a non-trivial complexity.

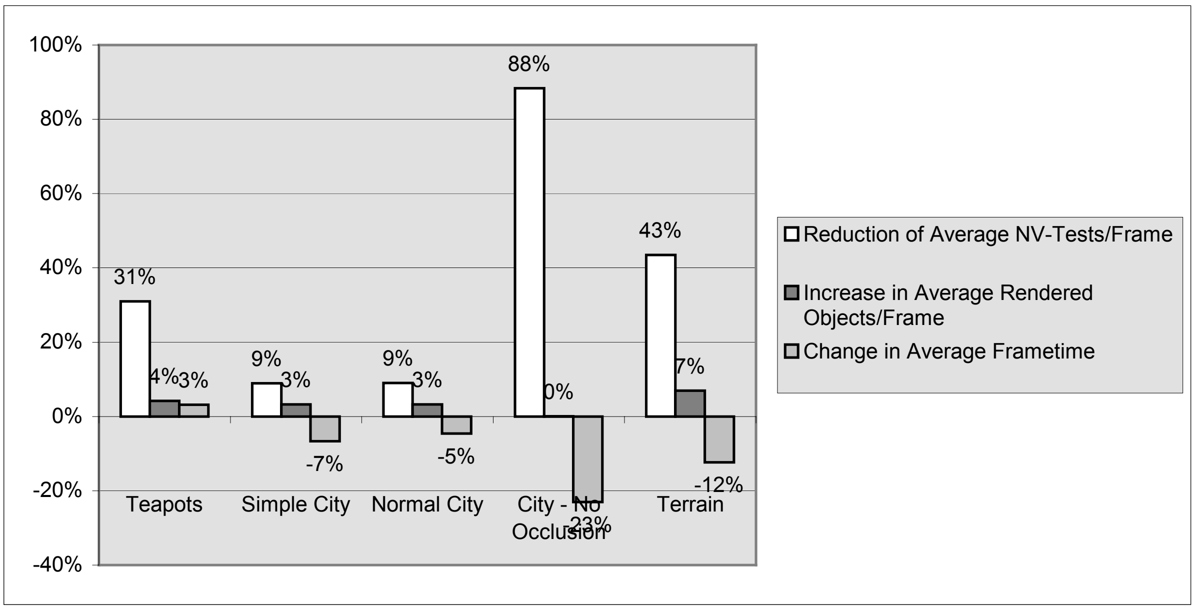


Fig. 3.16: The effect of *Assumed Visibility* at VF_{max} = 10 and VF_{min} = 1 compared to the basic OG-based approach (both without any further improvements): The average number of occlusion queries is reduced at the cost of rendering some occluded objects as well.

The essence of Fig 3.16 is that *Assumed Visibility* reduces the number of occlusion queries the more the higher the percentage of visible objects is compared to the overall number of objects within the view frustum. The costs seem to be moderate yet can nevertheless do harm if the additionally rendered objects are complex enough (like the teapots).

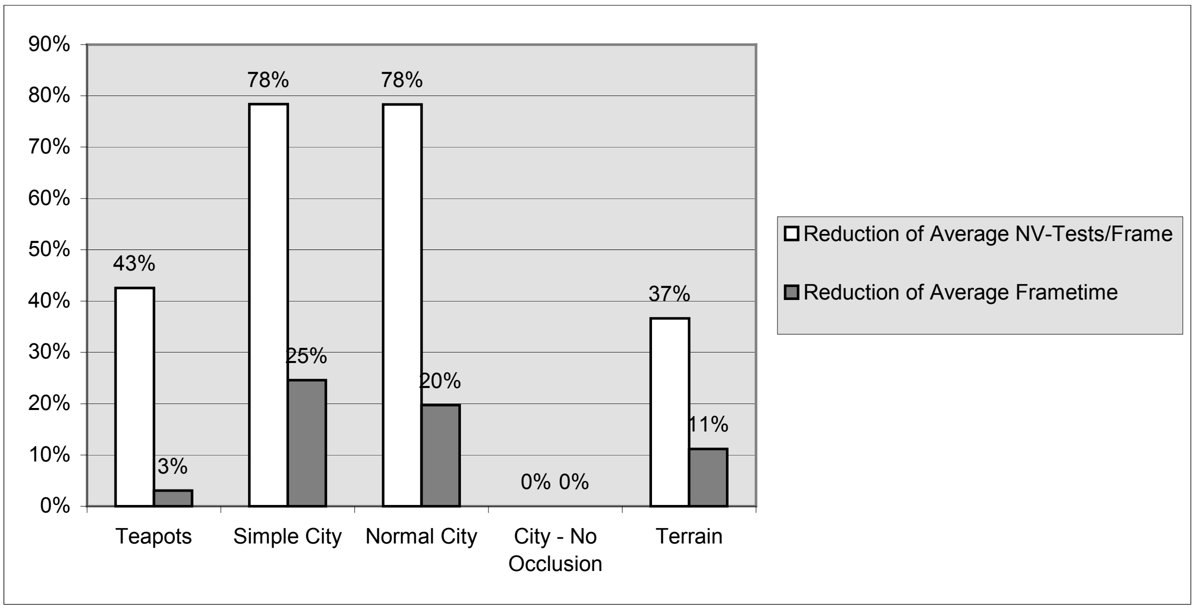


Fig. 3.17 The effect of *Early Rejection* (without further improvements): Both the number of tests and the frame times could be reduced.

Fig 3.17 shows that employing *Early Rejection* is a reasonable complement to *Assumed Visibility*: While the latter one optimizes the usage of occlusion queries for visible objects, *Early Rejection* succeeds to reduce the number of occlusion queries especially in cases of much occlusion. The increases of overall performance – particularly in the city scenes – are remarkable too.

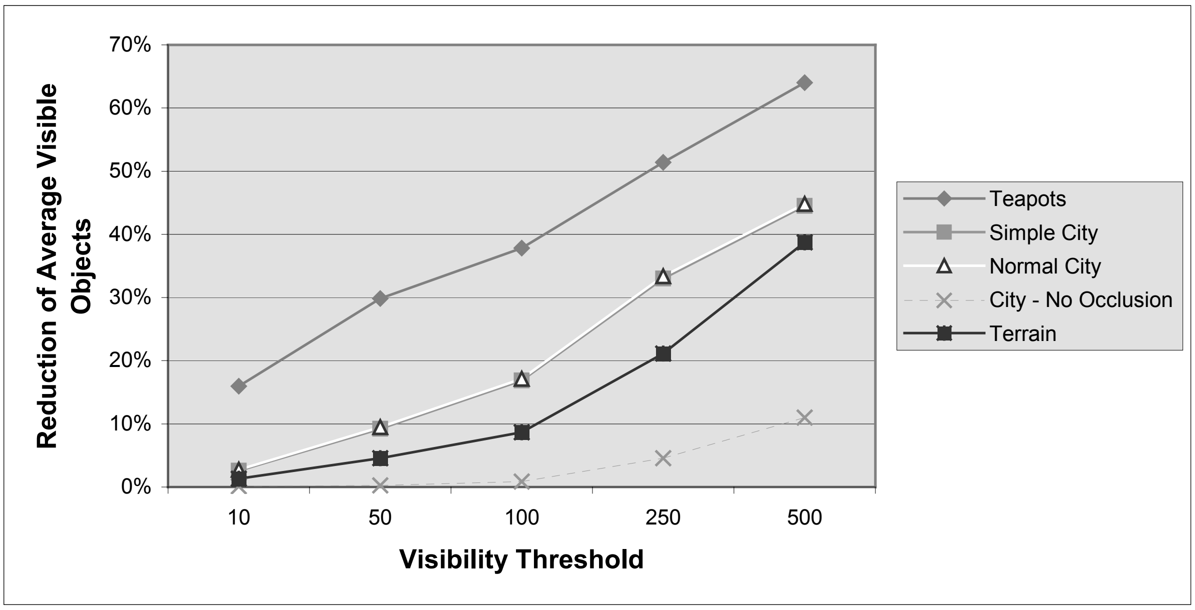


Fig. 3.18: The effect of various values for T_V on the average number of visible objects compared to rendering with a threshold of zero.

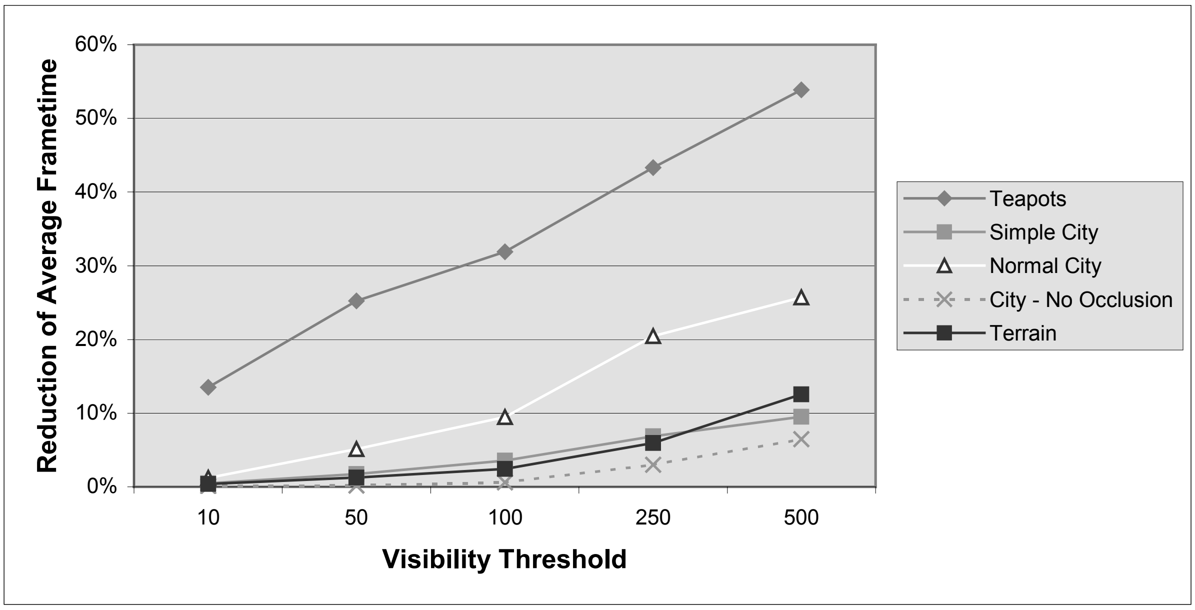


Fig. 3.19: The effect of various values for the visibility threshold on the average frame times compared to rendering with a threshold of zero.

Fig. 3.18 and Fig. 3.19 basically show that the benefits of a visibility threshold depend largely on the scene, but can be substantial where many objects are visible to a small percentage (like in the teapot scene). However, it must be emphasized that a threshold of 500 already led to ugly optical distortions in almost all scenes – and especially in the teapot scene.

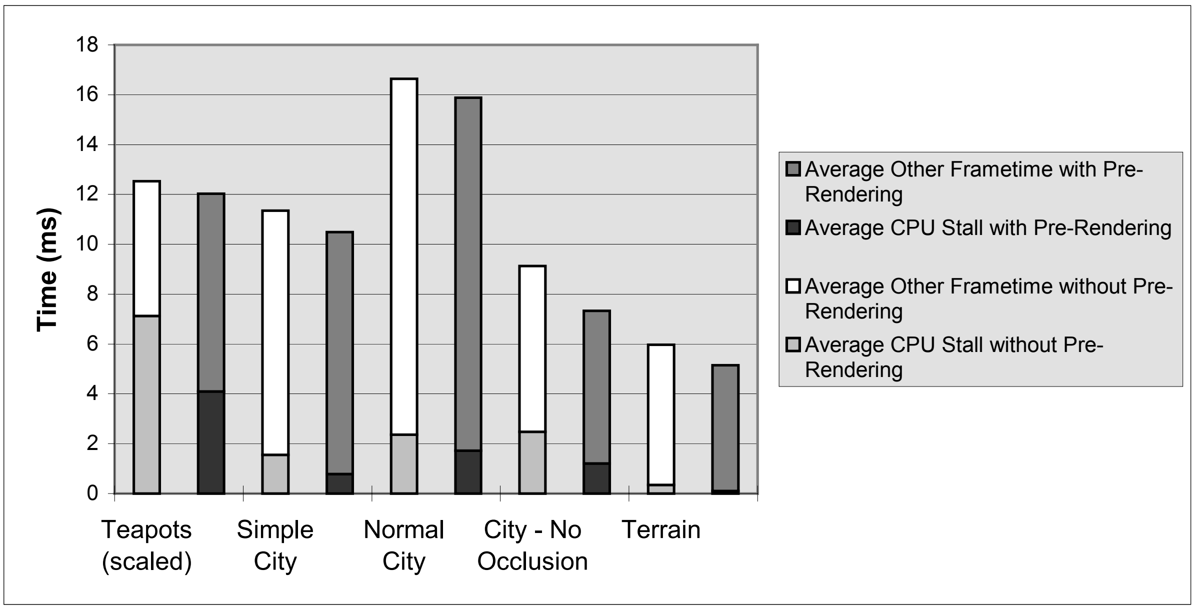
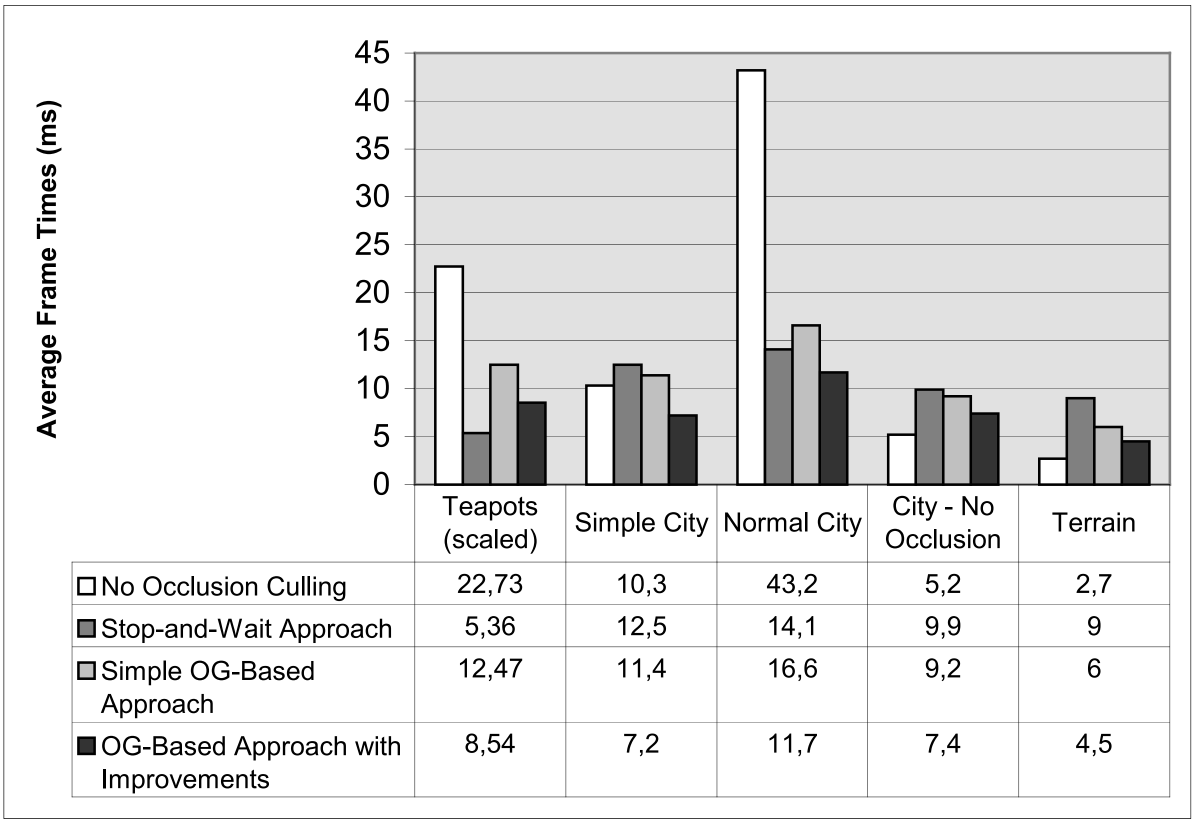


Fig. 3.20: The effect of *Pre-Rendering* compared to rendering without it. Both tests are measured with *assumed visibility* at $VF_{max} = 10$, but utilizing no other improvements and decomposing the average overall frame time.

Fig. 3.20 shows that *Pre-Rendering* improves the performance yet to a smaller amount than one might have expected, if done separately. However, it is more effective in combination with *Early Rejection* which reduces the overwhelming number of occlusion queries.



	Teapots (scaled)	Simple City	Normal City	City - No Occlusion	Terrain
□ No Occlusion Culling	22,73	10,3	43,2	5,2	2,7
■ Stop-and-Wait Approach	5,36	12,5	14,1	9,9	9
▨ Simple OG-Based Approach	12,47	11,4	16,6	9,2	6
■ OG-Based Approach with Improvements	8,54	7,2	11,7	7,4	4,5

Fig. 3.21: Comparison of the average performance of all approaches presented so far for all walkthroughs. Note that the times of the teapot walkthrough have been divided by 10 in order make all values lie approximately within the same range.

Fig. 3.21 demonstrates the average performance for the various walkthroughs. The improvements refer to $VF_{min} = 1$, $VF_{max} = 10$, $T_V = 50$ along with applying *Pre-Rendering* and *Early Rejection*. Note that these parameters are by no means optimal for all scenes: For instance, the results for the teapot scene could definitely be enhanced by a lower value for VF_{max}. Nevertheless, the same values were used for all scenes on purpose since this approach is meant to be of general nature and should yield good performance for arbitrary scenes without special tuning. For even more detailed results, refer to chapter 5, where exact plots of the varying performance throughout the various walkthroughs can be found.

It can be seen that basically two properties determine the success of the OG-based approach:

- The average amount of occlusion.
- The average object complexity.

Firstly, scenes with a considerable amount of occlusion are actually sped up remarkably (especially with the improved version) while the last two scenes suffer from any attempt of occlusion culling so far (in the terrain scene, this is more due to the low object complexity, though).

Secondly, the gains are greatest where the average object complexity is high. Yet especially in these scenes, the stop-and-wait approach performs unexpectedly well. This leads to the comprehensible insight that the more complex an object is, the more essential is it that is not rendered in vain.

In conclusion, the (non-hierarchical) OG-based approach together with the introduced improvements succeeds in increasing the average performance of scenes with enough occlusion. Although single enhancements as such might have had a quite disappointing effect, the accumulated improvement is evident and holds for all scenes. As it will be shown in the next chapter, it can still be enhanced for some scenes by making use of a hierarchy. However, its applicability as well as the optimal values for the parameters of the improvements depends to a great extent on the scene – even more precisely on the exact position within the scene as it is obviously the case for the city scenes. Besides, it should not be ignored that the whole approach with all improvements has grown quite complex which could prove too cumbersome to implement for a practical application. Finally, it is a reasonable approach that helped to gain many insights and can even be a good choice in rare cases as reasoned in chapter 5. Yet for the vast majority of scenes it does not yield the optimal performance possible with the NV_occlusion_query extension.

4 Hierarchical Occlusion Tests

While the non-hierarchical OG-based approach as presented in chapter 3 succeeds under certain circumstances to speed up the rendering process, it is generally speaking not good enough. Despite of all improvements, the main problem is still a big overhead that grows linearly with the number of objects, and the fact that many objects simply consist of too few triangles to justify an own occlusion query.

This chapter introduces spatial hierarchies being a well-known way to accelerate all sorts of computations in computer graphics. After motivating this step, techniques for creating spatial data structures being appropriate for our purpose are compared and assessed. Then it is demonstrated how to incorporate them into the existing OG-based approach. Finally – yet probably most important – an alternative, comparatively simple approach not relying on any form of OG is presented that will turn out to be superior in many cases.

4.1 Motivations and Considerations

While the number of occlusion queries could be significantly reduced by assuming objects as visible for several frames (*Assumed Visibility*) and rejecting definitely occluded objects without test (*Early Rejection*), essential problems still remain:

- Constructing the OG for many objects takes much time.
- Many objects tend to be too simple to justify their own occlusion query.

The first problem was partly solved by the introduction of *Pre-Rendering*, where the construction of the OG proceeds in parallel to GPU rendering. However, performing the necessary steps for several thousand objects is still a considerable overhead for the CPU that we must seek to reduce.

As pointed out in chapter 3.3, an occlusion query is the more expensive compared to straightforward rendering of an object, the less complex the respective object is. Therefore the complexity of an object to be tested should exceed a certain triangle count in order to make occlusion queries efficient. On the other hand, simply rendering all objects with less than a certain number of triangles without testing them first can be no solution, since:

- Apart from being system-specific, an optimal value for this threshold is likely to vary with the current state of the command buffer and other traversal-related details – computing a reasonable threshold would therefore be difficult if not impossible.
- Automatically assuming simple objects as visible could interfere with techniques that seek to determine the visibility of objects by exploiting spatial coherence (like *Visibility Propagation* in [Bitt01b], or *Early Rejection,* as described in chapter 3.4.1.2): Actually occluded objects would have to be considered visible just due to their (small) complexity and could therefore make occlusion tests of adjacent objects necessary that could have been avoided otherwise.
- If a scene consists only of such simple objects, doing no occlusion culling is not an option.

Spatial hierarchies tackle both problems mentioned above: Informally put, by combining several objects which are close together to one elementary cell, we achieve both a reduction of the number of nodes contained in the OG and an increase of the average number of triangles per item, and doing so hierarchically permits a flexible adaptation to actual visibility.

However, this comes at the cost of a considerable rise of (original) objects being classified as visible.

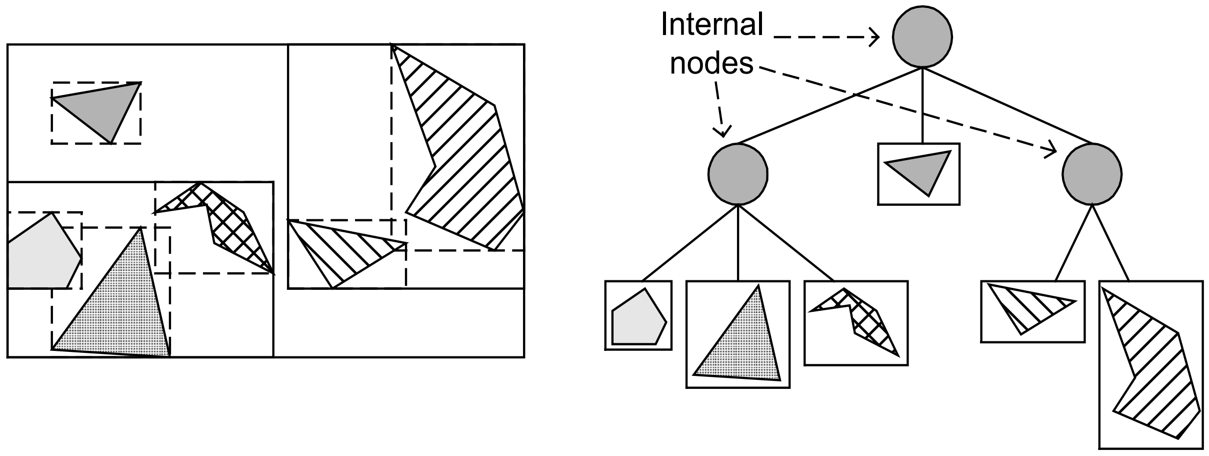


Fig. 4.1: A simple scene consisting of six objects and a way to organise their AABBs hierarchically.

Spatial hierarchies subdivide a certain n-dimensional space (3-dimensional in our context) in a recursively nested manner and are modelled using trees as data structure: Each node contains a subset of the space of its parent and its own space is subdivided further by its children. Therefore, the number of nodes grows exponentially with the depth within the tree. The actual geometry is usually stored within the leaves of the tree. Each node, including leaf nodes, has a bounding volume (an axis-aligned bounding box – abbreviated AABB – in our case) that encloses the geometry in its entire subtree – consequently, the root node contains the whole scene. The advantage is that every kind of traversal can stop at nodes where the criterion under examination (visibility in our context, yet also other properties like intersection with a ray are common) is not fulfilled for the whole space of the according bounding volume – the subtree does not need to be traversed any more. This way, the effort for queries can typically be reduced from O(n) to O(log n), provided that the tree is approximately balanced. One possibility of a spatial hierarchy is illustrated in Fig. 4.1.

Commonly desirable properties of spatial hierarchies are:

- The tree should be approximately balanced.
- It should fit the objects of the scene tightly. In other words, the surface area and the volume of the AABBs of internal nodes should be minimal.
- The depth of the tree should match the size and complexity of the scene in a way that the coarseness of the subdivision diminishes remarkably with increasing depth, but the tree does not become too flat to foil the benefit of logarithmic effort for traversals .

Such hierarchies are commonly used in computer graphics in order to exploit spatial coherence. In the context of the OG-based approach, we will use them as an alternative to *Early Rejection*, being one proposal presented in chapter 3.4.1.2 to exploit spatial coherence. Concerning occlusion culling, this implies that several objects that are close to each other tend to be classified alike. Instead of issuing a separate occlusion query for each object, we can test a bounding volume enclosing several objects. If this test indicates occlusion, we can skip the contained objects altogether. If parts of the bounding volume are visible, we are free to decide what to do next: Refine the result by testing more, yet smaller bounding volumes, or render all objects in the sub-tree without testing. Hence, by using a spatial hierarchy, we can choose the coarseness of the subdivision for certain parts of the scene and continuously adapt it in order to match the circumstances given by visibility. This principle is illustrated in Fig. 4.2: It shows the parts of the geometry of a city that are classified as visible, and a corresponding subdivision (AABBs rendered in wire-frame mode): orange boxes are considered visible, gray boxes are classified as invisible. The point is that the subdivision of the visible parts is much more fine-grained compared to the coarse subdivision of the occluded space. However, further details of this image like the exact kind of hierarchy applied are subject to later chapters.

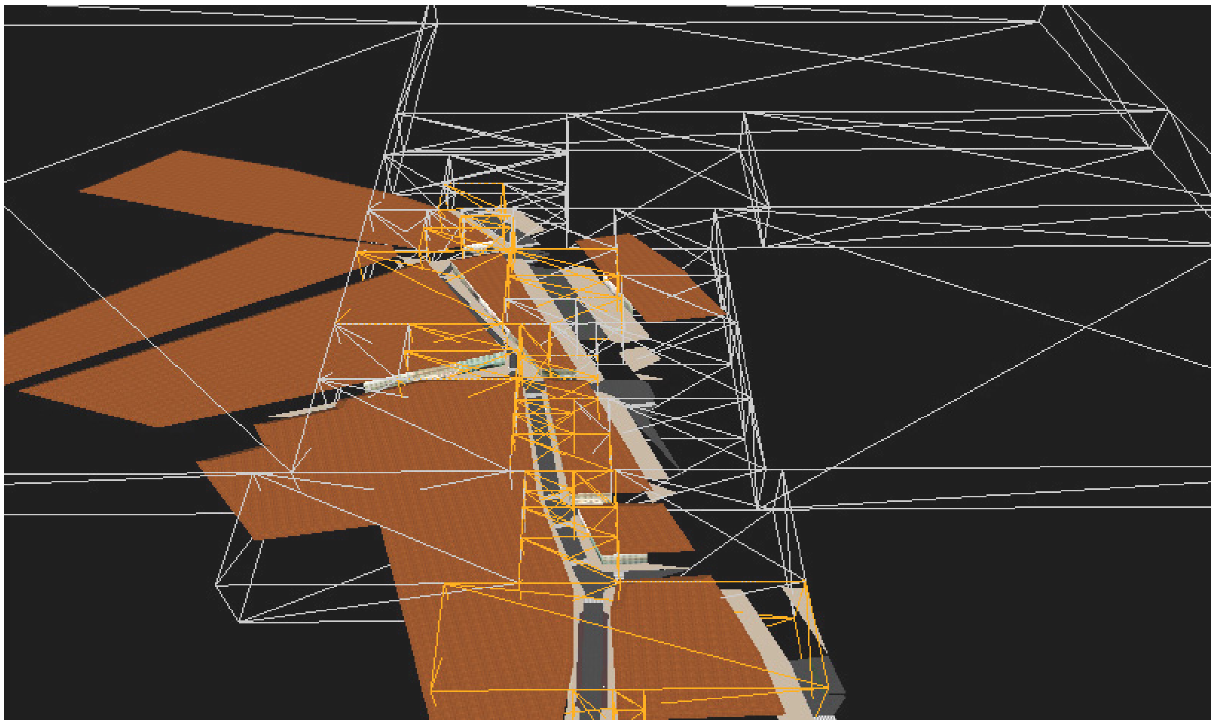


Fig. 4.2: An example for a subdivision of space: orange boxes correspond to visible nodes, the much coarser grey nodes indicate occlusion.

While the principles described so far apply to almost every spatial data structure, the details differ remarkably: A criterion to distinguish spatial data structures is whether they subdivide the space regularly (e.g., an octree) or irregularly (e.g., BSP trees), somewhat compromising simplicity with flexibility. A further aspect is whether the construction is done bottom up (simple objects are successively merged) or top down (the whole scene gets repeatedly broken down to increasingly smaller parts). Even considering only one type of hierarchy, different heuristics and values for thresholds are possible which influence its final properties. A good introduction to spatial data structures can be found in [Möll02], an even more detailed comparison can be found in [Same89].

In our context, incorporating spatial data structures into the OG-based approach does not only mean choosing an appropriate kind and finding reasonable heuristics for the construction that really succeed in reducing the number of objects (which will be dealt with in the next chapter), but also implies significant modifications to the traversal of the OG which will be subject of chapter 4.3. Occlusion tests in a hierarchical setting are also used by [Meiß99] in an approach which shows some similarities to the one we are going to develop in this chapter, yet they do not exploit temporal coherence within the hierarchy.

4.2 Creating Spatial Hierarchies

Since the form of the actual geometry of the objects is quite irrelevant for the OG-based approach, the creation of the hierarchy will only take into account AABBs of objects. Therefore we look for an approach that splits space in an axis-aligned manner. A further criterion is the time needed to create the hierarchy: Although it might be important to point out that the process of the creation does not need to fulfil real-time requirements, since it is done only once when a new scene is loaded, it should still not exceed 10 to 15 seconds in order to avoid annoying delays at start up.

Two different methods are presented: A bottom-up approach resulting in general *AABB hierarchies* and a top-down approach creating *axis-aligned BSP-trees* – also called *kd-trees*. These techniques will be assessed by means of the properties of the resulting tree. Although *octrees* are another famous spatial data structure, they are not considered here due to their similarities to kd-trees (which are basically a more general, irregular form of octrees). Polygon-aligned BSP-trees coincide in our case with axis-aligned BSP-trees, since the AABBs we use are per definition axis-aligned.

4.2.1 A Bottom-Up Technique

Initially, a set of independent objects is given, each one having an AABB assigned. An obvious way to organize them hierarchically is to combine adjacent AABBs successively until one AABB contains the entire scene. This is a typical bottom-up process for creating hierarchies.

Before describing some of the involved considerations, I want to refer to [Bare96] for another bottom-up approach, called *BOXTREE*. Besides, [Gold87] presents methods for the evaluation and automatic creation of general bounding volume hierarchies by incremental tree insertion. This chapter describes a different bottom-up approach. However, anticipating the results, the obtained trees do not match the requirements of chapter 4.1. Therefore, this method has finally been rejected in favor of the top-down approach presented in the next chapter. Still I want to outline the approach briefly.

Two questions guide the construction:

- Which objects should be merged next?
- How should they be merged?

The first question addresses the detection of two objects that can be tightly combined by a common AABB. Once a pair of nodes has been chosen for the next merge operation, the second question deals with the decision if one node should become the child of the other, or a common parent node should be introduced.

The approach discussed here repeatedly picks two nodes and merges them until only one overall node is left. Let *T(N)* denote the number of triangles contained by the subtree of the node *N* and let *S(N)* be the surface area of its AABB. Then

$$D(N) = \frac{T(N)}{S(N)}$$

expresses something like the density of triangles contained by the node. Note that it is not a density in the common sense, though, since the term ‘density’ is usually defined as a division by a volume. However, it has turned out that dividing by the surface is more suitable, because troubles with flat objects can be avoided this way, which would have a volume of zero.

Given two arbitrary nodes A and B, $A \neq B$,

$$Q(A,B) = \frac{D(A+B)}{D(A)+D(B)}$$

is a measure how much the density will decrease for a common AABB: Possible values for Q lie within (0, 1], where greater values mean that the density will diminish less (1 as result means that both AABBs coincide). According to chapter 4.1, the spatial hierarchy should fit the objects of the scene tightly. Since losing density is equivalent with approximating the objects less tightly, it is preferable to combine those two nodes which maximize Q. Determining the pair that maximizes Q basically implies comparing each node to each other node, but the quadratic effort of this search can be avoided if we insert all nodes into a *regular grid* and consider only nodes within the containing and neighboring cells.

The next step is to perform a merge of the selected nodes. While the heuristic for choosing two nodes works quite well, no method seems to be really satisfactory, since all applied heuristics fail to create trees with properties as requested in chapter 4.1. One way is to simply introduce a common parent node for each pair. This usually works well at the beginning but gradually one node turns out to dominate, growing bigger by swallowing most of the smaller nodes, yielding quite unbalanced trees. Enforcing a balanced tree during the construction impaired the tightness of the bounding volumes, on the other hand. This approach was finally rejected in favour of the simpler top-down techniques as described in the next chapter.

In general, while these flaws may be solved somehow, bottom-up approaches tend to generate trees with many overlapping nodes, which is usually undesirable, and they do not permit to sort the nodes implicitly as is possible with spatial subdivisions like the kd-tree.

4.2.2 A Top-Down Technique

Top-down techniques create hierarchies by repeatedly subdividing space. The first step is usually to compute a bounding volume (an AABB in our case) for the whole scene. This volume gets split according to some considerations into two or more parts. Each part gets subdivided again yielding an even more fine-grained decomposition, until a node is considered simple enough.

The majority of hierarchy construction algorithms are top-down approaches. This discussion focuses on the specific type of *kd-trees*. This decision is due to several advantages of kd-trees (partly according to [Havr00]):

- Kd-trees allow flexible positioning of the splitting planes, which results in various sizes of the elementary cells. These cells adapt well to the geometry of the scene.

- Unlike the more general bounding-volume hierarchies, kd-trees contain no overlapped elementary cells.
- Approximate depth-sorting can be done implicitly by traversing the tree appropriately.
- Kd-trees can be used to topologically model many other spatial subdivisions: for instance uniform and non-uniform grids and octrees can be seen as special cases of kd-trees.
- It is theoretically possible (although of no practical use for our purpose) to extend kd-trees for a space of arbitrary dimension.
- The underlying data structure of kd-trees is the well understood and simple binary tree.

Summarizing, kd-trees offer a good compromise between flexibility and simplicity and therefore they have been chosen as hierarchy for the two hierarchical approaches presented in this chapter. Fig. 4.3 depicts a typical kd-tree: Note that the triangle in the bottom left belongs to both C and D, thus it is contained by both leaves. Furthermore, note that the decomposition of space as shown in Fig. 4.2 is a kd-tree as well (although parts were culled away by view-frustum culling). For more elaborate examinations on kd-trees than given by this theses, refer to [Same90] and [Havr00].

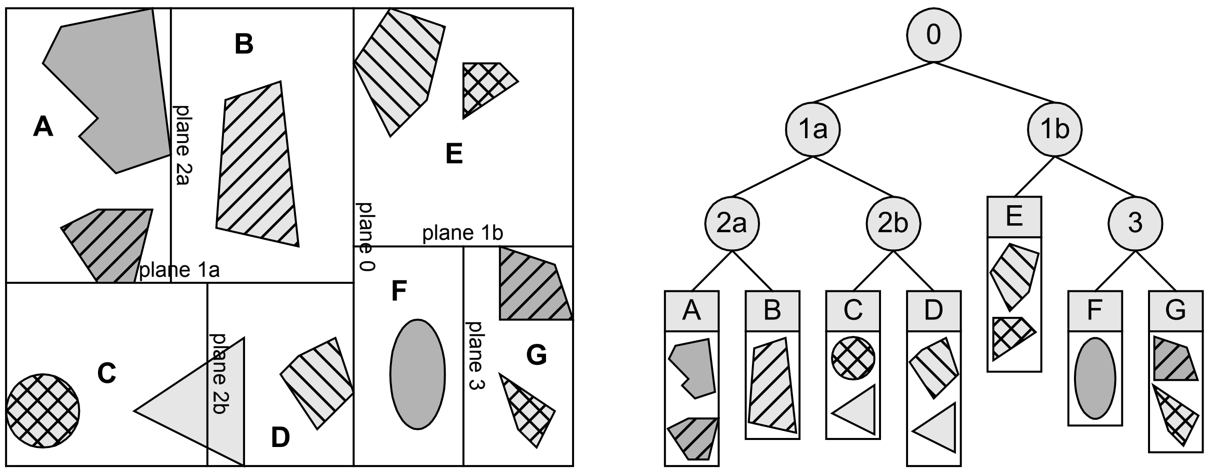


Fig. 4.3: A constellation of objects being subdivided and the resulting kd-tree.

Basically, the creation of kd-trees follows a pattern that can be outlined this way:

- Decide if the current node is simple enough and terminate if so.
- Select an axis at which the n-dimensional box is to be split.
- Compute the exact position of the split within this axis.
- Classify the geometry as part of the left or right (or possibly both, see below) half.
- Proceed with both halves recursively.

Let us have a closer look at the various steps: First of all, a test decides whether the node needs further refinement or if it is already simple enough. Several factors might be taken into consideration, for instance the volume of the node, its surface area, its complexity in terms of contained triangles and the depth within the tree. The implementation of this master thesis uses the following thresholds:

- *TKd_O*: Threshold for the number of scene-graph objects referenced by a node. It ensures that objects being both big and complex are still not subdivided further, which would be useless, as they get rendered as a whole anyway.

- *TKd_T*: Threshold for the sum of the number of triangles of all objects referenced by a node. It prevents nodes from getting too simple and thus makes sure that occlusion queries can pay off.
- *TKd_S*: Threshold for the surface of the AABB of that node. This value is not specified absolutely, but in terms of a percentage with respect to the surface of the AABB of the whole scene (e.g. 0,05%, defined as 1/2000). This threshold restricts the minimal size which is necessary to ensure the termination in some tricky constellations.

If any actual value of a node drops below the respective threshold, the node is considered as simple enough and thus not split any further. Note that these thresholds have a tremendous effect on the behaviour and performance: The higher the values, the less nodes will result from the creation process. This may reduce the effort for some computations, yet a coarse subdivision may have negative impacts (in our case for instance, all objects of a leaf which is considered as visible are classified as visible as well, which may cause a significant increase in the number of rendered objects – see the chapters 4.3.3 and 5 for detailed figures).

Common strategies for selecting the axis of subdivision are:

- Cycle through all axes.
- Always split the largest side of the box.

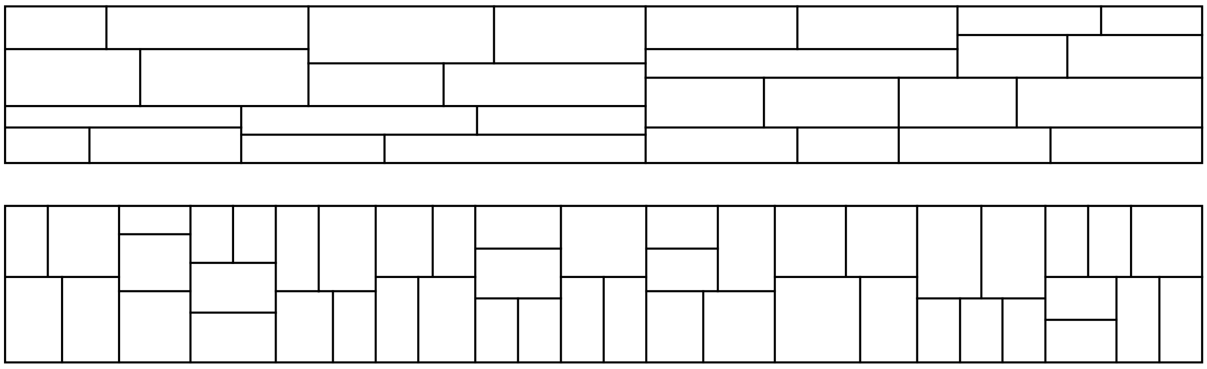


Fig. 4.4: A strictly cyclic axis selection (above) versus selecting the largest side (below).

The second strategy was used within the implementation due to cases as shown in Fig. 4.4: If the initial box is not approximately a cube, obstinately cycling through all axes may yield rather non-uniform cells. This is undesirable since objects far from each other might get tested together. However, splitting the largest side of the box requires the storage of the axis with each node.

Computing the exact position of the split plane is a decisive and computationally expensive step: The applied heuristic should seek to create two approximately equally complex parts, but the ultimate goal is to generate trees which conform to the criterions stated in chapter 4.l. In addition to the factors relevant for the termination criterion, the formula could try to minimize the number of objects that get intersected by the split plane, since these objects are referenced by multiple nodes, which can be undesirable. The implementation for this master thesis permits to choose between two heuristics – in both cases, potential splitting planes are the parallel planes of all AABBs lying within a certain band around the center of the box:

- $q =$ number of objects intersected by the split plane.
- $q = S_L * N_L + S_R * N_R$ with $S_{L/R}$ being the surface of the left and right part, respectively and $N_{L/R}$ being the number of objects in the respective part.

The first heuristic stresses simplicity and can be computed quickly. The second heuristic originates from the field of ray-tracing, where it was proven to be a good choice as described by Havran [Havr00] (the theoretical background can be found there as well), and it has been applied in order to assess if it works well in the context of occlusion culling too. Either way, the splitting plane with the minimal value for q is chosen. As illustrated in the results section, both heuristics show approximately the same performance most of the time.

The next step is to assign the geometry to the respective parts. The classification is obvious for all items entirely contained in one part. However, although it is undesirable, it is usually not avoidable to intersect objects which overlap both parts. One strategy is to split the geometry itself down to the triangle level so that each triangle uniquely belongs to one cell. In our case, though, objects are considered as atomic, which means that instead of splitting them, they might be referenced by multiple nodes (like the triangle in Fig. 4.3): An object gets rendered if any cell containing it is visible – the implementation has to take care that this rendering is done only once per frame. Nevertheless, the fact that these objects will overlap multiple cells does not affect the AABBs of the cells: They do not need to contain all referenced objects entirely, only the space of the cell itself. The parts of an object lying outside the AABB of one cell are within the AABBs of other cells. Therefore, assuming visibility of the cell D in Fig. 4.3 and occlusion for C, the triangle contained in both nodes will be rendered completely while the other contents of C get culled.

Despite the advantages listed above, kd-trees have disadvantages as well:

- Adjacent objects can be in completely different branches of the tree.
- The AABB of a cell can be a too conservative approximation of its contained objects.

The first problem applies to all top-down approaches: Once two objects lie in different parts, they are handled separately, even if they are in fact very close together and thus combining them would be advantageous. The earlier in the construction process such adjacent objects are separated, the worse. One can only attempt to tackle this by constructing sophisticated heuristics that take such considerations into account.

The second point refers to situations as shown in Fig. 4.5: The space actually occupied by the objects is small compared to the AABBs of the cells. Since the probability of an AABB to be classified as visible increases with its size, we should seek to minimize them. This can be done easily by storing two different AABBs for each cell: One that contains the whole space as usual and one that is basically the intersection of the cell with an AABB around all of its objects. This is illustrated by the dotted lines in Fig. 4.5. The cost for this tighter match is the fact that modifications to the set of objects managed by the tree will generally also invalidate several AABBs which can cause expensive recalculations.

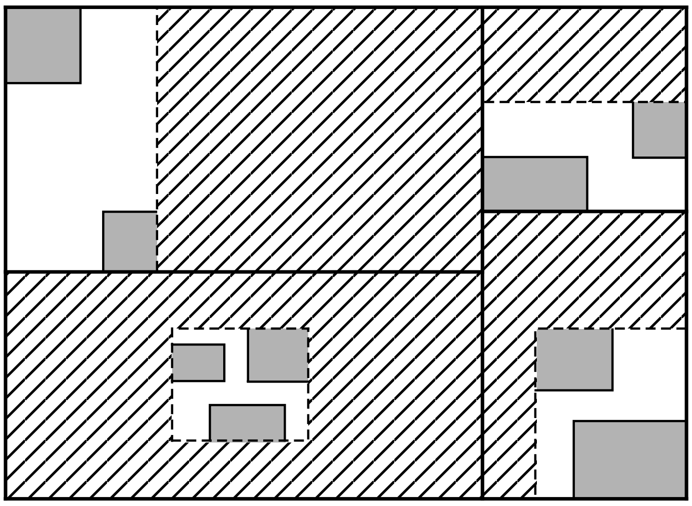


Fig. 4.5: A kd-tree with tight AABBs: The hatched space can be saved if not the cells themselves are used as AABBs.

4.3 Occlusion-Graph Based Approach

So far, we, we have motivated the introduction of a hierarchy and have reasoned why kd-trees are the data structure of choice. This chapter explains how a kd-tree can be incorporated into the OG-based approach, while chapter 4.4 will present an independent approach which is based on kd-trees as well.

As mentioned above, the creation of the hierarchy is a separate step which is done only once when a new scene is loaded (at least for the static geometry). Therefore, this chapter assumes that the kd-tree already exists and focuses on those task which are done each frame:

- Traversal of the kd-tree.
- Traversal of the OG.

Generating the OG is still done each frame anew, but this procedure is not affected by the introduction of a hierarchy apart from the fact that it gets created out of kd-tree nodes.

4.3.1 Traversal of the Hierarchy Using Temporal Coherence

Originally, the whole scene graph itself had to be traversed to gather all objects of interest. As shown in appendix A, arbitrary data structures can be used to support the scene graph – a simple list there, and a kd-tree here. While traversing a list is a trivial task, this is different for kd-trees, since all of its properties should be exploited as much as possible: As mentioned above, kd-trees allow for the sorting to be done implicitly. Another issue is view-frustum culling, which can also be enhanced by exploiting the hierarchical setting. Finally, the purpose of the hierarchy is to combine several nodes, which implies that the traversal does not go down to the leaves each time. How this can be attained will be a central question. Summarizing, we have to deal with three issues:

- Obtaining a sorted set of nodes by traversing the kd-tree appropriately.
- Optimising view-frustum culling for a hierarchical setting.
- Improving the traversal by exploiting coherence.

As described in [Möll02], we can achieve a rough front-to-back sorting by traversing the tree as follows: Each node must be aware which axis has been split (if any at all). For an arbitrary internal-node, the traversal continues on the side of the split plane where the viewpoint is located. When encountering a node where the traversal should stop (assume for now that this only applies to leaves, yet later in this chapter, this can also be the case for certain internal-nodes), we append the respective node to the end of the set of objects for which the OG is to be constructed. An example for such an order is given in the left half of Fig. 4.6. Note that this order is independent of the viewing direction.

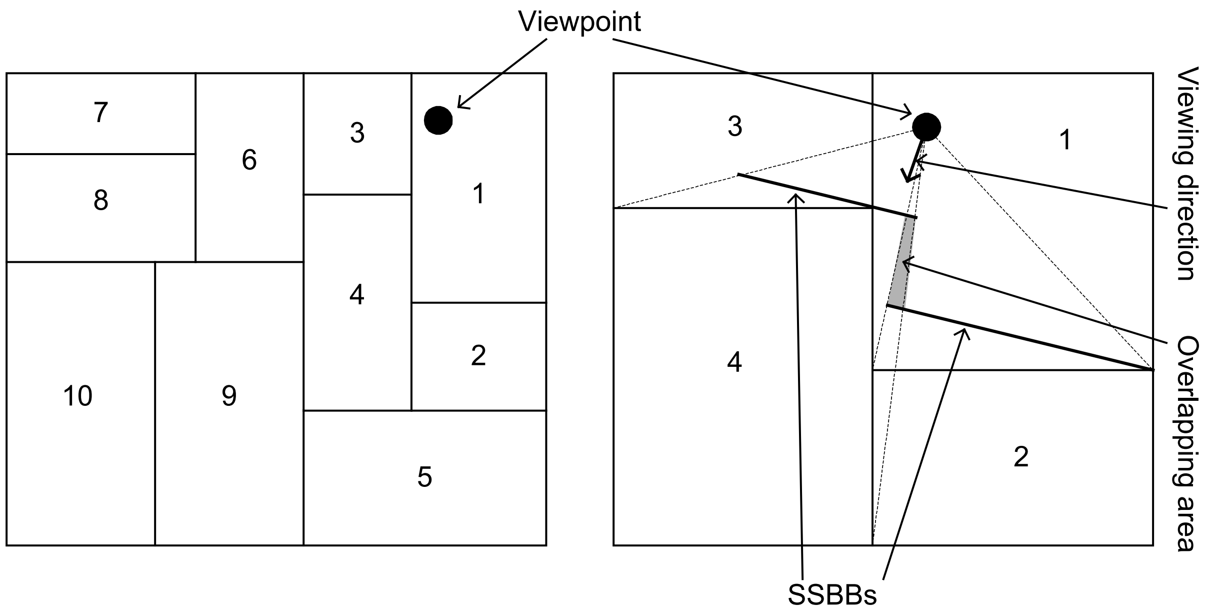


Fig. 4.6 : The left half shows a viewpoint-dependent ordering of the cells; the right half illustrates that although the order is not strictly front-to-back, earlier cells will never be occluded by later ones.

This algorithm does not yield an exact depth-sorting; for example, in Fig. 4.6, some parts of cell number 4 are definitely closer to the viewpoint than any point of cell number 2. Although it was said in chapter 3.2.2.2 that all objects must be sorted by their distance from the near plane, this does no harm: As shown in the right half of Fig. 4.6, it is guaranteed in such a case that later objects will never occlude earlier ones. More precisely, even if the screen-space bounding box (abbreviated SSBB) of a later node (the fourth node in the right half of Fig. 4.6) has a smaller depth value than the SSBB of an earlier node (the second node in this example), for those parts where they overlap, the earlier cell will lie in front of the later one – thus, we will get a correct order for the inverse Z-buffer (IZB) and for the occlusion queries.

The next issue is view-frustum culling within a hierarchical setting. A good reference about this topic is [Assa99], therefore, I will merely briefly outline those suggestions which have been considered in the implementation. As a starting point, the equations of all six planes forming the viewing frustum in world space must be extracted from the parameters of the camera or the OpenGL projection matrix (see [Morl00], how this can be achieved). The following points are some relevant aspects and possible improvements when doing view-frustum culling for a hierarchy of AABBs:

- The most basic approach is to test all 8 vertices against all 6 clip planes, thus one object might need up to 48 tests for a complete classification. To test a vertex, it is inserted into the respective plane equation and the sign of the result tells whether it is in front of it or behind. If all eight vertices are in front of all planes (assuming inward-looking planes) the AABB is completely within the view frustum, if all vertices are behind any plane, it lies completely outside and can thus be culled. Otherwise, it intersects the view frustum.

- The number of tests can be reduced significantly since it is sufficient to test two vertices per plane: Given the normal vector of the plane equation, one can easily compute those two vertices forming the diagonal of the AABB that is most closely aligned by checking the signs. The so called p-vertex has the greatest signed distance from the plane, whereas the n-vertex has the smallest signed distance of the whole AABB (the latter one is also used as distance of the according SSBB for the occlusion queries).
- Temporal coherence can be exploited by the so called *plane-coherency test*: If an AABB fails the test for a certain plane, it is likely that it will fail the test for the same plane the next time as well. Beginning with this plane the next frame, we have a good chance of classifying an AABB as outside with the effort of a single test.
- If the AABB of a certain node is entirely in front of a clip plane, this is also valid for the AABBs of all nodes within the whole subtree. Therefore, we can exploit the hierarchical setting by passing a mask (implemented as a bitfield) from the parent to its children indicating which planes can safely be ignored.

Finally it must be explained how we can make use of temporal coherence. This concept is not only significant for the hierarchical OG-based approach, it is also crucial for the approach presented in chapter 4.4. These ideas are closely related to [Bitt01b].

So far, we make use of the hierarchy only for view-frustum culling, yet all nodes not culled away are traversed down to the leaves. This yields a great number of objects which must be considered by the OG, even if the majority turns out to be actually occluded. Now, the idea is to stop the traversal at internal nodes which can already be classified as definitely invisible, thus skipping the traversal of the subtree. Unfortunately, this does not work for entirely visible internal nodes as well, since – unlike occlusion – the visibility of an internal node says nothing about the visibility of its sub-nodes. For this reason, we do not distinguish between entirely visible and partially visible nodes (unlike [Bitt01b]).

The most basic approach (from now on referred to as *basic hierarchical approach*) for doing occlusion culling in a hierarchical setting can be outlined as follows (note that this method is not directly suitable for the purpose of gathering nodes for the OG, yet it is described anyway for the sake of comparison and since it will gain relevance in the next chapter): Once a node has passed view-frustum culling, it is tested for occlusion (with an occlusion query in our case). If it passes this test as well, these steps are repeated first for the child closer to the viewpoint (as described above for the sorting) and after this subtree has been completely traversed, for the other child – visible leaves simply render their geometry. Since this basically boils down to a roughly front-to-back sorted stop-and-wait testing of all nodes within the tree, it may incur significant latencies. However, lacking further information about the nodes like for instance an OG or a previous visibility classification, it is the only possible method to traverse the tree without risking the loss of any occlusion.

The way how the kd-tree is traversed by the hierarchical OG-based approach is superior to the basic hierarchical approach, as it exploits temporal coherence: It maintains a set of nodes (from now on referred to as '*cut*'), which are exactly those where the traversal stops and consequently the ones for which the OG gets constructed. This cut comprises all internal nodes where the whole subtree is classified as invisible, and visible leaves. Instead of issuing occlusion queries for all nodes encountered during traversal, the algorithm collects all nodes of the cut passing view-frustum culling and passes them on to the generation of the OG. When the viewpoint changes, this cut will typically have to adapt as well: Some previously occluded internal nodes may become visible – thus the cut moves down the tree, we call this situation a *pull down* – whereas other nodes become occluded and might be combined on a higher level – then the cut moves upwards, which is accordingly called *pull up*. Initially, the cut is assumed to consist of all leaves, but this typically changes after the first frame. Temporal coherence is exploited in so far as the classification of the previous frame is used to construct the cut for the current frame. Fig. 4.7 depicts a tree along with the respective classifications of all nodes and the according cut and also demonstrates a pull-up and a pull-down for a changed classification.

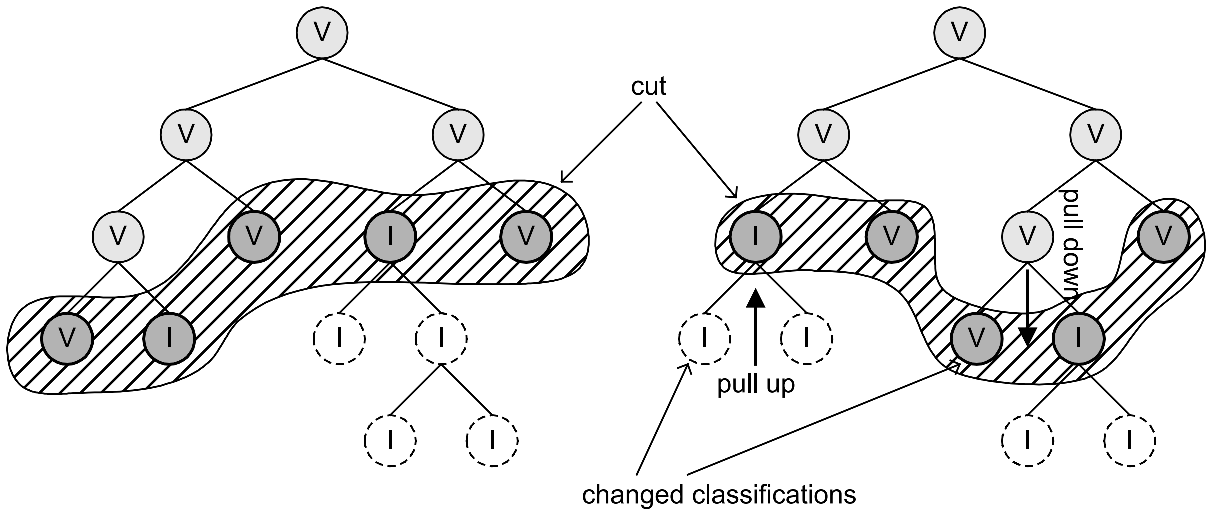


Fig. 4.7: A tree where all nodes are classified as either partly visible (V) or invisible (I) and the resulting cut. The right tree demonstrates the effect on the cut when the classification changes. Note that light grey nodes are merely traversed but not occlusion tested, and dotted nodes are not even traversed, only nodes belonging to the cut are both traversed and tested.

Note that the cut need not be stored separately – it suffices when nodes are aware of their former classification.

This paragraph describes the realization of pull ups, which is partly done during the traversal of the kd-tree and partly within the traversal of the OG. Within the kd-tree traversal, the visibility classification of the previous frame is considered for each traversed node in order to determine the cut for the current frame. Afterwards, each traversed node is assumed to be proven invisible within the current frame (at this place regardless of the actual visibility) which is basically equivalent to – temporarily – pulling the cut of the subsequent frame (not to be confused with the cut of the current frame which has been determined by then and is not lost by doing so) up to the root, being the most radical pull up possible. This obviously wrong classification is corrected during the traversal of the OG: The visibility of all nodes of the cut of the current frame is determined. Each node being proven visible sets all its parents within the kd-tree up to the root visible as well, thus removing them from the cut of the subsequent frame again (this is probably a bit confusing: The cut of the next frame, which has temporarily been pulled up to the root, is moved downwards again, but this action is still part of the 'pull up', for the cut will be at most moved down to the level of the current frame, but not below). The idea is that only internal nodes where all children are classified as invisible stay within the cut. This way, the extent of the pull up is not restricted by any means.

Pull downs can only be performed when the visibility has been determined, which is not before the traversal of the OG. They are required when previously invisible internal nodes have become visible and will be dealt with in the next chapter.

Summarizing, listing 4.1 shows a pseudo code for the overall kd-tree traversal.

***Listing 4.1**: Kd-tree traversal:*

```
Traverse(Node N)
{
  if (InsideViewFrustum(N))
  {
    if (IsLeaf(N))
    {
      // Assumed Visibility
      if (AssumedVisible(N)) Render(N);

      AppendToCut(N);
    }

    else if (Classification(N) == INVISIBLE)
    {
      AppendToCut(N);
    }

    else // visible internal node
    {
      // this might get overwritten
      SetClassification(N, INVISIBLE);

      if (ViewPoint[GetSplitAxis(N)] < SplitValue(N))
      {
        // the left child is closer
        Traverse(GetLeftChild(N));
        Traverse(GetRightChild(N));
      }

      else
      {
        // the right child is closer
        Traverse(GetRightChild(N));
        Traverse(GetLeftChild(N));
      }
    }
  }
}
```

4.3.2 Changes to the Occlusion Graph Traversal

Once the set of objects for the current frame has been determined, the generation of the OG remains unaffected by the introduction of a hierarchy. Basically, the traversal of the OG stays the same as well, hence the pseudo code of listing 3.1 (in Chapter 3.2.3.3) is still applicable. However, two modifications are necessary in order to realize pull ups and pull downs as discussed above:

- Each visible node must propagate its classification to all of its parents.
- Visible internal nodes of the kd-tree require a pull down.

Both modifications concern visible nodes, therefore an implementation (as discussed in this chapter) essentially extends the line of listing 3.1:

```
    if (Visible = TRUE) Render(N)
```

The first modification is part of the realization of pull ups, as explained in the previous section, and its implementation is quite straightforward: Each node has to be aware of its parent node within the kd-tree. Since this parent knows in turn its own parent node, this yields a sequence for each node up to the root. If an arbitrary node has been proven visible, all nodes along this sequence are set visible as well.

Handling the second task appropriately is much more sophisticated: If an internal node *N* turns out to be visible, it has obviously come into sight since the last frame. Now the challenge is that its subtree within the kd-tree must of course be rendered and tested, yet these nodes are not contained in the OG. Besides, the successors of *N* within the OG should only be dealt with after the subtree of *N* has entirely been drawn to avoid any loss of occlusion. Therefore they are not freed by *N* itself, but by the last node of its subtree.

One strategy for realizing pull downs is to classify all nodes within the subtree of *N* as visible without test which means rendering all leaves immediately, which is the most massive pull down possible. Consequently, all leaves will belong to the cut the next frame and get classified independently, probably leading to a significant pull up the frame after. Doing so is simple, yet a very bad idea since this inevitably causes a huge jitter for even slight movements. Apart from that, the majority of the objects within the subtree would be rendered in vain since with high probability, only a small fraction is actually visible. In fact no leaf needs to be visible at all since the classification of *N* can be due to its large AABB, and even its direct children can still be invisible as their AABBs match the actual geometry more tightly.

A better strategy is the basic hierarchical approach as presented in chapter 4.3.1: It issues a test for the child closer to the viewpoint and handles its subtree entirely before issuing the test for the second child. The problem of this method are the CPU stalls as reasoned above. A supposed way to overcome this is by increasing parallelism through immediately issuing the occlusion queries for both children. However, this turns out to be a bad idea, since the tests will not detect any sort of self-occlusion within the subtree (one part of the tree occluding another) which is in fact a common source of occlusion. If we are unlucky, even the whole tree might get falsely classified as visible this way, causing a big jitter again.

Because we do not have any further information about the nodes within the subtree of *N* (they are neither contained by the original OG, nor have they been classified in the previous frame), we can not get around applying the basic hierarchical approach. One modification has been tried, though: The closer child of a visible internal node was also assumed visible without test since I supposed that this was usually the case. However, it turned out that this assumption was not beneficial to performance; therefore I rejected the modification and returned to the unchanged basic hierarchical approach.

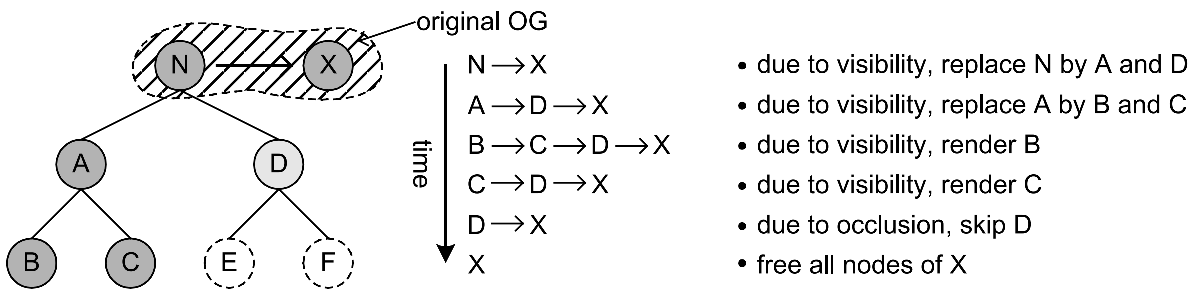

original OG
N
X
A
D
B
C
E
F
time
N ⟶ X
A ⟶ D ⟶ X
B ⟶ C ⟶ D ⟶ X
C ⟶ D ⟶ X
D ⟶ X
X
• due to visibility, replace N by A and D
• due to visibility, replace A by B and C
• due to visibility, render B
• due to visibility, render C
• due to occlusion, skip D
• free all nodes of X

Fig. 4.8 : The collaboration of the traversal of the subtree of a visible internal node and the OG.

Concerning the implementation, nodes being direct or indirect children of *N* are treated much like normal nodes of the OG. However, since these nodes are added to the OG afterwards, the implementation has to take care that all according variables of their data structures are set appropriately (which requires computing the SSBB for instance). In order to ensure the correct sorting, the child being closer to the viewpoint is assigned the other child as mere successor within its 'own' OG, while this latter child receives the complete list of successors from *N* – this way, these successors are freed after having rendered the last node of this subtree. Fig. 4.8 illustrates this by means of an example: It shows a possible subtree of *N* and how it might get traversed when assuming visibility for all nodes except for *D* and when presuming that *A* is closer to the viewpoint than *D* and *B* being closer than *C*. The right part shows the various steps in terms of the state of the temporary OG of this subtree, which is part of the overall OG (*X* denotes all successors of *N* and can actually be an arbitrarily large set of nodes).

4.3.3 Results and Discussion

This section extended the OG-based approach by using a hierarchy in order to reduce the number of objects making up the OG. Techniques were discussed to create spatial hierarchies resulting in the decision for kd-trees as most appropriate for our purpose. It was pointed out which issues must be paid attention to when traversing this tree, and in which way the traversal of the OG is affected.

This chapter also highlights some aspects of the hierarchical OG-based approach and discusses if this modification succeeds or fails to enhance overall performance. All measurements have been taken using the same five walkthroughs on the same system as presented in chapter 3.5. In all cases the improvements of chapter 3 have been applied:

- *Assumed Visibility* (VF_{min} = 1, VF_{max} = 10) together with *Pre-Rendering*.
- *Early Rejection*.
- Visibility threshold T_V of 50.

Besides, all hierarchical tests employ view-frustum culling, use the cut and order the objects implicitly by traversing the kd-tree as explained in chapter 4.3.1: Since these techniques are proven, no separate measurements have been taken in order to demonstrate their efficiency. For related results, refer to the according literature.

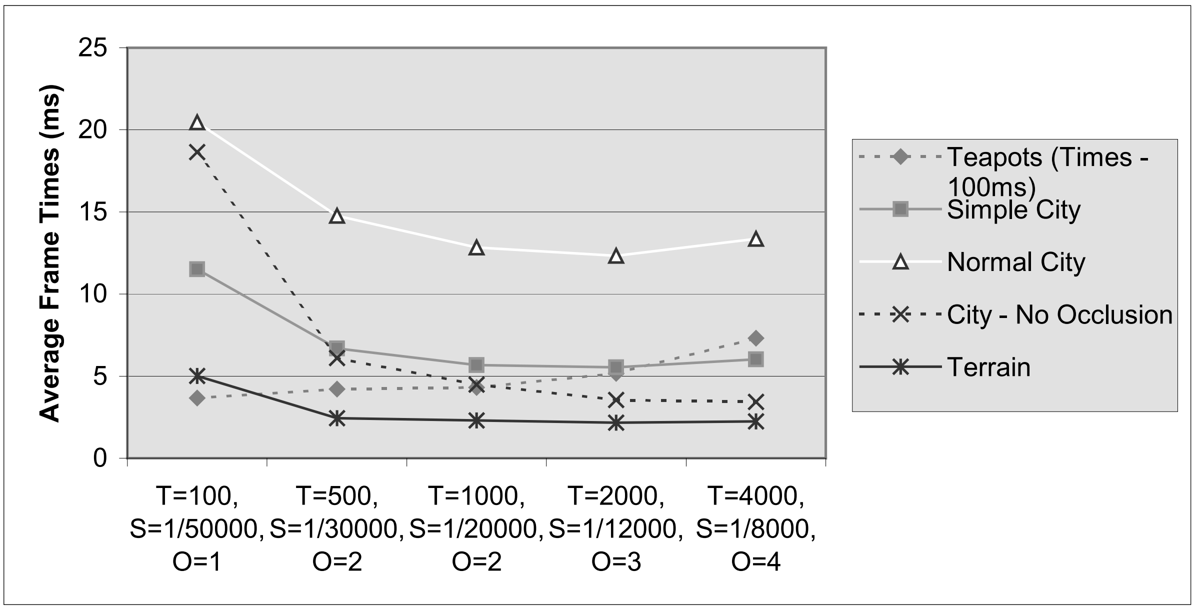


Fig. 4.9: Impact of the termination criterion used during the kd-tree construction on the overall performance. T stands for TKd_T, S for TKd_S and O for TKd_O. TKd_O, TKd_T and TKD_S constrain the properties of internal nodes as described above: In order to be split any further, a node must lie above all three thresholds.

The size of the kd-tree is determined by the termination criterion used for its construction, which affects the performance substantially as illustrated by Fig. 4.9: The number of nodes (the granularity of the leaves) decreases to the right. Evidently, having too many nodes has a negative effect on the performance since the geometry is too simple to justify its own occlusion query (it fits well that this does not apply to the teapot scene where each teapot is already quite complex). If the granularity becomes too coarse, many objects are rendered in vain thus decreasing performance again. For both city scenes as well as for the terrain, a setting of $TKd_T = 2000$, $TKd_S = 1/12000$, $TKd_O = 3$ (constraining triangles, surface and objects – see the image text) has been found optimal. Therefore, these values will be used from now on for all following tests.

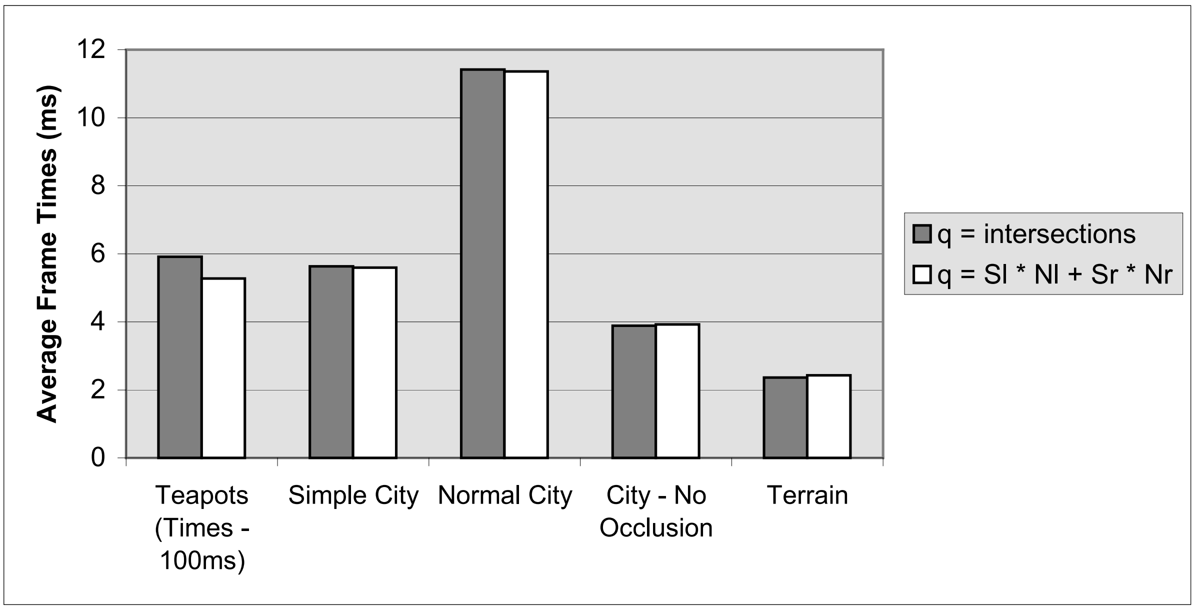


Fig. 4.10: Effect of the heuristic used during the kd-tree construction on the overall performance. Note that in order to make all results lie approximately the same range, 100 has been subtracted from the times of the teapot scene (which makes the difference much less than it might appear).

Fig. 4.10 shows that the two heuristic applied to select the optimal split value basically have the same effect on the performance. This might be a surprising result yet it holds for all scenes. Still it can not be ruled out that a heuristic could be found that improves the performance perceptibly. Due to (very slight) advantages, the second heuristic is used from now on.

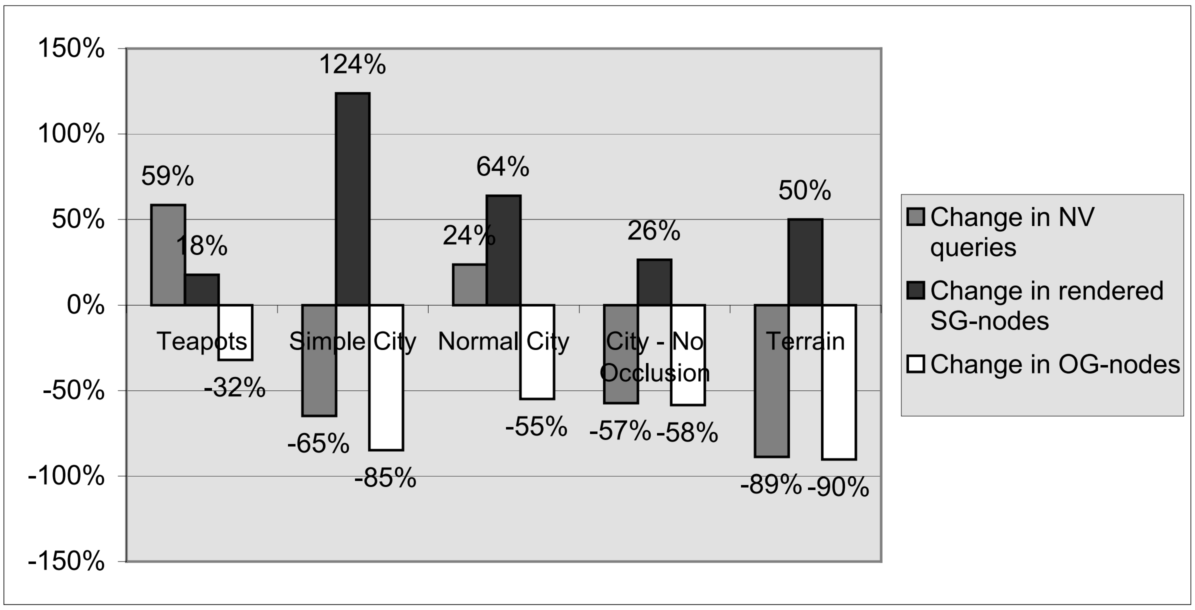
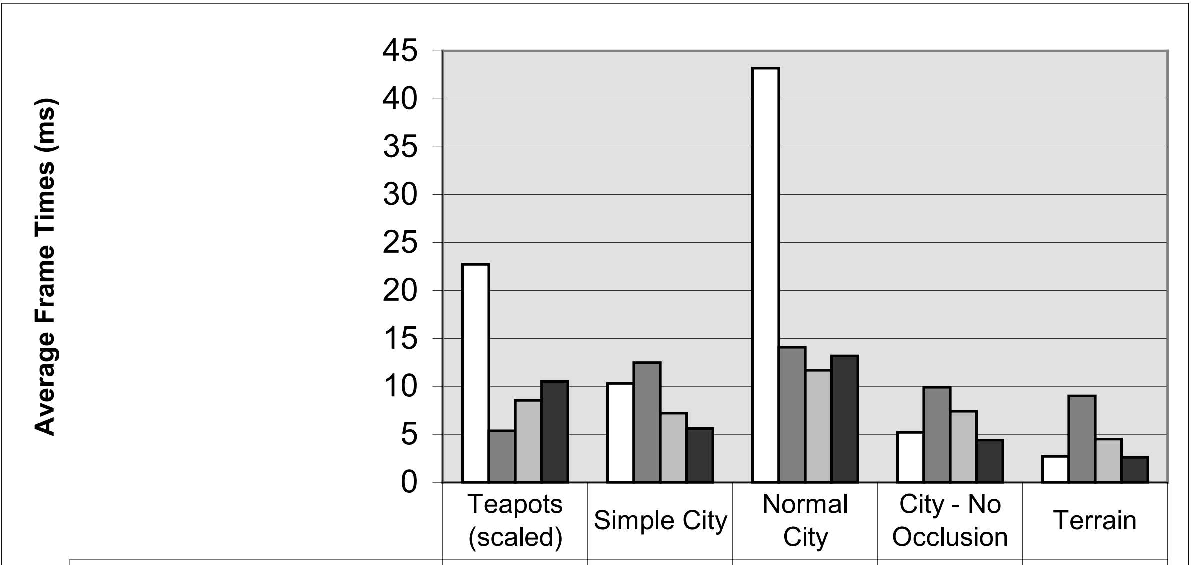


Fig. 4.11: The effect of the hierarchical OG-based approach compared to the non-hierarchical OG-based approach.

Fig. 4.11 illustrates how radically the introduction of a hierarchy affects the OG-based approach: The number of nodes making up the OG could (in some cases dramatically) be reduced. On the other hand, although a certain increase in the number of visible objects might have been expected, the actual amount is rather unpleasant. Especially where the geometry tends to consist of few triangles, many objects must be gathered to one node. Thus even if just a small part of the resulting SSBB used for testing is actually visible, still all objects are rendered. However, as shown in Fig. 4.9, accepting this additional rendering effort is for many scenes still better than doing tests in a more fine-grained fashion.

Apart from that, having less nodes may have led to the assumption that the number of occlusion queries can consequently be lessened as well. Obviously, this is not necessarily true: On the one hand, the regions that are simplified the most are those that are completely occluded anyway (and which cause hardly any tests due to *Early Rejection*). On the other hand, one must keep in mind that doing a pull down may take more tests than the number of contained leaves. Therefore, if the cut is constantly subject to strong variations, more tests might be required than without the hierarchy (this is particularly true for the teapot scene where hardly any temporal coherence is present for the remote teapots). Furthermore, due to the increased size of the testing entities as well as the still very conservative approximation with SSBBs, comparatively more tests indicate visibility on average, which interferes with *Early Rejection*.



	Teapots (scaled)	Simple City	Normal City	City - No Occlusion	Terrain
☐ No Occlusion Culling	22,73	10,3	43,2	5,2	2,7
▨ Stop-and-Wait Approach	5,36	12,5	14,1	9,9	9
▨ Non-Hierarchical OG-Based Approach	8,54	7,2	11,7	7,4	4,5
■ Hierarchical OG-Based Approach	10,51	5,6	13,2	4,4	2,6

Fig. 4.12: Comparison of the average performance of all important approaches presented so far for all walkthroughs. Note that the times of the teapot walkthrough have been divided by 10 in order make all values lie approximately within the same range.

Finally, does it pay off to do the OG-based approach in a hierarchical setting? Fig.4.12 compares the average frame times. For more detailed plots, refer to chapter 5. While some scenes could be sped up, others are slowed down which makes a generally valid answer difficult.

As already pointed out in chapter 3.5, an essential criterion for occlusion queries is the average object complexity: It decides if an additional occlusion query does pay off in order to refine the classification. In the teapot scene for instance, the costs of unnecessarily rendering many objects exceed the benefits of issuing less tests, while the situation is contrary in the scenes with little occlusion. Besides, the teapot scene suffers from a high dynamics of the cut due to little temporal coherence for the remote objects as reasoned above. The result of the complex city might be astonishing: When examining the exact plot of this walkthrough (as given in chapter 5), it turns out that the frame times for the 'usual' walkthrough situation – for example going along a street in a normal person's view – were reduced by and large, yet that the peak load has increased dramatically which is once more due to the raised number of rendered objects. The scene with no occlusion has – surprisingly – even become faster than without any occlusion culling. However, these performance gains originate from the introduction of the visibility threshold (which does not render almost imperceptibly small parts) and maybe a slightly more efficient traversal (e.g. exploiting the hierarchy for view-frustum culling) rather than from actual occlusion culling itself.

Summarizing, as with most occlusion culling approaches, it must be decided for each scene anew if doing occlusion culling with the hierarchical OG-based approach makes sense. This will generally rather be true for scenes with significant occlusion on a coarse level: This means that – as with the terrain scene or the walkthrough without occlusion – the classification is unlikely to switch back and forth often within a small area. In other words: A hierarchy is a way to exploit spatial coherence and it will be the more successful, the more distinct the property of spatial coherence is for a certain scene. Still, one might be able to optimize the approach for a given setting by finding the most appropriate parameters for the creation of the kd-tree.

In conclusion, the hierarchical OG-based approach yields pretty satisfactory results for scenes with a high spatial and temporal coherence. However, the sometimes dramatic overestimation of visibility might be unbearable for many scenes. Besides, the incorporation of a hierarchy has further increased the complexity of the already quite complex non-hierarchical OG-based approach. Thus the results may not justify the high programming effort. Besides, dealing with dynamic scenes has become even harder due to the static hierarchy; even more disadvantages are listed in section 4.4.1. Finally, the NV_occlusion_query extension has still a greater potential. Therefore – although some results might be encouraging –a still easier and generally better approach is desirable.

4.4 Efficient Hierarchical Temporal-Coherence Based Approach

This chapter presents an independent approach for doing occlusion culling with hardware occlusion queries, which is different from the OG-based approaches discussed in previous chapters. Although the insights gained with the OG-based approaches had of course a strong influence on its design, it is a completely independent algorithm that does not rely on the OG in any form. As its name suggests, the main characteristics of this new approach (from now on abbreviated as *HTC-based approach*) are its application within a hierarchical setting and the exploitation of temporal coherence in two ways.

The first part of this chapter explains why the OG-based approach was not considered to be sufficient, and how these experiences could help to design an even better algorithm. The following two parts deal with the algorithm itself and significant issues concerning its implementation, before the final part presents the according results and figures.

4.4.1 Motivations and Considerations

This approach was devised for two reasons:

- Major flaws of the OG-based approach are inherent and can only be mitigated yet by no means entirely solved.
- To examine how well the OG-based approach is actually doing, i.e., for the sake of comparison.

To begin with, let us analyze the main problems of the OG-based approach and their implications for the design of an alternative algorithm:

- Despite of all improvements, it can not be ruled out that the OG might be created for a lot of objects, which means a significant CPU overhead. The less occlusion we have, the more painfully this overhead will affect frame times. Consequently, a new approach should not suffer from any remarkable CPU overhead.
- Although the OG-based approach succeeds in reducing the time of CPU stalls significantly compared to the basic stop-and-wait algorithm, it fails to cut them down to a negligible amount. This is due to the fact that it often lacks alternative work even if the algorithm is aware that fetching the next test result would incur a CPU stall. Therefore a new approach should (nearly) always be able to provide the CPU with reasonable alternative work if the result of the oldest outstanding occlusion query has been proven unavailable yet.
- The effect of *Early Rejection* – this improvement turned out to be crucial in order to justify the construction of the OG at all – is diluted when applying the approach in a hierarchical setting. Basically, the problem caused by a host of occlusion queries for occludees is tackled in two different ways then. Since a new approach will do without OG anyway, this topic need not be heeded any further.
- SSBBs must be computed for the IZB and are moreover necessary as approximations for *Early Rejection* as reasoned in section 3.4.1.2, but they have two major disadvantages: Firstly, their computation out of AABBs is very expensive and secondly, they are extremely conservative (much more than AABBs), which causes many objects to be falsely classified as visible. Therefore, the new approach should do without SSBBs and use AABBs as test geometry instead.
- Finally, the OG-based approach along with all its improvements has grown quite big and complex, which might hinder its application for other purposes than scientific investigations. Hence, simplicity and brevity emerge as design goals for a new algorithm.

In order to be able to cope with an abundance of objects properly, using a hierarchy is a requirement for the new approach. As reasoned in chapter 4.2, kd-trees proved to be a good compromise between flexibility and simplicity, so a kd-tree will be the core of the new algorithm as well. Everything that was said in chapter 4.2.2 concerning its construction also applies to the HTC-based approach; in fact, the respective code was hardly modified for the new approach at all.

As a consequence of the first point of the list above, it is not affordable to traverse the scene graph each frame because of the massive CPU overhead. Besides, doing so would contradict the demand for a hierarchy. Therefore, as explained in appendix A, we use a kd-tree as additional data structure which only references single nodes of the scene graph. Furthermore, avoiding any CPU overhead means that no time must be wasted by doing any pre-traversals of the kd-tree before starting the actual work of testing and rendering. Everything must be accomplished within one traversal.

The second point implies the maintenance of two different sources of meaningful work: Preferably, the results of previous occlusion queries are fetched, yet if no results are available at the moment, a second source should always be able to supply the CPU with further work. A prerequisite for doing so is the ability to check if outstanding test results are available without incurring any stall. Observations show that such checks are almost for free.

Finally, it should be emphasized that – unlike the OG-based approach – it was no requirement for the HTC-based approach to rule out any loss of occlusion caused by incorrect sorting of rendering and testing, which turned out to be a too strong restriction for the OG-based approach (even more so as the OG-based approach actually does lose occlusion anyway owing to the usage of SSBBs and – in the hierarchical version – combining many objects within the elementary cells of the hierarchy). It is tolerable if a few objects might be rendered in vain as long as the benefits of this increased flexibility outweigh the costs entailed by such renderings.

4.4.2 Algorithm Description

The central idea of the HTC-based approach is to exploit temporal coherence of the visibility classification in order to increase the parallelism within the application: The CPU and the GPU can act more independently of each other, if the time when an occlusion query can be issued for a node does not depend on the classification of any other node (in contrast to the OG-based approach).

Basically, the HTC-based approach is solely a new combination of ingredients which have already been developed for the hierarchical OG-based approach. Apart from the fact that it does not rely on an OG, it partly resembles this approach very strongly since it is basically a mixture of methods used within the traversals of the OG and the kd-tree. The new algorithm is based on the following techniques (the discussion assumes that these techniques are familiar):

- Approximate front-to-back kd-tree traversal (chapter 4.3.1).
- Having a queue of issued occlusion queries as one source of work, and another data structure for alternative work for the case that no results are currently available (chapters 3.2.3.3 and 3.2.3.4).
- Maintenance of a *cut* (chapter 4.3.1).
- Application of *Assumed Visibility* (3.4.1.1) and a visibility threshold (3.4.2.1).
- Exploiting the hierarchy for view-frustum culling (chapter 4.3.1).

Apart from *Assumed Visibility*, which is rather an extension than a part of the core algorithm, temporal coherence is primarily exploited by maintaining a cut. Summarizing the description of chapter 4.3.1, the idea is to subdivide all nodes of the kd-tree into three categories: Nodes above the cut are only traversed but not considered for occlusion queries, nodes belonging to the cut are both traversed and occlusion tested and nodes below the cut are neither traversed nor handled in any other form (provided that the cut needs no adaptation). Whether a node belongs to the cut is determined by the visibility classification of all nodes in the former frame: The cut comprises all invisible internal nodes where the whole subtree is invisible as well and all visible leaves.

Very briefly, the algorithm can be outlined as follows: The kd-tree is traversed once per frame in a depth-first manner that yields an approximate front-to-back sorting of the nodes and performs view-frustum culling. When the traversal encounters a node which belongs to the cut, it issues an occlusion query in order to check if the previous visibility classification is still valid. Furthermore, if the respective node has been proven visible in the previous frame, it is rendered immediately after issuing the occlusion query (without waiting for the result!) to permit the occlusion of further nodes. In addition to the actual traversal, the algorithm repeatedly checks the availability of occlusion results. If these results indicate changes in the visibility classification with respect to the previous frame, pull ups and pull downs must be performed to adapt the cut accordingly. As possible improvements, occlusion tests can be saved by reusing results indicating visibility for several frames (*assumed visibility*), and considering only such nodes as visible where more than a specified number of pixels have passed the occlusion test, allowing to trade off quality for speed.

The algorithm comprises two major parts which alternate permanently:

- It traverses the kd-tree in a front-to-back order down to the cut and deals with the nodes of the cut passing view-frustum culling as suggested by the respective previous classification, which may involve rendering and issuing occlusion queries.
- By evaluating the results of the occlusion queries, it ensures that the cut and the classifications of its nodes are still correct, and updates them if needed.

As with the traversal of the OG, the termination condition is when no part requires further progress (there are no occlusion queries waiting, and the kd-tree has already been entirely traversed). Furthermore, processing available results is preferred: As long as test results are present, they are fetched and handled appropriately; only if querying would incur a CPU stall or no tests are currently enqueued, we turn to the alternative work in form of continuing the traversal of the kd-tree.

4.4.2.1 Kd-tree Traversal

This part of the algorithm can briefly be characterized as follows:

- Deal with all nodes recursively down to the cut. For each node, start with the subtree being located on the same side of the split plane as the viewpoint, yielding a roughly front-to-back sorting (as described in chapter 4.3.1).
- View-frustum culling is performed for each traversed node, taking advantage of the hierarchy (as described in chapter 4.3.1).
- Invisible nodes of the cut within the view frustum are only tested but not rendered, since they are assumed to stay invisible.
- Visible nodes of the cut (which must be leaves) are tested (unless this test is skipped due to *Assumed Visibility*) and immediately rendered afterwards since the test is presumed to indicate visibility. Note that this instantaneous drawing is crucial since this is the only way that potential occlusion of the next few nodes in the traversal is made possible.

Since we lack any information which objects will overlap in the final image, we can not get around dealing with all objects in a roughly front-to-back manner. In order to maintain the cut, each node stores the information of its visibility classification from the previous frame. This way, leaves and invisible internal nodes – generally speaking all nodes making up the cut – need not be traversed any further. Again, this cut is likely to require adaptations, referred to as pull ups and pull downs, for every modification to the viewpoint or the viewing direction. The latter ones are more unpleasant as they entail the traversal of a subtree with the basic hierarchical approach; for more detailed information about adapting the cut, refer to the next chapter.

After outlining the traversal, let us consider its implications: The advantage of relying entirely on the former visibility classification is that no node is constrained by the progress in the visibility classification of any other node. However, this comes at the cost of possibly rendering a few occluded nodes as well: False classifications can only be ruled out under the condition that the cut has not changed within the recent two frames (and even longer if we take *Assumed Visibility* into account). For instance, if a leaf *N* has become occluded since the last frame, this is noticed only after *N* has been rendered unnecessarily, yet it will not be rendered the next frame. Formally, let '$T_k(N)$' denote the issuing of an occlusion query for *N* within the *k*-th frame. Similarly, '$R_k(N)$' indicates that *N* is rendered and '$F_k(N)$: *X*' means that the result of the occlusion query for *N* is fetched and – anticipating the next chapter – the visibility classification of *N* is possibly inverted to *X*. Furthermore N^I and N^V mean that *N* is classified as invisible or visible, respectively, before applying the according action. Using this formalism, the situation described above (a leaf *N* has become occluded) can be described as follows:

Frame k: $T_k(N^V) \rightarrow R_k(N^V) \rightarrow F_k(N^V): I$

Frame k+1: $T_{k+1}(N^I) \rightarrow F_{k+1}(N^I): I$

If a previously occluded node *X* has come into view, hereby occluding another formerly visible node *Y*, it takes even two frames (without *Assumed Visibility*) to adapt the classification of *Y*: In the first frame, *Y* is tested before *X* (more precisely: the visible nodes of its subtree) is rendered, thus *Y* is found visible again. In the second frame, the test for *Y* is issued after *X* has been rendered and consequently proves it invisible, yet as explained above, this happens after *Y* has been rendered once more:

Frame k: $T_k(X^I) \rightarrow T_k(Y^V) \rightarrow R_k(Y^V) \rightarrow F_k(X^I): V \rightarrow R_k(X^V) \rightarrow F_k(Y^V): V$

Frame k+1: $T_{k+1}(X^V) \rightarrow R_{k+1}(X^V) \rightarrow T_{k+1}(Y^V) \rightarrow R_{k+1}(Y^V) \rightarrow F_{k+1}(X^V): V \rightarrow F_{k+1}(Y^V): I$

Frame k+2: $T_{k+2}(X^V) \rightarrow R_{k+2}(X^V) \rightarrow T_{k+2}(Y^I) \rightarrow F_{k+2}(X^V): V \rightarrow F_{k+2}(Y^I): I$

4.4.2.2 Updating Classifications and the Cut

After issuing an occlusion query for a node, this node is enqueued in order to delay fetching its result until this result is actually present (note that nodes are tested in order to verify their classification, not to constrain the time of their rendering). In the best case, all results confirm the classifications of the respective nodes, thus neither the nodes themselves, nor the cut need any updating. However, the vanishing of a previously visible node – apart from updating its classification – can entail a pull up, just like the appearance of a previously invisible node requires a pull down (if this node is not a leaf anyway). As reasoned in chapter 4.3.2, performing a pull down means traversing the according subtree with the basic hierarchical approach, which is conceptually a traversal within the traversal and must of course be appropriately interwoven with the rest of the work.

As explained in chapter 4.3.1, the basic hierarchical approach means traversing a tree in a front-to-back fashion, just like the HTC-based approach does it. However, the difference is that an occlusion query is issued for each node encountered throughout the traversal and the work can only be continued after the result is present – hence the full stall is incurred each time.

4.4.3 Implementation Issues

While the concepts of the HTC-based approach were described in the previous chapter, this section briefly covers some implementation related aspects:

- Doing the front-to-back kd-tree traversal.
- Performing pull ups.
- Performing pull downs.
- Realizing assumed visibility.
- Managing the OpenGL state for testing and rendering.

Let us begin with the kd-tree traversal: As explained in chapter 4.3.1, a front-to-back traversal can be achieved by always starting with the part which lies on the same side of the split plane as the viewpoint. Dealing with all nodes in the correct order can be ensured by maintaining a stack for the storage of nodes which are yet to be traversed. This stack initially contains merely the root node. Each time the foremost node is removed, and only if this node does not belong to the cut, it pushes its children in reverse order onto the stack (of course this only applies to children passing view-frustum culling). This way, the whole subtree of the closer child will be traversed before the other child gets processed.

Pull ups are basically done quite the same way as with the OG-based approach (see chapters 4.3.1 and 4.3.2): Each node encountered throughout the stack-based kd-tree traversal is assumed to be part of the cut the next frame (this implies that we assume invisibility too). On the other hand, every visible leaf informs all of its parents up to the root that they are visible as well and can therefore not belong to the cut. As a consequence, if no direct or indirect child is proven visible for an arbitrary internal node, this node remains classified as invisible and thus stays within the cut until the next frame where it is handled accordingly.

As mentioned before, performing a pull down for an internal node requires a traversal using the basic hierarchical approach. Although the rest of the HTC-based approach does not constrain nodes by the progress of other nodes, doing so is inevitable in the case of the basic hierarchical approach in order to avoid losing too much occlusion within the subtree itself (see section 4.3.2). This traversal within the traversal is implemented as follows: A stack is maintained for each pull down traversal the same way as for the overall traversal (described in two paragraphs above), where the topmost node is occlusion tested and enqueued just like any other tested node. If this test indicates invisibility, the node is popped off the stack without replacement; otherwise, its children are pushed onto the stack in reverse order of their distance. The work is continued by issuing an occlusion query for that node of the stack which is topmost then. Unlike the overall traversal, the stack is not modelled as an independent data structure, but each node contains a pointer to the node below in the stack (if a node is not part of a stack – which is the case for most nodes, as this pointer is only used for pull downs – or if it is the bottommost element, this pointer is null). Summarizing, an own stack is maintained for each traversal (one overall traversal and several pull downs), but the queue for outstanding occlusion-test results is the same throughout the whole algorithm.

The improvements of *Assumed Visibility* (chapter 3.4.1.1) and the visibility threshold (chapter 3.4.2.1) are applied in the same manner as within the OG-based approaches. Unlike the latter enhancement, incorporating *Assumed Visibility* requires minor modifications: As reasoned above, AABBs are used as geometry for the NV queries instead of SSBBs. While this means no great difference for the code of the testing itself, it involves a revision of the formula we evaluate for computing the number of frames an object is assumed to stay visible: Since we don't have any screen-space projection, we lack the information how many pixels are potentially covered by the geometry – hence we can not compute the percentage how much of an object was found to be visible. Generally speaking, this percentage must be approximated one way or the other. A reasonable way would be to estimate the area of the screen space projection (an according formula can for instance be found in [Coor96]). The alternative used for the implementation to this thesis compares the number of visible pixels, which is the result of the NV query, to the overall number of pixels of the output device. A certain percentage P_V is presumed to indicate full visibility. Assuming for instance a resolution of 640 x 640 and a P_V of 2 (which has been found a reasonable value in practice), test results of 8192 visible pixels upwards are considered fully visible. Thus if for instance 3000 pixels pass an occlusion query, a visibility of 36,6% is presumed. Apart from that, the number of frames without test are still linearly interpolated as shown in chapter 3.4.1.1.

Another topic is ensuring the correct OpenGL state without unnecessarily switching back and forth between the rendering and testing modes. In the traversal of the OG, NV queries and renderings were done in an alternating fashion. In the HTC-based approach, we can't predict in which order tests and renderings will occur. This problem was tackled by switching the state lazily: Every time that a certain state is required, we check if this state isn't already set anyway. If it is not, it is changed accordingly.

Finally, listing 4.2 shows the HTC-based approach in C-like pseudo code. `ClassifyParentsAsVisible(N)` is part of the pull down: It sets all parents of the node N (up to the root) as visible, thereby removing them from the cut of the subsequent frame.

***Listing 4.2**: The Efficient Hierarchical Temporal-Coherence Based Approach:*

```
//---- initialisation
Stack.Clear(); Queue.Clear(); Stack.Push(Root);

while (NOT Stack.Empty()) OR (NOT Queue.Empty())
{

  //---- part one: updating cut and classification
  while (NOT Queue.Empty()) AND
        ((TestAvailable(Queue.Front())) OR
        (Stack.Empty()))
  {
    N = Queue.Dequeue();

    if (VisiblePixels(N) > VisibilityThreshold)
    {
      ClassifyParentsAsVisible(N);

      if (IsLeaf(N))
      {
        Render(N);

        ComputeVisibleFrames(N); // Assumed Visibility
      }

      else
      {
        // pull down
        RemoveFromCut(N);
        AddToCut(N->Left); AddToCut(N->Right);

        if (InViewFrustum(CloserChild(N)))
          PullDownStack(N).Push(CloserChild(N));

        if (InViewFrustum(DistantChild(N)))
          PullDownStack(N).Push(DistantChild(N));
      }
    }
```

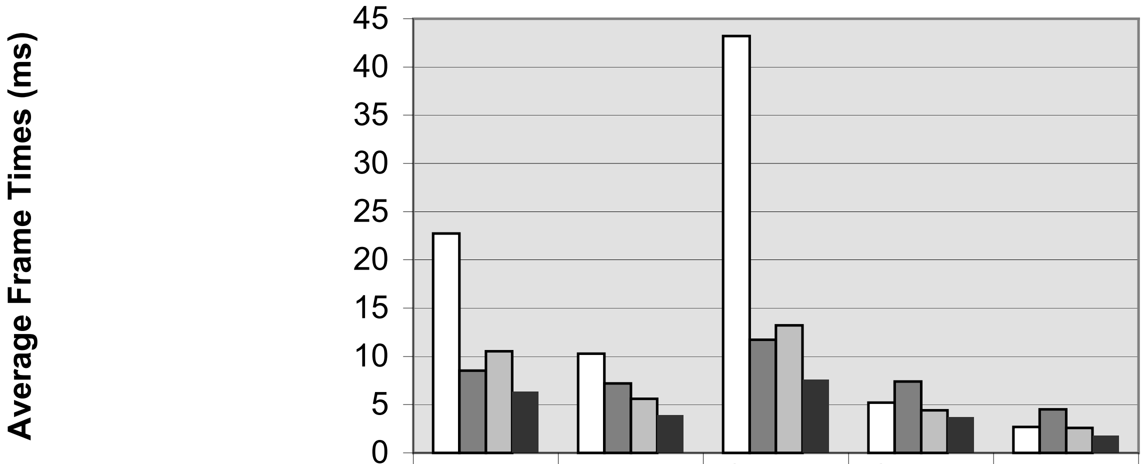
```
      if (NOT PullDownStack(N).Empty)
      {
        M = PullDownStack(N).Pop();

        NVTest(M); Queue.Enqueue(M);
      }
    }

    //---- part two: kd-tree traversal
    if (NOT Stack.Empty())
    {
      N = Stack.Pop();

      if (IsWithinCut(N)) OR (IsLeaf(N))
      {
        if (NOT Visible(N)) OR (NOT VisibleFrames(N) > 0)
        {
          NVTest(N); Queue.Enqueue(N);
        }

        if (VisibleFrames(N) > 0)
        {
          DecrementVisFrames(N); SetVisible(N);
        }

        if (Visible(N)) AND (IsLeaf(N))
        {
          Render(N); ClassifyParentsAsVisible(N);
        }
      }

      else
      {
        // pull up
        AddToCut(N);

        if (InViewFrustum(DistantChild(N))
          Stack.Push(DistantChild(N));

        if (InViewFrustum(CloserChild(N))
          Stack.Push(CloserChild(N));
      }
    }
  }
```

4.4.4 Results and Discussion

In this section, the HTC-based approach has been introduced as another possibility to use the NV_occlusion_query extension in a hierarchical setting exploiting both spatial (by the hierarchy) and temporal (by the cut and *Assumed Visibility*) coherence. As before, all measurements have been taken using the same five walkthroughs on the same system as described in chapter 3.5.

This section basically confines itself to presenting the average frame times of the walkthroughs, since the figures of chapter 4.3.3 concerning the parameters for the construction of the kd-tree apply to this approach as well and an examination of the effect of

Assumed Visibility and the visibility threshold can be found in chapter 3.5. For a detailed comparison of other aspects than the performance itself (e.g., number of visible scene-graph objects, number of NV queries or average CPU-stalls) and the exact performance plots refer to the comparison of all approaches in the next chapter.



	Teapots (scaled)	Simple City	Complex City	City - No Occlusion	Terrain
□ No Occlusion Culling	22,73	10,3	43,2	5,2	2,7
▩ Non-Hierarchical OG-Based Approach	8,54	7,2	11,7	7,4	4,5
▨ Hierarchical OG-Based Approach	10,51	5,6	13,2	4,4	2,6
■ HTC-Based Approach	6,35	3,9	7,6	3,7	1,8

Fig. 4.13: Comparison of the average performance of all introduced approaches. Note that the times of the teapot walkthrough have been divided by 10 in order make all values lie approximately within the same range.

The most striking result is that the HTC-based approach clearly outperforms both hierarchical and non-hierarchical OG-based approach – regardless of the scene. This insight is even more pleasing, as this approach is much simpler than the others and thus of a higher practical relevance. Obviously, this approach seems to be an appropriate way to ensure a constant supply of meaningful work to the GPU while maintaining a reasonable accuracy in the classification – in spite of the fact that the HTC-based approach does without strict constraints, but classifies each node independently.

The advantages of the HTC-approach are without doubt the satisfactory performance and its comparative simplicity. However, one might suppose that it suffers from a tendency to classify even more objects as visible than the hierarchical OG-based approach. Maybe somewhat surprisingly, this is not the case (see the comparison of chapter 5 for exact numbers) – on the contrary: Apart from the absence of any CPU-overhead combined with a reduction of CPU-stalls, the main reason for the speed-ups is the less conservative approximation of the geometry used for the NV queries (AABBs instead of SSBBs), which prevents a lot of false classifications. This way, even the teapot scene, which is very sensitive to the correctness of the classification, shows remarkable performance gains, because less teapots were rendered than with the non-hierarchical OG-based approach (let alone the hierarchical version).

Anticipating chapter 6.2, one might ask if any further enhancements to the HTC-based approach are still imaginable. One idea could be to combine entirely visible subtrees as well (so far, this is only done for entirely invisible subtrees). However, it turns out that this is practically impossible since generally speaking, even the full visibility of a common AABB doesn't suffice to rule out that some leaves may still be occluded due to self-occlusion within the subtree. Besides the problem of an abundance of small visible leaves causing many NV queries is already tackled by the introduction of *Assumed Visibility* and the effort for the traversal as such is likely to be negligible. Another idea is to switch to rendering without occlusion culling when almost no occlusion is encountered any more, which would probably increase performance even more for many scenes; the challenge is to find out when occlusion culling should be turned on again.

Summarizing, the HTC-based approach shows the highest performance of all algorithms presented in this master thesis and can therefore be seen as its main contribution. It is both generally applicable and still relatively simple. Besides, due to the tighter approximations, it tends to classify less objects as visible than the hierarchical OG-based approach, for example. As far as all results show, it has no serious drawback that wouldn't be inherent to any occlusion culling algorithm (e.g. that a certain overhead is inevitable). Eventually, it might turn out that it is less suitable for incorporating dynamic objects, since this requires inserting them into the kd-tree each frame anew which might become costly if done very often.

5 Discussion and Comparison of Approaches

In this master thesis, three different approaches for doing occlusion culling with the NV_occlusion_query OpenGL extension have been described and discussed. Two of them are based on the non-hierarchical and hierarchical, traversal of a directed acyclic graph – the occlusion graph (OG) -, while the third one depends solely on temporal coherence in a hierarchical setting.

Each approach has already been thoroughly discussed more or less on its own (some comparison between various approaches has been necessary already in the previous chapters, though). The purpose of this section is to provide an overview about all approaches, their performance and other important aspects.

To begin with, let us specify what the various approaches actually refer to:

- **No occlusion culling**: Simply traversing the original scene-graph and rendering all objects within the view-frustum. Note that this is the only actually conservative approach in this comparison: While all other test employ a visibility threshold, this is obviously impossible in this case, since no occlusion queries are issued.
- **Stop-and-wait approach**: Sorting all objects within the view-frustum in a strictly front-to-back fashion (using a quick-sort) and testing them this way. AABBs are used for testing. All objects where at least 50 pixel pass the occlusion query (T_V = 50) are considered visible (and thus rendered),. This algorithm is also referred to as **Exact Visibility**.
- **Non-hierarchical OG-based approach**: The approach presented in chapter 3 together with all improvements, SSBBs are used for testing. The parameters are set as follows: Assumed visibility VF_{max} = 10, VF_{min} = 1, T_V = 50.
- **Hierarchical OG-based approach**: The approach presented in chapter 4.3 together with all improvements. The parameters for constructing the kd-tree are set as follows: TKd_O = 3, TKd_S = 1/12000, TKd_T = 2000 with the heuristic $q = N_l * S_l + N_r * S_r$. The parameters of the optimisations are set to the following values: VF_{max} = 10, VF_{min} = 1, T_V = 50.
- **HTC-based approach**: The approach presented in chapter 4.4. The kd-tree was constructed with these settings: TKd_O = 3, TKd_S = 1/12000, TKd_T = 2000 with the heuristic $q = N_l * S_l + N_r * S_r$. The parameters of the optimisations are set to the following values: VF_{max} = 10, VF_{min} = 1, T_V = 50. AABBs are used for testing.
- **Optimal (precalculated visibility)**: This is not an actual approach, but refers to the time it would take to render all visible objects (assuming a visibility threshold of 50 again) if this visibility was known without any computations.

The measurements refer to the duration of the whole rendering – thus until all committed work has entirely been handled by the GPU (this can be ensured by calling `glFinish()`).

The figures 5.1 to 5.5 depict the exact plots of the frame times for all approaches and all test scenes.

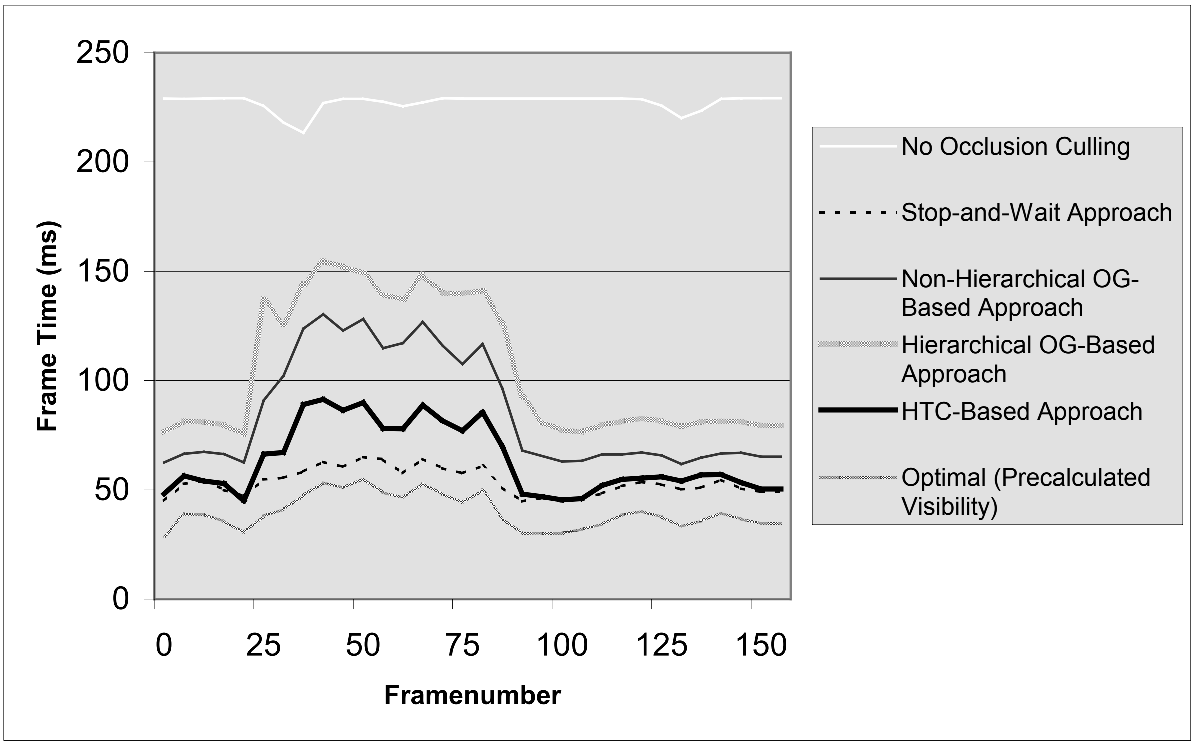


Fig. 5.1: Comparison of the frame times of the walkthrough of the teapot scene.

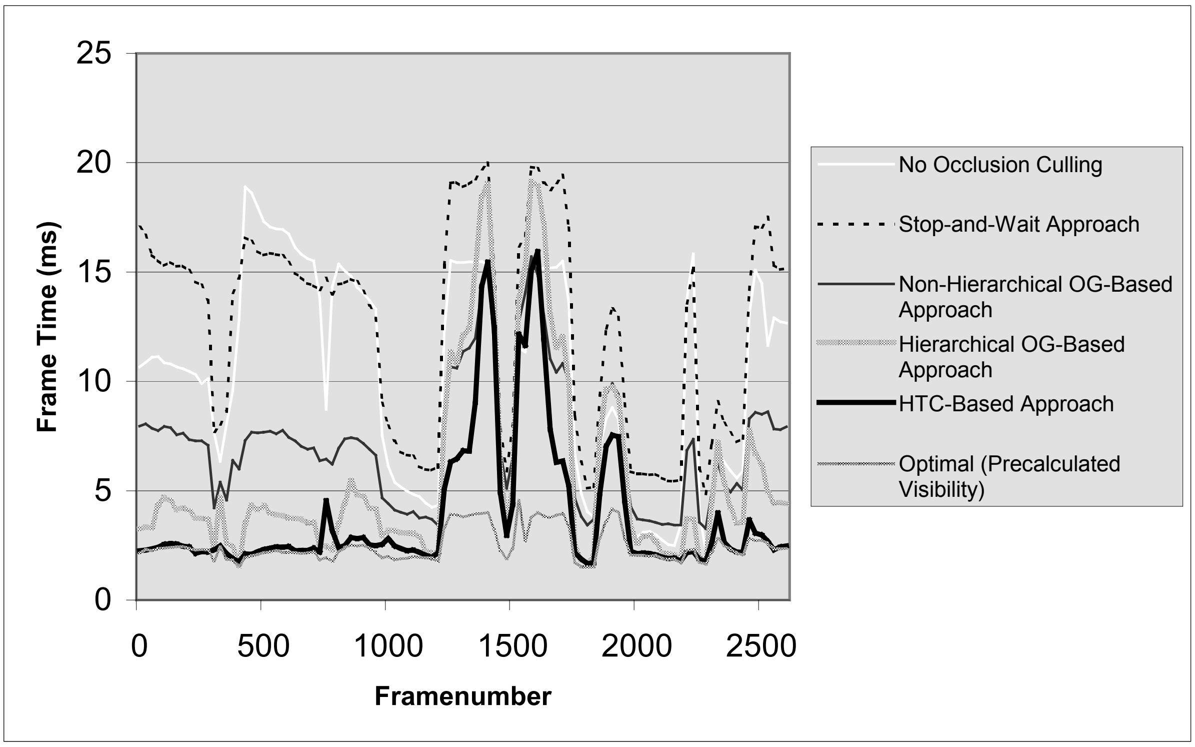


Fig. 5.2: Comparison of the frame times of the walkthrough of the simple city model.

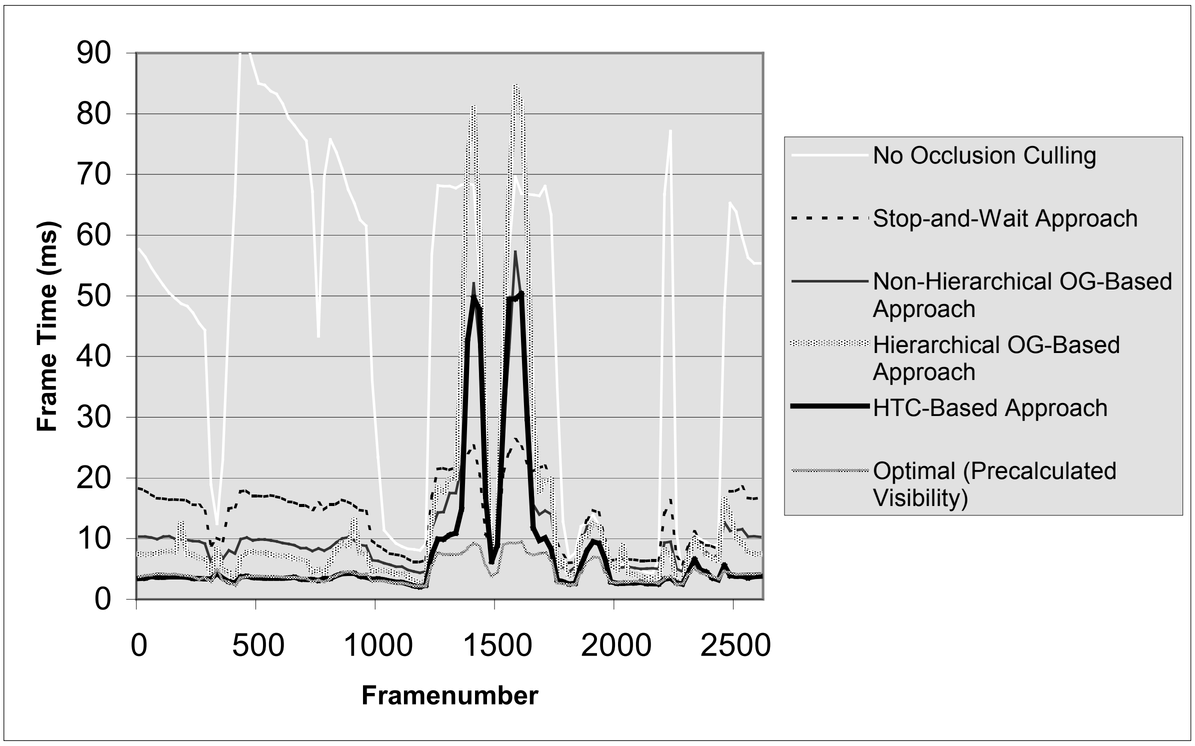


Fig. 5.3: Comparison of the frame times of the walkthrough of the complex city model.

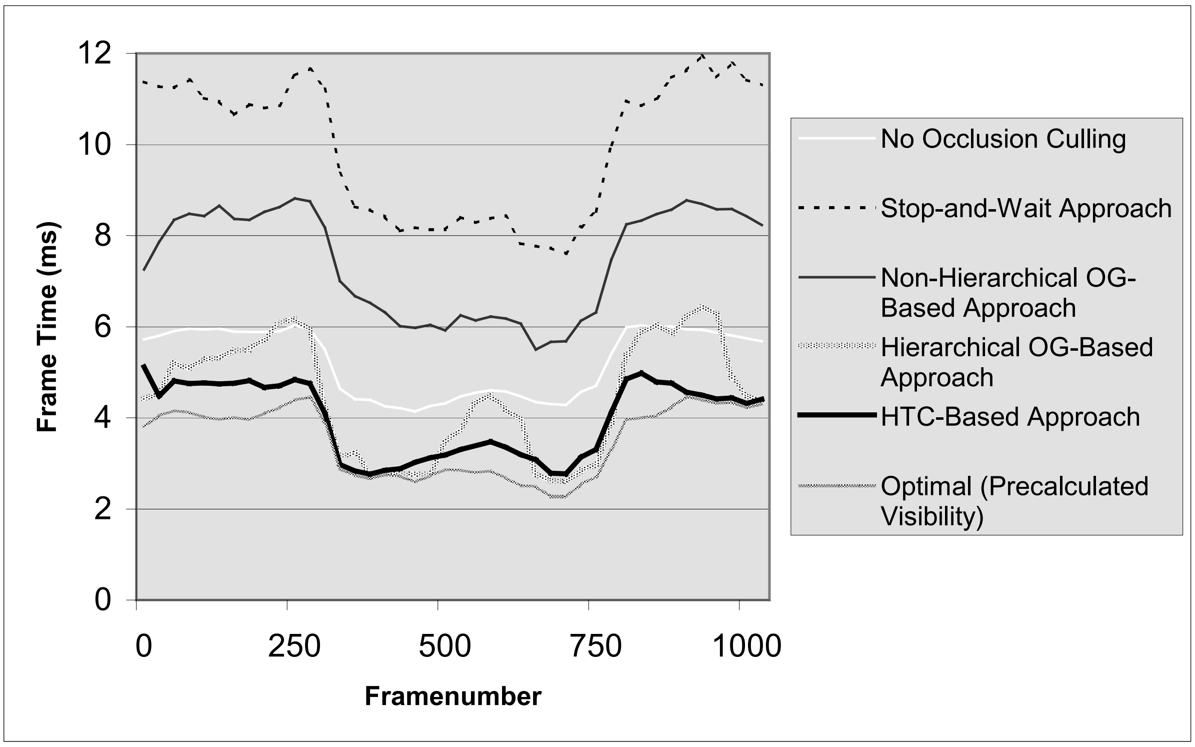


Fig. 5.4: Comparison of the frame times of the city walkthrough when no occlusion is present.

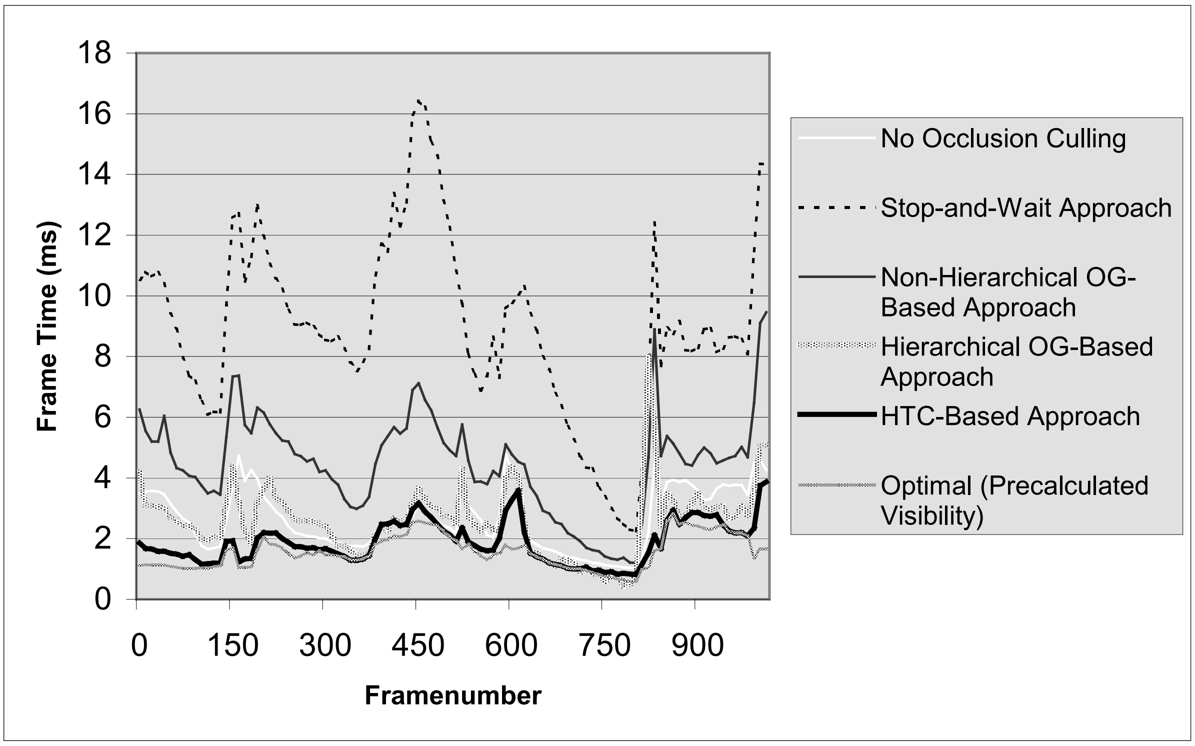


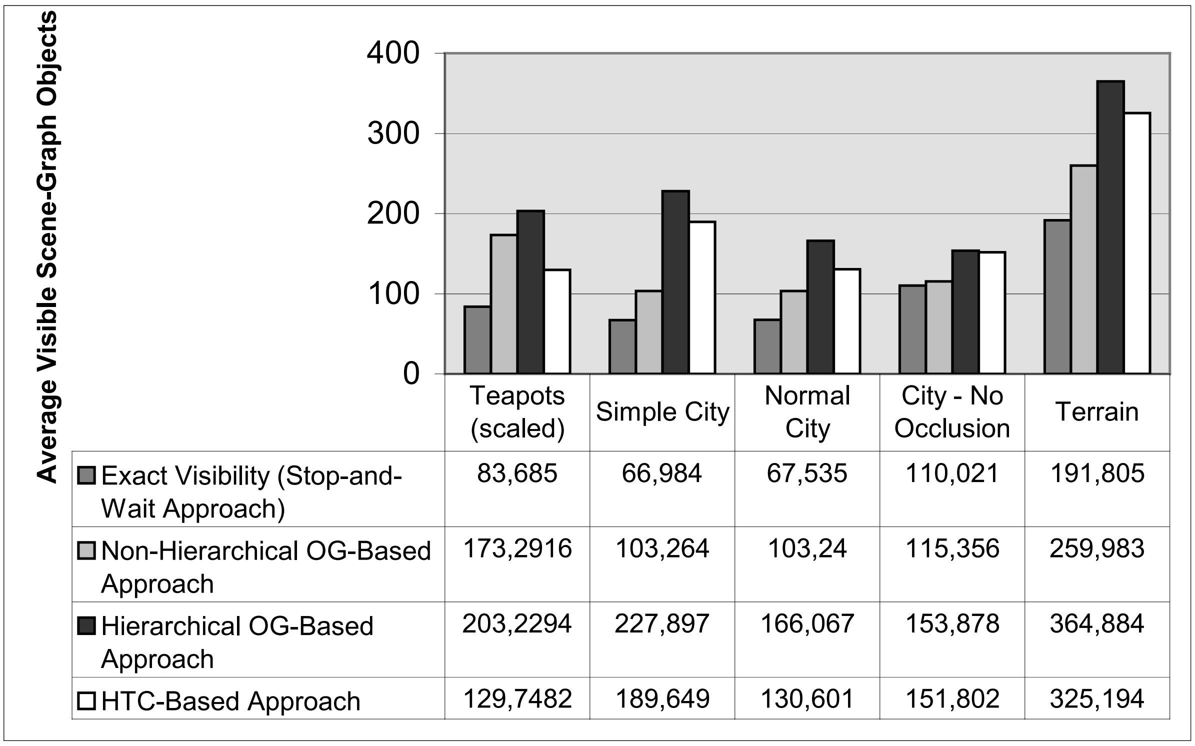
Fig. 5.5: Comparison of the frame times of the walkthrough of the terrain scene.

A few facts should briefly be pointed out:

- The performance of the various approaches differs very much from scene to scene and varies strongly even within one walkthrough. The figures highlight pretty clearly that doing no occlusion culling at all can be a good idea for some scenes while it is unbearable for others.
- As reasoned in the description of the walkthroughs in chapter 3.5, a few figures reveal strong peaks (especially in the city walkthroughs when lifting the camera above the height of the roofs). Obviously, some approaches are more susceptible to peaks while performing better in the 'usual' case (particularly the hierarchical OG-based approach) than others. One has to assess for each application anew if such peaks are tolerable.
- The two main characteristics of a scene that determine the performance of the various approaches are the average amount of occlusion and the average complexity of an object. If both (and especially the latter) are high, even the stop-and-wait approach can yield better results than the more sophisticated OG-based approaches due to the more precise classification.
- The HTC-based approach performs nearly always better than the other approaches and shows less distinct peaks than the OG-based approaches. Sometimes it is even close to the optimal time. This confirms the assessment that it is actually a very reasonable way to do occlusion culling with hardware occlusion queries.

In the following, some other aspects apart from the performance itself are compared. It must be emphasized once more that many characteristics are not inherent for a specific approach, but are rather valid only for the very parameters used for these tests. Furthermore, it must be stressed that the concrete values to which the parameters were set are by no means claimed to be optimal in all cases, but turned out to be a satisfactory compromise for all test scenes with respect to performance.

Since the teapot scene differs significantly from all other scenes concerning its complexity, the values had to be divided by a scale factor in order to make all results lie approximately within the same range (as it has already be done in various figures up till now). Otherwise, the differences within the other scenes would have been hardly perceptible. However, this has no impact on the correctness of the tendencies for the teapot scene.



	Teapots (scaled)	Simple City	Normal City	City - No Occlusion	Terrain
▨ Exact Visibility (Stop-and-Wait Approach)	83,685	66,984	67,535	110,021	191,805
▨ Non-Hierarchical OG-Based Approach	173,2916	103,264	103,24	115,356	259,983
■ Hierarchical OG-Based Approach	203,2294	227,897	166,067	153,878	364,884
□ HTC-Based Approach	129,7482	189,649	130,601	151,802	325,194

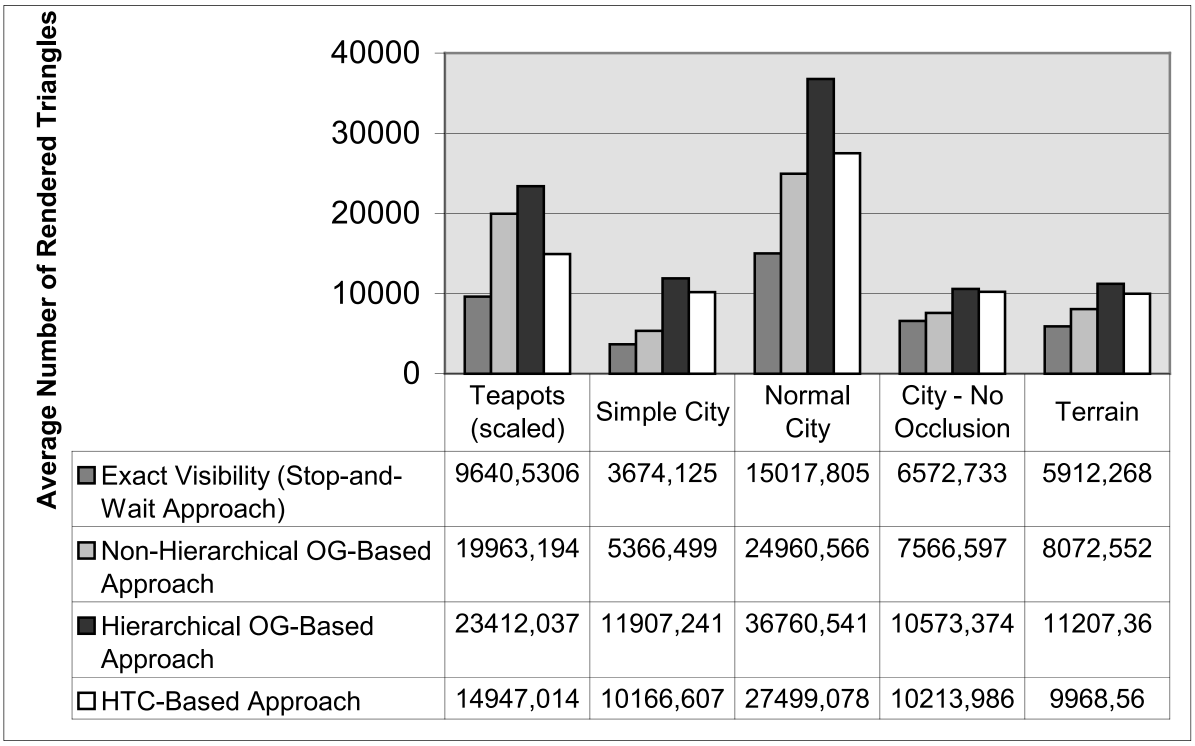
Fig. 5.6: Comparison of the number of scene-graph objects classified as visible (and thus rendered) by the various approaches. The values of the teapot scene were divided by 5.

Fig. 5.6 illustrates how many objects are on average classified as visible with the various approaches:

- Hardly surprising, the stop-and-wait approach (= exact classification) usually yields the most exact classification, since it issues the tests on a per-object basis and loses no occlusion. Note however, that the number is still an over-estimation as the geometry itself is approximated by AABBs.
- The non-hierarchical OG-based approach is in most cases just slightly worse than the exact classification. The difference originates from the application of *Assumed Visibility* and the more conservative approximations by SSBBs instead of AABBs.
- The hierarchical OG-based approach is worst in all test scenes. In addition to the application of *Assumed Visibility* and the approximation by SSBBs, the coarser subdivision (tests are not done on a per-object basis any more) increases the number of visible objects considerably. Besides, due to the on average larger SSBBs, the effect of the visibility threshold is somewhat diminished.
- The results of the HTC-based approach are hard to generalize: On one hand, it suffers from the same coarse subdivision as the hierarchical OG-based approach. Moreover, it loses a little bit of occlusion in cases where previously visible nodes have become occluded. On the other hand, the application of AABBs as testing geometry has turned

out to have a remarkably positive effect so that its classifications are in general more precise than the hierarchical OG-based approach.

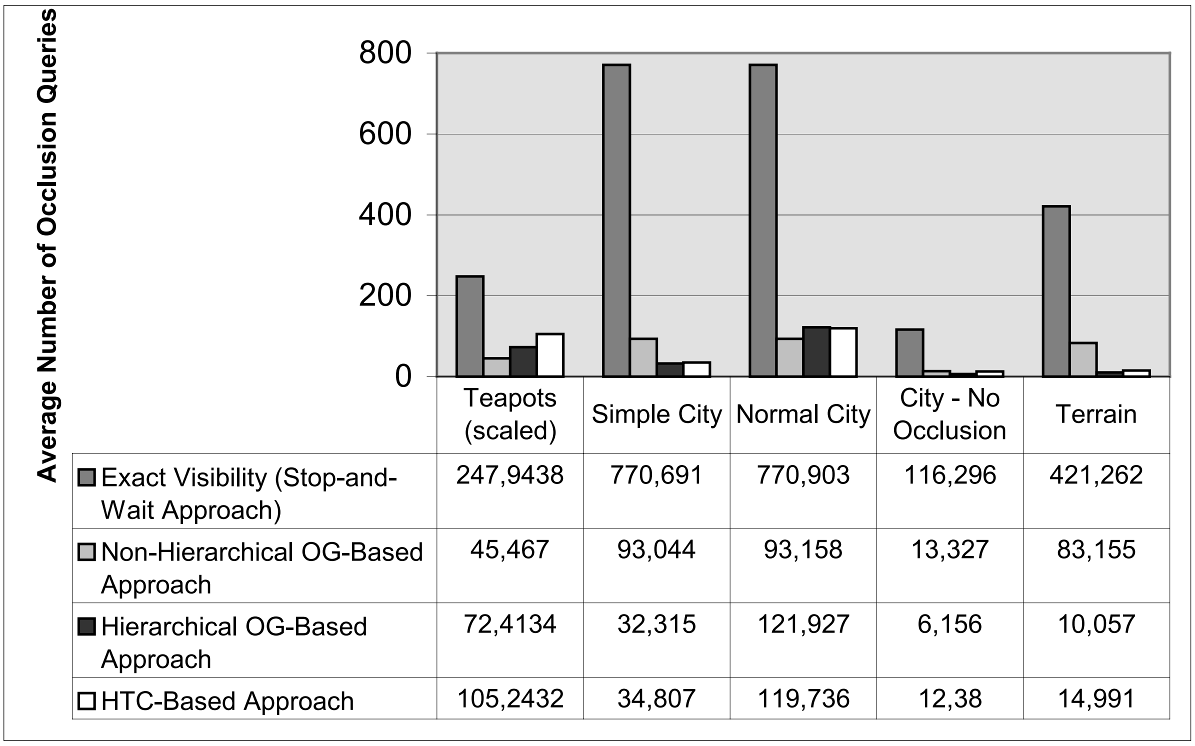
The results of Fig. 5.7 are tightly related to those of Fig. 5.6 (basically it boils down to multiplying them with the average number of triangles per object) and thus need no separate discussion but are provided for the sake of completeness.



	Teapots (scaled)	Simple City	Normal City	City - No Occlusion	Terrain
▩ Exact Visibility (Stop-and-Wait Approach)	9640,5306	3674,125	15017,805	6572,733	5912,268
▨ Non-Hierarchical OG-Based Approach	19963,194	5366,499	24960,566	7566,597	8072,552
■ Hierarchical OG-Based Approach	23412,037	11907,241	36760,541	10573,374	11207,36
□ HTC-Based Approach	14947,014	10166,607	27499,078	10213,986	9968,56

Fig. 5.7: Comparison of the average number of triangles rendered with the various approaches. The values of the teapot scene were divided by 100.

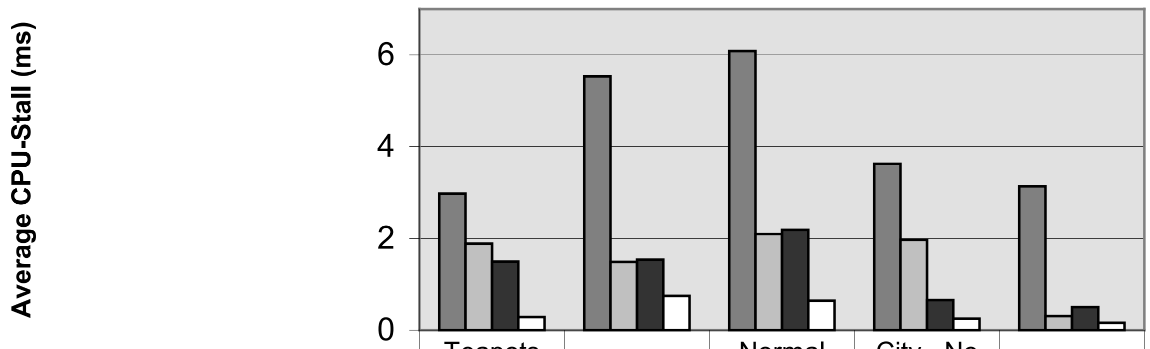
Fig. 5.8 illustrates that the number of occlusion queries has successfully been reduced by both OG-based approaches as well as the HTC-based approach. This was achieved on one hand by the application of *Early Rejection* and *Assumed Visibility* and on the other hand by incorporating a hierarchy which combines several objects and manages to classify vast parts as invisible with only few tests.



	Teapots (scaled)	Simple City	Normal City	City - No Occlusion	Terrain
■ Exact Visibility (Stop-and-Wait Approach)	247,9438	770,691	770,903	116,296	421,262
▨ Non-Hierarchical OG-Based Approach	45,467	93,044	93,158	13,327	83,155
■ Hierarchical OG-Based Approach	72,4134	32,315	121,927	6,156	10,057
□ HTC-Based Approach	105,2432	34,807	119,736	12,38	14,991

Fig. 5.8 Comparison of the average number of occlusion queries per frame for the various approaches. The values of the teapot scene were divided by 10.

Finally, Fig.5.9 compares the average time wasted by CPU-stalls. This duration can be determined by measuring the execution time of the API call that fetches a result: Since the overhead for requesting an available result is negligible, this duration reflects the time of the stall quite precisely.



	Teapots (scaled)	Simple City	Normal City	City - No Occlusion	Terrain
▨ Exact Visibility (Stop-and-Wait Approach)	2,9779	5,536	6,088	3,624	3,139
▨ Non-Hierarchical OG-Based Approach	1,8856	1,488	2,096	1,967	0,307
■ Hierarchical OG-Based Approach	1,4939	1,537	2,188	0,655	0,505
□ HTC-Based Approach	0,2895	0,746	0,64	0,253	0,163

Fig. 5.9: Comparison of the average time per frame spent waiting for results of occlusion queries for the various approaches. The values of the teapot scene were divided by 10.

As expected, much CPU-time is lost with the stop-and-wait approach. The OG-based approaches usually take less than half the time (which one is actually better differs from scene to scene) and the HTC-based approach manages to avoid them almost completely. This aspect is somewhat remarkable indeed: Although the OG fulfills its purpose to increase the parallelism between CPU and GPU (at least once it has been created), doing entirely without any constraints is still better. As reasoned in previous chapters, the key is both being able to query if a result is present without incurring any stall and having enough alternative work for the CPU if no result is available at the moment. This paradigm is also applied by the OG-based approaches, yet obviously the tight constraints often foil an alternative supplement of work.

The conclusion of all results is that the HTC-based approach is most suitable for the vast majority of cases, yet the question remains whether the OG-based approaches have their right to exist as well. Apart from being a subject of research that has served well to gain many insights and thus being an important step towards the HTC-based approach, the hierarchical version is actually probably useless due to its tendency to classify more objects as visible than any other approach and the considerable effort for a proper implementation. The situation is a little bit different with the non-hierarchical version: Although it is still more complex than the HTC-based approach and definitely has some flaws – the significant CPU overhead most likely being the worst – it usually yields a more precise classification than the HTC-based approach as it operates on a per-object basis (instead of a per-node basis of a hierarchy). Therefore, if the geometry of all objects is really extremely complex, which does not only mean many triangles but also many textures and possibly expensive effects like pixel shaders, the more accurate classification could outweigh the all additional costs so that it would actually exceed the HTC-based approach. Since the geometry tends to grow more and more complex while the CPUs become constantly faster, this aspect is likely to become even more important in the future.

Still one might wonder, if doing occlusion culling with either the non-hierarchical OG-based approach or – more likely – the HTC-based approach will always pay off. The answer is no:

- Scenes with no occlusion at all are still slowed down. At first sight, this might contradict the results with the HTC-based approach for the according test-scene, but as mentioned in section 4.3.3, these gains originate from the visibility threshold and the hierarchical traversal, not from the occlusion culling itself: The latter is also possible without the HTC-approach and the visibility threshold trades off image quality for speed – if one is not willing to tolerate any visibility errors, these gains must not be taken into account.
- For static scenes like the interior of buildings, calculating visibility offline (e.g. using PVS) is probably still the better choice – at least for doing occlusion culling at a coarse level, yet refining the result by the HTC-based approach can be a reasonable option.
- All approaches presented in this master thesis are definitely not adept for a few non-adjacent, moving objects like the characters in a computer game. In this case, simply issuing an independent test for each character is definitely more suitable.

Finally, it must be pointed out that neither the HTC-based approach, nor the non-hierarchical OG-based approach can guarantee a fixed frame rate. At least the HTC-based approach will in many cases increase the frame rate compared to rendering without occlusion culling, but to which extent depends on many factors and can not be determined by the user. If such a constant frame rate is in fact required, it only remains to compromise image quality with speed. This can be achieved for instance by adapting the visibility threshold T_V with respect to the current performance instead of using a static value or by using LOD (see chapter 2.4).

6 Conclusions

The driving motivation for this research was to sustain high frame rates in order to convey the impression of fluid motion in real-time rendering by relieving the GPU from drawing objects being occluded by others. The goal was to develop an approach for solving the visibility problem with the following properties: The approach must be based on the application of hardware occlusion queries and should be fast by maximizing the parallelism between CPU and GPU, generally applicable as well as realizable with a sensible effort. This final chapter presents the most important insights and proposes some ideas how the work could be further extended and improved.

6.1 Conclusion

Computing visibility for an arbitrary three-dimensional environment is a non-trivial problem. It is a big challenge to design a general algorithm that yields good results under all possible circumstances. Many purely CPU-based approaches have been presented so far, whereas there has been little work on solutions which are optimised for the use of specialized hardware extensions. The main difference is the requirement to shift as much work from the CPU to the GPU as possible while maintaining parallel execution and still avoiding a too conservative estimation of the set of visible objects.

Due to its major improvements, which consist of the ability to deal with more tests concurrently, to ask for the availability of results without incurring a CPU-stall and the more detailed information about the actual amount of visibility, the NV_occlusion_query extension is a very reasonable utility that permits the design of algorithms which are superior to approaches using the more limited HP_occlusion_test extension. However, an important insight is that even when not considering potential stalls, occlusion queries are inherently expensive and should therefore be minimized. The duration of stalls, on the other hand, is impossible to predict and the overall amount of wasted CPU-time does not grow with the number of issued occlusion queries in a linear fashion.

Tackling this problem by querying the accessibility of results can only be a solution when an alternative supplement of work for the CPU exists. In this context, the amount of constraints given by the introduction of an occlusion graph has turned out to be counter-productive, as it often prevents the availability of further work.

In order to obtain satisfactory results, every kind of coherence must be exploited as much as possible. However, while this applies to the majority of scenes, in some cases the absence of coherence can actually deteriorate performance when coherence is assumed. The extent to which temporal and spatial coherence of a scene is supposed to occur is determined by various parameters which have – in addition to the algorithm itself – a tremendous impact on the overall behaviour and performance. When optimising them for specific scenes, one usually faces tradeoffs. For instance, a reduction of occlusion queries is normally compromised with a less precise classification of visibility.

An important aspect is the insight that an occlusion query only has a chance to pay off in cases where the original geometry exceeds a certain complexity. The decision if a further refinement is desirable should be based on this concern. Although for the sake of simplicity only the triangle count of an object has been considered to determine its complexity, the costs of rendering also comprise its textures (which could squander precious bandwidth and might interfere with caching strategies) and eventually expensive computations like pixel shaders to achieve realistic shading. Taking into account these factors as well could lead to further increase in the efficiency. A recent proposal for rendering time estimations can be found in [Wimm03].

The most important contribution of this thesis is the hierarchical temporal-coherence based approach (HTC-based approach). Its design is based on insights gained with both OG-based approaches and takes into consideration most aspects mentioned in this conclusion. The HTC-based approach has been proven to outperform any other tested algorithm for the majority of scenes. It manages to avoid CPU-stalls almost entirely, it does not show any significant CPU-overhead, it does not suffer from a remarkable complexity, and it yields an acceptably accurate visibility classification. However, in rare cases, the usually tighter set of potentially visible objects of the non-hierarchical OG-approach could outweigh the additional costs, and for scenes without noteworthy occlusion, the best performance is sometimes still achieved when doing no occlusion culling at all.

6.2 Further Ideas

As reasoned above, the HTC-based approach already offers an adept solution to the problem of occlusion culling with hardware occlusion queries in many cases. In order to increase its generality even further, some enhancements are still imaginable.

The probably most urgent modification for many applications is the incorporation of dynamic objects. These objects should be free to be added or removed as desired and are not bound to any fixed position, orientation or size. A typical example are the characters of a computer game, but generally speaking all items that are possibly subject to any change fall into this category. It seems reasonable to explicitly distinguish between static and dynamic geometry, as dealing with the latter tends to involve more effort. As a proposal, the spatial hierarchy could be constructed out of the entire static geometry, while all dynamic objects are inserted each frame anew without affecting the structure of the spatial hierarchy itself, and removed after completing the traversal. Respective research would have to take into account issues like the validity of AABBs (along with a quick computation and potential caching strategies), examinations about the applicability of the formula used for temporal coherence, as well as a possibly more accurate visibility classification for dynamic objects (on average, dynamic objects are modeled in more detail resulting in a higher complexity). If the number of dynamic objects becomes overwhelming, even handling them in a completely separate way might pay off (see [Ratc01]). Respective algorithms could be performed subsequent to the actual HTC-based algorithm for the static geometry.

Further ideas concern the ability of the algorithm to adapt to the current circumstances in an even more distinct way: Especially the formula for computing *VF*, the number of frames an object is assumed to stay visible, could yield more precise results by incorporating further aspects like the current speed of the observer, both the viewing direction and the direction of movement as well as maintaining a small history about the recent development of visibility for each object. As always, care must be taken that the additional computational costs do not exceed the resulting benefits.

As already mentioned, making the visibility threshold T_V dependent on the past few frame times instead of using a fixed value could help to sustain a certain frame rate by compromising image quality with speed. Methods of control engineering like using a hysteresis must be employed in order to avoid constant strong overshooting.

Finally, within some areas of a scene, doing no occlusion culling at all might be the best idea. Therefore the algorithm could be extended in a way that permits to disable occlusion culling automatically if the number of occludees drops below a certain threshold. While being switched off, a few stochastically chosen objects could still be tested to decide when occlusion culling should be turned on again.

Appendix A – Implementation Details

Pre-Allocating Memory

Throughout the implementation to this master thesis, several data structures support both all kinds of traversals and – in the OG-based approaches – the creation of the OG. Common data structures have been used like for instance heaps, queues, or hashtables, which make intensive use of dynamic memory – mostly for allocating and freeing single elements of lists comprising some kind of content and the pointer to another element. Using `new` and `delete` each time would slow down execution dramatically, since both operators are quite complex functions after all. The solution is to *pre-allocate* the memory. This means to allocate the memory once and reuse it each frame without returning it to the operating system. This strategy can either be realized by knowing the exact number of needed elements (or more precisely: knowing an upper boundary), or – more flexibly – by encapsulating the allocations as a class implementing a pool for a specific type that permits dynamic growth. Instead of calling the ordinary `new` operator, something like `pool.new()` must be called to obtain a valid pointer. When all elements of the pool are currently in use, it automatically increases its size by allocating a certain number of further elements. At the beginning or at the end of a frame, the pool must be reset which clears an internal counter so that further memory request would return the same pointers as previous requests. Summarizing, memory pre-allocation proved to be an appropriate way to enhance performance without notably complicating the implementation.

Maintaining Data Structures in Addition to the Scene Graph

Within all approaches presented in this master thesis, the scene has been modelled by using both a scene graph as well as a – hierarchical or non-hierarchical – other data structure. The scene graph was not sufficient on its own for the following reasons:

- A traversal took much too long for our purpose.
- Objects were not uniquely identifiable.
- Other spatial data structures had to be employed for acceleration purposes.

The first point meant quite a tough problem at first: Within complex scenes, a complete traversal took sometimes even longer than the time available for the whole frame (about 16ms when requesting 60fps). The exact duration varied very much depending on the number of objects that could be culled away on a higher level by view-frustum culling, yet generally speaking, another solution had to be found – especially for the OG-based approaches, where all objects must be known before the creation of the OG can be started.

The second point refers to the fact, that nodes can be referenced by multiple parents (as described in section 2.3). This is problematic, when one has to store additional data for each instance (meaning appearances on the screen), in contrast to each node. Since the pointer itself is not sufficient, techniques must be applied where a key is computed in connection with a hash value derived from the transformation matrix, yet this is an additional effort that becomes costly when done very often (besides it is clumsy).

Furthermore, scene graphs are a convenient way to store the contents of the scene, but they are by no means the most appropriate data structure with respect to acceleration. Therefore, it is common practice to employ a scene graph for the storage of the actual scene description, but to make use of other spatial data structures in order to speed up rendering. The latter do not replace the scene graph entirely, but reference single nodes within. While the geometry and all drawing attributes stay within the scene graph, all data related to acceleration (AABBs, previous visibility classification, etc.) is stored within the supporting spatial data structure – and this can be done for each visual instance, which renders elaborate hashing techniques unnecessary. The coexistence of a scene graph and another data structure (a list, but it could be any other data structure as well) is illustrated in Fig. A.1: Note that single scene-graph nodes can get referenced by multiple list elements.

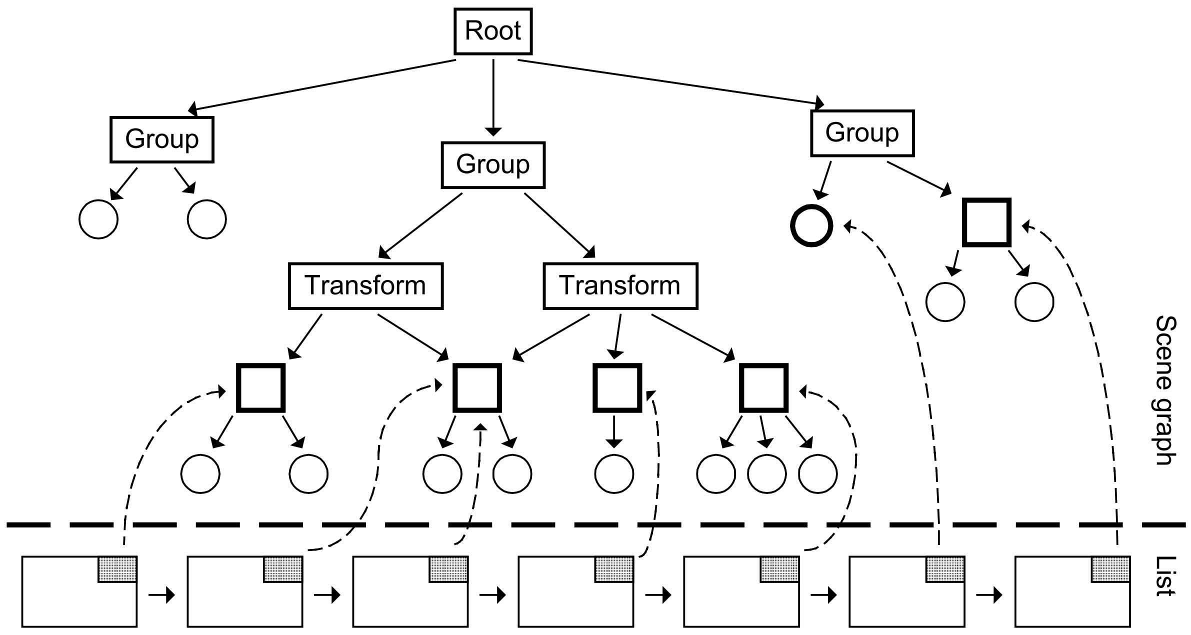


Fig. A.1: The scene graph is supported by a list referencing nodes. Boxes stand for group nodes in some form whereas circles represent leaves (e.g. actual geometry, lights,…).

Concerning the implementation, the scene graph is traversed once when a new scene has been loaded, gathering all relevant instances together with their transformation matrices, and this data is used to construct the supporting data structure (which is a kd-tree for the hierarchical approaches). However, doing so is not for free, for subsequent modifications of the scene graph are problematic: These scene-graph nodes and their sub-graphs being referenced by the other data structure still get traversed as usual, when they are about to be rendered, so subsequent changes automatically take effect. However, altering any super ordinate transformation can invalidate AABBs which must be cached for speed reasons (calculating the AABBs for all objects each frame is no option). Therefore, the system would need to detect such modifications in order to selectively update the affected AABBs. Moreover, it would also have to be noticed somehow when new objects are added or existing ones get removed.

One possibility to address these issues might be to distinguish between static geometry which is assumed to stay where it is and dynamic geometry which would have to be checked for changes at each frame – this approach is pursued by some commercial rendering engines. In this case, a spatial hierarchy might comprise static geometry only whereas the dynamic objects are handled separately in order to avoid costly updates and both sets are merged later in the frame. Dealing with an increasing number of dynamic objects requires a more elaborate organization for the dynamic geometry than a simple list as well: [Ratc01] presents an efficient way to maintain a hierarchy for objects in motion. However, it should be emphasized that these issues have not been considered by the implementation in any form.

Bibliography

[Aila00] Aila, T., and V. Miettingen, "*Umbra Reference Manual*", Hybrid Holding Ltd., October 2000.

[Aire90] Airey, John M., "*Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*", Ph.D. Thesis, Department of Computer Science, University of North Carolina at Chapel Hill, July 1990.

[Akel88] Akeley, Kurt, and T. Jermoluk, "*High-Performance Polygon Rendering*", SIGGRAPH '88 Proceedings, pp. 239 – 246, 1988.

[Assa00] Assarsson, Ulf, and Thomas Möller, "*Optimized View Frustum Culling Algorithms for Bounding Boxes*", journal of graphics tools, vol. 5, no. 1, pp. 9 – 22, 2000.

[Bare96] Barequet, G., B. Chazelle, L.J. Guibas, J.S.B. Mitchell, and A. Tal, "*BOXTREE: A Hierarchical Representation for Surfaces in 3D*", Proceedings of Eurographics '96, pp. 387 – 396, 1996.

[Bart98] Bartz, Dirk, Michael Meißner, and Tobias Hüttner, "*Extending Graphics Hardware for Occlusion Queries in OpenGL*", Stephen N. Spencer, ed., 1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware, pp. 97 – 104, September 1998.

[Bitt98] Bittner, Jiří, Vlastimil Havran, and Pavel Slavik, "*Hierarchical Visibility Culling with Occlusion Trees*", Franz-Erich Wolter and Nicholas M. Patrikalakis, ed., Proceedings of the Conference on Computer Graphics International 1998, pp. 207 – 219, June 1998.

[Bitt01a] Bittner, Jiří, and Jan Přikryl, "*Exact Regional Visibility Using Line Space Partitioning*", Technical Report TR-186-2-01-06, Institute of Computer Graphics and Algorithms, Vienna University of Technology, March 2001.

[Bitt01b] Bittner, Jiří, Vlastimil Havran, „*Exploiting coherence in hierarchical visibility algorithms*", Journal of Visualization and Computer Animation 2001; 12, pp. 277 – 286, December 2001.

[Broc02] Brockhaus, Manfred, Anton Ertl, Andreas Krall, "Übersetzerbau", script to the according lecture held at the Institute of Computer Languages, Department of Compilers, Vienna University of Technology, 2002.

[Clark76] Clark, James H., "*Hierarchical Geometric Models for Visible Surface Algorithms*", Communications of the ACM, vol. 19, October 1976.

[Cohe02] Cohen-Or, Daniel, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand, "*A Survey of Visibility for Walkthrough Applications*", IEEE Transactions on Visualization and Computer Graphics, 2002.

[Coor96] Coorg, Satyan, and Seth Teller, "*Temporally Coherent Conservative Visibility*", Proc. 12th Annual ACM Symposium on Computational Geometry, pp.78 – 87, May 1996.

[Corm90] Cormen, T.H., C.E. Leiserson, and R. Rivest, "*Introduction to Algorithms*", MIT Press, Inc., 1990.

[Crai02] Craighead, Matt, "*NV_occlusion_query Specification*", NVidia Corporation, 2002. http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt

[Deco99] Decoret, Xavier, Gernot Schaufler, François Sillion, and Julie Dorsey, „*Multi-layered Impostors for Accelerated Rendering*", Proceedings of Eurographics '99, vol. 18, no. 3, pp 61 – 72, 1999.

[Down01] Downs, L., T. Möller, and C. H. Séquin, "*Occlusion Horizons for Driving through Urban Scenery*", 2001 Symposium on Interactive 3D Graphics. ACM SIGGRAPH, 2001.

[Dura97] Durand, Frédo, George Drettakis, and Claude Puech, "*The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool*", SIGGRAPH '97 Proceedings, pp. 89 – 100, August 1997.

[Dura00] Durand, Frédo, George Drettakis, Joëlle Thollot, and Claude Puech, "*Conservative Visibility Preprocessing Using Extended Projections*", SIGGRAPH 2000 Proceedings, pp. 239 – 248, July 2000.

[Eber00] Eberly, David, "*3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*", Morgan Kaufmann Publishers Inc., 2000

[Eccl00] Eccles, Allen, "*The Diamond Monster 3Dfx Voodoo 1*", Gamespy Hall of Fame, 2000.
`http://www.gamespy.com/halloffame/october00/voodoo1/`

[Egge02] Eggert, David W., Kevin W. Bowyer, and Charles R. Dyer, "*Aspect Graphs: State-of-the-Art and Applications in Digital Photogrammetry*", Proceedings ISPRS 17th Cong.: International Archives Photogrammetry Remote Sensing, pages 633 – 645, August 1992.

[Funk03] Funkhouser, Thomas A., and Carlo H. Séquin, "*Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments*", SIGGRAPH '93 Proceedings, pp. 247 – 254, August 1993.

[Gamm94] Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides, "*Design Patterns*", Addison-Wesley, 1994.

[Garl97] Garland, Michael, and Paul S. Heckbert, "*Surface Simplification Using Quadratic Error Metrics*", Proceedings of SIGGRAPH 97, pp. 209 – 216, August 1997.

[Gieg02] Giegl, Markus, and Michael Wimmer, "*Unpopping: Solving the Image-Space Blend Problem*", submitted to Journal of Graphics Tools, 2002.

[Gold87] Goldsmith, Jeffrey, and John Salmon, "*Automatic Creation of Object Hierarchies for Ray Tracing*", IEEE Computer Graphics and Applications, vol. 7, no. 5, pp. 14 – 20, May 1987.

[Gour71] Gouraud, H., "*Continuous Shading of Curved Surfaces*", IEEE Transactions on Computers, vol. C-20, June 1971.

[Gree93] Greene, Ned, Michael Kass, and Gavin Miller, "*Hierarchical Z-Buffer Visibility*", SIGGRAPH '93 Proceedings, pp. 231 – 238, August 1993.

[Havr00] Havran, Vlastimil, "*Heuristic Ray Shooting Algorithms*", Ph.D. Thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, November 2000.

[Hear94] Hearn, Donald, and M. Pauline Baker, "*Computer Graphics*", Second Edition, Prentice-Hall, Inc., 1994.

[Helm94] Helman, James L., „*Architecture and Performance of Entertainment Systems*“, SIGGRAPH 94 Course Notes: Designing Real-Time Graphics for Entertainment, 1994.

[Hong97] Hong, L., S. Muraki, A. Kaufman, D. Bartz, and T. He., "*Virtual voyage: Interactive navigation in the human colon*", SIGGRAPH '97 Proceedings, pp. 27 – 34, 1997.

[Huds97] Hudson, T., D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang, "*Accelerated Occlusion Culling using Shadow Frusta*", Proceedings of the 13th Annual ACM Symposium on Computational Geometry, pp. 1–10, June 1997.

[King00] King, Yossarian, "*2D Lens Flare*", in Mark DeLoura, ed., Game Programming Gems, Charles River Media, pp. 515 – 518, 2000.

[Kolt00] Koltun, Vladlen, Yiorgos Chrysanthou, and Daniel Cohen-Or, „*Virtual Occluders: An Efficient Intermediate PVS Representation*“, 11th Eurographics Workshop on Rendering, pp. 59 - 70, 2000.

[Kuma96] Kumar, Subodh, and Dinesh Manocha, "*Hierarchical Visibility Culling for Spline Models*", Graphics Interface '96, pp. 142 – 150, May 1996.

[Leng98] Lengyel, Jerome, "*The Convergence of Graphics and Vision*", Computer, pp. 46 – 53, July 1998.

[Maci95] Maciel, P., and P. Shirley, "*Visual Navigation of Large Environments Using Textured Clusters*", Proceedings 1995 Symposium on Interactive 3D Graphics, pp. 96 – 102, 1995.

[McCu00] McCuskey, Mason, "*Using 3D Hardware for 3D Sprite Effects*", in Mark DeLoura, ed., Game Programming Gems, Charles River Media, pp. 519 – 523, 2000.

[McRe99] McReynolds, Tom, David Blythe, Brad Grantham, and Scot Nelson, "*Advanced Graphics Programming Techniques Using OpenGL*", SIGGRAPH '99 course notes, 1999.

[Meiß99] Meißner, Michael, Dirk Bartz, Tobias Hüttner, Gordon Müller, and Jens Einighammer, "*Generation of Subdivision Hierarchies for Efficient Occlusion Culling of Large Polygonal Models*", Technical Report WSI-99-13, WSI/GRIS, University of Tübingen, 1999.

[Möll02] Möller, Thomas Akenine, and Eric Haines, "*Real-Time Rendering*", Second Edition, A K Peters, 2002.

[Morl00] Morley, Mark, "View-Frustum Culling", 2000. http://www.markmorley.com/opengl/frustumculling.html

[Nade98] Nadeau, David R., Michael J. Bailey, and Michael F. Deering, "*Introduction to Programming with Java3D*", SIGGRAPH '98 Lectures, http://java.sun.com/products/java-media/3D/collateral/class notes/java3d.htm

[Patt98] Patterson, David A., and John L. Hennessy, "*Computer Organization & Design*", Second Edition, Morgan Kaufmann Publishers, 1998.

[Ratc01] Ratcliff, John w., "*Sphere Trees for Fast Visibility Culling, Ray Tracing, and Range Searching*", Mark DeLoura, ed., Game Programming Gems 2, Charles River Media, pp. 384 – 387, 2001.

[Salo02] Salomon, B., K. Hillesland, A. Lastra, D. Manocha, "*Fast and Simple Occlusion Culling using Hardware-Based Depth Queries*", submitted to Journal of Graphics Tools, 2002.

[Same89] Samet, Hanan, "*The Design and Analysis of Spatial Data Structures*", Addison-Wesley, 1989.

[Scha95] Schaufler, Gernot, "*Dynamically Generated Impostors*", GI Workshop on "Modeling – Virtual Worlds – Distributed Graphics", D.W. Fellner, ed., Infix Verlag, pp. 129 – 135, November 1995.

[Scha97] Schaufler, Gernot, "*Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes*", Eurographics Rendering Workshop 1997, pp. 151 – 162, 1997.

[Scha00] Schaufler, Gernot, Julie Dorsey, Xavier Decoret, and François Sillion, „*Conservative Volumetric Visibility with Occluder Fusion*", SIGGRAPH 2000 Proceedings, pp. 229 – 238, July 2000.

[Scot98] Scott, N., D. Olsen, and E. Garnett, "*An Overview of the VISUALIZE fx Graphics Accelerator Hardware*", Hewlett-Packard Journal, pp. 28 – 34, May 1998.

[Shad96] Shade, J., D. Lischinski, D. Salesin, T. DeRose, and J. Snyder, "*Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments*", SIGGRAPH '96 Proceedings, pp. 75 – 82, August 1996.

[Shir93] Shirman, Leon A., and Salim S. Abi-Ezzi, "*The Cone of Normals Technique for Fast Processing of Curved Patches*", Proceedings of Eurographics '93, vol. 12, no. 3, pp. 261 – 272, 1993.

[Stan02] Staneker, Dirk, "*A First Step towards Occlusion Culling in OpenSG PLUS*", 1. OpenSG Symposium Darmstadt, WSI/GRIS, University of Tübingen, 2002.

[Tell91] Teller, Seth J., and Carlo H. Séquin, "*Visibility Preprocessing For Interactive Walkthroughs*", SIGGRAPH '91 Proceedings, pp. 61 – 69, July 1991.

[Watt93] Watt, Alan, "*3D Computer Graphics*", Second Edition, Addison-Wesley, 1993.

[Wern94] Wernecke, Josie, "*The Inventor Mentor*", Addison-Wesley, 1994.

[Wimm01] Wimmer, Michael, "*Representing and Rendering Distant Objects for Real-Time Visualization*", Ph.D. Thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, June 2001.

[Wimm03] Wimmer, Michael, Peter Wonka, " *Rendering Time Estimation for Real-Time Rendering*", Eurographics Symposium on Rendering 2003, 2003.

[Wonk99] Wonka, Peter, and Dieter Schmalstieg, "*Occluder Shadows for Fast Walkthroughs of Urban Environments*", Computer Graphics Forum, vol. 18, no. 3, pp. 51 – 60, 1999.

[Wonk00] Wonka, Peter, Michael Wimmer, and Dieter Schmalstieg, "*Visibility Pre-processing with Occluder Fusion for Urban Walkthroughs*", 11th Eurographics Workshop on Rendering, pp. 71 – 82, 2000.

[Wonk01] Wonka, Peter, Michael Wimmer, and François X. Sillion, "*Instant Visibility*", Proceedings of Eurographics 2001, Vol. 20, No. 3, pp. 411 – 421, September 2001.

[Woo99] Woo, Mason, Jackie Neider, Tom Davis, Dave Shreiner, "*OpenGL Programming Guide*", Third Edition, Addison Wesley, 1999.

[Zang98] Zhang, Hansong, "*Effective Occlusion Culling for the Interactive Display of Arbitrary Models*", Ph.D. Thesis, Department of Computer Science, University of North Carolina at Chapel Hill, July 1998.