Diplomarbeit

Shadow Volumes in Complex Scenes

ausgeführt am
Institut für Computergraphik und Algorithmen
der Technischen Universität Wien

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

durch

Christian Steiner
Matrikelnummer 9926216
Wagnerstrasse 29/12
A-2371 Hinterbrühl

Unterschrift

Datum

Abstract

Over the last 10 years, significant progress has been made in the field of computer graphics, especially in real-time rendering. Most noteworthy, the use of dedicated graphics accelerator hardware has found its way from the professional to the consumer market, and their ever increasing power has allowed for rendering almost photorealistic virtual environments while still maintaining interactive framerates.

Despite this rapid development, a significant element of computer graphics has been neglected for a long time. Shadows do not only provide more realistic-looking scenes, but also aid the viewer in perceiving spatial relationships. However, due to the additional computational requirements, it has been impossible for graphics accelerators to render shadows and keep framerates high enough to maintain the feeling of immersion for a long time. Although there are several approaches to realize shadows, dynamic environments with a multitude of light sources and complex objects still make high demands on the hardware.

In this thesis, a technique is presented to improve the performance of shadow volumes in *complex* scenes that consist of a large number of individual objects. Several optimization techniques have already been proposed that target applications where the rasterization of the shadow volume polygons is the main bottleneck. However, these optimizations usually assume that the number of individual objects in the scene is rather small compared to the number of geometric primitives (triangles). In such scenes, calculations can be accelerated by using low-polygon approximations of the actual geometry. Most of these existing optimization techniques relieve the graphics hardware at the cost of increased CPU load. If the CPU is already at peak load, these techniques do not achieve any performance gain, but rather worsen the bottleneck in the CPU stage, resulting in an even lower performance.

This thesis first presents an overview of current state-of-the-art shadowing techniques that are based on standard shadow volumes. Then, we will try to adapt parts of these techniques to work in complex scenes. Specifically, we will improve visibility determination for *shadow volume culling* in GPU-demanding scenes with lots of individual objects, as well as present a method for the fast creation of segmented (*clamped*) shadow volumes that tightly fit the shadow-receiving geometry using vertex programs (*shaders*).

Kurzfassung

Während der letzten 10 Jahre wurden auf dem Gebiet der Computergraphik und insbesondere dem Echtzeit-Rendering bedeutende Fortschritte erzielt. Nicht nur griff die Verwendung von dedizierter Hardware zur Berechnung dreidimensionaler Bilder vom Profi- nun auch ins Consumer-Segment über, auch die Leistungsfähigkeit der Grafikbeschleuniger ist in einen Bereich gestiegen, der es ermöglicht, nahezu fotorealistische virtuelle Umgebungen bei dennoch interaktiven Bildwiederholraten zu erzeugen.

Trotz dieser rasanten Entwicklung wurde ein wesentliches Element der Computergraphik lange Zeit vernachlässigt. Schatten sorgen nicht nur für realistischere Szenen, sondern unterstützen den Betrachter auch beim Erfassen räumlicher Zusammenhänge. Allerdings war die Grafikhardware lange Zeit nicht in der Lage, die zusätzlichen Berechnungen ohne signifikante Leistungseinbußen durchzuführen. Obwohl es mittlerweile mehrere Ansätze zur Realisierung von Schatten gibt, stellen auch heute noch dynamische Umgebungen mit vielen Lichtquellen und komplexen Objekten hohe Anforderungen an die verwendete Hardware.

Diese Diplomarbeit untersucht Methoden, um die Leistung des traditionellen shadow volume Algorithmus in *komplexen* Szenen zu verbessern. In diesem Zusammenhang nennen wir eine Szene komplex, wenn sie nicht nur eine hohe Anzahl an Geometrie beinhaltet, sondern diese auch auf viele individuelle Objekte verteilt ist. Es existieren bereits eine Reihe von Optimierungen, die auf Anwendungen abzielen, in denen die Rasterisierung der Schattenpolygone der Hauptflaschenhals ist. Diese Optimierungen gehen jedoch überwiegend davon aus, dass die Szene eine begrenzte Anzahl an individuellen Objekten enthält, die ihrerseits durchaus komplex sein können (d.h. aus vielen Dreiecken zusammengesetzt sind). In diesem Fall können viele Berechnungen vereinfacht werden, in dem anstelle der tatsächlichen Geometrie einfachere Annäherungen verwendet werden. Die meisten existierenden Optimierungen entlasten die Grafikhardware zulasten der CPU, weswegen sie stark von approximierter Geometrie profitieren. Wenn jedoch die CPU bereits stark ausgelastet ist, versagen diese Methoden, da sie den Flaschenhals in der CPU noch verstärken und somit sogar eine Verschlechterung der Performance bewirken.

Diese Diplomarbeit beschreibt zunächst den aktuellen state-of-the-art bei shadow volumes. Daran anschliessend werden Teile dieser optimierten Methoden genauer

untersucht und für die Verwendung in komplexen Szenen mit vielen Objekten angepasst. Wir präsentieren eine Verbesserung bei der Visibility-Bestimmung für Shadow Volume Culling bei anspruchsvollen Szenen mit vielen individuellen Objekten und präsentieren weiters Methoden zur schnellen Erstellung von segmentierten Shadow Volumes (*Clamping*).

Contents

1 Introduction

1.1 Real-Time Rendering

Like photorealistic rendering, real-time rendering is a part of the field of computer graphics. While photorealistic rendering deals with the appearance of the generated images, real-time rendering mainly focuses on creating those images fast enough to keep the viewer immersed in the scene. To achieve this goal, the images must be created at a rate that does not permit the viewer to distinguish individual frames. This requirement imposes a natural limit on the quality of the generated images. In many cases, a trade-off must be found between the visual acuity of the images and the speed at which the images can be generated (*rendered*).

The term *framerate* is used to describe the rate at which images are displayed on the screen. Its unit is *frames per second* (fps). Haines and Akenine-Möller [6] classify an application as *real-time* if its framerate is at least 15 fps. The individual perception of interactivity, however, greatly depends on the scene. While a slow walkthrough in a static scene may create a somewhat interactive feeling at as little as 6 frames per second, computer games with a lot of movement in a highly dynamic environment should provide at least 60 fps to avoid disturbing visual artifacts [52]. Generally, the framerate should not exceed monitor refresh rates to avoid *ghosting*.

Real-time rendering mainly deals with the creation of three-dimensional virtual environments. A few years ago, the task of creating those environments was performed entirely by the computer's main processor (CPU). With the exception of professional solutions like workstations manufactured by SGI (also known as Silicon Graphics), dedicated graphics hardware was practically nonexistent. In recent years, however, ever-increasing demands of computer graphic's killer application - computer games - have sponsored an amazing development in the consumer segment. With the introduction of the 3Dfx Voodoo 1 in 1996 [31], more and more of the 3D rendering process was shifted to dedicated graphics accelerators, also called *GPUs*, relieving the main processor from the demanding task of computing three-dimensional images. Henceforth, GPUs have increased dramatically both in power and complexity, allowing for realistic images while providing the necessary framerates for an interactive experience (Figure 1.1). While not absolutely required for real-time rendering, graphics accelerator hardware has become a necessity for almost all of today's real-time applications.

Figure 1.1: Luna, NVIDIA's new mascot character. Rendered in real-time with multi-layered and translucent clothing, reflections, displacement mapping. *(Image courtesy of NVIDIA Corp.)*

1.2 Shadows in Real-Time Rendering

Shadows play an important role in computer graphics. First of all, scenes that lack any kind of shadow don't create the impression of being realistic. A viewer will notice the absence of shadows immediately, which prevents the viewer from getting immersed in the scene. In a time of ever-increasing realism in virtual environments – be they computer games or architectural walk-throughs - , shadows have become practically a necessity to sustain the illusion of the artificial world. Perhaps even more important is the role of shadows in the viewer's perception of the scene. Without any shadows at all, the placement of objects relative to each other is often not clear [91]. A shadow serves as an anchor for an object, tying it to the ground or a certain spot in the air. For example, imagine a plate sitting on top of a table. If the plate does not cast a shadow onto the table, we cannot discern whether the plate is on the table or hovers slightly above it (see Figure 2.19).

Another property with shadow that is important with real-time rendering is the fact that they need not be physically accurate to create the correct impression. Similar to reflections, the human eye is rather forgiving and does not immediately recognize shadows that have a wrong shape. Wanger concludes that it is usually better to have incorrect shadows than not having any shadows at all [91]. This fact has been exploited especially in computer games for a long time (see Figure 1.2). Quite often, a dark circle beneath the feet of a person or a dark rectangle beneath a car is sufficient

Figure 1.2: Left: A character from the game Deus Ex (2001) with a simple circle as a shadow. Right: A garage with cars that cast fake shadows.

to create the illusion of a shadow and enhance the perceived realism of the scene. These "fake shadows" can be easily implemented using texturing, multitexturing and blending even on older graphics hardware.

However, with the rapid development of consumer graphics hardware, the demand for realistic scenes has risen just as fast, if not faster. With the capabilities of today's state-of-the-art graphics hardware, fake shadows are no longer satisfactory (especially in the gaming industry). Several techniques for rendering physically accurate shadows, even soft shadows, have existed for quite some time. These have become feasible only recently, made possible by the power that is provided on modern graphics hardware. One of them, namely stencil shadow volumes, is the main focus of this work.

1.3 Goals of this work

Together with shadow mapping, the shadow volume technique is among the most prominent ones for doing shadows in real-time rendering. In the last years, they have become increasingly popular with computer games, given the rapid development of consumer graphics accelerators. However, though they produce good-looking results, shadow volumes are still problematic in large scenes that contain a lot of geometry. They increase the number of polygons in the scene as well as burden the rasterizer stage, because shadow volumes tend to extend to infinity and thus cover a large portion of the screen. The goal of this work was to implement shadow volumes in a way that ensures interactive framerates in large urban scenes. To this end, we improve the standard shadow volume algorithm as well as reduce the number of shadow volume by *culling* irrelevant shadow casters (i. e., casters that do not occlude

Figure 1.3: This figure shows a typical view of the city model that was the basis for this thesis.

any visible object) and prevent shadow volumes from being drawn over empty space by *clamping* them to tightly fit around the shadow receiving geometry.

1.4 Main Contributions

The research done for this master thesis is based on the application of the shadow volumes algorithm in large environments, where the standard algorithm fails due to the increased overhead and load on the graphics hardware. As such, we use an existing optimization technique as a starting point and perform an in-depth analysis of its shortcomings, followed by several improvements to achieve interactive framerates in our target scenes.

The cornerstones of this work are as follows:

- First, we analyse several existing shadow volume optimization techniques (section 3.2). We also examine the interaction between different techniques and their behavior in different circumstances (scenes). Specifically, we analyse the *CC Shadow Volumes* algorithm described by Lloyd et al. [69] and the steps involved - shadow volume culling and clamping. Though designed to work in scenes that are comprised of many triangles, this technique does not provide acceptable performance in our target scenes. Therefore, we adapt and improve several steps involved in the original technique.

- Since the original algorithm does not yield acceptable performance, we develop a new, optimized technique to perform the culling of shadow volumes (section 4.3). This technique is based on the *Coherent Hierarchical Culling* visibility determination algorithm presented by Bittner et al. [13].
- To further increase both performance and accuracy of the algorithm, we apply an effective way to focus the light's view of the scene, which is needed for shadow volume culling (subsection 4.3.1). The focussing uses geometric intersection calculations that execute fast on the CPU and results in a highly improved culling speed without introducing much additional computational overhead.
- To speed up the computation of segmented (clamped) shadow volumes, we present a technique that uses the stencil buffer to compute occupied intervals and then uses a vertex shader to create multiple shadow volume segments (subsection 4.4.3).

Together, these contributions enable the interactive rendering of complex scenes that not only consist of many triangles, but also of a large amount of individual scene objects.

1.5 Structure of this document

This thesis is structured as follows. In chapter 2, we will cover concepts that are necessary to fully comprehend the remainder of this work, such as the rendering pipeline, programmable capabilities of recent graphics hardware, and theoretical concepts like scene graphs, visibility, and an overview of the most prominent shadowing techniques. The shadow volume algorithm is described in chapter 3, followed by an overview of the current state-of-the-art optimization techniques. chapter 4 first defines what makes a scene *complex* in this context, then closely examines key parts of the optimization techniques. First, we will explain the specific problems, then try to adress some of these deficiencies and improve them to work with complex scenes. Finally, in chapter 5 we will further discuss the pros and cons of the optimization techniques and closely examine their influence on their performance with two examples of complex scenes (the city scene[1] and a "light" version of the PowerPlant model[2]). Finally, chapter 6 briefly recaps the introduced algorithms and gives an outline of "open issues" and what can be improved in future work. The appendix contains an overview of the rendering engine that was used in this work as well as provide implementation-specific details.

[1] http://www.cg.tuwien.ac.at/research/vr/urbanmodels/index.html
[2] http://www.cs.unc.edu/~geom/Powerplant/

2 Related Work

2.1 3D Graphics Packages

Graphics packages contain a number of convenience functions for tasks common in computer graphics, like clearing a display screen, initializing parameters, or drawing primitives on a screen [47]. A primary goal in standardized graphics software is portability. Standardizing graphics software allows software to be easily moved between different hardware systems and be used in different applications and implementations. The lack of standards would complicate the transfer of programs designed for a specific hardware platform to another by requiring extensive rewriting of the programs.

In the following sections, we will examine two of the most prominent graphics libraries that are used for current real-time applications: Microsoft DirectX and OpenGL.

2.1.1 Microsoft DirectX

DirectX is a multimedia API that provides a standard interface to interact with graphics and sound cards, input devices and more. The first version appeared in 1995 and was then called "GameSDK". In its original form, it was targeted at developers using C and C++. Only with the release of its first managed offspring, DirectX 9.0, it has become possible to use C# or VB.NET with DirectX.

Virtually every modern graphics accelerator supports DirectX functions directly in hardware.

2.1.2 OpenGL

Like DirectX, OpenGL is a software interface for graphics applications. It was introduced by Silicon Graphics in 1992 and has come a far way since then. In 2000, SGI released its OpenGL implementation as open source, simplifying the development of hardware-accelerated drivers for many different operating systems.

Today, OpenGL is the best choice for developing portable, interactive 2D and 3D applications. Its specification is governed by the OpenGL Architecture Review Board (ARB), which consists of many of industry's leading graphics vendors (specifically NVIDIA and ATI) as well as a number of other companies (e. g., Microsoft) and is responsible for defining conformance tests and approving the latest OpenGL enhancements. Since there is a standard procedure for new extensions being incorporated into the OpenGL standard, the library evolves, though slower than DirectX, in a controlled manner.

Most OpenGL functions are executed in hardware on practically every modern graphics accelerator.

2.2 The Graphics Rendering Pipeline

The *graphics rendering pipeline* (or simply the pipeline) represents the core of real-time graphics. Its main function is to generate (*render*) a two-dimensional image that is then displayed on a viewing device. Specifically, it takes a viewpoint (or virtual camera), three-dimensional objects, their textures, light sources, lighting models and possibly more as input and outputs a two-dimensional array of pixels. The pipeline performs tasks like transforming vertices between different coordinate systems, lighting, clipping, texturing, and antialiasing.

In the real world, pipelines appear in many different functions, places, and physical forms. Possibly the most prominent representatives are ski lifts, oil pipelines, and assembly lines. A pipeline's main characteristic is its segmentation into several interconnected stages. Entities (skiers, assembly parts, oil) can move from one stage to the next only when the subsequent stage has been vacated (that is, has finished its task). Hence, the overall speed of the pipeline is determined by its slowest element, no matter how fast the other stages may be. Ideally, using a pipeline with n stages should result in a speed-up of a factor n with respect to a non-pipelined construction. For example, the overall throughput of a ski lift is proportional to the number of chairs or bars it contains.

Each pipeline stage is executed parallel to all the others, but it is stalled until the subsequent stage has finished its task. For example, if the seat attachment stage of a car assembly line takes three minutes, and every other stage takes two minutes, the fastest rate that can be achieved is still one car every three minutes. All other stages are idle for one minute until the seat attachment stage has completed its task. The slowest stage in a pipeline is called the *bottleneck*, because it determines the speed of the complete construction. One of the most challenging problems in designing an efficient pipeline is the distribution of tasks in several smaller, equally-sized subtasks, to keep stage idle times as low as possible.

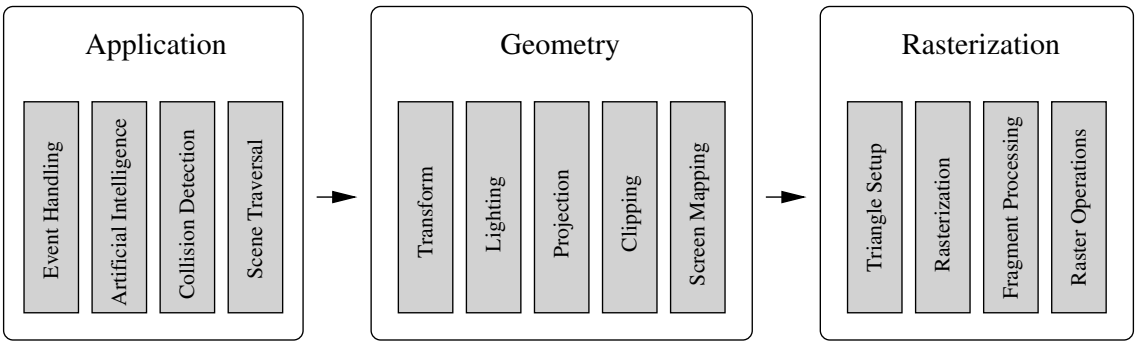

Figure 2.1: The traditional 3D rendering pipeline.

In the context of real-time rendering, the very same pipelining architecture can be found. At a coarse level, this pipeline can be subdivided into the three *conceptional* stages *application*, *geometry*, and *rasterization*. The pipeline can also be subdivided into *functional* stages. A functional stage performs a given task, but does not specify how this task is accomplished. From a low-level point of view, the pipeline consists of multiple *pipeline* stages. Each pipeline stage is executed simultaneously with all other pipeline stages. Pipeline stages may also be parallelized to achieve higher throughput (and higher rendering performance).

2.2.1 Application Stage

The application stage always executes in software. Hence, it is totally up to the developer to change the implementation. Common tasks for the application stages are input processing, collision detection and artificial intelligence (important for computer games). At the end of the application stage, it produces *rendering primitives* (points, lines, triangles, textures, ...) that are then fed into the subsequent stages of the pipeline. This implies that an important task of the application stage is to apply acceleration techniques like hierarchical view frustum culling (section 2.4) to reduce the number of primitives, which decreases the load on the following stages, thus increasing overall performance.

2.2.2 Geometry Stage

The geometry stage performs per-polygon and per-vertex operations. First, model- and viewing *transformations* are applied to all vertices. Depending on the chosen lighting model, the *lighting equation* is evaluated at each vertex. After lighting, the geometry stage performs a *projection* onto the viewing plane. The two most common kinds of projections are *orthographic projection* and *perspective projection*, which is a closer approximation to how human vision works. Primitives that lie partly within

the viewing volume are then *clipped*, so that each outside vertex is replaced by a vertex that is located directly at the intersection with the viewing volume. The clipped primitives are finally mapped to two-dimensional *screen coordinates*. The z coordinate is stored as well and later used in the z-buffer for visibility determination. Screen coordinates together with the z coordinate are also called *window coordinates*.

Obviously, the geometry stage performs a very demanding task: even for a single light source, each vertex requires approximately 100 floating point operations in the fixed-function pipeline[4].

2.2.3 Rasterizer Stage

Where the geometry stage operates mainly on vertices, the main task of the rasterizer stage is to compute the colors of individual pixels, given the transformed and projected vertices, colors and texture coordinates. The process of converting primitives to pixels is called *rasterization* or *scan conversion*. The information for each individual pixel is stored in a two-dimensional, rectangular array of colors called *color buffer*. The rasterizer operates on *fragments*, which is a data structure containing color, alpha, and z values. The number of fragments that can be processed in the rasterizer per second is called *fill rate*. Current state-of-the-art graphics hardware has a typical fill rate of several gigapixels per second.

Visibility is resolved in this stage as well. This means that while the whole scene has been rendered, the color buffer should ultimately contain only the colors of primitives that are visible from the point of view. Most graphics accelerator hardware accomplishes this with the *z-buffer* (also called *depth buffer*) algorithm. In addition to the color buffer, a second array (the z buffer) is used with the same dimensions. For each pixel, it stores the z-value from the viewpoint to the currently closest primitive. Whenever a primitive is rendered to a certain pixel, its color information is written to the color buffer only if the new z value is nearer than the value stored in the z buffer (which means that the new primitive occludes the previous primitive at this point). This algorithm allows the primitives to be rendered in any order, which is another reason for its popularity (however, to keep the number of z-writes low and increase performance, primitives should be rendered in a front-to-back order). This does not apply to (partially) transparent objects, these must be rendered after all fully opaque primitives and in back-to-front order.

Aside from the z and color buffers, there are a number of other common buffers. The *stencil buffer* which is a part of OpenGL usually holds one to eight bits per pixel. Primitives can be rendered into the stencil buffer just like into the color buffer using various functions that modify the stencil value. This value can then be used to control rendering into the color and z buffer. The stencil buffer is a powerful and

flexible tool to create special effects. The sum of all buffers available on a system is usually called *framebuffer*.

More information on rasterization can be found in [5] and [81].

2.3 Programmable Graphics Hardware

Since the release of the first dedicated consumer graphics hardware, the 3Dfx Voodoo 1, in 1995, graphics accelerator hardware for PCs have come a far way. With the introduction of the NVIDIA GeForce 256 in 1999, an important part of the rendering pipeline (section 2.2) was moved from software to the graphics hardware. Called *Hardware Transform & Lighting* (T&L), this hard-wired unit transforms vertices according to the supplied model- and viewport transformations and evaluats the basic *Phong* lighting model at each vertex [37]. Hardware T&L has paved the way for virtual worlds with higher polygon counts, making the scenes look more realistic.

While preserving CPU power for other tasks, hardware T&L came at the cost of decreased flexibility: the advantage of software T&L is its free programmability. Developers are free to chose their own lighting models, if they wish to do so, whereas with hardware T&L, they are stuck with per-vertex Phong lighting, which, though fast, does not offer the best image quality. To alleviate these disadvantages and still keep transformations and lighting in hardware, vertex shaders were introduced.

In general terms, a *vertex shader* is a freely programmable unit incorporated on the graphics hardware. It can modify per-vertex properties such as its color, normal, texture coordinate, and position. When the vertex shader is enabled, the hard-wired transform and lighting unit (sometimes also referred to as the *fixed-function pipeline*) is bypassed and no longer available. Its place in the pipeline is replaced by the programmable shader, which executes a series of commands specified by the user to accomplish its task. Vertex shader programs are usually stored in a form of assembly language. However, several high-level languages exist to create shader programs (often referred to simply as *shaders*) very similar to computer programs. A compiler then creates the assembly output. Applications that use vertex shaders work on all systems that support the API. This means that if the graphics hardware does not support the shader directly, the API emulates the program in software. Of course, hardware shaders perform much faster than the software emulation does.

A vertex shader program processes each vertex passed in. For each entering vertex, an output is generated. The shader can neither create vertices, nor destroy them. Furthermore, there is no mechanism to pass results from one vertex on to another one. In the first version of the vertex shader specification, the assembly language contained 17 different instructions, and shader programs (often called just *shaders*), were limited to 128 commands. These instructions are tailored towards graphics

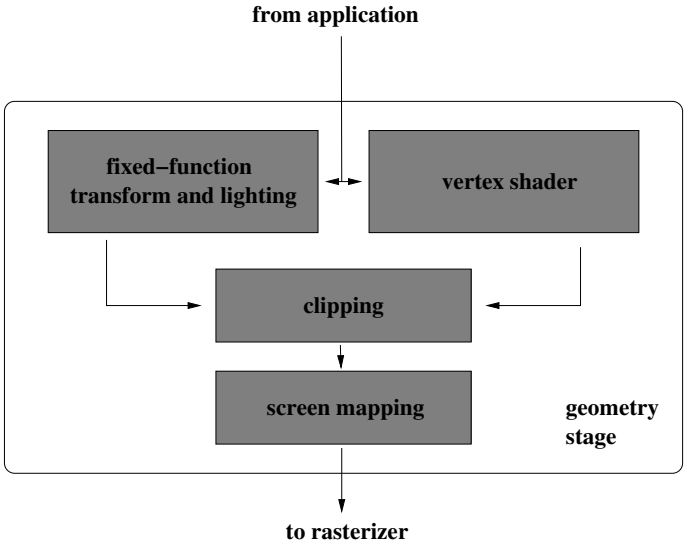

Figure 2.2: The geometry stage with a vertex shader. The vertex shader can be enabled to circumvent the fixed-function transform and lighting unit.

applications and include operations like three- and four-element dot products, inverse square root, reciprocal, and lighting operations. All these instructions execute in a single clock cycle. However, there are no early return statements, so a vertex program is always executed in constant time, regardless of any conditions encountered during the course of the program. This means that shorter shaders execute faster than longer, more complex, shaders. With newer hardware, newer specifications support longer programs and provide more sophisticated instructions, like loop and branch instructions (see [32, 72] for more information about vertex shader instructions).

Freely programmable units can also be found in the rasterizer stage. Unlike vertex shaders, which operate on a per-vertex basis, *pixel shaders*, also called *fragment shaders*, take place on a per-pixel basis during a rendering pass (Figure 2.3). The idea is the same as with vertex shaders: a series of instructions written by the user operate on a given data set (constants, interpolated values from the rasterizer, texture lookup values) and compute the color for this fragment. Pixel shaders can perform dependent texture reads, modify the z value of the fragment, and even a number of many other operations that do not exactly fit into the texture stage concept, thus providing a flexible way to create realistic-looking illumination models and effects.

Vertex and pixel shaders can be used to create many different effects. A number of examples are listed below.

- Object deformation (twist, bend, taper, procedural, page curls, water ripples, ...)
- Lens effects (making objects on the screen appear fish-eyed, underwater, ...)

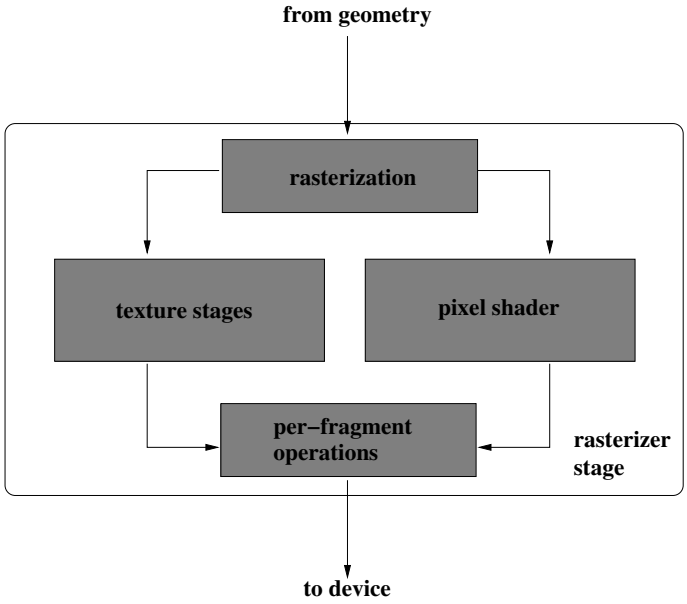

Figure 2.3: A fragment shader is a programmable hardware unit that can replace the traditional multitexture pipeline and operates on fragments.

- Creation of primitives by sending degenerate meshes down the pipeline and extruding them as needed
- Sophisticated lighting models (Phong, per-pixel, ...)
- Bump mapping
- Reflections and refractions

2.4 Scene Graphs

A *scene graph* (SG) is a high-level tree structure that represents a given scene. Unlike other hierarchical spatial data structures like BSP trees or bounding volume hierarchies (BVH), a scene graph usually contains more than just the geometry: it may contain the scene's geometry, textures, transformations, levels-of-detail (LODs), rendering properties, light sources and many other entities. When the scene is rendered, the tree is traversed in depth-first order. For example, transformations can be put in an internal node and then be applied to all objects contained in its sub-tree. Further examples for internal nodes are light sources and materials. Leaf nodes often contain geometry (Figure 2.5).

When nodes are allowed to have multiple parent nodes (i. e., nodes may share their children), the tree structure is called *Directed Acyclic Graph* (DAG) [24]. Scene graphs are often DAGs, because they permit *instantiation*. Instantiation allows

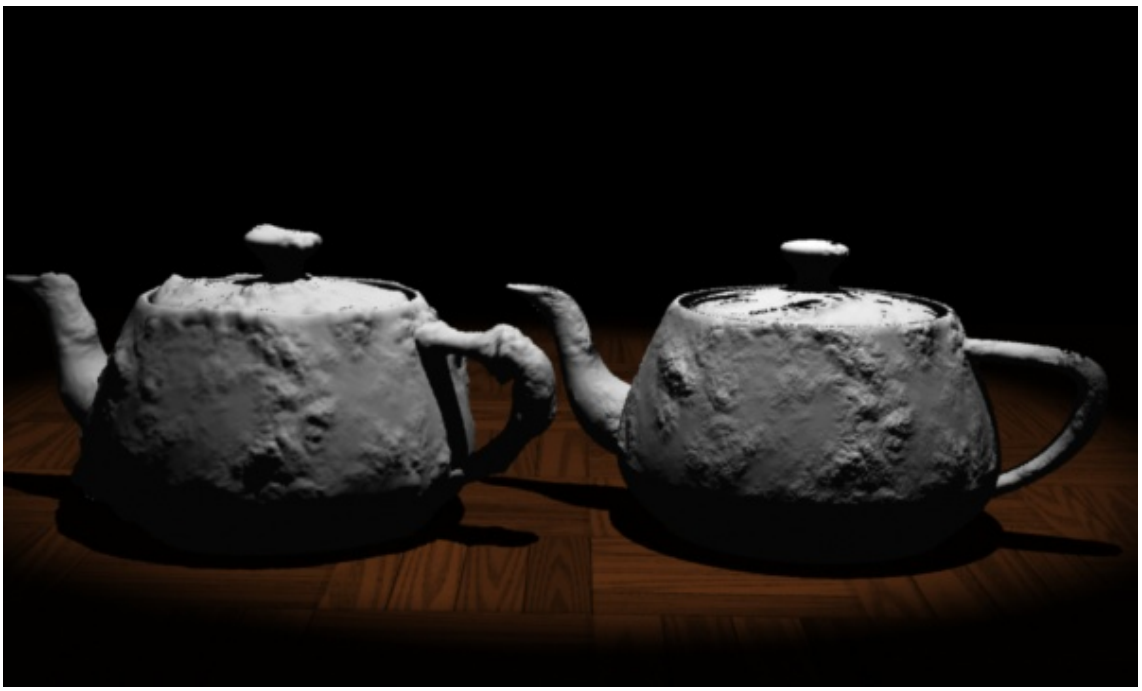

Figure 2.4: Two effects created using programmable shaders. On the left, the teapot is rendered using displacement mapping, which simulates surface structures without adding geometry. On the right, simple bumpmapping is used. Note how the illusion of the ripples is lost at the silhouette of the right teapot, whereas displacement mapping distorts the silhouette as well.

having several copies (instances) of the same object without having to replicate its geometry.

Each node in the scene graph is usually associated with a *bounding volume* (BV) which encloses its subtree. The bounding volume can be boxes, spheres, or even more complicated representations like *k-DOPs*. A k-DOP (*discrete oriented polytope*) is the volume formed by the intersection of a set of slabs [57] and often used for fast intersection tests [63]. These bounding volumes can then be used for acceleration techniques like *hierarchical view frustum culling* [22], where each node with a bounding volume is tested against the view frustum. If the bounding volumes intersects or lies completely inside the frustum, scene graph traversal continues and the child nodes are tested. If the bounding volume is completely outside the frustum, the node is not processed further (Figure 2.6).

In dynamic scenes with moving objects, the SG, specifically BVs and transformations, has to be updated. This can be done recursively on the tree structure. DAGs can overly complicate this process and are hence often avoided, or used in a limited form that allows only leaf nodes to be shared [30].

Prominent examples for SG-based graphic engines are *Open Inventor* [92] and *Java3D* [75].

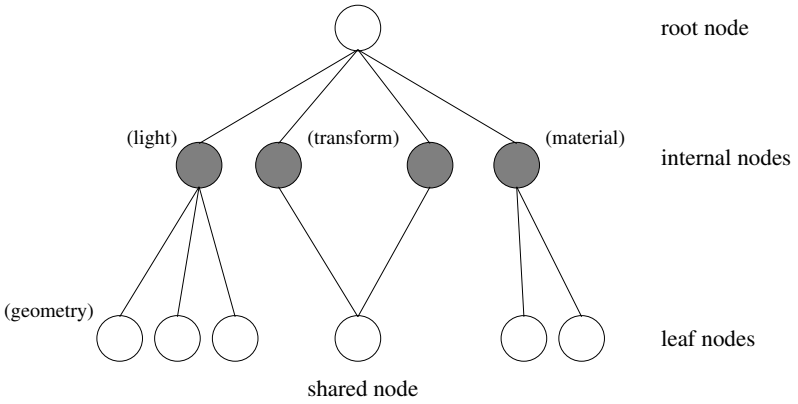

Figure 2.5: A scene graph with internal nodes, leaf nodes, and a single shared leaf node.

2.5 Visibility

In computer graphics, shadowing is closely tied to visibility determination, because the problem is quite similar. A shadow occurs whenever an object (the *shadow receiver*) is occluded from the light source (by a *shadow caster*). In this section, we will examine the visibility problems and explore several solutions. Later on, we will take this understanding and apply it to shadowing to filter out redundant and unnecessary shadow casters (section 4.3). A in-depth analysis of visibility in computer graphics can be found in [14].

Visibility is a hot topic in computer graphics. In a typical scene, only a fraction of all objects that make up the world is visible for a given viewpoint. In this context, we call an object that is hidden from view by another object an *occludee*, while the occluding object is called the *occluder*. As we have seen previously (section 2.2), a hardware construction called the z buffer is responsible for resolving visibility in the final image by keeping track of the distance for the currently nearest object. The z values of any incoming fragment that maps to the same screen-space location is compared to the value stored in the z buffer and written to the color buffer only if the new fragment's depth is lower. Otherwise, the new fragment is discarded.

Even though it correctly solves visibility, the z buffer is not the smartest mechanism in all respects. First and foremost, its location is at the end of the rendering pipeline. This means that objects that are occluded still have to pass through the complete geometry stage (undergoing transformations, lighting, clipping, ...) and the rasterizer. The final image will be correct, but for the graphics hardware, hidden objects impose as much processing work as do visible objects. Moreover, if the objects in the scene are not rendered in front-to-back order, the values in the color and z buffers are written multiple times. Because buffer writes are expensive and limit the performance of the rasterizer stage, ideally all objects in a scene should be rendered in front-to-back order, which usually requires the use of proper spatial data structures (e. g., a k-D tree).

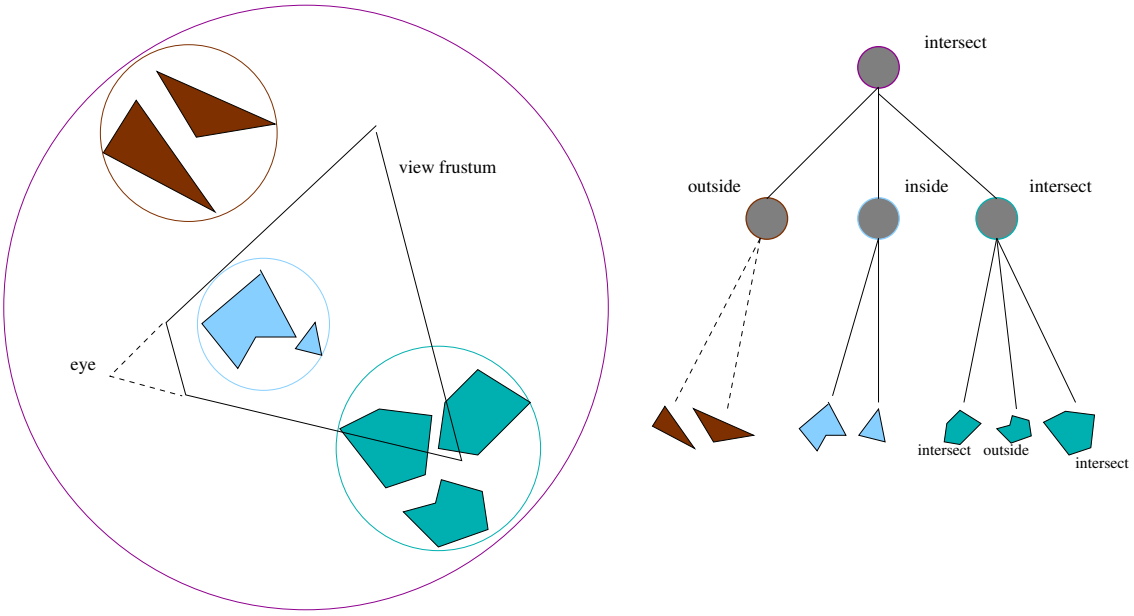

Figure 2.6: Hierarchical View Frustum Culling is performed on a BVH or scene graph (shown to the right). Intersection tests are only performed as long as the parent node's status is *intersected*.

Basically, we can distinguish between three different circumstances that cause objects to be invisible from the eye (Figure 2.8). A camera never covers the whole world. Rather, its viewing volume can be defined by six border planes (near, far, left, right, top, and bottom). The exact shape of the viewing volume (henceforth called *frustum*) depends on the kind of projection used. For orthographic projections, it is a cuboid, whereas for perspective projections, it is a truncated pyramid. Objects are visible only if they lie at least partly within the frustum. It is relatively easy for an application to identify objects that lie completely outside the view frustum by using geometric intersection tests with some kind of bounding volume that is associated with every object. Preventing objects that lie fully outside the view frustum from being sent through the pipeline is called *view frustum culling* [22]. If an object lies completely within the view frustum, it is not necessarily visible. Depending on the placement of objects in the world, several objects are completely occluded by other objects. The process of identifying these objects in the application before sending them down to the pipeline is called *occlusion culling*. Finally, even objects that passed view frustum culling and occlusion culling may contain triangles that are not visible. For any closed (and opaque) object, only those triangles can be seen that face toward the viewer (front-facing). The object's back side is completely hidden by the front side, and all triangles that comprise the back side can be culled. This process is called *backface culling* and usually performed by the graphics hardware, but can be turned on and off using appropriate API calls. The GPU detects back-facing triangles by testing the vertex order of the projected triangle.

The remaining objects are those that are at least partially visible in the final image.

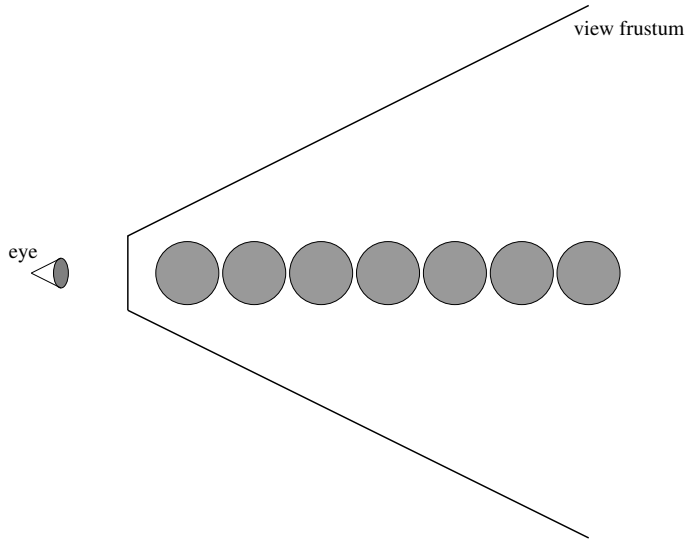

Figure 2.7: An illustration of occlusion. Several spheres are placed in a straight line exactly behind one another. From the viewpoint of the eye, all spheres except the first one are completely hidden. Nonetheless, all spheres have to pass through the complete rendering pipeline. Depending on the order in which the spheres are drawn, the z value is modified up to seven times (if rendering back-to-front), resulting in a high depth complexity and high fill costs.

The set of these objects is called *Exact Visible Set* (EVS). The process of preventing presumably invisible objects from entering the pipeline is called *visibility culling*. Back-face culling, view-frustum culling and occlusion culling are all different forms of visibility culling. Ideally, the result of visibility culling should be exactly the set of visible objects in the final image (EVS). In a sense, this is exactly what the z buffer does, but not without having to send every object through most of the rendering pipeline. The idea behind efficient occlusion culling algorithms is to perform visibility tests as early as possible and so avoid sending data through much of the pipeline. In practice, methods that perform exact visibility detection are generally not feasible because of their complexity and the additional computational overhead. Therefore, most visibility algorithms compute an approximation of the EVS, called *Potentially Visible Set* (PVS), rather than the EVS itself. The quality of these algorithms can be defined as how closely the generated PVS matches the EVS. Depending on how the PVS relates to the EVS, different kinds of visibility algorithms can be distinguished. *Conservative* algorithms generate a visible set so that $PVS \supseteq EVS$. This means that the PVS usually contains some objects that are intrinsically invisible. Image quality, however, does not suffer, and the small overhead is mostly acceptable. Conservative algorithms are generally preferred. *Aggressive* methods compute the PVS so that $PVS \subseteq EVS$. Ideally, the PVS is identical to the EVS, but occasionally some visible objects are incorrectly classified as hidden. This can lead to noticeable visual artifacts. *Approximate* algorithms are those for whom neither assumption holds. Sometimes hidden objects stay in the PVS, in other cases some visible objects are lost. Approximate algorithms are the

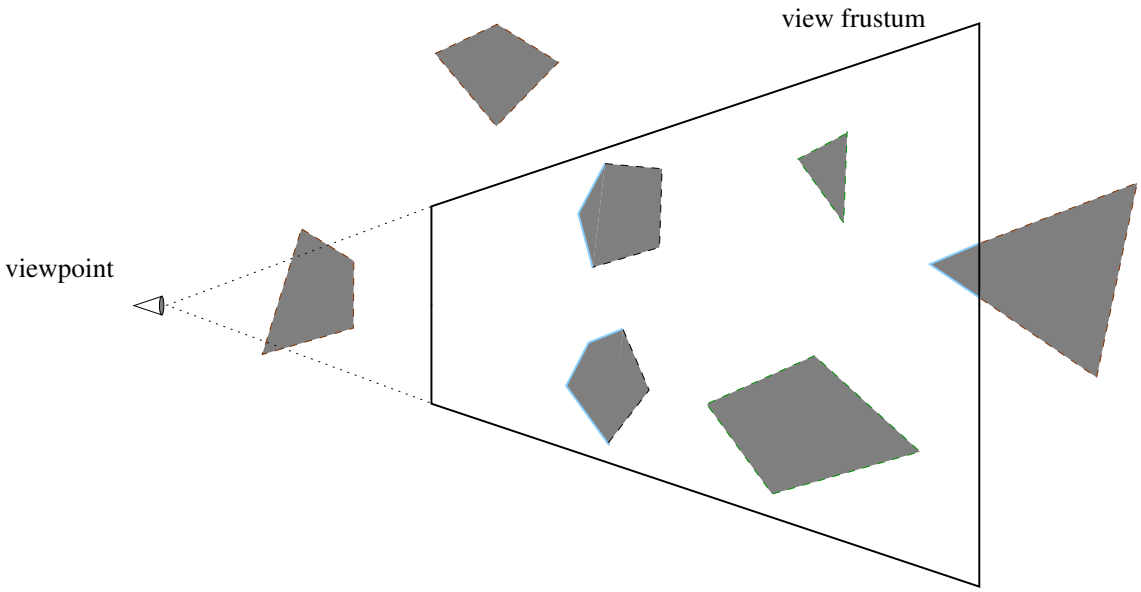

Figure 2.8: Visibility circumstances. Objects with a solid blue outline are visible. Objects with a dashed brown outline lie outside the view frustum and are invisible (the leftmost object is in front of the frustum near plane and hence invisible, too). Objects with a dashed green outline are within the view frustum, but occluded from the eye by other objects. A dashed, black outline indicates polygons that face away from the viewer (backfaces) and are not visible, either.

least desirable because their results are often difficult to predict.

Visibility algorithms can be categorized as those that operate in *object space*, *image space*, and *line space*. Object-space algorithms operate in the three-dimensional world and usually use geometric computations. Image-space algorithms perform visibility testing in two dimensions after some kind of projection. Line space methods operate in dual space [12, 15, 66]. Every point of interest maps to a ray in this space. The idea behind this technique is that the visibility tests are simpler or can be performed more efficiently in this space.

An important property with visibility detection is the fact that the cumulative occlusion of multiple objects can be greater than the sum of what each object would be able to occlude on its own (Figure 2.9). This implies that it is not sufficient to determine visibility by testing against every occluder separately. Rather, the union of all occluding objects must be computed before testing. The merging of several occluders is referred to as *occluder fusion* [100]. Another important observation with occlusion culling is that the size of the assumed occluder is in no way related to the number of objects it can occlude. Even small objects can be excellent occluders, depending on their distance to the viewpoint as well as to the hidden objects [1].

On a coarse level, all occlusion culling algorithms can be classified as either *from-point* or *from-region* (see Figure 2.10). When from-point visibility is used, the visibility is calculated for any given point in space. From-region visibility, on the

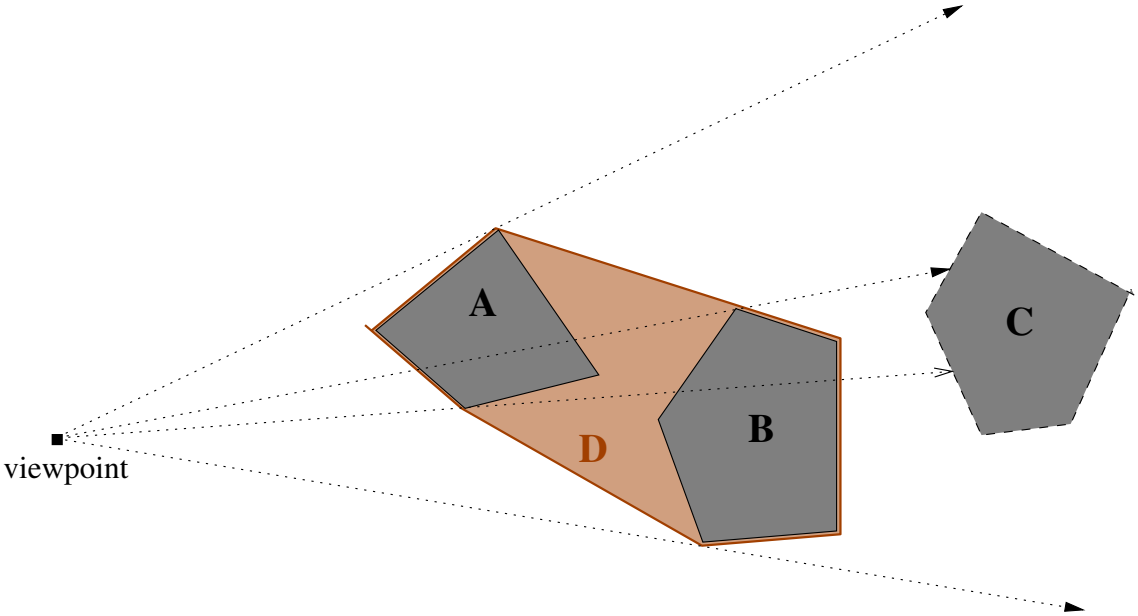

Figure 2.9: Occluder fusion. With occluder A alone, object C would be visible. The same holds for occluder B. However, with both occluders, object C is completely hidden. To correctly determine visibility, occluder fusion must be performed on A and B by creating the convex hull, resulting in occluder D.

other hand, is computed for a cell, which is a well-defined region in space, usually a box or a sphere. With from-region visibility, an object can only be classified as invisible when it can not be seen from *any* point within the viewing cell. Basically, the advantage of from-region over from-point visibility is that once it is computed for a certain cell, the results can usually be used for several frames, as long as the viewpoint stays within the cell. However, this advantage comes with the cost that it is often more time-consuming to compute than from-point visibility. Therefore, rather than being computed online, from-region visibility is often precalculated in a preprocessing step. This is in fact the main reason that this type of algorithm was developed. With from-region algorithms, occluder fusion is even more important, because the occluded space for from-region algorithms is smaller.

2.5.1 From-Point Visibility

From-point occclusion culling determines visible objects from a given point in space (normally the viewpoint) in a certain viewing direction. At any given point in time an object is either visible or hidden. The tests are view-dependant and usually performed online (at runtime) for every new frame. Hence, a main design criterion for from-point algorithms is *speed* and *scalability*. Speed is always an issue in real-time rendering, and even more so for routines that have to be executed every frame.

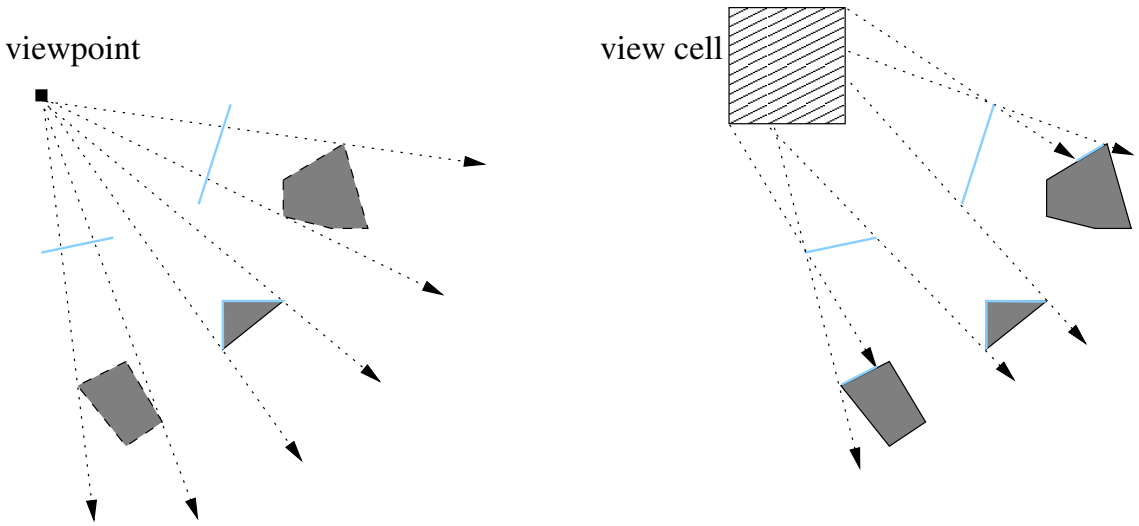

Figure 2.10: From-point versus from-region visibility. From-point visibility is depicted on the left. Here, the left- and rightmost objects can be discarded, since they are not visible from the viewpoint. With from-region visibility, shown on the right, all objects must be rendered because they are (partly) visible from some location within the view cell.

Without sufficient scalability, an algorithm that operates well on simple scenes may severely break down performance in more complex scenes. Developing algorithms that satisfy these criteria while still being as exact as possible is a challenging task.

Algorithm 2.1 describes a generic occlusion culling algorithm that takes cumulative occlusion into account [99]. The central part of the algorithm is the function `isOccluded()`, which is often called the *visibility test*. Its task is to check whether an object is visible from a given viewpoint or not. G is the set of geometric primitives that are to be rendered. The algorithm distinguishes between an occlusion representation O_R and a set of (potential) occluders, P. If an object is found to be hidden, it is not processed further, because we know that it will not contribute to the final image. Otherwise, the object becomes an occluder itself. However, updating O_R involves geometric computations (computing the convex hull of all objects contained in O_R) and is expensive. Therefore, new occluders are first moved to the temporary holding area P. If P is considered complex enough by some heuristic, the *occluding power* of all objects in P is used to update O_R. This process is usually called *multi-pass occlusion culling*. The rate at which O_R is updated is significant in determining the overall behavior of the algorithm. Some update O_R with every new occluder, skipping P completely. This technique is called *progressive occlusion culling*. Other algorithms just select a few large occluders in advance and do not update O_R at all. For this algorithm, as well as many others, the order in which the objects are drawn is important. To improve performance, the objects should first be sorted roughly front-to-back according to their distance from the viewpoint, then rendered in this order.

```
O_R = empty;
P = empty;
for all objects g ∈ G do
  if isOccluded(g, O_R) then
    Skip(g)
  else
    Render(g);
    Add(g, P);
    if LargeEnough(P) then
      Update(O_R, P);
      P = empty;
    end if
  end if
end for
```

Algorithm 2.1: Pseudocode for a general occlusion culling algorithm after Zhang. G contains all the objects in the scene, and O_R is the occlusion representation. P is a set of potential occluders that are merged into O_R when it contains a sufficient number of objects.

A conceptually interesting technique for from-point occlusion culling was developed by Greene et al. and is called *Hierarchical Z-Buffering* (HZB) [44, 42]. The algorithm operates in image space and organizes the z buffer as an image pyramid, which is called *z pyramid*. The base level of the pyramid is the standard z buffer. This is the finest level and has the highest resolution. At all other levels, each z value is the farthest value in the corresponding 2x2 window of the adjacent finer level. The top level consists of a single value, which is the farthest distance in the complete scene. Visibility testing is performed in a hierarchical way, starting at the top of the pyramid. If the nearest depth of an object is greater than the value on top of the pyramid (the farthest point in the scene), the whole object must be entirely hidden and can be skipped. Otherwise, testing continues down the pyramid until an area is found to be occluded, or until the bottom of the pyramid is reached. If a z value in the final image has been modified, the changes are propagated up through the coarser levels of the pyramid. To make this technique useful in real-time rendering, Greene suggested modifying the hardware z-buffer to support HZB, which requires the use of a z pyramid instead of the z buffer. More recently, Greene came up with an even more streamlined hardware implementation [43]. This system enables the application to read the top of the pyramid from the hardware, which can be used to perform culling at the very beginning of the rendering pipeline. In the absence of this kind of hardware support, existing z buffer hardware can be used to accelerate the HZB algorithm by exploiting temporal coherency [44].

Another visibility algorithm that operates in image space is the *Hierarchical Occlusion Map* (HOM) [100, 99]. Similar to the HZB, this algorithm work in a hierarchical way and can even take advantage of existing graphics hardware capabilities. This technique is characterized by dividing the occlusion test into a two-dimensional over-

lap test in the xy-plane and a comparison of depth extents in the z direction. The occlusion maps that are used for the overlap tests are obtained by reading back the contents of the color buffer after the occluders have been rendered with a white color on a black background. By doing this, occluder fusion is achieved automatically (similar to the HZB algorithm). A hierarchical pyramid of occlusion maps (the HOM) is then created by subsequently averaging over 2x2 pixel blocks. For any given level, a bright value (near white) indicates that most of the pixel it represents are occluded. The two-dimensional overlap tests are done in a similar way as the test for the HZB algorithm described above. After the overlap tests, the depth test decides whether a potential occludee is actually behind the occluders. One of the techniques originally described by Zhang utilizes a software z buffer with a coarser resolution than the final image which stores the farthest z value for each region (unlike the standard z buffer, which stores the nearest z value).

An interesting algorithm that focuses on speeding up the rendering of urban environments, like cities and villages, uses *Occlusion Horizons* [96, 95, 28]. This technique is based on horizons in image space. The observation that is exploited in urban scenes is that the majority of occluders are solid buildings through which the observer cannot see, and that the buildings are connected to the ground. Scenes that fulfill these requirements are called $2\frac{1}{2}D$, meaning that the depth complexity along any axis is never greater than one. $2\frac{1}{2}D$ scenes can still contain arbitrary three-dimensional objects. The idea behind this algorithm is to move a plane that is parallel to the viewer's near plane away from the viewer through the scene. As the plane cuts through objects (e. g., buildings), the occlusion horizon is created. Objects that lie completely below the horizon can be culled conservatively (see Figure 2.11). A nice feature of this method is the fact that it automatically accumulates the occluding power of all encountered occluders (occluder fusion).

The occlusion techniques presented so far suffer from a distinct disadvantage. With the exception of some suggestions by Greene [43], they all rely primarily on the CPU. The cost of executing the occlusion testing can often outweigh the performance gain by reducing the number of primitives that have to be processed by the graphics hardware, especially if the CPU is already the bottleneck of the application. *Hardware occlusion queries* aim at solving this problem by exploiting the graphics hardware not only for rendering the final image, but also for supporting occlusion culling by utilizing the speed of the rasterizer unit. They operate in image space and intend to determine the occlusion of a certain object by scan-converting its primitives and comparing the resulting fragments against the values in the z buffer. The return value of the query indicates whether any fragments passed the z test or not. Hence, hardware occlusion queries can be seen as a possible implementation for the `isOccluded()` function in the general occlusion culling algorithm (algorithm 2.1). Hewlett-Packard was the first to implement a hardware-based occlusion culling unit in the VISUALIZE fx graphics hardware [84], usually referred to as the HP-occlusion test. HP's OpenGL implementation used this technique automatically by querying bounding volumes for sufficiently large display lists [26]. This culling method is only

Figure 2.11: The occlusion horizon for an urban scene.

worthwile for bounding volumes that contain a large number of primitives, because the latency of individual queries can be a relatively long time, in which hundreds or thousands of triangles could be displayed.

Occlusion queries are usually used in conjunction with bounding volumes that are associated with the scene objects. For every object in the scene, its bounding volume is rendered with an occlusion query. If the bounding volume is found to be visible, the object is rendered. This technique works particularly well with hierarchical spatial data structures like BVHs and scene graphs. For internal nodes whose bounding volume is hidden, the entire subtree can be culled away.

An OpenGL extension that models the behavior of the original HP-occlusion test exists with the name `HP_occlusion_test`. Beginning with the GeForce 3 graphics accelerator, NVIDIA supports an extension for a similar occlusion query, `NV_occlusion_query`, often called NV query. The NV Query improves upon the HP test in that multiple queries can be issued in parallel without needing to wait for the result of the query, which extends the possibilities when used in an algorithm. Furthermore, the NV query returns the number of fragments n that actually passed the z test, as opposed to a boolean flag that merely indicates whether any of the fragments passed. Therefore, if $n = 0$, the entire bounding volume is hidden, and the contained objects can be safely discarded. If $n > 0$, at least some of the fragments passed the depth test, and the object must be considered visible. An application generally compares n with some threshold value to determine if the object is actually rendered. The threshold value is a measure of the *aggressiveness* of the occlusion culling. Recently, the OpenGL ARB approved a vendor independent extension for occlusion queries, `ARB_occlusion_query`, which behaves very similarly to the NV query.

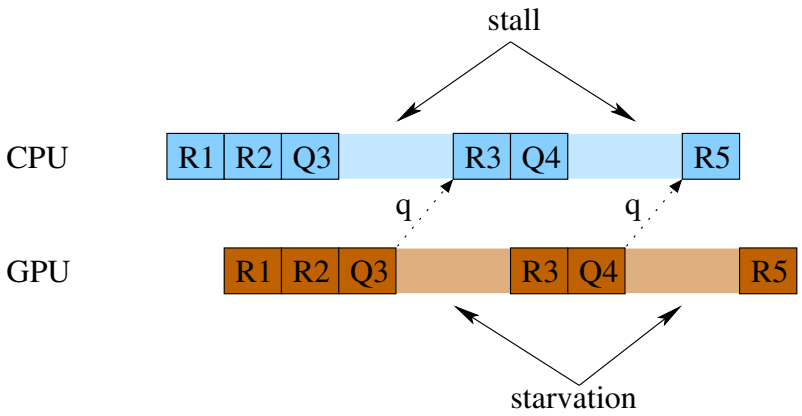

Figure 2.12: CPU stalls and GPU starvation. The R_i denote rendering tasks, Q_i occlusion queries. Object 3 is found visible and then rendered, whereas object 5 is determined hidden and skipped.

An advantage of hardware occlusion queries is their easy integration into existing graphics hardware. They do not require any fundamental changes to the underlying architecture, except some way to report the query result back to the application. A second advantage is the fact that during the time span between issuing the query and the availability of its result (which can take a considerable amount of time), the CPU is free to perform other tasks. The CPU is only stalled if it asks for the query result too early, before it is actually available. This implies that applications that employ hardware occlusion queries must be designed accordingly to distribute the tasks on the CPU to avoid stalls as much as possible. Simply replacing the occlusion test (`isOccluded()` in algorithm 2.1) is usually not sufficient.

As mentioned briefly in the previous paragraph, the main problem with hardware occlusion queries is the latency between issuing the query and the availability of the result. This latency is mainly due to the time it takes the pipeline to process the geometry, but also the time it takes the graphics hardware to report the result back to the host CPU. Problems that arise from the query latency are *CPU stalls* and *GPU starvation* [13]. CPU stalls are caused by the CPU waiting for the availability of the query result. During this time, it does not feed new data to the GPU, which can lead to an empty pipeline after the query is finally finished (starvation), see Figure 2.12 for an illustration. GPU starvation should generally be avoided as much as possible, since it is a waste of resources and mostly a sign of a poorly designed application. Therefore, the main challenge when designing applications that use hardware occlusion queries is to avoid the CPU stalls by using the query latency time to execute other, independent tasks.

Hardware occlusion queries are particularly useful in conjunction with hierarchical spatial data structures. They allow for efficient culling of large blocks of the scene with few queries, thereby exploiting spatial coherence. The following algorithms work particularly well with kD-trees, but are general enough to work with other data structures as well.

A simple method that exploits hierarchical data structures is called the *hierarchical stop-and-wait method* [13]. Basically, this algorithm tests every tree node that passes view frustum culling with a hardware occlusion query, then waits for the result. For internal nodes that are found visible, the algorithm recursively traverses the node's subtree, ideally in front-to-back order. If the node is a leaf, its contents are rendered. Nodes that are determined hidden are not processed further. Simple as this technique is, a big disadvantage is that it has to wait for the result of the occlusion query before tree traversal can continue. Whenever the result is not yet available, the CPU is stalled, which usually leads to GPU stalls as well and results in a significant performance penalty. Therefore, this algorithm is not the best choice in all respects. The individual tasks are distributed poorly, resulting in a waste of resources. Together with the overhead of the occlusion queries themselves, the sum of all these penalties can even result in a performance loss, compared to an implementation using pure view frustum culling.

Coherent hierarchical culling (CHC), presented by Bittner et al. in 2004 [13], is designed to improve the stop-and-wait method in several ways. This is accomplished mainly by exploiting the temporal coherence of visibility classification, i. e., it is assumed that the visibility changes little between successive frames. Moreover, occlusion queries are issued *without* waiting for their result. Rather, they are stored in a queue until they are known to have been carried out by the GPU. The algorithm distinguishes between different kinds of tree nodes. In this context, leaf nodes and nodes that were classified as hidden in the previous frames are called *termination nodes*. Similarly, internal nodes that were previously classified as visible are called *open nodes*. For every frame i, the set of termination nodes is denoted as T_i, and the set of open nodes O_i. In practice, the sets of opened and termination nodes are not stored separately, but kept as flag values within the node data structure.

To render the scene, the algorithm proceeds as follows. In a frame i, the tree is traversed in front-to-back order until a termination node is encountered (a node $n \in T_{i-1}$). If the termination node is a leaf node, n must have been visible in the previous frame. The algorithm issues an occlusion query, which is stored in the query queue. Then, the associated geometry is rendered immediately, without waiting for the result of the query. If the node is not a leaf node, it must have been invisible in the previous frame. In this case, an occlusion query is issued, but the algorithm does not immediately traverse the subtree. Whenever the visibility classification changes for a node in the tree, this information must be propagated upwards. In this case, the CHC algorithm performs a visibility *pull up*. This means that an interior node is classified as invisible if and only if all of its children are invisible. Even if a single child node is visible, the node remains opened. Similarly, if an internal node changes from invisible to visible, this information is passed on to its child nodes (*pull down*). However, a child node that 'inherited' a visibility change is not automatically visible. See Figure 2.13 for an illustration.

Listing 2.1 on page 39 shows a sample implementation for CHC. To prevent the

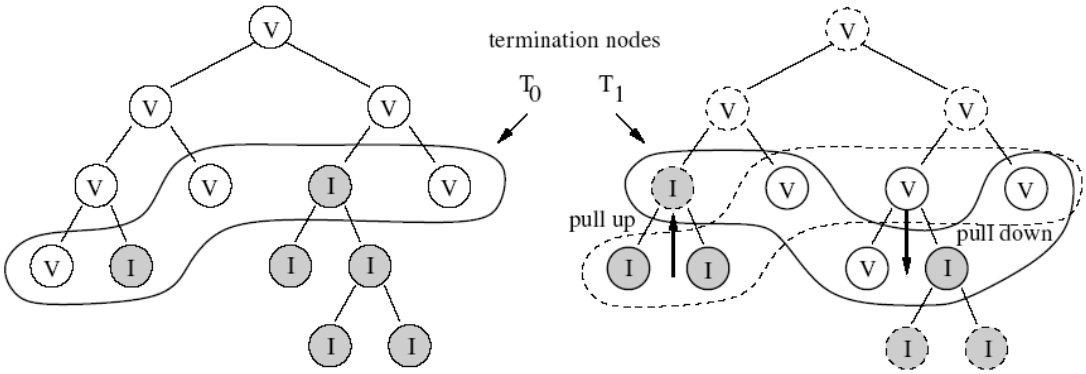

Figure 2.13: Visibility classification with CHC in a tree structure in two successive frames. *V* nodes are visible, *I* nodes are invisible. Nodes on which occlusion queries were performed in the frame are shown with a solid outline. The right figure also illustrates visibility pull-up and pull-down. *(Image courtesy of J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer [13])*

multiple rendering of leaf nodes in the same frame, the renderer stores the frame number in which the node was rendered last and compares it to the current frame.

Coherent hierarchical culling increases the performance of visibility determination by reducing the total number of issued occlusion queries as well as balancing the traversal of the data structure to prevent the CPU from having to wait for the result of a single query. The number of queries is reduced mainly due to the hierarchical approach: by testing internal nodes, entire subtrees can often be discarded without the need to test any further nodes. Plus, the algorithm avoids testing previously opened nodes by assuming that nodes that are classified as visible in one frame stay visible for a certain number of frames. In addition to reducing the number of occlusion queries, CHC also reduces CPU stalls and GPU starvation by interleaving the issuing of occlusion queries with the rendering of geometry that is known to be visible. Ideally, the results of the queries become available in between the traversal of subtrees and CPU stalls are eliminated altogether. This also implies that the tree is not rendered in a strict depth-first order anymore. Instead, subtrees are traversed as the result for their respective occlusion queries become available. Therefore, care has to be taken to keep track of the rendering state and perform as little state changes as possible.

2.5.2 From-Region Visibility

From-region visibility tests attempt to solve the visibility problem for a region of space, as opposed to a single point. Objects are classified as occluded if and only if they are hidden from all points within the region of interest (the *viewing cell*). The

idea behind such techniques is that the visibility information usually stays valid for several frames, as long as the viewpoint stays within the viewing cell. The drawback is that from-region visibility computations are much more complex than from-point techniques. Therefore, they are often performed offline in a preprocessing step. Then, at runtime, the precomputed visibility information is used, usually resulting in a negligible overhead compared to from-point approaches. Naturally, precalculating visibility offline is of little use in dynamic scenes where objects are moved or modified in any other way.

Maybe one of the most renowned techniques for from-region visibility culling is *portal culling*. Designed specifically for interior walkthroughs of architectural models, this technique was first introduced by Airey [3, 2] and later improved by Teller, Séquin and Hanrahan [88, 87, 89]. The motivation for all portal culling algorithms is the fact that the high number of walls in indoor scenes make for excellent occluders. Portal culling algorithms divide the scene into *cells* that usually correspond to rooms in a building. The doors and windows that connect individual cells are called *portals*. This decomposition is usually done in a preprocessing step, either from hand or automatically. In every frame, the algorithm then iteratively fits a frustum through each visible portal (e. g., door or window) and performs a variant of view frustum culling (Figure 2.14). The viewer is located in cell A, which is rendered as usual together with its contents. Cells B, C, and D directly connect to cell A. The portal to cell B, however, is outside the original view frustum and therefore ommitted from further consideration. The portals to cells C and D are visible. The view frustum is therefore diminished so that it passes straight through the connecting portals. The contents of C and D are then rendered with these diminished frusta. In the next step, the neighboring cells of C and D are examined. E is visible through the reduced frustum through D, so a new frustum is fit through the portal connecting E to D and the contents of E rendered according to this new frustum. Now, no more portals are visible through any of the frusta, and rendering stops. The algorithm discards cells F and G because they are not visible from the viewer.

Objects that are located within a certain cell are stored in a data structure associated with the specific cell. For each cell, this data structure also stores the adjacent cells and the portals through which they are connected in an adjacency graph, which is usually created manually (though Teller proposed a way to automatically generate it [87]).

Portals can be used in many other ways as well [6]. They can be used to create mirror reflections by reflecting the viewer's position and direction in the plane of a specific portal that is to act as the mirror. Other transformations can be used to create simple refraction effects.

Another technique, developed by Wonka et al., is called *occluder shrinking* and can use a from-point occlusion algorithm to generate from-region visibility [97]. The algorithm tries to extend the validity of from-point visibility from a given point p to a region in space by shrinking all occluders in the scene by a given distance d.

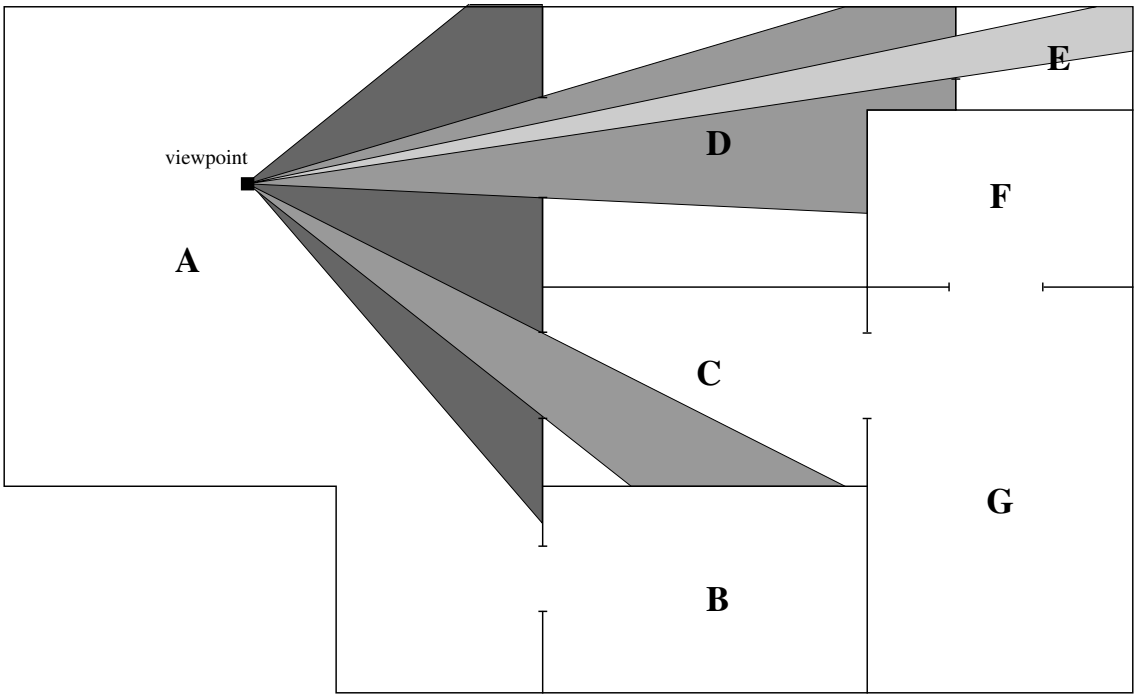

Figure 2.14: Portal culling. Cells are labeled from A to H and are connected by portals. The figure depicts how the original view frustum is extended into the cells B, C, D, and E. Cells G and H are hidden, they can be discarded. Geometry is rendered only if it is contained in one of the frusta. *(after T. Möller [6])*

This is a conservative approximation of from-region visibility within a viewing cell that is centered around p with a radius of d (Figure 2.15) [97]. The optimal value for d varies from scene to scene. It should be adjusted so that the spheres with radius d are small compared to the size of the average occluder. This is important to keep the visibility solution using shrunk occluders similar to the solution using regular occluders, for any given viewpoint. However, as d gets smaller, the number of spheres needed to cover the view cell increases, and the visibility calculation takes more time. Therefore, the choice for d affects the ratio between speed and efficiency.

To use this technique for on-the-fly visibility determination, occluder shrinking is often used in conjunctio with another technique known as *frustum growing* [98]. This means that the original frustum must be grown to make up for changes in the viewer's position and view direction. Let's assume that the movement rate of the viewer is restricted to a certain speed, v. Then the visibility for the view cell d is valid for $t = \frac{d}{v}$ seconds. These parameters should be chosen so that the time t spans at least a few frames. This way, visibility calculations need not be performed every new frame, exploiting *temporal coherence*. Further methods based on occluder shrinking and frustum growing include an algorithm called *instant visibility* [98], as well as *virtual occluders* [65].

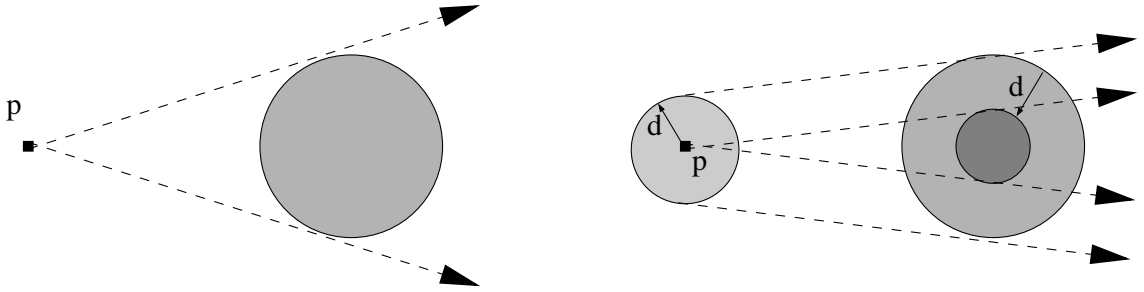

Figure 2.15: An illustration of occluder shrinking. The left image shows the viewpoint p and the original occluder. On the right, the occluder is shrunk by a factor d, which mathematically equivalent to an expansion of the viewpoint p to a sphere with the radius d.

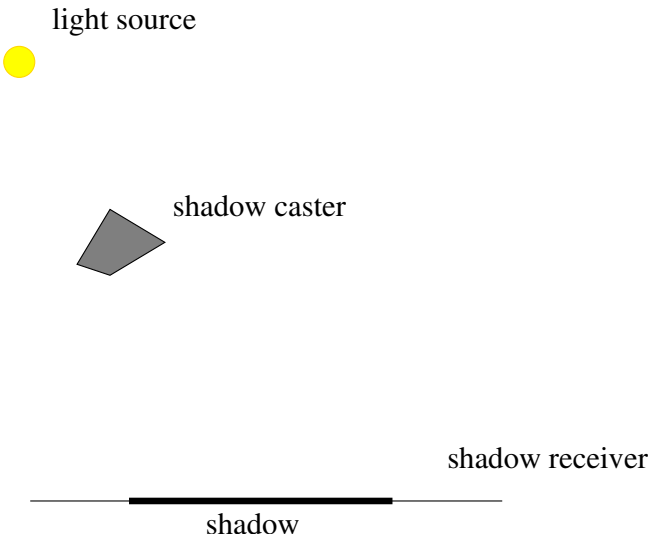

Figure 2.16: Terminology used throughout this document.

2.6 Shadows in Computer Graphics

Shadows are the result of the interaction between three components: light sources, shadow receivers and shadow casters (Figure 2.16).

A *light source* is an object that models a primary source of light, i. e., some physical entity that produces light. Light sources in the real world emit energetic particles (*photons*) that travel through space and may encounter physical obstacles. Depending on the obstacle's surface, they are either absorbed or reflected. Humans "see" an object whenever a photon that bounced from that object's surface hits the cone cells in the eye.

All real-world light sources are *area light sources* (Figure 2.17), which means that they have a physical extent. In computer graphics, however, *point light sources* (Figure 2.18) are often used to approximate area light sources. They have no physical

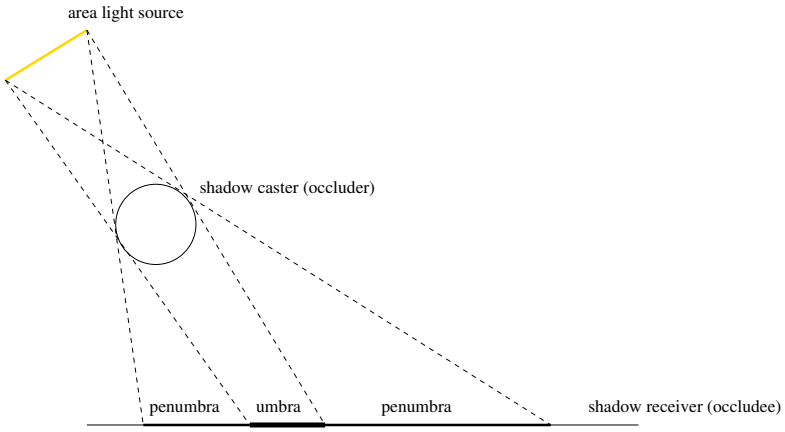

Figure 2.17: Umbra and penumbra regions of an area light source.

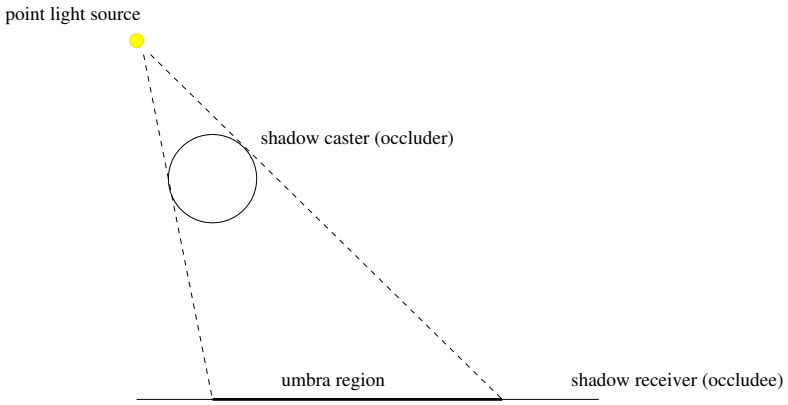

Figure 2.18: Umbra region of a point light source.

area, but can be described mathematically as a singularity (a point in space). Area lights can be simulated by using a number of point lights sampling the original light source area.

A *shadow receiver* is an object that can receive light from a light source. Usually, all objects in a scene are shadow receivers. Each surface point of a shadow receiver fulfills one of three conditions. If the point is visible from the light source, it is lit. If it is totally hidden from the light source by other objects, it is in shadow and not lit at all. The sum of all points totally hidden from the light is called the *umbra region* of the shadow. Other points may be hidden only partially from the light source, receiving some light from a part of the light source. Those partially-lit points form the *penumbra region* of the shadow. Together, umbra and penumbra compose the shadow. Obviously, with point light sources, each point is either totally lit or totally invisible. Hence, shadows cast by a point light source don't have a penumbra region.

Objects that partially or completely block photons from reaching a shadow receiver are called *shadow casters*, since they cast shadows on the shadow receivers. If an

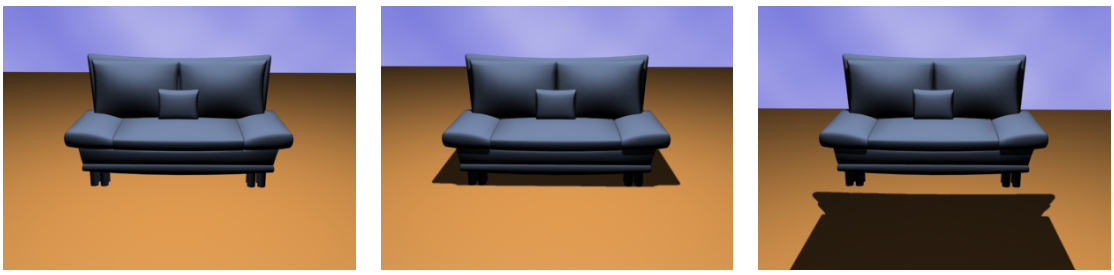

Figure 2.19: This figure demonstrates the importance of shadows in perceiving spatial placements.

Figure 2.20: Hard shadows (left) versus soft shadows (right). *(Image courtesy of ARTIS, Laboratoire GRAVIR)*

object casts a shadow onto itself, *self shadowing* occurs.

2.6.1 The Role of Shadows in Virtual Environments

Shadows have been neglected in real-time applications for a long time. Shadows require a significant amount of processing power, and it was simply not possible to perform shadowing while keeping the applications real-time. Nonetheless, shadows are an important part of computer graphics. They make scenes appear more realistic, and a user will immediately notice if a scene lacks shadows, which prevents him from becoming immersed in the virtual environment. Secondly, perhaps even more important, they are crucial to the correct spatial perception of the scene [91]. In many situations, shadows are the anchors that tie objects to a certain position in space. Without the shadow, the object's exact location is often unclear. See Figure 2.19 for an illustration.

2.6.2 Hard shadows vs. soft shadows

As explained above, real shadows are composed of an umbra region and a penumbra region. Points that lie within the umbra receive no light from that light source source at all, whereas points in the penumbra region are partially lit. The partially lit penumbra region produces a soft edge in the shadow, with the lighting increasing

toward the rim of the shadow. Shadows that consist of both an umbra and an penumbra region are called *soft shadows* and cast by area light sources.

In the real world, all light sources are area light sources; point light sources do not exist. Therefore, the human eye is used to seeing soft shadows. The use of hard shadows in computer graphic is sub-optimal, because they may be mistaken for real geometry. Nonetheless, point light sources are often preferred over area light sources because of their simplicity. Point light sources do not generate penumbra regions. Their shadows are hard-edged and consequently called *hard shadows*. In this thesis, we will focus on hard shadows only.

2.6.3 Traditional Shadow Algorithms

On a coarse level, algorithms for computing real-time shadows can be classified as either *object-precision* or *image-space* algorithms. Object-precision methods, also referred to as *object-precision* algorithms, operate in the three-dimensional world space and generate exact shadow boundaries mostly using geometric computations. Examples for object-space shadowing algorithms are *projection shadows*, which are described below, and *shadow volumes* (chapter 3). In contrast to object-precision techniques, image-precision shadowing algorithms operate in a two-dimensional space, typically after some kind of projection has been applied. Image-precision methods are generally faster, but the quality of the final image depends on the resolution of the two-dimensional image and is often lower than object-space shadows. The most renowned image-space shadowing technique are *shadow maps*.

Polygon Shadows

In 1978, Atherton, Weiler, and Greenberg presented a general purpose method for generating shadows using an object-space polygon clipping algorithm [8, 7]. Their algorithm relies on the fact that no shadows can be seen from the light's point of view. First, a camera is positioned at the location of the light source. All polygons are then clipped in a way so that all visible polygons are separate from all non-visible polygons (if necessary, triangle subdivision can be performed during this step as well). After this step, the visible polygons correspond to non-shadow polygons, whereas the non-visible polygons are shadow polygons. Turning a light source on and off simply changes the way the shadow polygons are displayed - illuminated or not. As long as the objects in the scene and the light source keep their relative distance, this shadow information need not be recreated, even if the viewpoint changes, which made this approach attractive for architectural walkthroughs. Furthermore, the method can be easily extended for multiple light sources by repeating the clipping stage for each light source and marking each shadow polygon with the responsible light source.

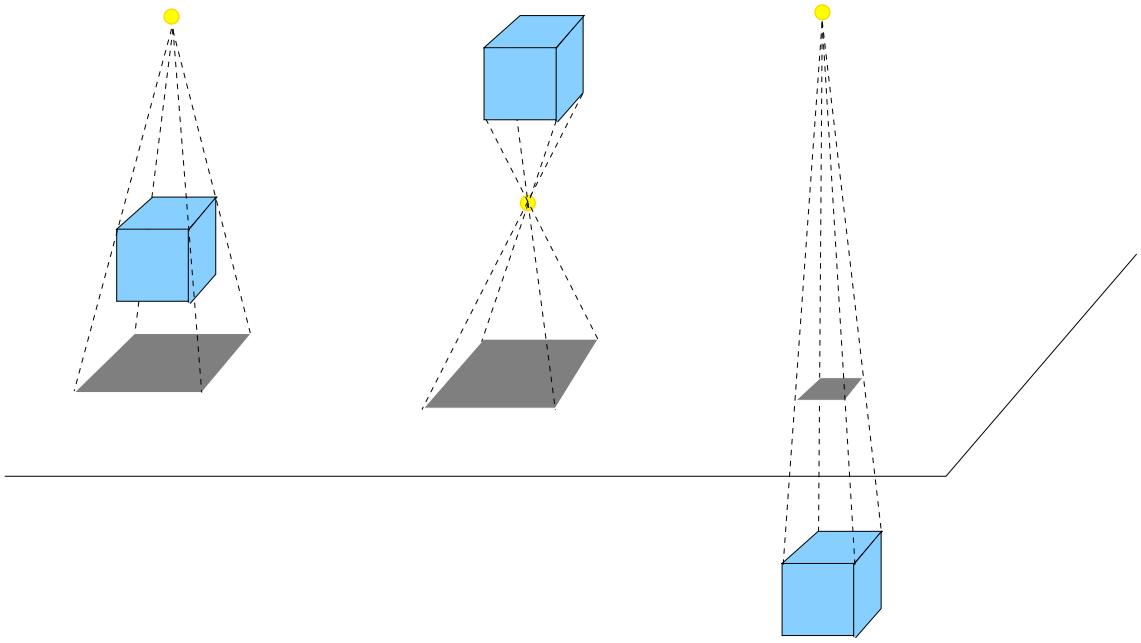

Figure 2.21: Left: a correct projected shadow. Middle: If the light source is below the topmost point on the object, an antishadow is generated. Right: Objects below the far side of the surface plane produce false shadows.

A severe disadvantage with this technique is the fact that its performance scales linearly with the number of triangles in the scene, making it practically impossible to use in larger scenes without some kind of optimization to reduce the number of triangles that take part in the clipping process.

Projection Shadows

Shadows that are created by projecting a three-dimensional object onto a two-dimensional surface (left of Figure 2.21) are called *projection shadows*. This projection can be easily achieved by multiplying the object's coordinates with a special 4x4 matrix [16, 90].

To create an object's shadow, the object is transformed with the projection matrix and rendered a second time with a dark color and no illumination. However, because the matrix will project the vertices exactly onto the surface plane, surface points may poke through the shadow due to floating point errors. One simple workaround is to add an offset to the plane the object is projected upon so that shadow polygons are always a bit "above" the surface polygons. Getting this bias right can be tricky: if the shadow is too high, the illusion of the shadow will be lost, if the shadow is too low, some pixels of the surface may still shine through. Furthermore, the bias depends on the angle of the surface normal toward the viewer. If this angle increases,

the bias must also increase. OpenGL, provides a function `glPolygonOffset` that performs exactly this task. It takes a constant bias parameter and computes the final offset by taking the polygon orientation into account. This way, the offset increases together with the surface's angle. Another method that does not need any offset is to draw the surface polygons first, then draw the projected shadow polygons with the z-test off, and finally render the rest of the geometry as usual. This way, the projected polygons always superimpose the surface polygons.

Though they are very easy to understand conceptually, in practice projection shadows have several flaws and disadvantages. The first issue concerns the borders of the surface plane: the projected shadows may easily fall outside the plane polygons. This problem can be solved by using the stencil buffer. In a first step, the surface polygons are drawn both to the screen and the stencil buffer. Then, with z-tests disabled and doing stencil tests, the projected shadow polygons are drawn only where the receiving polygons were draw previously. Finally, the rest of the scene is rendered. This way of rendering projected shadow polygons works only if the shadows are fully opaque. Transparent shadows, where the material of the underlying surface shines through, require more care. Simply rendering the projected polygons as semi-transparent works only for convex objects, because with concave objects, more than two points on the object may be projected to the same point on the plane. Again, the stencil buffer can be used to solve the problem by ensuring that each pixel is covered once at most [62].

Another drawback with this method is that the projected shadow polygons have to be recreated every frame, even if the scene has not changed at all. An improvement that works well in practice takes advantage of the fact that shadows are view-independent, i. e., they don't depend on the location of the viewpoint. Here, the shadows are rendered into a texture that is then applied to the surface plane. The shadow texture is recreated only when the light source or any shadow casting or receiving object moves.

An important aspect with projected shadows is the relative position of light source, shadow caster and shadow receiver. If caster and light source are on opposite sides of the shadow receiver plane, *false shadows* are generated, because objects beyond the surface plane would not cast any shadows (Figure 2.21, right). Light sources that lie between the shadow caster and receiver produce *antishadows* (Figure 2.21, middle).

Several techniques exist to use projective shadows to generate soft shadows, e. g., by sampling the area light source with a number of point light sources [53, 49].

Shadow Maps

Shadow maps were proposed by Lance Williams in 1978 [93]. The idea behind this technique is to render in a first step the scene from the position of the light

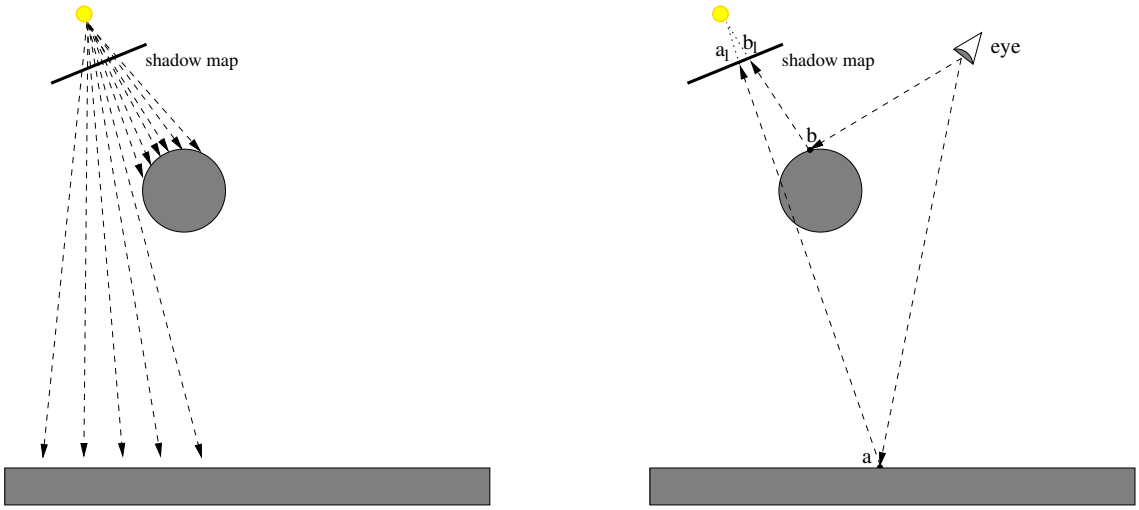

Figure 2.22: The concept of shadow mapping. On the left, a depth map (the shadow map) is created from the light's view. On the right, the scene is rendered from the position of the eye. After projecting point a to the light's view, its depth value is found to be significantly greater than the value stored in the map (which is the depth of the corresponding point on the sphere). Hence, the lighting equation is not evaluated at point a. Point b on the sphere projects back to a depth value that is approximately the same as the value stored in the shadow map - point a was visible from the light, and so is not in shadow.

source that is to cast shadows into the z-buffer. For each pixel, the z-buffer now contains the distance of the closest object to the light source. In this context, the depth map, essentially a 2D function, is usually called *shadow map*. To make use of the shadow map, the scene is rendered in a second step from the position of the viewpoint. However, before displaying a rasterized fragment, it is projected back into the light view, and its depth is compared to the value stored in the depth map. If the fragment is farther away from the light source than the value in the shadow map, it is in shadow; otherwise, it is not. Figure 2.22 depicts the two rendering steps.

Standard texture mapping hardware can be used to implement shadow maps. First, the shadow map is generated. Then the scene is rendered using only ambient lighting, creating a depth map to resolve visibility. In practice, the depth map is copied back from the z-buffer into a 2D texture, where it is stored as a 8-bit color channel (which is often not enough to produce pleasing results). In the next step, shadow testing is performed: the value in the depth map is compared to the value in the shadow map, after it has been transformed from the coordinate system of the light to the coordinate system of the viewpoint. This transformation can be accomplished by using eye-linear texture coordinate generation, e. g., OpenGL's `GL_EYE_LINEAR` texgen [59], which relies on projective texturing [48, 85], or by using multi-texturing and OpenGL's texture combiner extension (`EXT_texture_env_combine`) as suggested by Heidrich [51]. Depending on the outcome of this comparison, an additional flag value

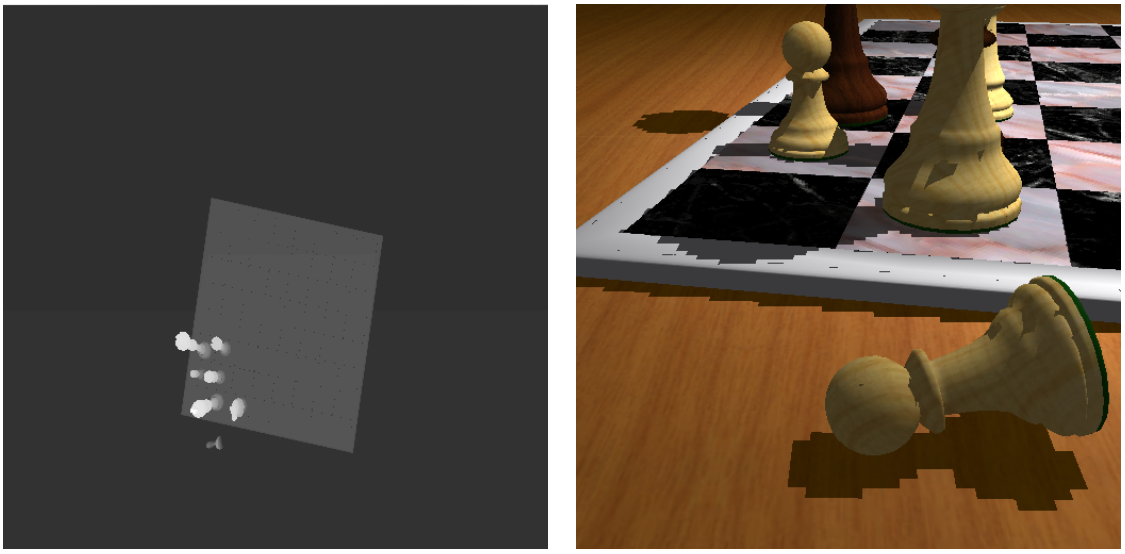

Figure 2.23: Aliasing problems in shadow mapping. On the left, the shadow map is shown. On the right, the limited resolution of the depth map causes severe aliasing artifacts, amplified by the close proximity of the viewpoint to the object. *(Images courtesy of Marc Stamminger and George Drettakis)*

is set for each pixel. If the two z values are approximately identical, the flag value is set to 1, otherwise, it is set to 0. Finally, the whole scene is rendered again, this time using diffuse lighting. The final color of each pixel is computed by additive blending the color from the full rendering pass, multiplied by the flag value, over the ambient color.

In 2002, the OpenGL ARB approved two extensions that simplify the use of shadow maps. The shadow test can now be performed simply by using a texture filtering operation (`ARB_shadow`), which performs the texture lookup and sets an internal flag value, which is then used to modulate the color in the color buffer. `ARB_depth_texture` incorporates new internal texture formats which allow for higher-precision shadow maps. With the advance of programmable graphics hardware, shadow mapping can be implemented using vertex and pixel programs.

The shadow map itself has to be recreated only if the scene changes, i. e., the light source or any object has moved. An advantage of this method is that it is well supported on graphics hardware, and that it scales well with respect to the scene complexity.

As an algorithm operating in image space, shadow maps are prone to *aliasing* problems. This happens when multiple screen texels map to a single texel in the shadow map (due to shadow map undersampling), resulting in jagged shadow edges (Figure 2.23). Crucial for obtaining good-looking results is the setup of the light frustum: it should be limited to fit tightly to the shadow-casting objects. However, whenever the viewpoint can move arbitrarily close to a shadow edge, aliasing al-

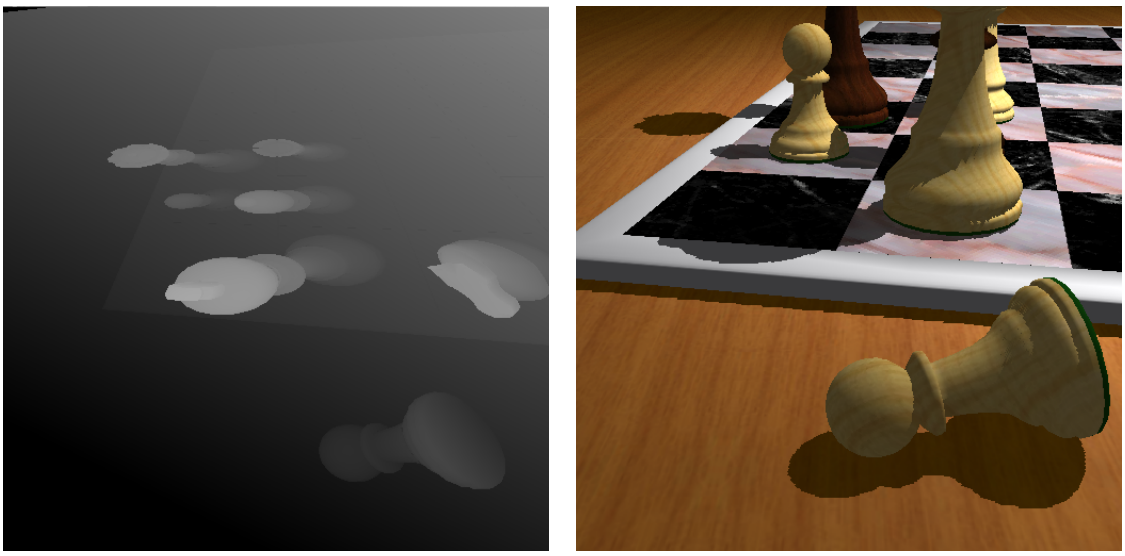

Figure 2.24: A perspective shadow map, shown to the left. Perspective shadow maps can improve shadow quality, compared to standard shadow maps, by concentrating the map resolution where appropriate. *(Images courtesy of Marc Stamminger and George Drettakis.)*

ways occurs. This is particularly evident in scenes with a high depth range, because nearby shadows need a high resolution, whereas for distant shadows, a lower shadow map resolution would be sufficient. Fernando et al. propose a hierarchical caching scheme using a combination of hardware and software to provide shadow maps with arbitrarily high resolutions (*adaptive shadow maps*) [34]. A different solution to reduce aliasing artifacts, called *percentage closer filtering* (PCF), was introduced by Reeves et al. [79]. In contrast to blending depth map values, which can lead to pixels being wrongly in shadow [59], PCF averages the boolean comparison results within the extents of the filter kernel, smoothing the jagged shadow edges.

A technique introduced by Drettakis and Stamminger [29] creates a single *perspective shadow map* by using a non-uniform parameterization. This distorted shadow map provides increased resolution for near objects and decreased resolution for objects that are farther away, the idea being to first map the scene to eye post-perspective space, and only then create the shadow map (Figure 2.24). A problem with this method is that the perspective transformation changes the light source and may introduce singularities. *Light space perspective shadow maps* use a variable perspective mapping specified in light space [94], which eliminates singularities, thus avoiding most of the problems of perspective shadow mapping.

Other problems may occur due to the limited numerical precision of the z-buffer. When transforming surface points that are not in shadow from the eye's point of view to the light's coordinate system, the value should ideally be identical to the value stored in the shadow map. However, in practice, the projected value will be slightly above or below the shadow map value. If the value is larger, then the point will be considered as in shadow, even though it should be lit. This phenomenon is

called *self-shadow aliasing*. To counter this problem, Williams simply subtracts a constant bias from the z-values after they have been transformed to light space [93]. Kilgard uses OpenGL's `glPolygonOffset` to offset the depth value in the shadow map back and compensate for the slope of the polygon [59].

A limitation common to most shadow mapping techniques is that the light source is assumed to be located somwhere outside of the scene. This is because the light looks at the scene through a single frustum, just like the eye does. Point lights inside the bounds of a scene cannot render the whole scene into a single shadow map, but must use a six-sided cube, similar to what is used in cubic environment mapping [41]. However, even then it can be problematic to get the shadows to join properly along the seams of the cube, because parameters (like bias) can be different along such edges than they are on the rest of the surfaces. Brabec et al. suggest using parabolic projections for use with hemispherical and omnidirectional light sources [18].

```
TraversalStack.push(kDTree.Root);
while (!TraversalStack.empty() || !QueryQueue.Empty()) {
  // 1st part: process query queue
  while (!QueryQueue.empty() &&
         (queryResultAvailable(QueryQueue.front()) || TraversalStack.empty())) {
    N = QueryQueue.pop();
    nVisiblePixels = queryResult(N);
    if (nVisiblePixels > visibilityThreshold) {
      pullUpVisibility(N);
      Traverse(N);
    }
  }

  // 2nd part: traverse tree
  if (!TraversalStack.empty()) {
    N = TraversalStack.pop();
    if (isInsideViewFrustum(N)) {
      // identify previously visible nodes
      wasVisible = N.isVisible && !isLeaf(N);
      // identify previously opened nodes
      opened = wasVisible && !isLeaf(N);
      // reset node data
      N.isVisible = false;
      N.lastVisited = currentFrame;
      // do not test previously opened nodes
      if (!opened) {
        issueQuery(N);
        QueryQueue.push(N);
      }
      // traverse visible nodes
      if (wasVisible)
        TraverseNode(N);
    }
  }
}
```

Listing 2.1: C++-like pseudocode for coherent hierarchical culling (CHC).

3 Shadow Volumes

Shadow volumes were first introduced by Franklin Crow in 1977 [25]. Let's imagine a point and a triangle. Drawing lines from the point through the vertices of the triangle, we obtain an infinite pyramid. Considering only the part of the pyramid below the triangle yields a truncated infinite pyramid. Let's further imagine the point is actually a point light source. Then, all points outside the truncated pyramid are visible from the light (they are lit), while all points within the pyramid are not visible and in shadow. Hence, the truncated infinite pyramid is usually called the *shadow volume* (SV) of the point light source (Figure 3.1).

Rather than performing computationally expensive geometric intersection tests on all objects in the scene to find out which parts of an object are within the shadow volume of any other object, a ray-casting method can be used instead to determine if a given part of an object (a pixel, to be exact) is in shadow or not. For a given scene and viewpoint, we shoot a ray through each pixel until it hits the object that is displayed on the screen and keep track of the number of shadow volumes the ray enters and leaves. Each time the ray crosses a polygon of a shadow volume that is oriented towards the viewer (front-facing), we increment a counter (the ray enters a SV). When the ray crosses a shadow volume polygon that is oriented away from the viewer (back-facing), the counter value is decremented (the ray leaves a SV). Then, a pixel is in shadow if its counter value is greater than zero, otherwise, it is not. This principle works with an arbitrary number of shadow-casting polygons (Figure 3.2)

3.1 Stencil Shadow Volumes

Performing ray-casting geometrically is expensive and time-consuming. In 1991, Tim Heidmann presented a much smarter solution that cleverly utilizes the stencil buffer to do the counting [50]. Initially, the stencil buffer is cleared to zero. Then the algorithm proceeds in three steps. First, the whole scene is drawn into the frame- and z-buffers, using only ambient lighting. The z-buffer now contains a depth representation of the scene. Second, with z-buffer writes disabled (but still doing z-tests), front- and back-facing shadow volume polygons are drawn separately. When front-facing shadow volume polygons are drawn, the stencil value is incremented for this pixel if the z-test succeeds, indicating that this shadow volume pixel is

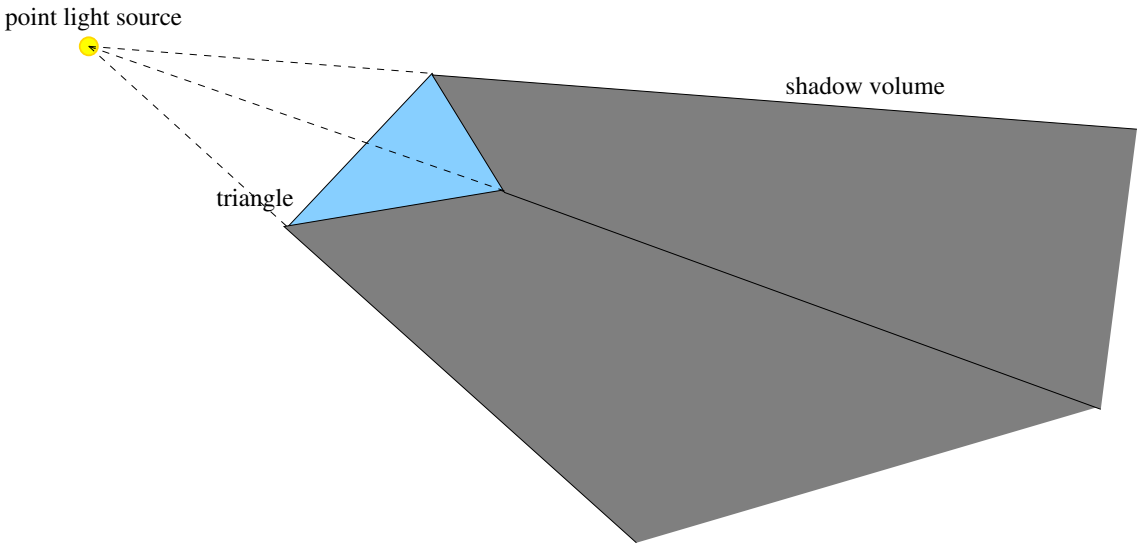

Figure 3.1: Shadow Volumes are formed by extruding an object's vertices away from a (point) light source to infinity.

visible from the eye. In our geometric interpretation (Figure 3.2), this is identical to the situation where the ray is entering a shadow volume. For drawing back-facing shadow volume polygons, the stencil value is decremented for this pixel if the z-test succeeds. Geometrically, this situation occurs when the ray from the eye leaves a shadow volume before hitting the object. After these two passes, regions in shadow are tagged by a non-zero stencil value. Recent graphics hardware (beginning with NVIDIA's GeForce FX) supports a special "two-sided" stencil test, where front- and back-facing polygons can be tested simultaneously in a single pass. Finally, the whole scene is rendered again, this time only with the diffuse lighting component, and displayed only where the value in the stencil buffer is zero. This value indicates that the ray has gone out of shadow volumes exactly as many times as it has entered them - i. e., this location is illuminated by the light. Because the stencil value is modified only if the z-test passes, this method is called *zpass* algorithm.

If the near plane of the view frustum intersects any of the shadow volumes, the zpass algorithm can produce incorrect results (Figure 3.3). The traditional way to solve the near clipping problem is to close the shadow volumes at the intersecting frustum plane by introducing additional geometry (*capping*) [10, 58, 74]. However, there are many different situations and special cases, so these methods are mostly not robust and general. In 1999, John Carmack from id software presented an alternate approach to avoid the clipping problem [58]; Bilodeau and Songy independently discovered the same method [11]. Logically equivalent to the zpass algorithm, the idea in this method, which is sometimes called *Carmack's reverse*, is to reverse the direction of the ray from infinity towards the eye. Instead of counting the number of shadow volume polygons *in front of* the object of interest, the number of shadow

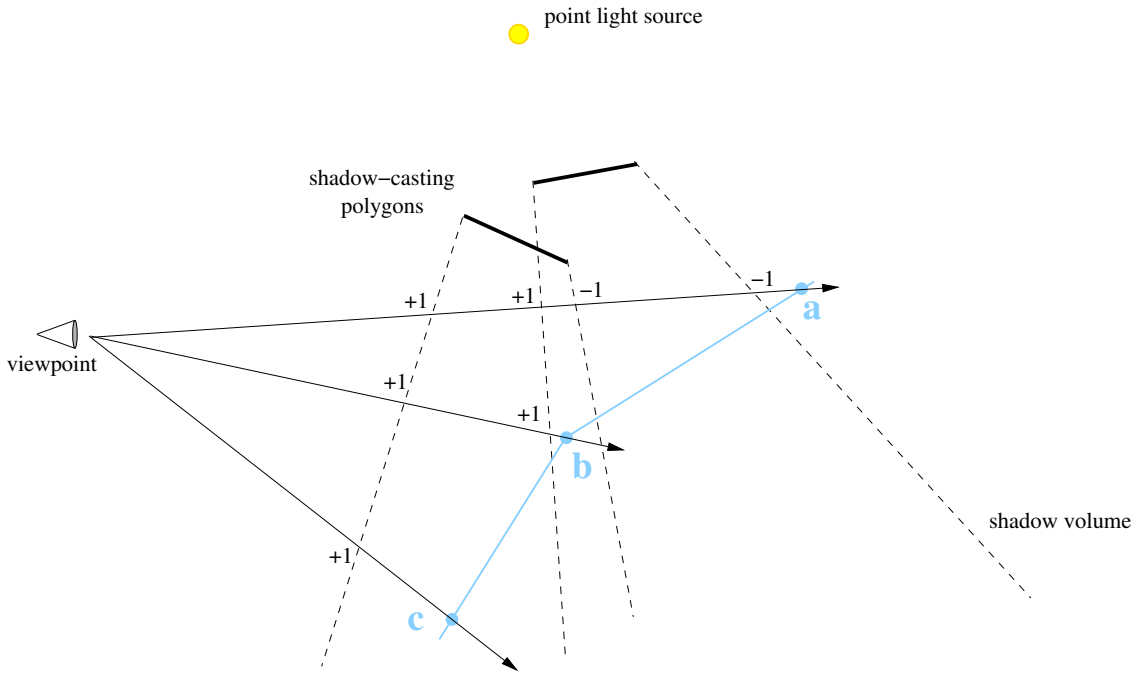

Figure 3.2: A two-dimensional view of counting shadow volume crossings. A ray is cast from the eye through each screen pixel. At point **a**, the ray has entered and left two shadow volumes. The counter value is zero, the point is not in shadow. Point **b** lies in the shadow volume of both polygons, the counter value is 2. Point **c** lies in a single shadow volume, the counter value is 1.

volume polygons *behind* the object are counted. Hence, in the first stencil pass, back-facing shadow volume polygons are rendered and the stencil value is incremented if the z-test fails. In the second stencil pass, front-facing shadow volume polygons are drawn and the stencil value is decremented if the z-test fails. Because the stencil value is modified only if the z-test fails, this algorithm is usually called *zfail*. By rendering only the hidden shadow volumes, the clipping problem at the near plane is avoided. However, the same problem now exists at the far plane: to properly maintain the count of entry/exit events, the shadow volumes must be closed at their far ends, and these far ends must be inside the far plane. With zpass, shadow volumes may intersect the view frustum's near plane. With zfail, shadow volumes can potentially intersect the far plane and cause similar shadowing errors. Another point worth mentioning is that the shadow volumes need not be closed at their near end for zpass, because these closing polygons are always made invisible by the scene geometry rendered in the ambient pass. With zfail, both near and far caps of the shadow volumes must be rendered.

Everitt and Kilgard discovered two solutions to the far clipping problem. The first, called *depth clamping*, is available as a hardware extension `NV_depth_clamp` since the NVIDIA GeForce 3 graphics accelerator. When using this extension, objects beyond the far plane are no longer clipped, but rather drawn on the far plane with

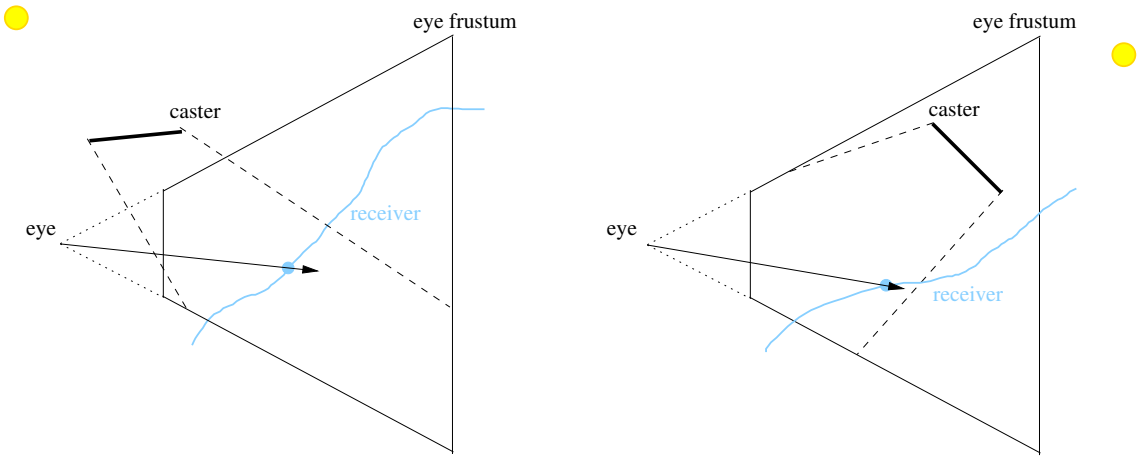

Figure 3.3: This figure depicts two cases where the zpass algorithm will fail. On the left, the shadow volume is clipped by the near plane of the view frustum. The pixel's stencil value is 0, although it should lie in the shadow. On the right, the shadow volume is clipped by the top plane, leading to a similar problem.

a maximum z depth. Both shadow volume edge and capping edges can be extruded out arbitrarily far, and will still be handled properly by the hardware.

The second solution Everitt and Kilgard presented is a pure software solution and does not require any hardware extensions at all. It exploits an important trait of four-dimensional homogenous coordinates commonly used in computer graphics. A three-dimensional point $\mathbf{p}$ can be represented in homogenous coordinates as the quadrupel $(p_x \cdot w, p_y \cdot w, p_z \cdot w, w)$. To obtain the original, three-dimensional point $\mathbf{p}$, the x-, y-, and z-coordinates are divided by the fourth coordinate, w. Therefore, points can be projected to infinity (along their position vector) by setting the w coordinate to zero. It is mathematically valid to think of points at infinity this way, and this fact is used in projecting a shadow volume's far cap vertices to infinity. Projecting shadow volume vertices to infinity this way does not yet solve the problem, because the far plane will still clip the shadow volume. The second key part of the solution is that the far plane itself is set to infinity, just like the far cap vertices of the shadow volume are. With the far plane at infinity, the zfail method works flawlessly. The shadow volume cap will be correctly rendered at infinity, and nothing will be clipped.

3.2 Shadow Volume Optimizations

Shadow volumes are created in object space and produce shadows with sharp, crisp edges. They do not suffer from any of the aliasing problems that are inherent in image-based algorithms (see page 2.6.3) and can be used on general-purpose

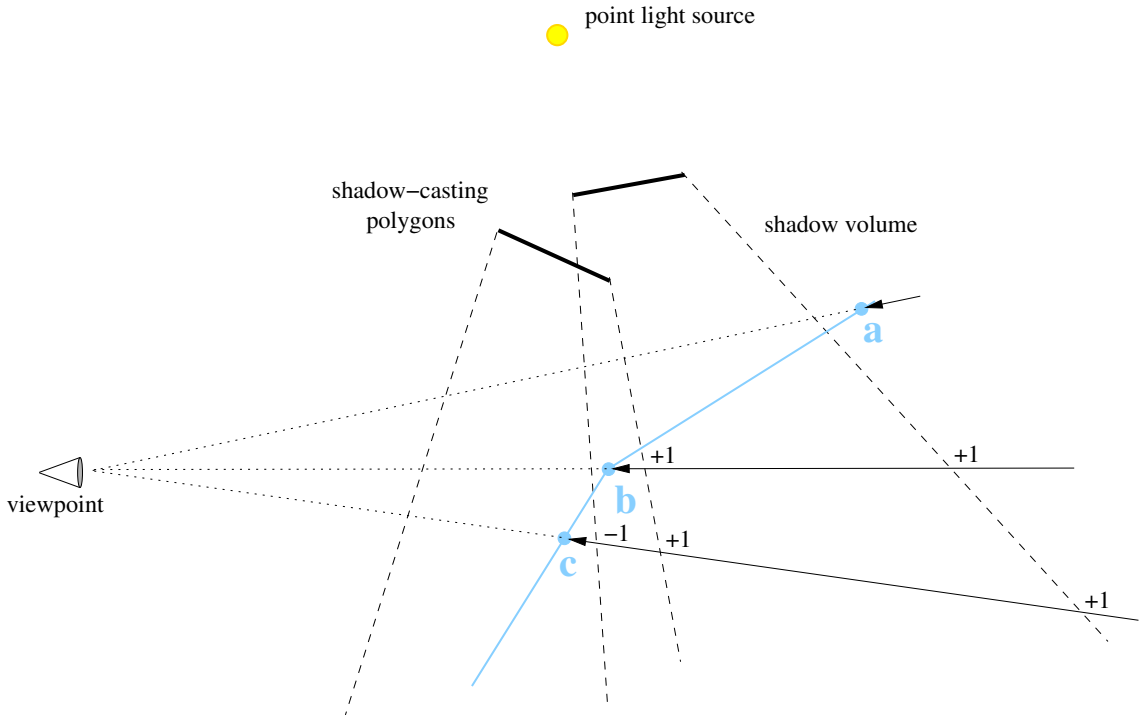

Figure 3.4: An illustration of the zfail algorithm. Rays are cast from infinity towards the eye, until an object is hit. The stencil value at each pixel is calculated as the difference between shadow volume entries and exits behind the object.

hardware, the only requirement being a stencil buffer (though Roettger et al. discuss a number of strategies for using color and alpha buffers to take the place of the stencil buffer [80]). However, there is a number of limitations and drawbacks. In this section, we will examine most of the shortcomings with shadow volumes and discuss techniques that try to alleviate some of these problems.

An obvious problem with the shadow volume algorithm is that they add significantly to the number of triangles in the scene - for each light, each occluding triangle introduces three additional quadrilaterals that must be properly extended and rendered into the stencil buffer. The problem is even more distinct with the zfail algorithm because the shadow volumes have to be closed both at the near and far ends. A commonly used technique to reduce the number of triangles is to determine *silhouette* edges and extrude only those. This works for all *manifold* objects (section 4.1). Several methods exist for silhouette detection (section 4.2).

Another area of concern is the fact that shadow volumes extend to infinity and often cover a large area on the screen, stressing the rasterizer unit of the graphics accelerator and burning fill rate. Several solutions have been proposed recently that adress this problem by trying to limit shadow volume polygons to those that are absolutely necessary for the correctness of the final image (see sections 3.2.3, 3.2.5

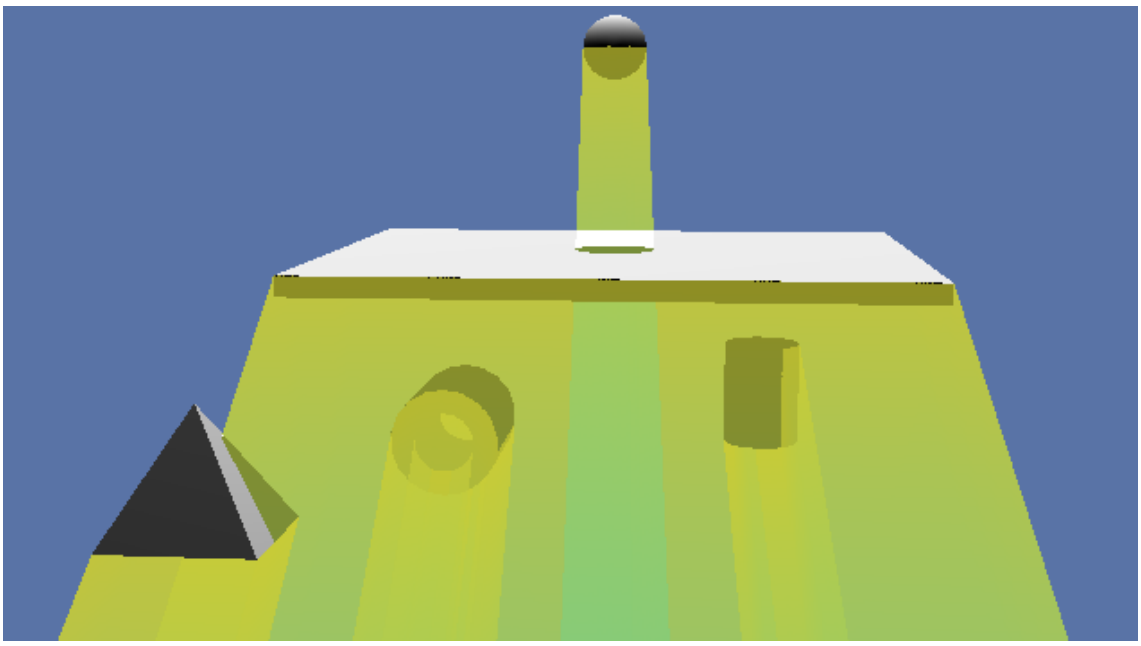

Figure 3.5: Shadow volumes (yellow gradient) in a scene with several objects. The point light is located directly above the quad.

and 4).

3.2.1 The zp+ Algorithm

Section 3.1 described two different techniques for stencil shadow volumes. Both first initialize the stencil buffer to zero and create a depth representation of the scene in the z buffer, then rasterize the shadow volumes. For every visible shadow volume pixel (i. e., fragment that passes the depth test), the zpass method increments the stencil value if the shadow volume is front-facing and decrements the stencil value if the shadow volume is back-facing. Finally, shadow regions are those with a stencil value greater than zero. This method fails when shadow volumes are partially clipped by the view frustum planes, which was the motivation for the development of the zfail method. Zfail moves the problem from the near clipping plane to the far clipping plane by processing only shadow volume fragments that fail (hence the name) the z test. The clipping problems at the far plane can be circumvented by extending the view frustum to homogenous infinity. However, this robustness comes at the cost of speed, because the shadow volumes must be closed, which requires the rendering of additional cap geometry. Furthermore, zfail cannot utilize early z culling, an optimization available in state-of-the-art graphics hardware that prevents occluded fragments from processing far into the pipeline.

A closer examination of the zpass algorithm reveals that the clipping problems are

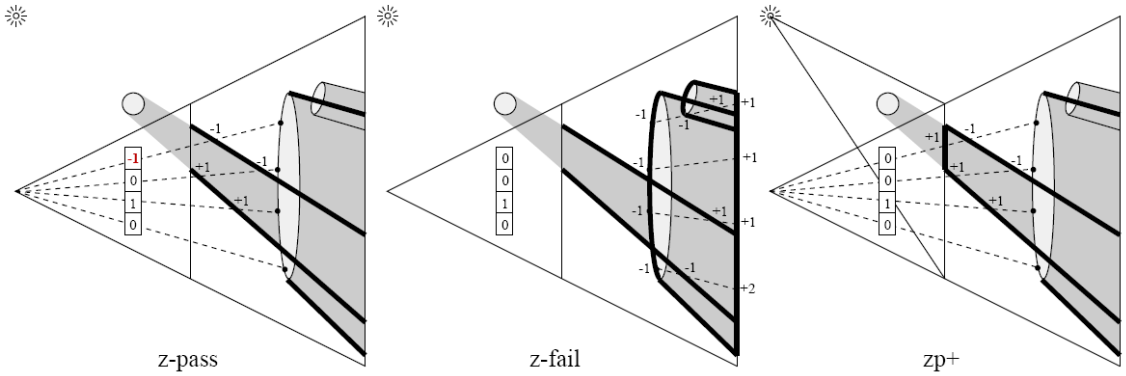

Figure 3.6: zpass vs. zfail vs. zp+. The zp+ algorithm initializes the stencil buffer to the proper values ordinarily clipped by zpass. Therefore it can process shadow volumes with the depth-culled speed of zpass and the robustness of zfail. *(Image courtesy of Hornus, Hoberock, Lefebvre, and Hart)*

due to the scene geometry within the pyramid-shaped volume between the viewer's near plane and the light source. Any occluder lying at least partly within this region casts a shadow volume that intersects with the near plane and causes zpass to produce incorrect results. A technique aimed at resolving this deficiency is the *zp+* algorithm, presented by Hornus et al. in 2005 [56]. This algorithm provides the robustness of the zfail technique while staying close to the speed of the zpass technique. ZP+ constructs in a separate pass a sheared frustum that extends from the light's position to the view frustum's near plane (Figure 3.6, right). Because the far clipping plane of this shared frustum coincides with the viewer's near plane, the frustum contains those parts of the scene geometry that would be clipped by the viewer's near plane. Rasterizing this geometry projects its fragments onto the original near clipping plane where it can be used to properly and robustly initialize the stencil buffer.

ZP+ involves the following steps:

1. Position a frustum at the position of the light source.
2. If viewer and light are on the same side of the viewer's near plane,
   a) then orient the light frustum parallel to the viewer's frustum
   b) otherwise, rotate the light frustum 180 degrees around the y axis.
3. Align the light frustum's far plane onto the viewer's near plane.
4. Rasterize front-facing (with respect to the light source) geometry, accumulating fragment counts in the stencil buffer
5. Perform standard zpass shadow computation, initialied with the values in the stencil buffer.

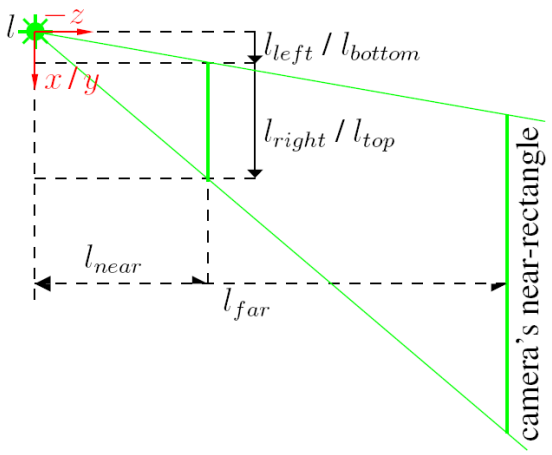

Figure 3.7: This figure illustrates how the parameters for the light frustum can be calculated. *(Image courtesy of Hornus, Hoberock, Lefebvre, and Hart)*

The light's modelview matrix $\mathcal{M}$ can be seen as a frame with origin at the light's position. It is setup so as to have the same up vector (y) as the viewer. The viewing direction ($-z$) is parallel to the viewer's direction and oriented towards the viewer's near plane. Therefore, the transformation from the viewer's modelview matrix to the light's modelview matrix involves a translation and, if the viewer and the light source are on opposite sides of the viewer's near plane, a rotation of 180 degrees around the up vector.

The light's projection matrix $\mathcal{P}$ is an off-centered, sheared perspective transformation with its far plane aligned with the viewer's near plane. The light's near plane should be placed as far away as possible to attain the highest possible precision. It can be computed using a simple procedure [56]. If the distance d from the light to the nearest occluder is known, l_{far} is the distance of the light's far plane, and d_{max} is the farthest distance from the light to the viewer's near plane, then the distance of the light's near plane can be computed as

$$l_{near} = \frac{l_{far} \cdot d}{d_{max}} \tag{3.1}$$

$\mathcal{P}$ can be setup using an API call like `glFrustum` as illustrated in Figure 3.7. However, due to numerical errors, it is often preferrable to setup P directly.

After the transformation matrices have been set up, the scene geometry is sent to the graphics pipeline to rasterize the near cap. To count the number of shadow volume entries for every pixel, the API states are set to cull either back- or front-facing faces (depending on whether the viewer and light source are on the same side of the near plane), increment the stencil value for every passing fragment and disable the z test. The rest of the algorithm is identical to the zpass method.

Rendering shadow volumes this way may lead to visual artifacts due to numerical

computation errors. When the viewer's near plane lies both in light and shadow, several lit pixels that should be dark or vice versa can be observed. These incorrect pixels lie exactly on the boundary of the near cap, along the edges where the shadow volume's sides and near-cap meet on the near plane. The screen-space coordinates of these edges are computed in two different ways. First, when rendering the near cap, they are the result of some silhouette edges' projection with the zp+ modelview matrix $\mathcal{M}$ and projection matrix $\mathcal{P}$. Later, when the sides are rendered the usual way, they are the coordinates of some clipped shadow volume side edges. Because of non-exact arithmetic precision, the results of the two transformations differ slightly, and those edges are no longer rasterized identical. The resulting artifacts can be eliminated by manually clipping the shadow volume edges on the viewer's near plane, e. g., by using a vertex program. The exact procedure is described in detail in [56]. In many cases, the artifacts are not disturbing enough to justify the additional effort.

3.2.2 Shadow Volume Reconstruction from Depth Maps

Michael McCool was the first to propose a hybrid algorithm combining shadow maps and shadow volumes. He introduced his algorithm for reconstructing shadow volumes from a depth map in 2000 [73]. It does not require a polygonal representation of the scene. Instead, somewhat similar to the shadow map method, it requires a depth map rendered from the light source location. The shadow map information is then used to reconstruct a polygonal shadow volume boundary that can be combined with an eye-view depth map using only a one-bit stencil buffer. Reconstruction is based on edge detection in the shadow map. The identified silhouette edges are then used to build a polygonal mesh representing the important shadow volume boundaries. This algorithm does not need any hardware extensions, hence it can be used with any hardware that generates a correct depth map.

The algorithm's key observation is the fact that depth samples in the shadow map, in conjunction with their pixel coordinates, describe scene points relative to the light position in an orthogonal device coordinate system. These points can be joined into a polygonal mesh and then transformed back into world space, using the inverse of the shadow map projection. Surfaces that are constructed in this way define the boundaries between shadow and light in the scene.

In the pure shadow volume algorithm, shadow volume boundaries may be nested, hence the need to sum front-facing polygons and subtract back-facing polygons. With McCool's technique, shadow volume boundaries consist of a single star-shaped surface. In this case, it is only necessary to keep track of the *parity* of the number of shadow volume polygons in front of the surface at each pixel. If the number of shadow volume boundary intersections along a ray from the eye to the first surface point at a pixel is odd, the pixel is in shadow. Otherwise, it is illuminated. To keep track of the parity, a single bit in the stencil buffer is sufficient. This bit is simply inverted whenever a shadow polygon fragment is drawn on top of it.

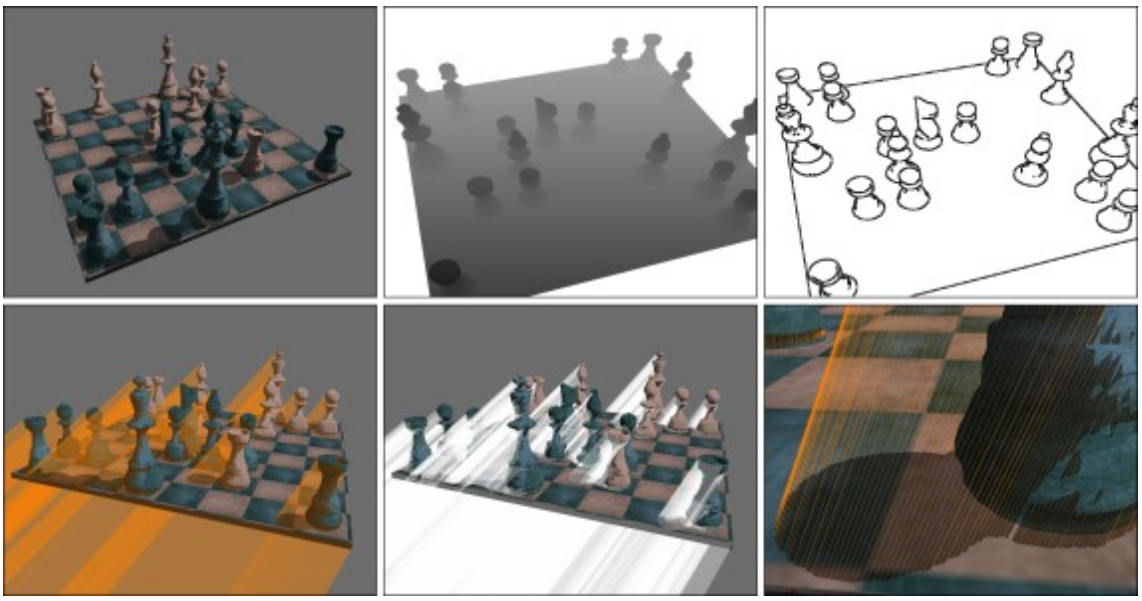

Figure 3.8: Shadow volumes reconstruction from depth maps. From left to right and up to down: scene, depth map created from the light position, edges detected in depth map, reconstructed shadow volumes, shadow volumes created from geometry of occluders (using potential silhouette edges), wire model of reconstructed shadow volumes in detail. *(Images courtesy of David Ambrož, CGG Prague)*

The algorithm inherits all of the advantages of the shadow map algorithm. In particular, since the shadows depend only on the values left in the depth buffer, it is non-intrusive - adding shadows to a scene will not require modification of the base rendering engine. Another advantage of this hybrid approach is that the generated shadow volumes are minimal in size. They extend from the shadow caster's surface to the shadow receiver's surface, but no further, whereas Crow's original shadow volumes extend outwards to the view far plane after clipping, stressing the rasterization hardware. However, the hybrid method also inherits some of the disadvantages of the shadow map algorithm. Aliasing is still a problem, being literally magnified when the shadow is projected, as is the precision problem of the z buffer and the shadow map. Another disadvantage is the large number of shadow polygons, which is inherited from the basic shadow volume algorithm.

Since this technique still suffers from the same problems as shadow mapping and only offers the advantage of a somewhat reduced number of shadow volume polygons, it is not often used in practice.

3.2.3 An Efficient Hybrid Shadow Rendering Algorithm

It can be shown that shadow map aliasing is only noticeable at the discontinuities between shadowed and illuminated regions, i. e., at shadow silhouettes [86]. Then

Figure 3.9: Comparison of image quality using a shadow map (left), shadow volumes (middle), and Chan/Durand's hybrid technique (right). A shadow map resolution of 1024x1024 pixels was used for the shadow map and hybrid algorithms. *(Image courtesy of Eric Chan and Frédo Durand)*

again, shadow volumes compute shadows accurately at every pixel, but this accuracy is in fact only needed at the shadow silhouettes. This observation suggests a hybrid algorithm that uses the exact shadow volume algorithm only at the shadow silhouettes and the faster shadow map algorithm everywhere else in the scene. In fact, this approach avoids not only aliasing problems with shadow maps, but also the high fill costs of the standard shadow volume algorithm by Crow.

The technique presented in this section was developed by Eric Chan and Frédo Durand [21]. In the first step, an ordinary shadow map is created, as described on page 34. This shadow map serves to identify shadow silhouette pixels and compute shadows for non-silhouette pixels. In the second step, the scene is rendered from the eye's viewpoint. Each fragment is transformed back to light space, where its depth is compared against the four nearest depth samples from the shadow map [86]. If the comparison results agree, the fragment is a non-silhouette pixel and is shaded according to the depth comparison result. Otherwise, the fragment is classified as silhouette. During this step, standard z-buffering is performed as well, to create the depth map needed for drawing the shadow volumes in the next step. After silhouette pixels have been identified, shadow volumes are drawn according to the standard shadow volume algorithm explained in section 3.1. The key difference in this technique, however, is that shadow volumes are rasterized only at silhouette pixels. Last, the shaded scene is rendered only at pixels with a stencil value of 0, thereby avoiding the shadowed regions in the scene.

This technique relies heavily on the graphics hardware to discard unnecessary shadow volume fragments as early as possible in the pipeline. To restrict rasterization of shadow volumes and stencil updates to regions containing silhouette pixels, Chan and Durand propose using *computation masks*. A computation mask is a device that allows masking of specific framebuffer addresses, so that the hardware can avoid processing pixels at those locations [21]. Current graphics hardware does not directly expose computation masks. However, the OpenGL extension `EXT_depth_bounds_test` can be used to simulate the behavior. The idea is to use a pixel shader to mask pixels by setting their depth values to a constant value *outside* the depth bounds. By enabling depth bounds in the subsequent steps, rasterized pixels with an outside

depth value can be discarded early in the pipeline.

3.2.4 Interactive Shadow Generation in Complex Environments

To address the problem of interactive shadow volume generation in static environments with a large number of geometric primitives, Lloyd et al. present a distributed method that utilizes a cluster of several workstations [39].

The algorithm uses a scene graph representation which is augmented with a sub-object hierarchy (that is, large individual objects that contain a high number of triangles are broken down into sub-objects with only a few triangles). The decomposition into sub-objects is done using a combination of partitioning and clustering algorithms to ensure that they have roughly the same size. Furthermore, the algorithm precomputes LODs for leaf nodes and HLODs (hierarchical LODs) for intermediate node. The HLOD that is associated with an internal node represents a simplification of all objects contained in the subtree of that node.

In the first step, the algorithm interactively computes the potentially visible set (PVS) for both the eye-view and the light-view. The PVS for the eye is denoted PVS_e, the PVS for the light view PVS_l. Levels-of-detail are used to accelerate this step by selecting the appropriate LODs for each view. First, an occlusion representation of the scene is generated. Then, scene graph culling is performed using hardware occlusion queries (subsection 2.5.1). The overall performance of this step is affected by two factors. The first factor is whether culling is performed on sub-object level or not. Sub-object culling takes more time because of the additional occlusion queries, but can result in a smaller PVS. The second factor is the number of triangles per primitive k. If k is large, the number of sub-objects per object is reduced, thereby fewer occlusion queries have to be issued. Low values for k result in a much smaller PVS. The smallest PVS for any given scene can be obtained using a k value of 1.

Every triangle in PVS_l is a potential shadow caster, and every triangle in PVS_e a potential shadow receiver. However, not all triangles in PVS_l actually cast shadows that are necessary for the correctness of the final image. To filter out unnecessary shadow casters in PVS_l, *cross culling* is performed between PVS_e and PVS_l. Cross culling first checks the visibility of the triangles in PVS_e with respect to PVS_l, then uses this information to obtain the set of actual shadow casters ($\mathcal{SC}$). Therefore, the triangles in PVS_e are partitioned into three subsets:

- **Fully-lighted** ($\mathcal{FV}$): These triangles are fully visible from the light view and are drawn completely lit.
- **Fully-shadowed** ($\mathcal{SR}_{\mathcal{F}}$): The exact opposite of $\mathcal{FV}$, these triangles are completely occluded from the light view and therefore are fully in shadow.

- **Partially-shadowed** ($\mathcal{SR}_\mathcal{P}$): These triangles are only partially visible from the light and must be considered for shadowing.

After partitioning PVS_e, $\mathcal{SC}$ is computed as the set of triangles in PVS_l that occlude $\mathcal{SR}_\mathcal{P}$.

Shadowing is done in the final step. Only triangles in $\mathcal{SR}_\mathcal{P}$ and $\mathcal{SC}$ are considered. The algorithm to create shadows is a variation of the classic polygon shadow technique by Atherton et al. (page 32). All triangles in $\mathcal{SR}_\mathcal{P}$ are clipped against the shadow frusta formed by each triangle in $\mathcal{SC}$. Therefore, the resulting shadow polygons are calculated by repeatedly clipping triangles in $\mathcal{SR}_\mathcal{P}$ against the planes of the shadow frusta.

If the number of the shadow polygons is very high, the clipping operations may not be performed at interactive rates on the CPU. Therefore, Lloyd et al. suggest a hybrid scheme that uses shadow maps instead of shadow polygons if they are very small or far away from the eye. Therefore, shadow polygons are computed only where they make a significant difference in image quality, and a shadow map is used everywhere else.

Because it is not (yet) possible to compute PVS_l, PVS_e and perform cross-culling and shadow generation on a single graphics processor at interactive rates, Lloyd et al. use a process-parallel algorithm and distribute the work on three graphics processors (on three different workstations). The algorithm is distributed as follows:

- GPU_1: computes PVS_e
- GPU_2: computes PVS_l
- GPU_3, CPU_1, CPU_2 and CPU_3: perform cross-culling, shadow generation and rendering of the final image

The communication between the workstations is synchronized using an acknowledge-based protocol.

The original implementation used three Dell Precision workstations, each with dual 1.8 GHz Pentium4 CPUs, 2 GB of main memory and a NVIDIA GeForce 4 Ti4600 GPU. With this configuration, Lloyd et al. are able to render 2M triangles in immediate mode and about 14M triangles in retained mode.

3.2.5 CC Shadow Volumes

Lloyd et al. present a method for accelerating shadow volumes in scenes where shadow volume rasterization is the main bottleneck [69]. This method is an improvement of the technique described in subsection 3.2.4 and decreases rasterization cost by two different techniques: shadow volume *culling* and *clamping*.

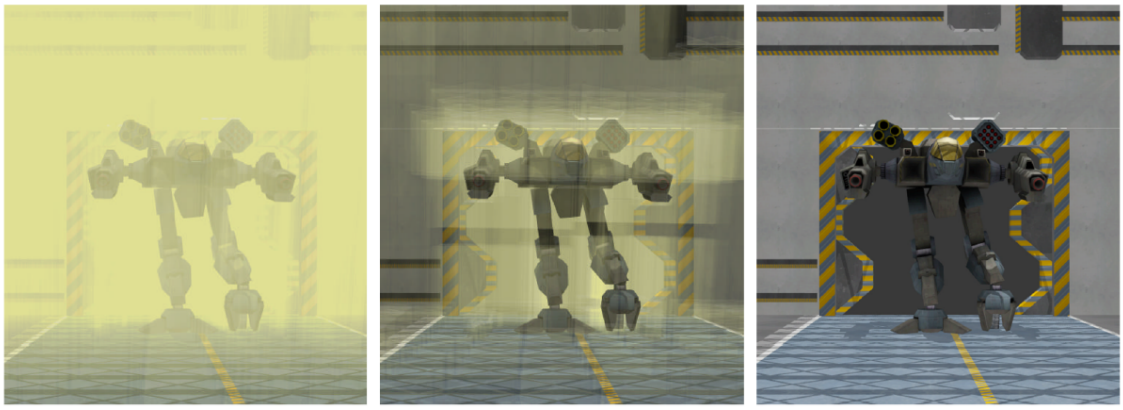

Figure 3.10: CC shadow volumes in a scene that consists of 96k polygons. Shadow volumes are colored in transparent yellow. Left: standard shadow volumes. Middle: CC shadow volumes. *(Image courtesy of Lloyd et al. [69])*

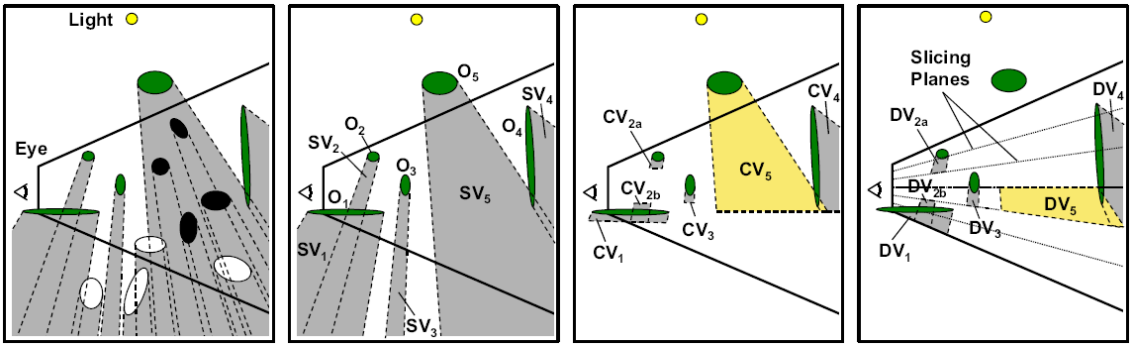

Figure 3.11: CC shadow volumes accelerate the standard shadow volume algorithm by first *culling* unnecessary shadow casters and then *clamping* surviving shadow volumes to fit tightly around the scene geometry. From left to right: standard shadow volumes, shadow volume culling, continuous shadow volume clamping, discrete shadow volume clamping. *(Image courtesy of Lloyd et al. [69])*

Culling

The culling step is very similar to the technique described in [39]. To recap, the culling of shadow volumes usually proceeds in two steps. First, all objects that are visible from the eye are detected, yielding the set of *potential shadow receivers* (PSR). Similarly, in the second step all objects are detected that are visible from the light source's point of view. Taking the previously identified objects in PSR into account, these objects can be additionally limited to those that actually occlude any visible objects from the eye. The second step yields the set of *potential shadow casters* (PSC).

Clamping

A closer examination of Figure 3.11 reveals that even after the culling step, the remaining shadow volume are largely unnecessary: only small parts of the shadow volumes actually cover any scene geometry. By clamping shadow volumes to regions that contain shadow receivers, we can save fill costs and increase overall performance.

Lloyd et al. perform clamping using two different techniques. *Continuous clamping* operates entirely on the host CPU and attempts to clamp shadow volumes to their z bounds in light space. *Discrete clamping* splits the viewing frustum by several planes into sections and utilizes the graphics hardware to find out which sections are occupied with shadow receivers. Like continuous clamping, it then limits the shadow volumes to these sections.

Continuous Clamping Continuous clamping is performed entirely on the CPU and does not need any dedicated graphics hardware. Clamping is done by transforming every object $\in$ PSR into the light's coordinate system, ideally the same that was used in the culling step. Then, the regions where the shadow volumes need to be drawn correspond to the intervals that are occupied by shadow receivers in the z direction.

Discrete Clamping In contrast to continuous clamping, *discrete clamping* uses the graphics hardware to test for shadow receivers within discrete shadow volume intervals. To this end, the view frustum is partitioned into slices by a set of similarly oriented planes. The discrete shadow volume intervals are determined by the intersection of the shadow volumes with the slicing planes. After the slicing planes have been constructed, discrete clamping uses image-space occlusion queries to test for slices that contain any shadow receiver, and shadow volumes are finally rasterized only in such slices.

3.2.6 Split-Plane Shadow Volumes

An interesting method for combining the robustness of the zfail algorithm with the speed of zpass was presented by Laine in 2005 [67]. The main idea behind this technique is the observation that since both zfail and zpass produce identical shadows, the choice between using zpass or zfail can be made *locally* as long as it stays consistent while rendering a single shadow volume. If this choice is made on a per-tile basis, it is possible to cull entire pixel tiles when it can be concluded that none of the pixels in that tile would cause stencil updates.

Both zpass and zfail adjust the stencil value for a given fragment depending on a given set of conditions. This adjustment can be formally expressed for both algorithms (Equation 3.2 and Equation 3.3).

$$\Delta S_{(zpass)} = \begin{cases} +1 & \text{if} \quad z_{frag} < z_{pixel}\ , \quad facing = front \\ -1 & \text{if} \quad z_{frag} < z_{pixel}\ , \quad facing = back \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

$$\Delta S_{(zfail)} = \begin{cases} +1 & \text{if} \quad z_{frag} \geq z_{pixel}\ , \quad facing = front \\ -1 & \text{if} \quad z_{frag} \geq z_{pixel}\ , \quad facing = back \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

By taking a pixel-dependent *split depth* (z_{split}) into account and comparing it against the depth stored in the depth buffer (z_{pixel}), an additional criterion can be applied to the stencil buffer adjustment ΔS. This comparison is referred to as the *split test*, and its result determines whether zpass or zfail update rules are used. The split depth z_{split} does not need to be the same for every pixel, but it must be ensured that z_{split} remains the same while a single shadow volume is processed. Thus, the result of the split test is consistent for individual shadow volumes. Using the split test yields new rules for stencil buffer adjustment, as denoted in Equation 3.4.

$$\Delta S_{(spsv)} = \begin{cases} +1 & \text{if} \quad z_{frag} < z_{pixel} < z_{split}\ , \quad facing = front \\ -1 & \text{if} \quad z_{frag} < z_{pixel} < z_{split}\ , \quad facing = back \\ -1 & \text{if} \quad z_{frag} \geq z_{pixel} \geq z_{split}\ , \quad facing = front \\ +1 & \text{if} \quad z_{frag} \geq z_{pixel} \geq z_{split}\ , \quad facing = back \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

Hence, if z_{pixel} is smaller than z_{split}, zpass stencil update rules are applied. Otherwise, zfail stencil update rules are used.

The motivation for adding the split test is that no stencil buffer update is required unless z_{pixel} is between z_{frag} and z_{split}. This by itself results in a reduction in the number of stencil buffer updates, provided that the split depth z_{split} is chosen adequately. Therefore, the challenge with this technique is the calculation of z_{split}. Laine suggests assigning a suitable, automatically constructed *split plane* for every

individual shadow volume and calculating the split depths based on the plane equation of that split plane. [67] describes two different methods for constructing the split planes and performing the split test against a plane.

A reduction of the number of stencil updates is not that much of an improvement, unless the number of processed pixels can be reduced as well. Moreover, the cost of performing the split test on a per-pixel basis may well surpass the benefit gained from reducing the number of stencil updates. Therefore, this technique requires an efficient hardware implementation that makes it possible to cull multiple pixels at once, thereby reducing the number of processed pixels. A possible implementation involves pixel tiles, per-tile depth tests and hardware per-tile split tests. For every pixel tile, z_{min} and z_{max} values are maintained, which enables the culling of complete pixel tiles.

4 Shadow Volumes in Complex Scenes

The aim of this thesis was to implement shadow volumes at interactive frame rates in complex urban scenes. As opposed to the technique described in subsection 3.2.4, the algorithm should run on a single workstation while still providing acceptable performance. Moreover, most optimization techniques that try to improve fill rate achieve their goal by increasing the load on the CPU and/or the geometry stage. Because all of these stages are comparably busy, such an approach does not improve the performance in these scenes.

A typical view of the city model can be seen in Figure 4.1. This scene consists of more than a million triangles. Furthermore, the triangles are distributed over a large number of small, individual objects. Hence, the scene graph representation of the scene is flat, but very wide, which requires significant processing power on the CPU. In the context of this thesis, we call scenes *complex* if they contain a lot of geometric primitives *that are distributed over a large number of small, independent objects*. This means that the robot scene used in subsection 3.2.5 does not qualify as a complex scene by this definition because the polygons are clustered in large, but few objects.

In common terms, scenes are called complex when they contain a sufficiently large number of triangles and vertices. In such scenes, shadow maps often do not generate pleasing results because of their aliasing problems. For shadow maps to work in those scenes, we would have to drastically increase the image resolution, which would in turn decrease performance. In contrast, shadow volumes do not have any aliasing problems, therefore they are still the first choice in complex scenes.

The shadow volume algorithm produces good results, but this comes at a severe performance hit. This performance hit takes place at three independent places:

- **CPU** The shadow volumes have to be constructed on the host CPU. Moreover, depending on the variation of the algorithm used, possibly silhouette edges must be computed, at the worst in every frame for dynamic scenes.
- **Geometry** Shadow volume polygons increase the number of geometric primitives (triangles) in the scene, therefore more vertices have to be processed in the geometry stage.

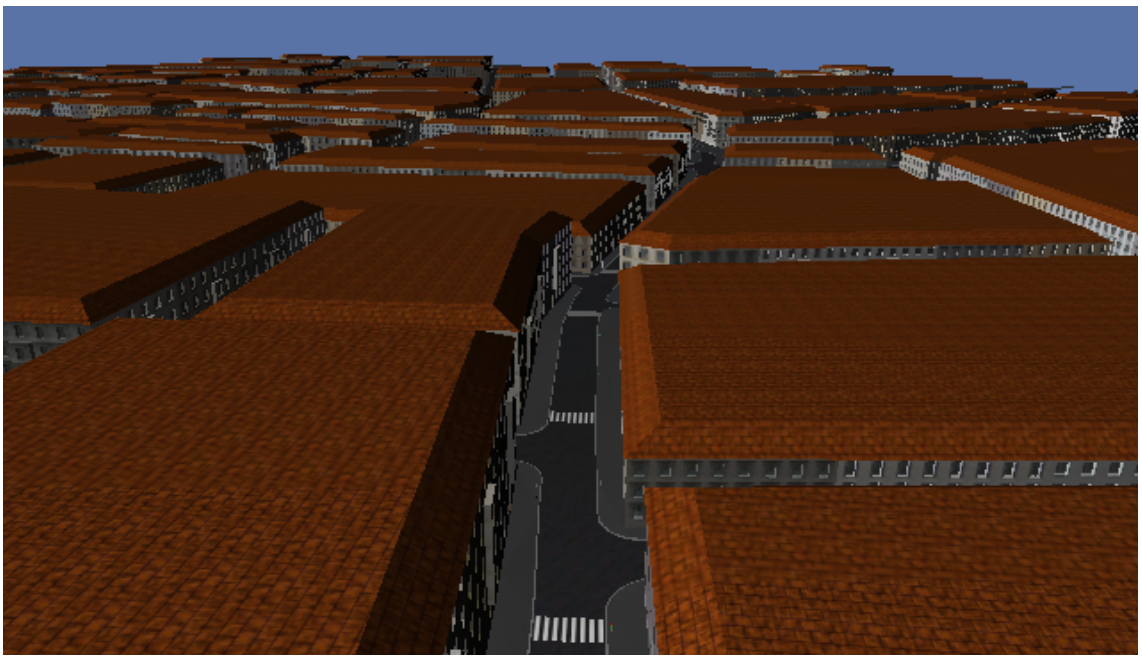

Figure 4.1: The city scene that was the basis for this thesis. The city is stored in a scene graph that consists of 17,420 individual geometric objects, some of which are instantiated for multiple use. The geometry itself is made up of 1,033,002 triangles and 872,715 vertices.

- **Rasterizer** Since shadow volumes usually extend to infinity, they cover large regions of the screen, which results in a lot of rasterized fragments and, together with the additional stencil writes, burns fill rate.

Implementing standard shadow volumes without optimizations in such a complex scene results in an unacceptable performance on even the latest graphics hardware (several seconds per frame on an NVIDIA GeForce 6600 GT). Furthermore, as we will see, many optimization techniques described in section 3.2 fail with this scene because of the additional requirement that the scene geometry is distributed over a large number of objects.

In the following sections, we will take a closer look at the different steps involved in performing the classical shadow volume algorithm and try to improve the various stages. To do this, we will take some of the optimization techniques described in section 3.2 and adapt them for our specific needs. In section 4.1, we will discuss existing techniques to enhance the generation of shadow volumes, ideally by using fast graphics hardware instead of software algorithms. Section 4.2 describes ways to determine an object's silhouette with respect to the light source, and how shadow volumes can be created at these silhouette edges. Constructing a shadow volume only at silhouette edges reduces depth complexity by decreasing the number of shadow volume polygons, which in turn decreases the number of buffer writes (stencil, z). Section 4.3 discusses strategies to reduce (*cull*) the number of shadow

casting objects. Not all objects in a complex scene cast relevant shadows, because some are completely enclosed in the shadow of another object, while others do not occlude any visible objects at all. The culling algorithms presented by Lloyd et al. ([69], see section 3.2) address this issue, however they are tailored toward scenes that contain only a few number of individual objects and do not produce acceptable performance in the city scene. Therefore, this chapter introduces an improved algorithm that works in arbitrary scenes that do not necessarily consist of only a few, clustered objects. Furthermore, we discuss how to apply a sophisticated technique to focus the light source's view on the relevant parts of the scene (which is necessary to achieve culling speed and accuracy). After culling unnecessary shadow casters, section 4.4 describes a technique to limit (clamp) shadow volumes to those regions in space that actually contain geometry, avoiding large shadow volumes that only cover empty space. In this section, we also present a technique that optimizes the rendering of the shadow volume segments by creating them on-the-fly in a vertex shader. Finally, all techniques are tied together in section 4.5, describing a complete algorithm for shadow volumes in complex scenes.

4.1 Creating Shadow Volumes

The construction of a shadow volume is a promising candidate for improving the CPU part of the shadow volume algorithm. Since a complex scene consists of a high number of shadow casters, this step has to be repeated many times. Therefore, improving the shadow volume construction step can result in a significant overall performance gain. In this section, we will take a closer look at some existing techniques for shadow volume creation.

To construct a shadow volume for a single triangle, each of the three vertices has to be duplicated and projected away from the light source. How exactly each vertex is projected depends on the specific light source. Directional lights, i. e., lights that are located infinitely far away from the scene, cast parallel light rays, hence all three vertices are projected along the same vector. For point light sources with a fixed position in the scene, each vertex must be projected along the vector from the light source to the vertex. Figure 4.2 clarifies the process.

To obtain the extruded point (usually called *far cap* vertex), the light vector $\vec{l}$ is multiplied by a parameter t and added to the original point p. Setting t to infinity yields an extrusion point $p_\infty = (l_x, l_y, l_z, 0)$. The sides of the shadow volume can be rendered by warping every edge of the object into a quadrilateral by duplicating and extruding each of its two vertices. The resulting quad can then be sent to the graphics pipeline, e. g., as a strip of two triangles. When the zfail algorithm is used (chapter 3), the shadow volume can be closed by rendering the original object two times, once with its original vertices, and a second time with the far cap vertex p_∞. For solid occluders, the number of triangles can be reduced somewhat by using

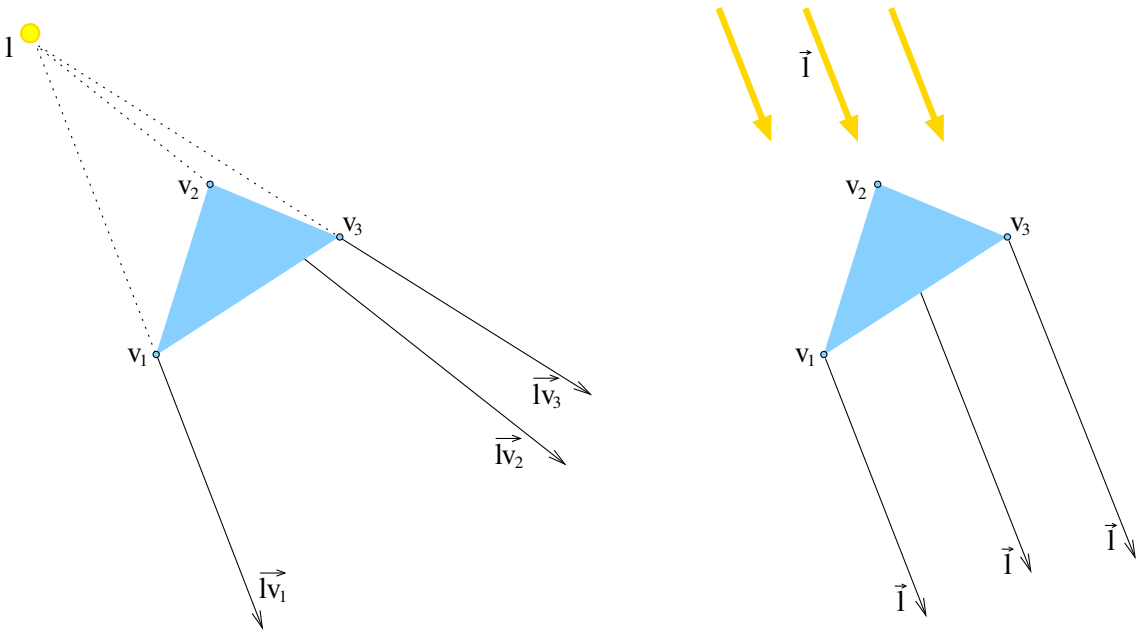

Figure 4.2: The creation of a shadow volume for a single triangle. With point light sources (left), points are extruded along the path from the light source (l) to the vertex (v_1, v_2, v_3), resulting in the vectors $l\vec{v}_1$, $l\vec{v}_2$ and $l\vec{v}_3$. If the light source is located at infinity (right), all incoming rays are parallel. In this case, the extrusion vector is the same for all vertices, namely the light direction vector $\vec{l}$.

only the set of triangles facing towards (or, equivalently, the set of triangles facing away from) the light to render the shadow volume caps. A big drawback with this simple technique, however, is the high number of triangles that make up the shadow volumes. Each triangle creates six additional triangles forming the shadow volume's sides, plus two triangles for the shadow volume caps if the zfail algorithm is used.

A smarter solution extrudes only the objects' silhouette edges to form the sides of the shadow volumes. This technique involves finding the silhouette edges of the occluding object (see section 4.2). For it to work, however, the object must be *manifold*. With manifold objects, none of the back-facing triangles are visible from outside the object, and every edge is part of exactly two triangles. A silhouette edge is then defined as an edge where one adjacent triangle face towards the light, and the other triangle faces away from the light. Therefore, only the silhouette edges need to create shadow volume quadrilaterals. We will examine methods for silhouette detection more closely in section 4.2. Generating shadow volumes in this way increases the load on the CPU, because the silhouette must be recalculated whenever the relative position of the occluding object and the light source changes, but can significantly cut down the number of shadow volume polygons.

Another interesting idea utilizes the vertex shader to create shadow volume polygons. Several different methods are thinkable. One technique, originally presented by ATI [45, 19], sends every occluder edge down the pipeline as a degenerate quadrilateral when constructing the shadow volume, supplying the normals for both adjacent triangles as well (Figure 4.3). Specifically, the original edge vertices are assigned one normal, the vertices from the replicated edge the other. For each of the two edges, the vertex shader checks the respective surface normal against the light direction. If the normal faces the light, the vertices are part of the light cap and passed through without change. If the normal faces away from the light, the vertices are part of the dark cap. In this case, the vertices are projected away along the vector from the light to the vertex. By doing this, shadow volumes are formed automatically. If the surface normals of both adjacent triangles face toward the light, the edge is rendered as a degenerated quadrilateral. If both surface normals face away from the light, the edge is still rendered as a degenerated quadrilateral, but moved to the far cap, thus closing the shadow volume. If the surface normals have opposite facings, the edge is part of the silhouette. In this case, one edge remains at its original position, while the other is projected away, forming a side quadrilateral of the shadow volume. An advantage of this method is that for static objects, the edge quadrilaterals can be formed once and then stored in a vertex buffer directly on the graphics hardware for fast access. This method does not require silhouette determination on the CPU, but doubles the amount of vertices in the scene.

A method used by BioWare in their game *Neverwinter Nights* to create shadow volumes on the fly first determines silhouette edges on the CPU, then uses the centroid of the occluding object to create a triangle strip of four triangles formed by six vertices for each silhouette edge. One triangle connects the centroid with

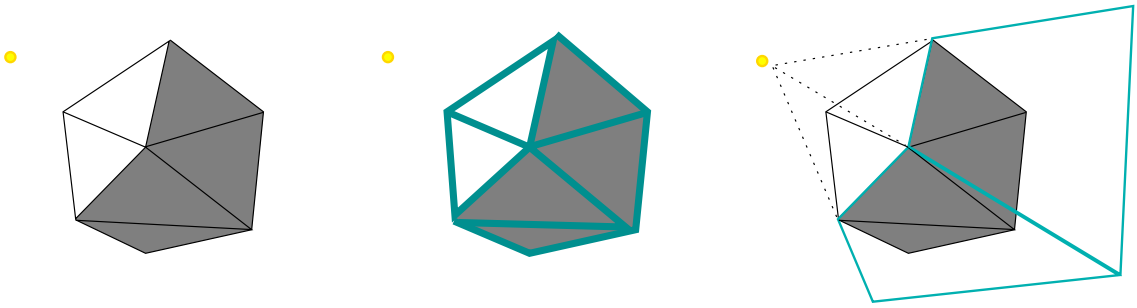

Figure 4.3: Constructing a shadow volume using vertex shader hardware. In the leftmost figure, the occluder is shown. In the middle, the edges are replaced by degenerate quadrilaterals with two identical vertices (thick outline). On the right, the vertex shader extrudes those edges found to be part of the silhouette away from the light source, thus forming the sides of the shadow volume.

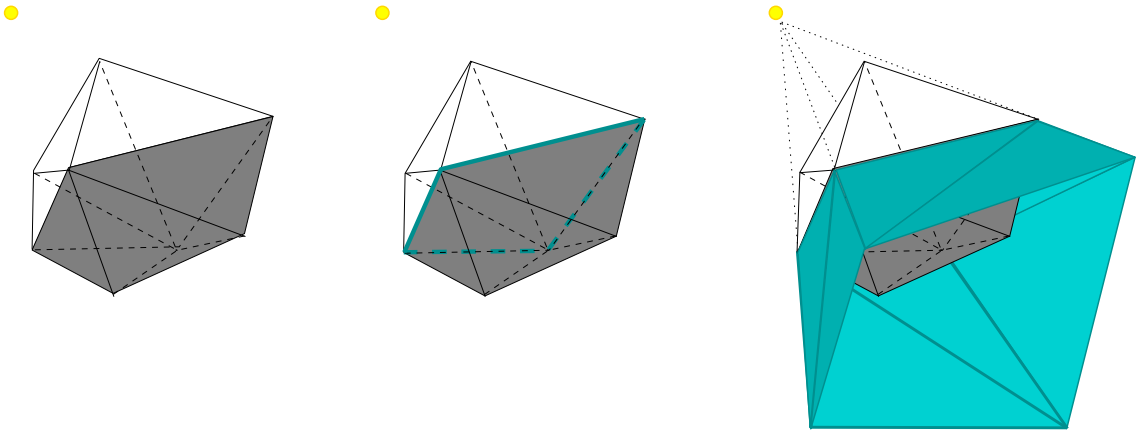

Figure 4.4: When silhouette edges are connected in a closed loop (middle), the side polygons of the shadow volume can be drawn as a triangle strip (right), reducing the number of vertices sent down to the graphics accelerator and conserving bandwidth. Manifold objects always have a closed silhouette.

the silhouette edge, two degenerate triangles that will form the side of the shadow volume using a copy of each silhouette vertex, and a last triangle back to a replica of the centroid. The vertex shader then projects the last three vertices in the strip the proper distance away from the object. So, in the end, the first triangle forms the light-side cap of the shadow volume (the *light cap*), the two middle triangles become the quadrilateral that forms the side, and the last triangle the dark-side cap (*dark cap*).

For every manifold object, the silhouette edges form a closed loop. This fact can be exploited by rendering the sides of the shadow volume as a triangle strip (Figure 4.4). Triangle strips are an optimized form of rendering. They cut down the number of vertices in the geometry by reusing vertices in adjacent polygons. However, creating triangle strips from arbitrary mesh geometry can be difficult. If the light source is

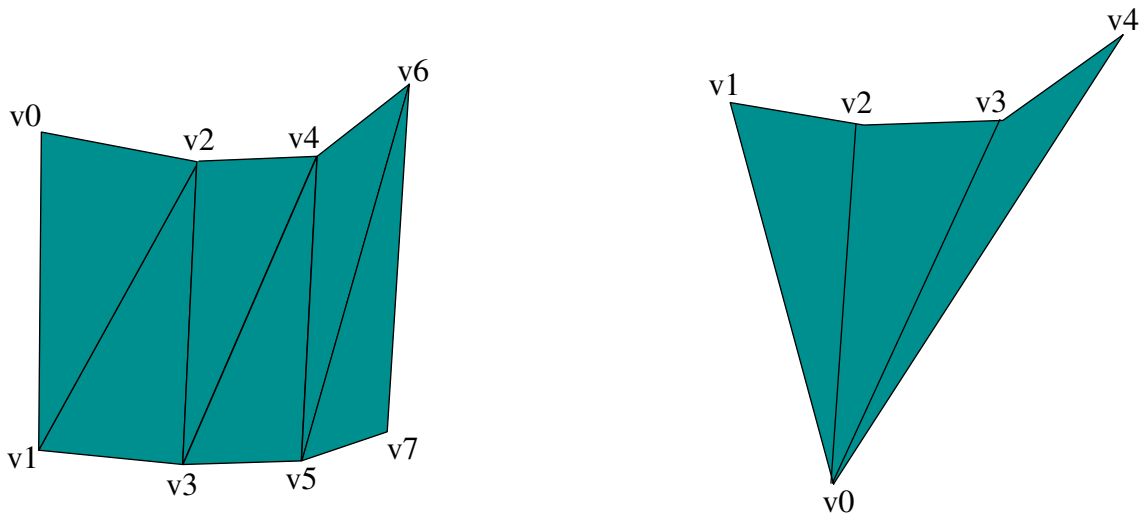

Figure 4.5: The difference between a triangle strip and a triangle fan. On the left, a triangle strip is used. Strips allow for shared vertices between adjacent triangles and reduce the number of vertices that are sent to the pipeline from 18 (each triangle has its own copy) down to 8. On the right, the side polygons are drawn as a fan, taking advantage of the parallel projection with directional lights. The fan further reduces the vertex count to 4.

infinitely far away, i. e., it is a directional light source, the shadow volume polygons can be further optimized. With directional lights, all vertices of the occluding object are projected along the same vector (Figure 4.2). Since this vector has infinite length, all points are projected to the *same* point at infinity. Basically, this has two advantages. First, the dark cap polygons degenerate into a single point and need not be drawn at all. This can significantly reduce the number of shadow volume polygons when using the zfail algorithm, where the shadow volumes have to be closed at both ends. Second, instead of rendering the side polygons of the shadow volume as a triangle strip, they can now be rendered as a triangle fan (Figure 4.5).

4.2 Silhouette Detection

As mentioned before (section 3.2), determining the silhouette of an occluding object reduces the number of shadow volume polygons and is thus an important step in shadow volume generation. Just like the generation of the shadow volume polygons, it is a starting point for performance optimizations. As such, a number of different silhouette detection algorithms have been developed in the past.

Basically, silhouette detection can take place either in object space, or in screen space. Screen space algorithms are usually based on 2D image recognition techniques and are useful if the main goal is the actual rendering of the silhouette, therefore we

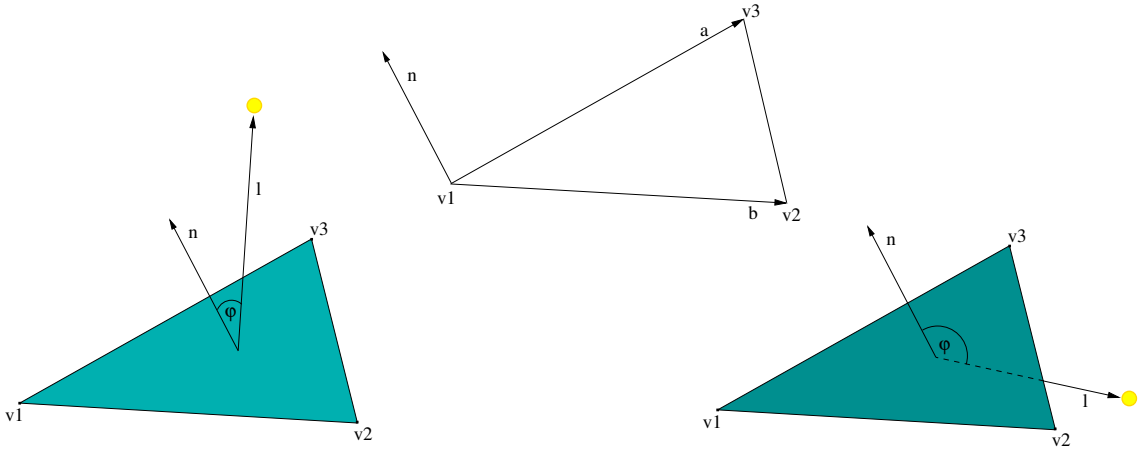

Figure 4.6: The angle between the triangle's surface normal and the incoming light vector determines the orientation of the triangle. The surface normal $\vec{n}$ is the binormal vector of the two plane vectors, $\vec{a}$ and $\vec{b}$, and calculated by building the cross product of these two vectors (top). On the lower left, the angle φ between $\vec{n}$ and the to-light vector $\vec{l}$ is lower than 90 degrees, and the dot product returns a value greater than zero. On the lower right, φ is greater than 90 degrees, and the dot product returns a value less than zero.

will concentrate on object space algorithms. These operate in three dimensions and usually produce a list of edges. For object space silhouette determination methods to work properly, the object must be a closed triangle mesh, i. e., there must be no "holes" that expose the object's interior, and every edge must be part of exactly two triangles. To avoid holes in the object's surface, all triangles must have a consistent winding, either clockwise or counterclockwise. Objects that satisfy these criteria are called *manifold*.

Silhouette edges are edges that connect a front-facing and a back-facing triangle (with respect to the light source). To determine the orientation of a triangle, the *dot product* of the triangle's surface normal with the vector formed reversing the incoming light ray is calculated. If the dot product results in a value greater than zero, the angle between the two vectors is smaller than 90 degrees, and the triangle faces towards the light. If the value is less than zero, the triangle faces away from the light (Figure 4.6). In summary, the inequality that has to be fulfilled for edges that are part of the silhouette can be expressed as

$$(\vec{n_1} \cdot \vec{l} > 0) \neq (\vec{n_2} \cdot \vec{l} > 0) \tag{4.1}$$

where $\vec{l}$ is the vector towards the light source, and $\vec{n_1}$ and $\vec{n_2}$ are the surface normals of the adjacent triangles.

There are many ways to calculate the set of silhouette edges, and most of them are CPU intensive. A very simple method is to loop through all edges and compare the orientation of the two adjacent triangles by *brute force* [71]. If they have opposite

orientation, the edge is added to the silhouette set. However, this method has linear complexity with respect to the number of triangles in the mesh. Moreover, every triangle is involved in three dot product computations. Obviously, this method is not very efficient. An optimization mentioned by Lander involves culling out edges that are inside planar polygons [68]. This means that the silhouette test is skipped for edges whose adjacent triangles lie in the same plane.

An algorithm called *edge buffer* was presented by Buchanan and Sousa [20]. Instead of iterating over all edges and testing adjacent triangles for their orientation, this method iterates over the triangles. Since the number of triangles is always less than the number of edges, this algorithm should run faster than brute force algorithms that perform testing for every edge. Iterating over a triangle list, the orientation of every triangle is tested. If the triangle is front-facing, the front-facing flag for each edge shared by this triangle is XOR'ed with 1. Likewise, for back-facing triangles, the back-facing flag for each edge is XOR'ed with 1. Upon completion, silhouette edges are those edges that have both their front-facing and back-facing flags set to 1. Interestingly, in practice the edge buffer algorithm runs slightly slower than the brute force method mentioned above. Hartner et al. ascribe this to the higher overhead cost because of the numerous XOR operations and edge table lookups [46].

Another technique is called *edge elimination*. This technique makes use of the fact that internal front-facing and back-facing edges are shared by exactly two triangles. The algorithm proceeds in a single step. It loops through all front-facing triangles and adds all edges that are part of these triangles to a silhouette edge set. If the edge was already contained in the set, however, it is removed. This is the case for all edges where both adjacent triangles are facing the light. After the loop terminates, the edge set contains all edges that connect a front-facing triangle with a back-facing triangle. Instead of looping through all front-facing eges, the algorithm can also iterate over the back-facing triangles and achieve the same result. For an efficient implementation, mesh data structures should be implemented using an indexed edge table, where every triangle is stored as indices into an array of edges. This algorithm is robust in that it also works for objects that have holes, i. e., objects with edges that are part of only one triangle. These "dangling" edges remain in the silhouette set and are ultimately considered part of the silhouette.

There are several silhouette determination methods that are based on *hierarchical culling*. Each of these algorithms requires to descend a hierarchical data structure at runtime. A method developed by Sander et al. involves creating a hierarchy of *normal cones* during a pre-process step (Figure 4.7). These can be used at runtime to cull large numbers of edges which are definitely not part of the silhouette. Any cones that intersect the light vector are discarded immediately, others that intersect the light vector have to be further analysed. The cones are organized in a hierarchical search tree. Whenever polygons within a certain node can be classified as all front-facing or all back-facing, this node can be discarded as not containing any silhouette edges. In addition to normal cones, several other variations of the hierarchical

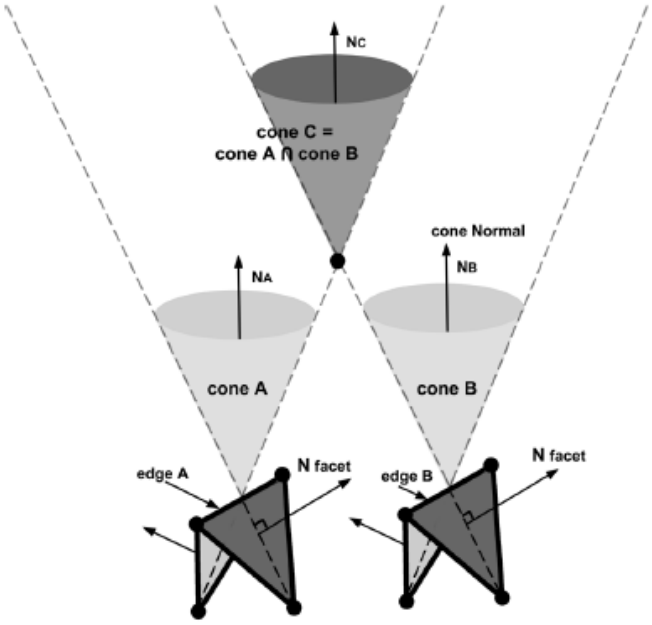

Figure 4.7: An example of a cone hierarchy built from cones that have similar dihedral angles, have similar cone normals, and are spatially close to each other. *(Image courtesy of A. Hartner, M. Hartner, E. Cohen, and B. Gooch, University of Utah)*

culling method exist. Hertzmann et al. use dual surface intersections [54], Pop et al. a wedge hierarchy [78].

Markosian et al. present a *probabilistic*, randomized search algorithm for rapidly finding silhouette edges [70]. This technique assumes that a silhouette always consists of a single closed curve, although there can be more than one silhouette curve on a surface. Furthermore, silhouette edges can belong to only one curve at a time. This does not necessarily mean that each vertex on the silhouette curve has only two incoming silhouette edges, although this would be preferable. Initially, a small number of edges based on their probability to be part of the silhouette is chosen, then tested to see whether the assumption is correct. The probability for being a silhouette edge correlates to the edge's *dihedral angle* (Figure 4.8). Edges that were part of the silhouette in the previous frames also have a high probability (*frame coherency*). Once these edges have been found, the algorithm starts by testing adjacent edges to see whether these are part of the silhouette as well. This process is repeated iteratively until the whole silhouette has been identified. Compared to many others, the advantage of this method is that it does not require any special data structures that are often expensive to set up. Therefore, it is suitable for use in dynamic scenes with animated meshes and/or light sources. A disadvantage with this technique is that some silhouette edges may be missed. This has to do with the termination criterion of the algorithm and how many edges are tested before it is assumed that there are no more silhouettes on the mesh. This value must be tuned according to the circumstances, which may involve the speed of the animation, the

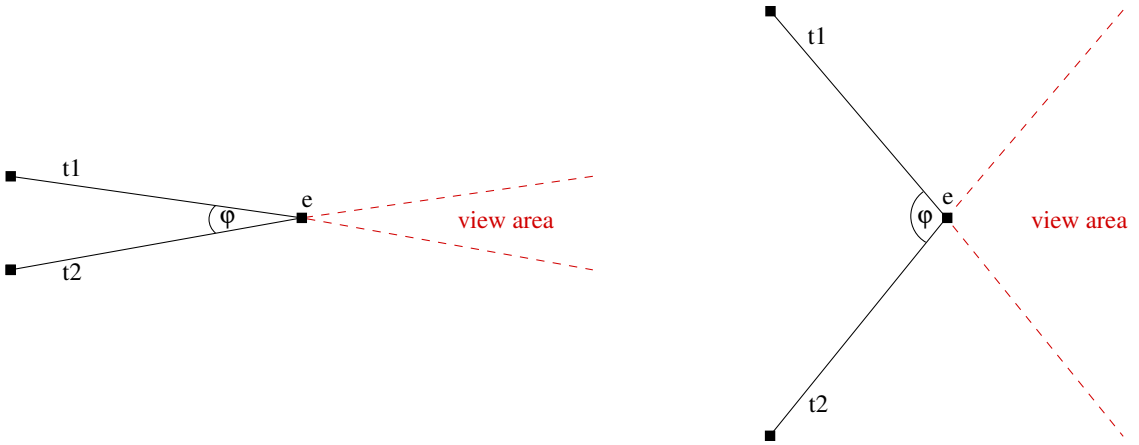

Figure 4.8: The dihedral angle of an edge and its correspondance to the view area. In this figure, the edge e connects the two triangles t_1 and t_2. Edges with a small dihedral angle φ have a higher probability of being part of the silhouette.

kind of meshes used, and how accurate the results should be. Compared to the brute-force algorithm, Markosian reports about a five times speedup.

A modified *Gauss map* can also be used to calculate the silhouette edges of a polygonal object [36, 9]. First, the object is placed at the origin of a bounding sphere (Gauss map). Each edge of the object now maps to an arc on the surface of the sphere. Silhouette edges can be extracted by intersecting the Gauss map with a plane. This plane passes through the origin of the sphere and is defined as perpendicular to the light vector. Arcs in the Gauss map that intersect with this plane correspond to silhouette edges of the object. A severe disadvantage with this method is that it only takes into account the direction to the light and not the distance. Hence, it will work only with orthogonal projections and cannot be used with omnidirectional point light sources.

A disadvantage common to all mentioned silhouette determination techniques is that they are quite CPU intensive. It is possible to move the process of determining silhouette edges to the vertex shader, using a technique similar to ATI's way of projecting shadow volumes on the fly (section 4.1). However, such methods are not applicable in this case because the silhouette data must be available on the host CPU for shadow volume construction. A possible bottleneck with CPU based methods is the nonsequential memory access [6]. Ordering mesh data structures (vertex, edge, and triangle lists) simultaneously in a way that allows for efficient caching is virtually impossible. Another area of concern is the linear complexity of most methods, resulting in an increasing number of silhouette tests for highly-detailed meshes that are comprised of lots of triangles.

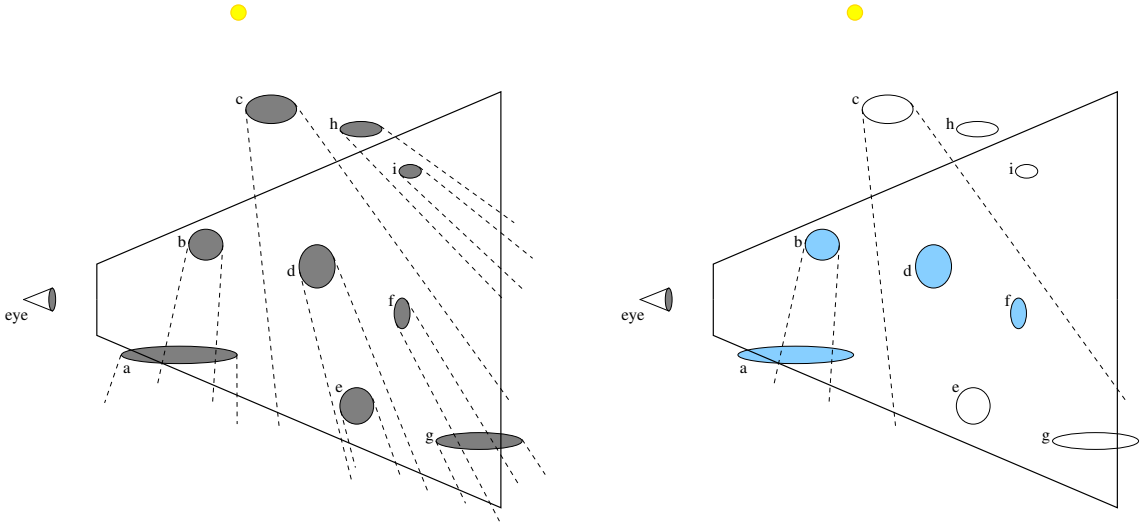

Figure 4.9: An illustration of shadow volume culling. On the left, no culling is done. A shadow volume is constructed for every object, resulting in 9 shadow volumes. On the right, a visibility test for the view point reveals 4 possible shadow receivers (objects a, b, d, f). Based on these results, a shadow volume is constructed only for objects b and c. Object i is not visible from the eye, hence no shadow volume is needed for object h. Objects d, e, f, and g lie totally within the shadow volume of object c.

4.3 Shadow Volume Culling

The standard shadow volume algorithm creates a shadow volume for every object in the scene. However, for large, complex scenes that are comprised of many objects, it is often not necessary to take all objects into account. Usually, there are many objects that do not contribute to the shadowing process, because they do not cast a shadow on any visible objects, or because they are completely enclosed in the shadow volume of another object (see Figure 4.9 for an illustration). The *CC shadow volumes* algorithm described in subsection 3.2.5 proposes a culling step to filter out unnecessary objects. This method, however, is not well suited for use in a complex scene. In this section, we will introduce a technique to perform shadow volume culling in complex scenes. Figure 4.10 shows shadow volume culling in a test scene, as compared to the standard shadow volumes illustrated in Figure 3.5.

The culling of shadow volumes usually proceeds in two steps. First, all objects that are visible from the eye are detected, yielding the set of *potential shadow receivers* (PSR). Similarly, in the second step all objects are detected that are visible from the light source's point of view. Taking the previously identified objects in PSR into account, these objects can be additionally limited to those that actually occlude any visible objects from the eye. The second step yields the set of *potential shadow casters* (PSC).

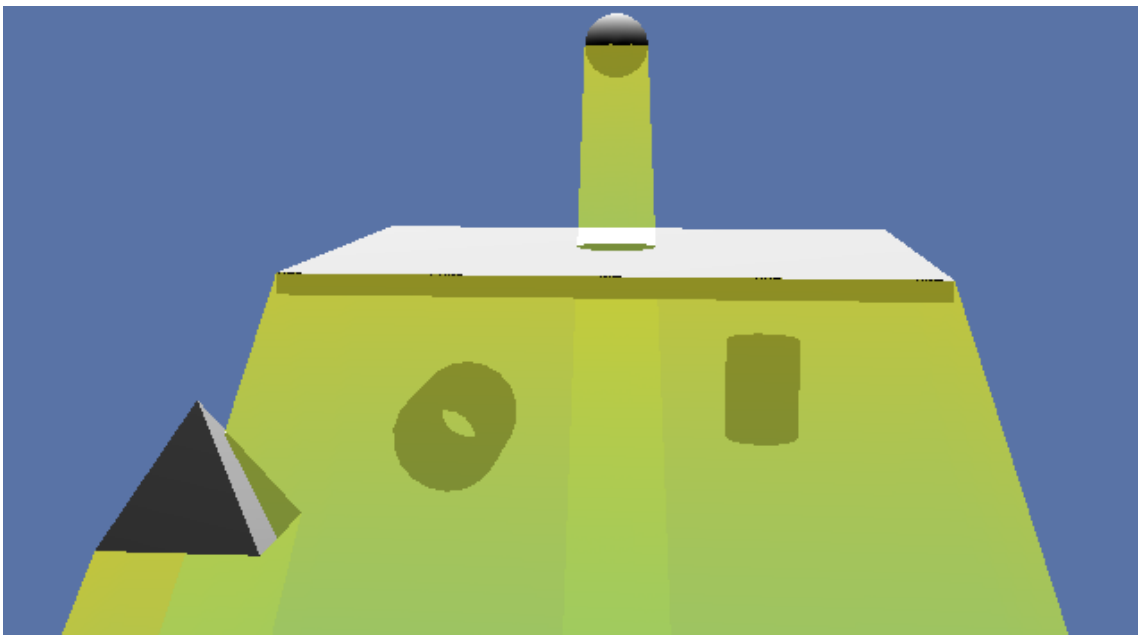

Figure 4.10: Shadow volume culling reduces the number of shadow volumes in the scene by discarding shadow casters that do not contribute to the final image.

4.3.1 Visibility in Light Space

In their paper, Lloyd et al. use image-based hardware occlusion queries to resolve the visibility [69]. To compute the set of potential shadow receivers (PSR), the scene is rendered from the viewpoint of the eye, creating a depth representation of the visible surface in the z buffer. Next, with z writes disabled, but still doing z testing, the bounding volume of every object in the scene is rendered with an occlusion query. If all fragments of a bounding volume fail the z test, the occlusion query returns zero. In this case, the object is completely occluded. Objects with a visible bounding volume are added to PSR. The potential shadow casters (PSC) are determined similarly, except that the scene is rendered from the viewpoint of the light source. However, this way PSC may contain objects that are visible to the light source, but do not occlude any objects in PSR. These unnecessary shadow casters can be removed by slightly modifying the algorithm. After the depth representation is created for the light, the objects in PSR are rendered into the stencil buffer, setting the stencil value to 1 if the z test fails. After this step, regions with a stencil value of 1 indicate areas of objects that receive shadows. Finally, when rendering the bounding volumes and performing occlusion queries, the stencil test is enabled as well, so that only those shadow casters are added to PSC that actually cover shadowed regions.

Though simple and easy to understand, this method for visibility detection has a distinct disadvantage: it is slow. Determination of the PSR requires two additional

rendering passes, plus another three per light source to compute the PSC. In our complex scenes, the cost of these additional passes surpasses any performance gains accomplished by having fewer shadow volumes. In complex urban scenes, performing shadow volume culling as described above results in an even lower performance than using standard shadow volumes (see chapter 5)

A smarter way is to use one of the fast visibility determination algorithms presented in section 2.5. The aim of visibility determination is usually to identify objects in the scene that are visible from the viewer's point of view, therefore preventing hidden objects from entering the rendering pipeline and wasting valuable hardware resources. With shadowing, the problem is identical, except that the visibility is not established from the viewer's position, but from the light source. The type of the light source dictates which visibility algorithms are applicable. For omnidirectional point light sources, a point-based approach must be used, whereas area light sources require a cell-based algorithm. Since this thesis deals only with point light sources and their hard shadows, we will concentrate on point-based visibility determination algorithms.

As explained in section 2.5, visibility culling consists of view-frustum culling, backface culling, and occlusion culling. View-frustum culling is used to filter out objects that do not lie within the light's region of influence (see subsection 4.3.3 for a more detailed explanation). Backface culling is performed automatically by the GPU when the lighting model is evaluated, because polygons facing backwards never receive any lighting from that specific light source. This leaves occlusion culling, which is very much applicable when doing shadows. Performing occlusion culling from the point of view of the light source, we can filter out all objects that are themselves occluded from the light and whose shadow does not contribute to the final image because it is totally enclosed within the shadow cast by another object. This way, the number of shadow volumes in the scene can be significantly reduced, saving both geometric as well as fill costs in the graphics hardware.

4.3.2 CHC Shadow Volume Culling

A well-suited technique to perform shadow volume culling is the coherent hierarchical culling algorithm described in subsection 2.5.1. Like the simple technique described at the beginning of this chapter, CHC uses hardware occlusion queries to resolve visibility. However, CHC traverses the scene graph in such a way that CPU stalls due to the unavailability of the query result are avoided. Ideally, any idle times are prevented altogether, resulting in a performance increase of 500 % and more. However, this speed improvement comes at the cost of a slightly less culling acuity, since CHC renders leaf nodes that were previously visible without waiting for the query result. Therefore, CHC can be slightly more conservative than the simple algorithm above, resulting in a higher number of shadow volumes in the scene. In

practice, the performance improvement of CHC over the exact algorithm turned out to more than compensate the higher geometry and fill costs.

CHC shadow volume culling can be easily integrated into the rendering framework. Similar to [69], we determine the eye-visible objects (PSR) in a first step. To do so, we invoke the CHC renderer, but instead of rendering visible objects into the framebuffer, they are stored in a list for later use. For the determination of the light-visible objects, it is important to use a *different* instance of the CHC renderer; otherwise, all frame-coherent visibility information for both the eye and the light view is lost. To include only objects in PSC that actually occlude any shadow-receiving object, all objects in PSR (as determined in the first step) are drawn into the stencil buffer, using the *eye*-instance of the CHC renderer. Although this introduces an additional rendering pass, it is very fast because it exploits the precalculated visibility information from the PSR pass. After initializing the stencil buffer, we use the *light* instance of the CHC renderer with stencil testing enabled to determine the objects in PSC. Again, instead of actually rendering visible objects into the framebuffer, we just store them in a list.

By following this approach, we exploit frame coherency in both PSR and PSC determination, which greatly improves the performance for the culling step. As mentioned in subsection 3.2.5 on page 53, the added overhead of the culling algorithm (lots of buffer writes that eat up any fill savings achieved by culling unnecessary shadow volumes) is one of the main bottlenecks of the CC shadow volume algorithm and the main reason why they don't achieve acceptable performance in complex scenes.

4.3.3 The Light Frustum

To perform visibility tests for the light source, a frustum must be constructed which represents the region that is influenced by the light source. The exact form of the frustum depends on the type of the light source. Directional light sources that are infinitely far away from the scene cast rays that are parallel to each other. Therefore, the frustum corresponds to an orthographic projection and must be chosen so that it encloses the whole scene. Point light sources can be either spot lights that influence a cone-like region in space, or omnidirectional lights that emit light in all directions. Spot light sources correspond directly to the camera analogy. Their opening angle φ is analogous to the field-of-view angle of a camera. The near plane is usually close to the light, and the far plane depends on its attenuation. Omnidirectional light sources are somewhat more complicated, because their field of view cannot be described with a single frustum. If they are located sufficiently far away from the scene (i. e., outside the bounding volume), they can be dealt with just like spot lights by fitting the frustum tightly around the scene. Omnidirectional light sources that are located within the bounding volume of the scene typically need to compute their visibility in several steps, e. g., in the front, back, top, bottom, left, and right

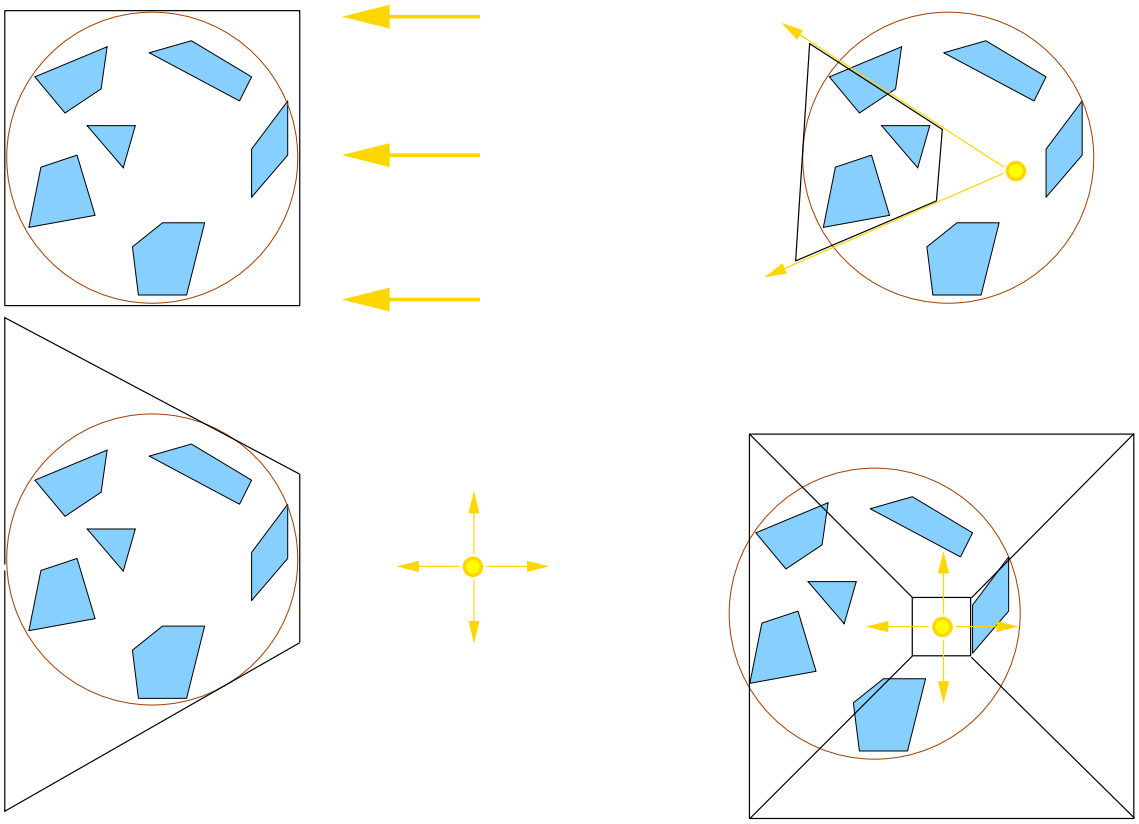

Figure 4.11: This figure illustrates how a frustum can be fit around the scene for different types of light sources. Top left: directional light sources correspond to orthographic projection, tightly fit around the bounding volume of the complete scene. Top right: spot lights are analogous to a viewer camera with perspective projection. They are parameterized by their position and field-of-view-angle φ. Bottom left: omnidirectional light sources that are positioned outside the bounding volume of the scene are similar to spot lights, however the frustum is fit around the bounding volume and φ is calculated accordingly. Bottom right: for omnidirectional light sources within the scene bounding volume, the visibility determination must proceed in six distinct steps. A frustum is constructed for each of the top, bottom, left, right, front, and back viewing directions.

directions, which can be expensive. Figure 4.11 illustrates the frusta for different types of light sources.

A light frustum that is constructed this way changes only if the scene changes, i. e., if the scene contains moving objects or if the position of the light source changes. It also usually encompasses a big part of the scene (for directional light sources and point light sources outside the bounding volume of the scene, it contains even the whole scene). Since visibility culling often operates in image space, precision may be lost for large scenes that contain lots of individual objects which are small compared to the size of the world. Objects are lost if they are too small to map to a single pixel in screen space, which may result in shadows popping in and out in consecutive frames, which in turn greatly impairs the realism of the scene. Given the fact that

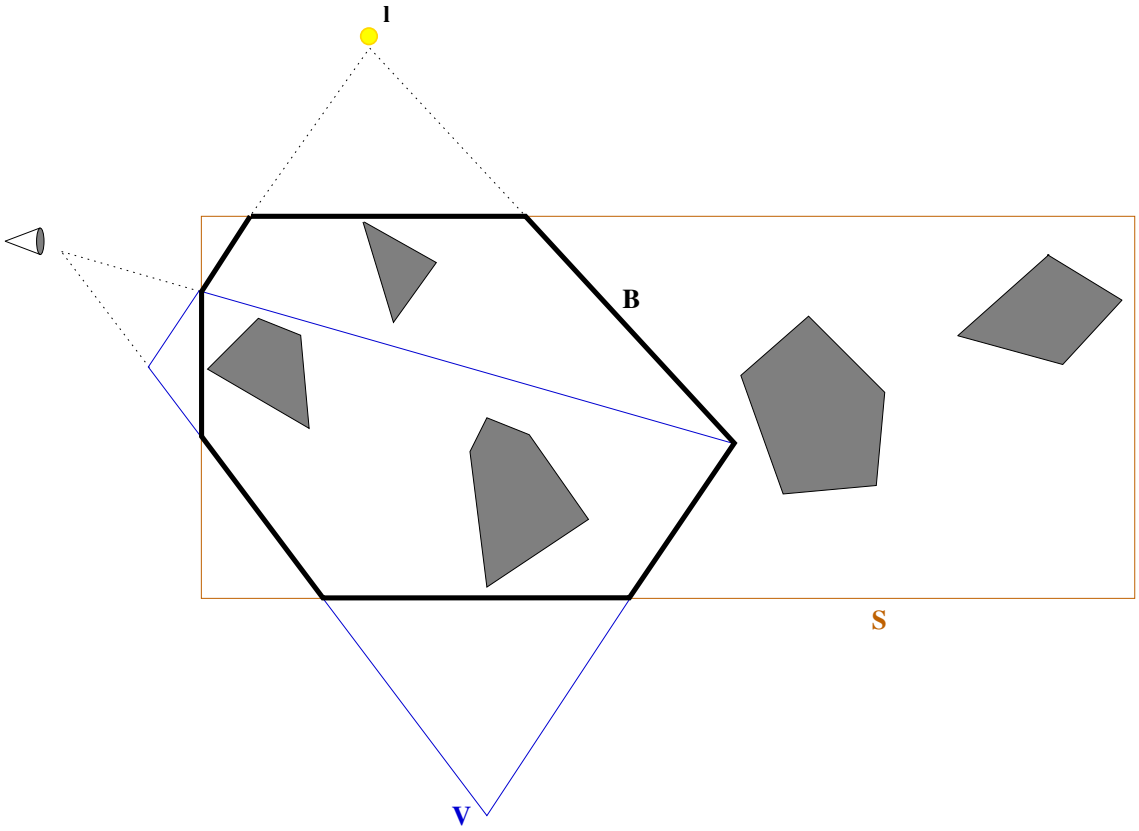

Figure 4.12: The construction of the intersection body B for a point light source. First, the convex hull of the view frustum V and the light position l is created. Then, the resulting body is clipped against the scene bounding volume B to avoid large frusta that cover vast regions of empty space and, consequently, result in a decreased precision when performing visibility culling.

the viewer can seldom see the whole scene, a lot of the light frustum is wasted on objects that are not even visible in the final image. Generally, it is better to limit the light frustum to those objects that actually contribute to the image.

The optimal way of focussing the light frustum to relevant parts of the scene is an issue that is also encountered in shadow mapping, which also profits from conserving image resolution. A way to focus the light view is described by Drettakis and Stamminger [29] and later refined by Wimmer et al. [94]. Here, the region of interest B is defined as the convex hull of the view frustum V and the light position l (for directional lights, this position is at infinity), clipped by the scene bounding volume S (Figure 4.12).

By clipping with S, large regions of empty space due to an over-dimensioned viewing frustum V are avoided. The resulting *intersection body* B defines a region that contains objects of interest with respect to light source visibility. Mathematically, the computation of B can be expressed as $B = (V + l) \cap S$, where $+$ denotes a union and $\cap$ an intersection operator. The resulting set operations are relatively easy to

implement, because all involved objects are simple convex polyhedra. Therefore, general purpose convex hull and polyhedra intersection algorithms can be avoided [94]. Listing 4.1 shows the construction of the intersection body in C-like pseudocode (from [94].

```
BRep tmp;   // a solid representation of an intersection object

Matrix4 invEyeProjView;
BRep sceneBV;     // the scene bounding volume (usually a box)

createSolidFrustum(tmp, invEyeProjView);

if (/* point light */)
  createConvexHullWithPoint(tmp, lightPosition);
else
  createConvexHullWithDirection(tmp, lightDirection);

clip(tmp, sceneBV);

return tmp;
```

Listing 4.1: Pseudo-code that shows the construction of the light's frustum.

`createSolidFrustum()` creates a solid view frustum representation by transforming the vertices of a centered 2-unit cube by the inverse of the combined view and projection matrix of the desired frustum.

`clip()` performs the intersection of two convex objects by clipping the first object against every plane of the second object and finally filling any holes with new polygons. Depending on the type of light source, the convex hull with the view frustum must be constructed in different ways.

`createConvexHullWithPoint()` creates the convex hull from a convex object and a point by removing all of the object's front-facing polygons with respect to the point, then closing the resulting hole with a triangle fan to the point. However, any algorithms that compute the convex hull for a set of points can be employed as well.

For directional lights, `createConvexHullWithDirection()` extrudes the light-facing polygons of the given convex body `tmp` a certain amount along the inverted light direction $-\vec{l}$. The surplus space in the resulting volume is finally clipped against the scene bounding volume.

Summarizing, focussing the light frustum to tightly fit the region of interest within the scene is an important part in light visibility determination. It is relatively easy

to implement and can be achieved without wasting too much processing power. Generally, the increased overhead due to the geometric intersection calculations is easily outweighed by the improved culling acuity.

4.4 Shadow Volume Clamping

Shadow volume clamping is originally part of the *CC shadow volumes* algorithm described in subsection 3.2.5. Let us examine how it fits in with complex scenes.

In the culling step, we removed unnecessary shadow volumes. A close examination of Figure 4.9 reveals that only small parts of the remaining shadow volumes actually cover shadow receivers. This observation indicates another disadvantage with standard shadow volumes: they extend to infinity. In screen space, the projected shadow volumes often cover a large number of pixels, which puts stress on the rasterizer engine. Moreover, the rendering of shadow volumes also modifies the stencil value, which further increases fill consumption[1]. Figure 4.9 suggests another way of improving standard shadow volumes. By tightly fitting the shadow volumes to regions in space that contain a shadow receiver (i. e., an object $\in$ PSR), the number of pixels that are covered by shadow volumes can be drastically decreased, which in turn results in a lower number of buffer writes and higher fill performance. The process of fitting the shadow volumes around the shadow receivers is called shadow volume *clamping*. See Figure 4.13 for an illustration. The use of shadow volume culling and clamping in a simple scene is shown in Figure 4.14.

Lloyd et al. suggest two different techniques to perform shadow volume clamping [69]. These are technically complementary, but both follow the same basic principle: they try to determine where interactions occur between a shadow volume and shadow receivers, then limit the shadow volume to these regions. The algorithm known as *continuous clamping* operates entirely on the host CPU and attempts to clamp shadow volumes to their z bounds in light space. *Discrete clamping* splits the viewing frustum by several planes into sections and utilizes the graphics hardware to find out which sections are occupied with shadow receivers. Like continuous clamping, it then limits the shadow volumes to these sections. In the following sections, we will examine each of the two techniques in more detail.

4.4.1 Continuous Clamping

Continuous clamping is performed entirely on the CPU and does not need any dedicated graphics hardware. Clamping is done by transforming every object $\in$

[1] On modern graphics hardware, render passes that do not affect the color buffer are often executed in an accelerated path, which results in a doubling of the nominal rasterizer speed on NVIDIA GeForce cards

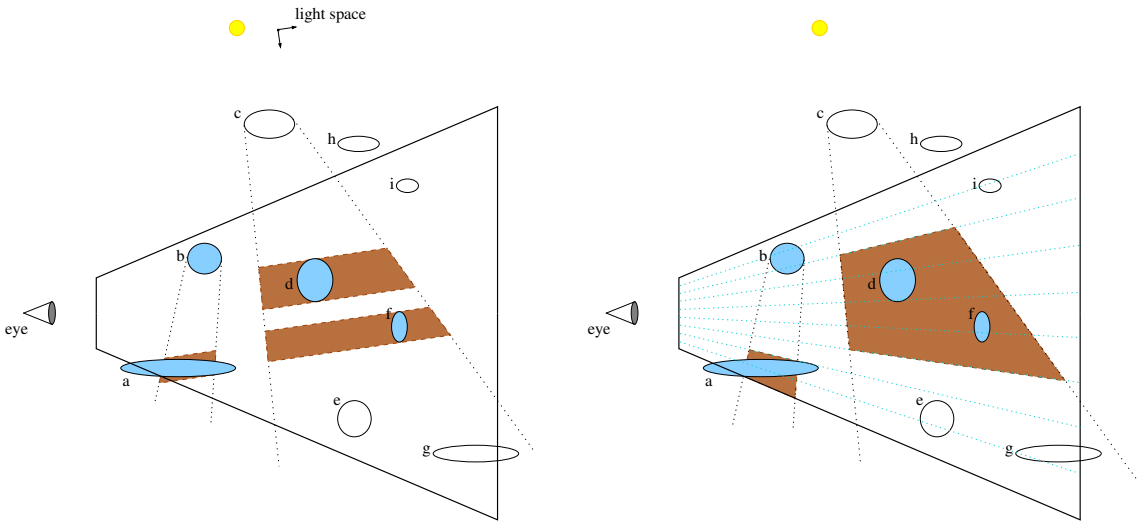

Figure 4.13: Two techniques to perform shadow volume clamping. Continuous clamping is illustrated on the left. All objects ∈ PSR are transformed into the light's coordinate system, and their z extents are used to clamp the shadow volumes around the shadow receivers. The right figure depicts discrete clamping. The view frustum is split into slicing planes, and shadow volumes are drawn only within slices that contain a shadow receiver. *(after Lloyd et al. [69])*

PSR into the light's coordinate system, ideally the same that was used in the culling step. Then, the regions where the shadow volumes need to be drawn correspond to the intervals that are occupied by shadow receivers in the z direction.

The algorithm proceeds as follows. First, all objects $o \in PSR \cup PSC$ are transformed into light space. In this space, an axis-aligned bounding box (AABB) is constructed for every object. To perform intersection tests, a technique similar to *dimension reduction* is utilized. The AABBs are projected onto the light's image plane, and the depth interval (z_{min}, z_{max}) of every AABB is stored. Then, two-dimensional bounding box overlap tests are performed on this plane to detect occupied z intervals according to the following rule: an object R possibly lies within the shadow of an object C if the projections of the bounding boxes on the light's image plane overlap and when z_{max} of R is greater than z_{min} of C. Naturally, objects may be mutually contained within each other's shadow volumes. Figure 4.15 gives an illustration.

The computation of occupied intervals after testing for overlaps in light space can be performed efficiently by keeping all objects in a sorted list and processing all shadow casters simultaneously. Every shadow caster stores a list of the intervals that are occupied by overlapping shadow receivers. The lists are initialized with the depth interval of the casting object itself to account for self-shadowing. Then, the shadow receivers are processed in order of increasing z_{min}, and the depth intervals of the shadow casters in which each receiver lies is updated. Since the shadow receivers are processed in order of increasing z_{min}, only the last interval in the list needs to be considered. There are three different ways in which the occupied intervals can be

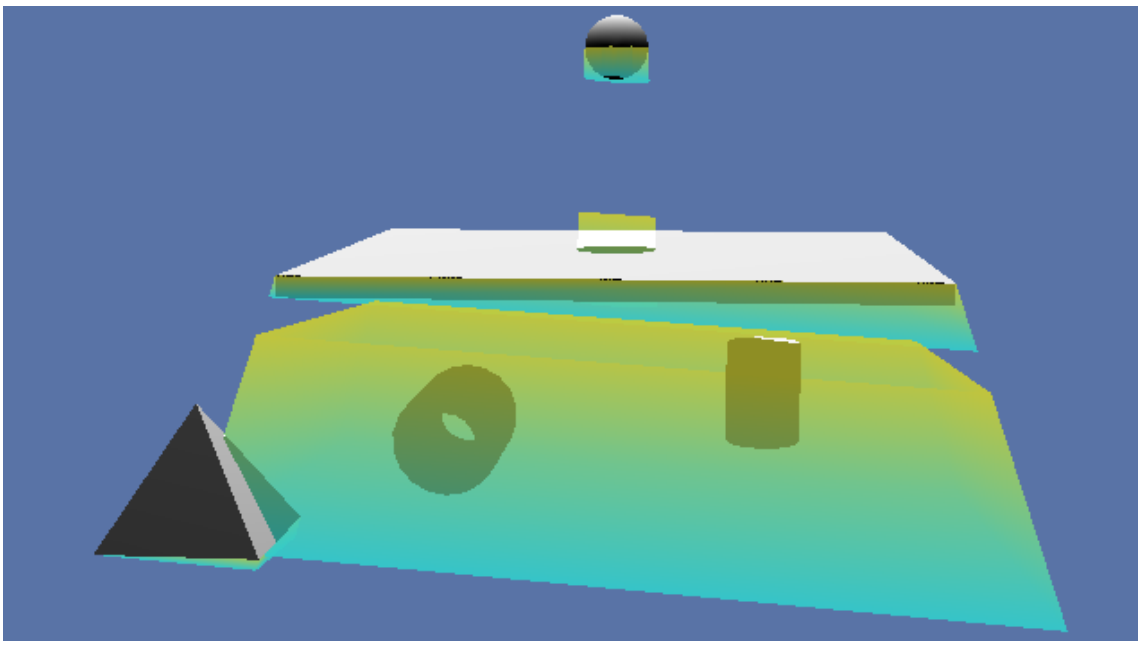

Figure 4.14: A scene that employs culling and (continuous) clamping to improve shadow volume performance.

updated:

1. The last interval in the list completely encloses the new interval. Nothing has to be done in this case.

2. The last interval in the list partly contains the new interval. In this case, z_{max} of the last interval is extended to include the new receiver.

3. The new interval is completely outside the last interval in the list. In this case, the new interval is appended to the back of the list.

Listing 4.2 gives an overview of the algorithm in pseudocode. In the pseudocode, `encounteredShadowCasters` is initially empty. Shadow casters are added only as they are encountered in the sorted list. This is to prevent incorrect shadow volume segments around objects that actually intersect the shadow caster in the light's view, but are closer to the light source than the shadow caster (the resulting shadows would be *above* the shadow casting object). `allObjects` is a list containing all objects of interest, i. e., $PSR \cup PSC$. This list is sorted by increasing z to facilitate the merging of new depth intervals. As the algorithm loops through `allObjects`, every object that is a shadow caster is added to `encounteredShadowCasters`. If the current object is a shadow receiver, overlap tests are performed for every encountered shadow caster. If the objects overlap, a new interval is added to the list of depth intervals of the shadow caster, according to the three possible cases explained above.

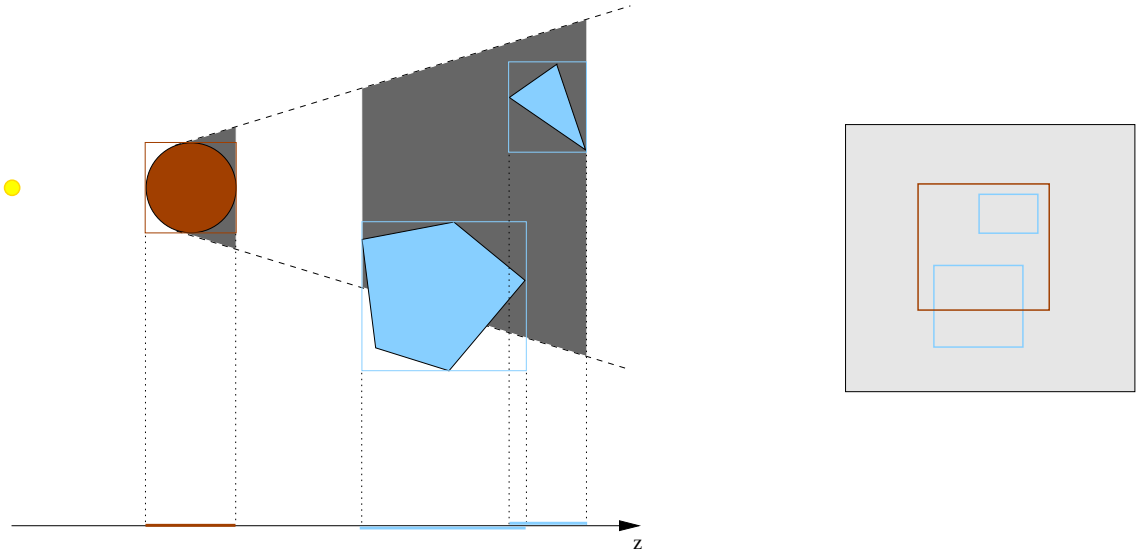

Figure 4.15: An illustration of continuous clamping. The left figure shows axis-aligned bounding boxes in light space. The AABBs are projected onto the light's image plane, where two-dimensional overlap tests are performed (right). *(after Lloyd et al. [69])*

For every depth interval in the list, a new shadow volume segment consisting of two sets of caps and an extra set of side polygons must be drawn in the stencil pass. If the gap between successive shadow volume segments is small in image space, the cost of rendering the additional geometry may outweigh the performance gain due to fill reduction. To determine whether or not the addition of a new interval in case 3 makes sense, a simple heuristic can be used [69]. Let V be the geometry processing cost of the new interval, and R the rasterization cost if the gap remains filled. Then, a new interval is only added if $V < R$. Otherwise, the interval is merged as in case 2, closing the gap. V and R can be computed according to equations 4.2 and 4.3.

$$V = (2C + 2S)v \tag{4.2}$$

$$R = Ar \tag{4.3}$$

C is the number of vertices in the shadow volume gap, S the number of vertices in the shadow volume silhouette, v the cost per vertex, A an estimate of the gap area in pixels, and r the rasterization cost per pixel. The exact values for r and v vary with different hardware accelerators and applications and can be determined empirically. An estimate of v can be obtained by rendering a "typical" set of shadow volumes at a very low resolution to eliminate rasterizer cost. Rendering this set at a high resolution and subtracting v yields r. This simple heuristic already provides reasonable results.

Intersection tests are expensive in scenes that contain a large number of objects. Without additional information about the scene and distribution of the objects,

```
List encounteredShadowCasters;
List allObjects;

encounteredShadowCasters.clear();
SortByZmin(allObjects);

for every object o in allObjects do
  if isShadowCaster(o) then
    encounteredShadowCasters.add(o);
  end
  if isShadowReceiver(o) then
    for every object c in encounteredShadowCasters do
      if overlap(o, c) then
        c.addInterval(o.zmin, o.zmax);
      end
    end
  end
end
```

Listing 4.2: Finding shadow volume intervals using sorted lists and sweep-and-prune intersection tests.

every object has to be tested against every other object. Dimension reduction techniques originate from the field of computational statistics, where often enormous amounts of data have to be processed. A similar problem exists with the quadratic complexity of bounding box intersection tests. Using dimension reduction to split the original three-dimensional problem into a two-dimensional xy-plane intersection problem and a separate test for the z axis simplifies the intersection tests themselves, but does not reduce the number of tests that have to be performed. For many scenes, it can be observed that the object distribution in the world does not change significantly between successive frames. Therefore, we can try to exploit *temporal coherence* to accelerate the tests, similar to the coherent hierarchical culling algorithm described in subsection 2.5.1. An efficient method that exploits temporal coherence by performing incremental computations is the *sweep-and-prune* algorithm developed by Cohen et al. [23]. It can easily be adapted for use with continuous shadow volume clamping [69].

Basically, the sweep-and-prune algorithm reduces the number of pairwise collision tests by eliminating the test for pairs that are far away. The algorithm proceeds as follows. First, axis-aligned bounding boxes are created. Every projected AABB consists of a minimum and a maximum coordinate in the x and y dimensions x_{min}, x_{max}, y_{min}, and y_{max}. These minima and maxima are maintained in two separate lists, one for each dimension. The lists are then individually sorted using *insertion sort* [64]. Additionally, for each bounding box pair, two separate overlap flags O_x

and O_y are maintained (one for each dimension). An AABB pair overlaps if and only if both flags are set. The key element of the algorithm is that the overlap flags are not computed in a separate step. Rather, they are maintained during sorting and modified whenever insertion sort performs a swap. The decision whether to toggle an overlap flag is based on whether the involved coordinate values both refer to bounding box minima, both refer to bounding box maxima, or one refers to a bounding box minimum and the other a maximum.

When a flag is toggled, the overlap status indicates one of three situations:

1. Both flags are now set. This means that the bounding boxes overlap in all three dimensions.
2. The bounding box pair had both flags set previously. In this case, the corresponding pair does no longer overlap.
3. The bounding box pair did not overlap in the previous frame, and does not overlap in the current frame either.

In every frame, the lists are re-sorted using insertion sort. When sorting is complete, an overlap test can be performed by simply comparing O_x and O_y for the corresponding pair. The list of depth intervals is maintained in a similar way, however there is no overlap flag for the z dimension.

The two main observations with sweep-and-prune are that the lists for the x and y dimensions are assumed to change little over successive frames, and the incremental operation of insertion sort which benefits significantly from almost-sorted lists. Thus, temporal coherence is exploited practically automatically.

To sum up, continuous clamping works entirely on the CPU, which keeps the graphics hardware free to perform other tasks. It can reuse the light coordinate system that was constructed in the culling step and relies on AABB intersection tests. Continuous clamping can in many cases generates tight fitting shadow volume segments. However, in scenes where the light direction is not aligned well with the orientation of the objects in the scene, the results may be poor. Continuous clamping fits shadow volumes precisely to the bounds of the shadow receivers in the z direction, but can overestimate the size of a shadow volume if only a small part of the shadow receiver lies in shadow because it always takes the entire extents of the receiver into consideration (Figure 4.16).

4.4.2 Discrete Clamping

Discrete clamping is an alternative technique that is in several ways orthogonal to the continuous clamping algorithm described above [69]. While the former operates entirely on the CPU, discrete clamping uses the graphics hardware to test for shadow receivers within discrete shadow volume intervals. To this end, the view frustum is

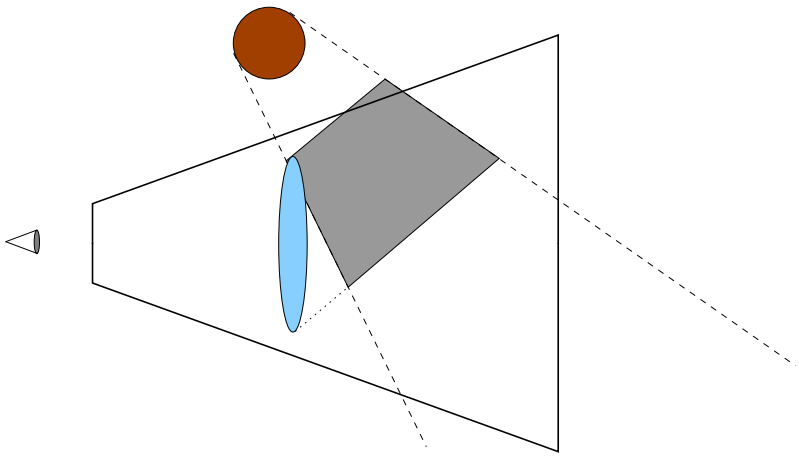

Figure 4.16: Continuous clamping can overestimate the shadow volume segments if only a small part of a receiver lies in shadow.

partitioned into slices by a set of similarly oriented planes. The discrete shadow volume intervals are determined by the intersection of the shadow volumes with the slicing planes. After the slicing planes have been constructed, discrete clamping uses image-space occlusion queries to test for slices that contain any shadow receiver, and shadow volumes are finally rasterized only in such slices.

As a matter of principle, any arbitrary plane can be used to partition the view frustum. A good choice for slicing planes are those that face the light source and pass through the viewpoint, splitting the image plane into strips of equal width [69]. Shadow volume segments created by such slicing planes cover an approximately equal area on the image plane, no matter how far away the shadow volume is from the viewpoint. Furthermore, with slicing planes oriented this way, the caps of the shadow volume segments need not be drawn, because they lie on a plane that passes through the viewpoint and do not affect any pixels. This choice of slicing planes is especially useful for point light sources that lie outside the view frustum.

Figure 4.17 illustrates how these planes can be created. Every slicing plane ϵ_i is defined by a point on the plane (in this case, the viewpoint $\mathbf{p}$) and its normal vector $\vec{n}_i$ pointing towards the plane's positive halfspace. We start with the top or bottom slicing plane, which coincides with the view frustum. The normal vector of this plane can be calculated using the eye's field-of-view angle. Furthermore, the angle between the normals in successive planes φ can be calculated by dividing the field-of-view angle by the number of desired planes. The normal vector of every subsequent plane $\vec{n}_{i+1}$ can then be computed by rotating n_i by ϕ.

To determine in which slices the shadow volumes have to be rasterized, Lloyd et al. describe a way to perform the discrete clamping computations in the light's view [69]. Every slice is rendered separately with a pair of user clip planes in back-to-front order. An occlusion query is issued for every shadow caster against a given slice. The depth test is set to `GREATER`, and the shadow caster is projected onto

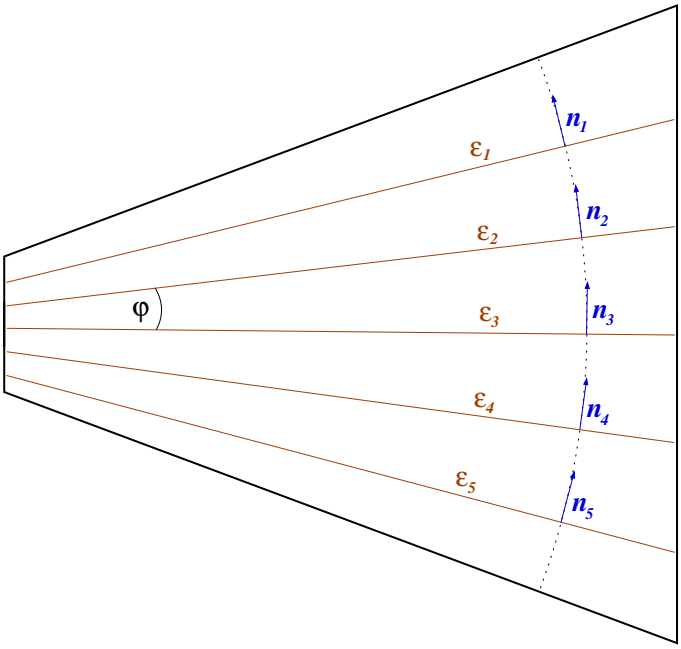

Figure 4.17: A view frustum partitioned by slicing planes. The planes face the light source, pass through the viewpoint and have the same angle φ, which is calculated as the field-fo-view angle of the eye divided by the number of slicing planes. To construct the slicing planes, a starting normal vector (e. g., , the normal of the bottom or top frustum plane) is continuously rotated by φ, yielding a normal $\vec{n_i}$ for every plane ϵ_i.

the bottom plane of the slice. This way, shadow receivers that lie within a slice but are not in shadow are discarded. If the occlusion query indicates that some pixels passed the test, the corresponding slice contains shadow receivers that are within the shadow volume of the occluder. Otherwise, the slice is empty or contains only objects that are not within this shadow volume. See Figure 4.18.

Projecting a shadow receiver onto the bottom plane of a given slice can be performed by the graphics hardware if a correct transformation matrix is supplied. The projection does not change the shape of the object in the light space, only the depth of the rasterized pixels. To setup the transformation matrix, the plane ϵ is represented by a quadrupel of homogenous coordinates (a, b, c, d). This quadrupel corresponds to the coefficients of the plane equation

$$ax + by + cz + d = 0 \tag{4.4}$$

a, b, and c are the x, y, and z coordinates of the plane normal $\vec{n}$, respectively. d is computed as the dot product of $\vec{n}$ and an arbitrary point on the plane (here, the viewpoint $\mathbf{p}$). The homogenous matrix $\mathbf{M}$ that projects the shadow caster onto ϵ through a center of projection $\mathbf{c} = (c_x, c_y, c_z, 1)$ can then be computed by

$$\mathbf{M} = \mathbf{c} \otimes \mathbf{p} - (\mathbf{p} \cdot \mathbf{c})\mathbf{I} \tag{4.5}$$

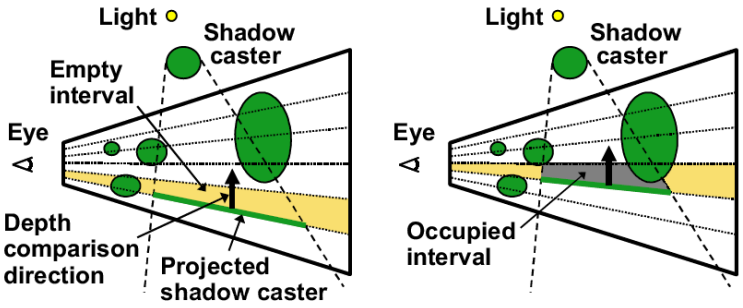

Figure 4.18: The identification of occupied slices. On the left, a shadow caster is tested against a slice, which is found empty. On the right, the following slice is determined occupied, so the shadow volume is rasterized within this interval. *(Image courtesy of Lloyd, Wendt, Gonvindaraju, and Manocha [69])*

where $\otimes$ denotes the outer product and $\mathbf{I}$ is the identity matrix. The center of projection $\mathbf{c}$ is the position of the light source. If $\mathbf{c}$ is located below the plane, $\mathbf{M}$ should be negated. This ensures that the fourth coordinate w is non-zero, which is necessary to prevent the point from being clipped by the graphics hardware. The projection matrix $\mathbf{M}$ correctly projects only objects that are closer to the plane than the light source. For objects that are farther away from the plane than the light source, the edges of their shadow volume must be extruded to infinity from the last plane onto which the object can be correctly projected.

This way of detecting occupied slices requires every shadow caster to be rendered for every slicing plane, because for every slice, every shadow caster must be projected onto the bottom plane. In the following, we describe an improved technique that utilizes the stencil buffer and needs every shadow caster to be rendered only once for all slices. With this technique, the idea is to render every individual shadow caster once into the stencil buffer. Then, in the light's plane, pixels with a stencil value other than zero correspond to regions that are in the shadow of this shadow caster. When testing the slices with occlusion queries, the stencil test is simply activated as well. With this improvement, the projection of the shadow caster is no longer necessary. With this improvement, the discrete clamping algorithm is outlined in the pseudocode in Listing 4.3.

As with all techniques operating in image space, the use of hardware occlusion queries in discrete clamping may lead to sampling errors due to the limited screen resolution (undersampling) and limited precision of the z buffer. These errors can cause sparsely occupied intervals to be incorrectly classified as empty, which results in holes in the shadows. A technique developed by Wonka and Schmalstieg [96] can be used to alleviate this problem by bloating the geometric primitives sufficiently to avoid sampling errors. Discrete clamping can be expressed as an interference computation problem between shadow casters and the objects within the view frustum slices. Lloyd et al. adapt the conservative overlap tests described in [40] to perform

```
Vector normal = viewer.frustum.top.normal;
Vector viewPoint = viewer.position;
Angle planeDifferenceAngle = viewFrustum.fov_y / numberOfSlices;

for every shadow caster c do
/* setup stencil for writing */
Render(c);
/* setup stencil for testing */
  for plane = 1 to numberOfPlanes do
    Vector newNormal = Rotate(normal, planeDifferenceAngle);
    Plane plane1 = CreatePlane(normal, viewPoint);
    Plane plane2 = CreatePlane(newNormal, viewPoint);
    /* set clipping planes plane1, plane2 */
    IssueOcclusionQuery();
    RenderScene();
    if (QueryResult() > queryThreshold)
      c.addInterval(plane1, plane2);
    normal = newNormal;
  end
end
```

Listing 4.3: A basic outline for constructing shadow volume intervals with discrete clamping.

reliable clamping by first constructing and bloating the solid representations of the shadow volumes [69]. With these, overlap tests can be performed with the bloated objects that are enclosed within the slices. This algorithm completely eliminates the image-precision artifacts, but is significantly slower.

4.4.3 Rendering Clamped Shadow Volumes

Shadow volume clamping produces shadow volumes that consist of individual segments that are not connected to each other. Consequently, such shadow volumes can no longer be constructed by simply morphing silhouette edges into quads and extruding the duplicated vertices to infinity. None of the techniques that construct shadow volumes directly in hardware (as explained in section 4.1 work with clamped volumes. In this section, we will first explore a general way to construct segmented shadow volumes on the host CPU. Then we will take a look at how to implement this technique on the graphics hardware.

Let us begin with continuous clamping. Continuous clamping projects the objects in the scene into the light's point of view. In this space, the z extents of the axis-aligned bounding boxes correspond to intervals that have to be occupied by shadow volumes. From a silhouette vertex' point of view, every silhouette vertex is projected onto two

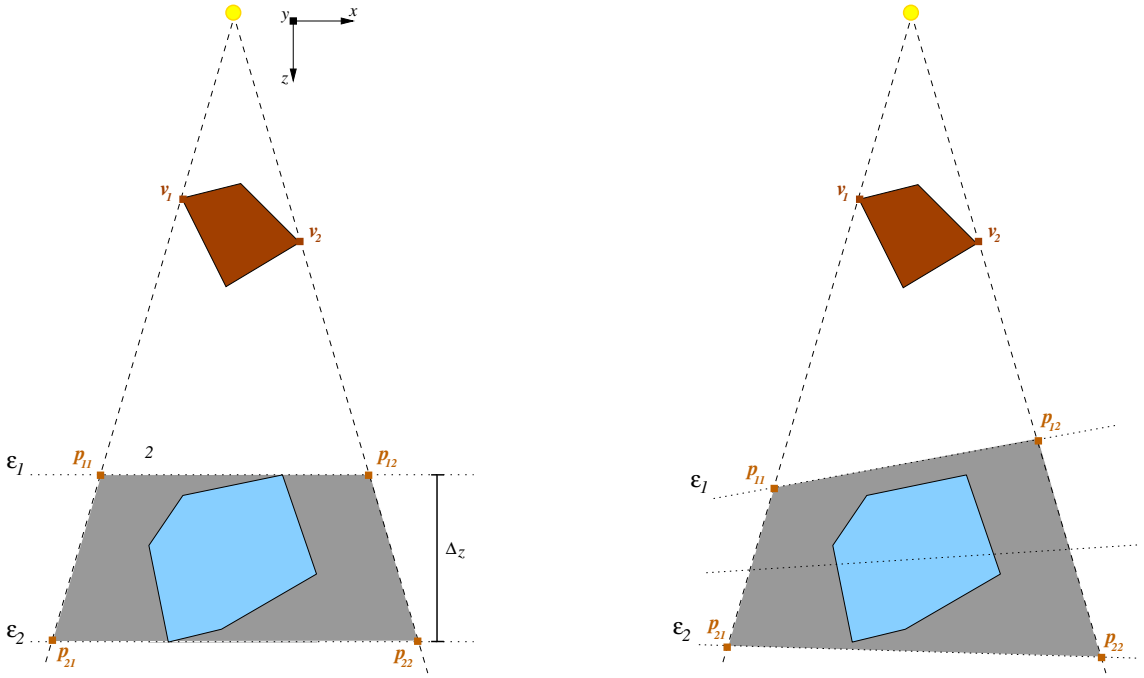

Figure 4.19: This figure depicts how shadow volume vertices are created for continuous clamping (left) and discrete clamping (right). The corner points v_1 and v_2 of the shadow receiver (dark brown) are projected onto the top (ϵ_1) and bottom (ϵ_2) planes of the shadow volume segment, yielding the projected points p_{11} and p_{12} on the top (forming the segment's *near cap*) and p_{21} and p_{22} on the bottom (forming the segment's *far cap*).

planes. The first is a plane that is parallel to the light's xy plane and passes through the point on the receiver that is nearest to the light (the *top plane*). Similarly, the second plane is also parallel to the light's xy plane, but passes through the farthest point on the receiver (the *bottom plane*). Due to the nature of the continuous clamping technique, all projection planes are parallel to each other. The left of Figure 4.19 illustrates the projections and the vertices that are created.

As hinted by Figure 4.19, the creation of the shadow volume segments with discrete clamping is very similar. Like with continuous clamping, all silhouette vertices are projected onto the top and bottom planes of an occupied segment. Only the planes themselves are different, because they are no longer parallel, but rather defined by the viewpoint through which they all pass and their normal vector, which is obtained by rotating the normal vector of the view frustum's top or bottom plane.

Obviously, to create shadow volume segments, all we have to do is supply the top and bottom planes of every occupied interval, then project silhouette and cap vertices of the shadow caster onto these planes. For point light sources, the vertices are projected through a center point $\mathbf{c}$, for directional lights all vertices are projected along the same projection vector $\vec{l}$. The nice thing with this observation is that it works for both continuous and discrete clamping, even though both techniques are

otherwise completely different. The process of projecting the vertices can easily be performed on the host CPU. The shadow volume segments obtained by continuous clamping are view-independent, therefore they need only be recomputed when the objects' positions relative to each other or the light source changes. This is not the case with discrete clamping, where the boundary planes of the shadow volume segments are defined by the viewpoint. Hence, whenever the viewpoint moves, the segments have to be recomputed. In larger scenes that consist of many objects, this can be expensive and decrease performance, especially if the CPU has already a lot to do.

A better solution relieves the CPU and relies on programmable graphics hardware to perform the projections. We pass the plane coefficients for the segment's lower and upper plane to the vertex shader. For point light sources, the light position is supplied as a vector with a w coordinate of 1, whereas the direction of directional light sources has a w component of 0. Then we let the graphics hardware project the shadow volume segment. The application simply renders the untransformed shadow volume vertices (usually, far cap vertices are marked by a w value of zero) once for every segment and passes the plane parameters on to the vertex shader. Listing 4.4 shows the pseudocode for the rendering part in the application. See section A.4 for a sample vertex shader program, written in NVIDIA's shading language Cg.

4.5 Putting the pieces together

Not all of the optimization techniques presented in the above sections can be used together simultaneously. For example, only one silhouette detection technique is implemented in a typical application. This section explains which techniques were actually chosen in the implementation of this thesis, as well as describing the idea behind the selection. In the next chapter (chapter 5), we will examine the performance improvement gained by these optimizations. Figure 4.20 gives a basic outline of the tasks that are performed by the application

Creating shadow volumes only for the shadow caster's silhouette edges may be the single most important optimization of all. Depending on the complexity of the shadow caster, it can greatly reduce the number of shadow volume polygons as well as the depth complexity of the shadow volumes, which relieves both the geometry stage as well as the rasterizer stage. On the other hand, CPU load increases because of the additional overhead of finding the silhouette. Basically, any of the silhouette detection techniques presented in section 4.2 can be chosen. Due to the data structures in the rendering engine, the edge elimination method was selected. This method also has the advantage that it is sufficient to compute intermediate data (normal vectors) once during the runtime of the application. In principle, the edge elimination algorithm loops through all polygons of the object and tests their orientation with respect to the light source. For these tests, the normal vector for every

```
List shadowVolumes;

BindShader(projectShader);

Vector4 lightVector;
if (isPointLight(l)) do
  lightVector = l.position;
  lightVector.w = 1;
else
  lightVector = l.direction;
  lightVector.w = 0;
endif
SetShaderParameter(lightVector);

for every volume v in shadowVolumes do
  List svSegments = v.segments;
  for every segment s in svSegments do
    SetShaderParameter(s.lower);
    SetShaderParameter(s.upper);
    Render(s);
  end
end
```

Listing 4.4: The rendering of shadow volume segments using a shader program that takes care of the segments' final projection.

polygon must be available. However, since the objects don't change their shape, the polygon normal must be calculated only once and can be reused later. For polygons that face towards the light source, all edges are stored in a list if they are not already contained in the list. Otherwise, they are removed. Therefore, all edges that remain in the list are either silhouette edges or "dangling" edges that are part of only one triangle and must be considered as well for correct shadow volumes. The orientation tests have to be done whenever the relative position of the object and the light source changes. For purely static scenes, the silhouette is determined only in the first frame. The edge elimination technique is exact, fast and rather simple to implement. However, for correct results, the objects need to be manifold, which is not the case with the city scene (Figure 4.1). Appendix A describes in detail how this problem was solved.

Chapter 3 describes two different algorithms for stencil shadow volumes, namely the zpass and zfail algorithms. Performance-wise, the zpass algorithm is preferrable, because it rasterizers less geometry. However, zpass is not robust and produces incorrect results when parts of the shadow volume are clipped against view frustum planes. The zp+ algorithm (subsection 3.2.1) improves zpass by initializing the

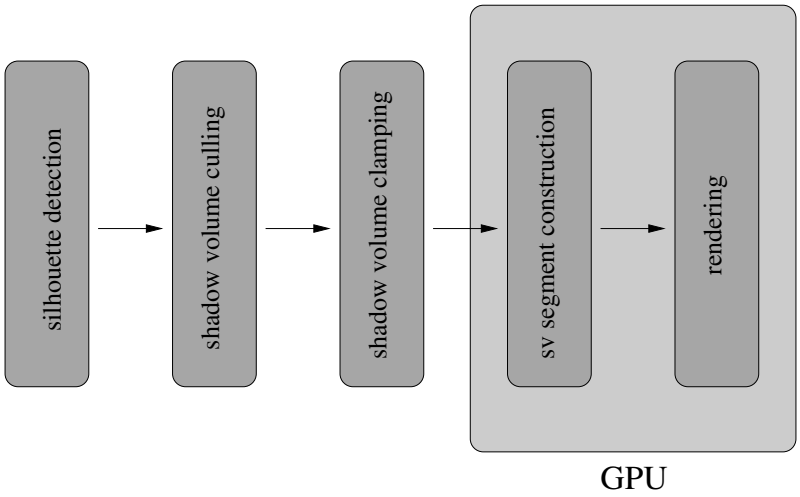

Figure 4.20: This figure illustrates the tasks that have to be performed for shadowing.

stencil buffer correctly for clipped shadow volumes. ZP+ does not need any special hardware extensions and requires only an additional render pass to rasterize the shadow caster with a special transformation matrix. Though mathematically exact, ZP+ still suffers from minor artifacts due to numeric errors in the transformation. In most cases, these can be safely ignored because they are nearly invisible. The nice thing about zp+ is that it is very easy to integrate into existing applications, requiring only a few lines of additional code. In our implementation, zp+ replaces the zfail algorithm because of its reduction of shadow volume polygons. See 5.3 for a comparison of the zpass, zfail and zp+ algorithms in the city scene.

To reduce the number of shadow casters, section 4.3 introduces a way to cull irrelevant shadow casters that do not contribute to the shadowing process. Culling is normally performed every frame. It can be expensive because the visibility computations can involve several rendering passes, depending on the chosen algorithm. For the city scene, both the exact algorithm as well as coherent hierarchical culling were implemented. The former is slower, whereas the latter is more conservative and does not filter out all irrelevant shadow casters.

Shadow volume clamping limits shadow volumes to those regions in space that actually contain shadow receivers. Both techniques have their advantages and disadvantages, their use should depend on the application. Applications where the CPU is already the bottleneck will not benefit from continuous clamping, and the same holds for applications that are GPU-bound and use discrete clamping. Theoretically, both techniques could be used together, using discrete clamping to further refine the results obtained from continuous clamping. However, in practice, this is not feasible because it involves complex geometric calculations with a large number of objects.

Finally, the culling and clamping results are used to construct shadow volume segments. To create the shadow volume segments, a shader program is used that

projects silhouette vertices into the occupied intervals on the fly. This relieves the CPU from performing a large number of geometric projections. With recent graphics hardware, this is almost as fast as using standard shadow volume extrusion (aside from the additional geometry due to the segment caps).

5 Discussion and Comparison

In this section, we will examine and compare the performance with the optimization techniques mentioned in the previous chapter. In particular, we will examine the performance gains of each technique in comparison to the standard stencil shadow volume algorithm. To measure the performance, NVIDIA's *NVPerfKit* tool was used, which gives access to low-level performance counters in the hardware and driver (see Appendix A). We use the city scene (Figure 1.3, 4.1) as our main test environment. The city scene comprises 1,033,002 individual triangles and 872,715 vertices and is organized as a scene graph with 17.420 individual objects, not counting object instancing for the smaller city objects (traffic lights, trees, ...). Our second test scene is the "light" version of the powerplant model that was released for public use by the University of North Carolina [1]. The original model is made up of 12,748,510 vertices, our light version is reduced to 3,067,884 triangles and 1,998,330 vertices in 18,427 distinct objects (Figure 5.1). The graphics hardware used for these performance tests is a mid-range NVIDIA GeForce 6600 GT with a clock of 500 MHz and 128 MB DDR memory.

Unless stated otherwise, all of the following performance plots were taken at a screen resolution of 800x600 pixels, with two light sources (a point light and a head light, which does not cast visible shadows). For the sake of comparison, Figure 5.2 shows the performance results for the city scene without shadows. The techniques that are compared in this section are:

- Standard shadow volumes limited to silhouette edges
- Shadow volumes with different culling algorithms
- Shadow volumes with culling and clamping

As it turns out, any performance measurement for standard shadow volume without any optimizations at all is practically useless, even if they are created on-the-fly in a vertex shader (section 4.1), because a single frame takes several seconds to complete. This is mainly due to the high count of geometric primitives (without silhouette determination, every triangle creates at least three additional shadow volume polygons) as well as the high overdraw which stresses the rasterizer.

Rendering shadow volume polygons only at silhouette edges is the single most important optimization technique to enhance performance. Identifying silhouette edges

[1] http://cs.unc.edu/~geom/Powerplant

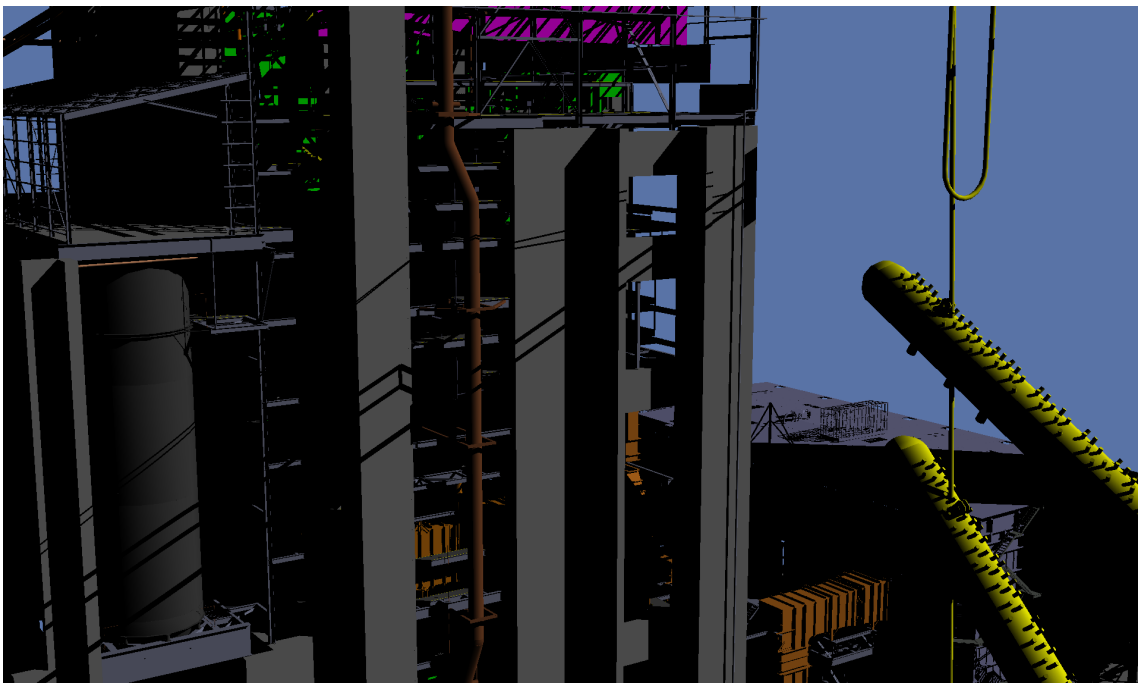

Figure 5.1: A view of the PowerPlant model used in the performance comparisons. The "light" version consists of 3.067.884 triangles and 1.998.330 vertices and uses the sections 2-14, 17, 18, and 21 of the original model.

involves a slight computational overhead but greatly reduces the number of shadow volume polygons. Furthermore, these computations have to be performed only when the relative position of the shadow caster and the light source has changed. Since the city (and the power plant as well) are static, it is sufficient to perform silhouette detection only at the first frame. The fact that the scene is static can be exploited in shadow volume creation as well, because the shadow volumes do not change. Even if they are constructed on the CPU, the overhead can be neglected because it occurs only in the first frame. Creating shadow volumes at silhouette edges only results in a performance boost to approximately 2 frames per second.

The zfail algorithm is a robust variation of zpass, however this robustness comes at the cost of a higher number of primitives that have to be rendered. In the city scene, zfail is about twice as slow as zpass (Figure 5.3). The use of the zp+ algorithm is almost as robust as zpass and for most viewpoints just as fast. zp+ involves only one additional rendering pass, in which the stencil buffer is initialized with the fragment counts of the objects that lie within the pyramid formed of the viewer's near plane and the light position. Since this pyramid is empty in most frames, it is vital to perform hierarchical view frustum culling in the zp+ pass to avoid traversing the complete scene graph and sending all primitives into the rendering pipeline. If the light source is very close to the near plane, and the near rectangle is large, the light pyramid is askew. In this case, zp+ may generate visible artifacts, because the

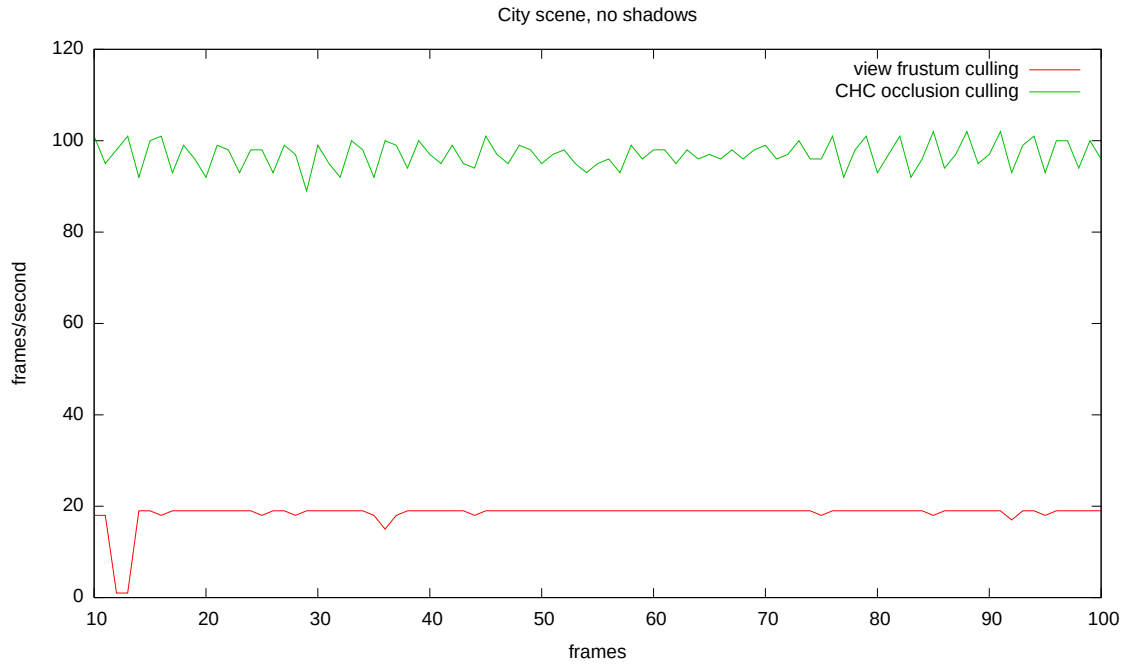

Figure 5.2: Performance in the city scene without rendering any shadow volumes, from a static viewpoint with moderate occlusion.

light projection and the viewer projection are not exactly aligned, due to numerical errors. In practice, the zp+ algorithm is usually perceived just as fast as the zpass technique, because HVFC takes care that only the necessary objects (those whose shadow volume is clipped) are sent down the rendering pipeline in the zp+ step. See Figure 5.3 for a performance comparison of the three algorithms.

The *CC shadow volumes* technique (section 4.3) describes two interesting possibilities for optimizing shadow volumes for complex scenes: shadow volume *culling* and *clamping*. Figure 5.4 shows performance plots for CC shadow volumes in the city scene.

Shadow volume culling is important for complex scenes because it limits the number of shadow casters and, consequently, the number of shadow volumes to those that actually generate visible shadows in the final image. A reduction of shadow volumes means a reduction of geometry and, more important, a reduced depth complexity and decreased fill requirements. However, the culling step itself can be expensive, because depending on the technique used, it may require several additional render passes, which in turn eat up any fill savings gained by reducing the number of shadow volumes. According to Figure 5.4, performing shadow volume culling in the city scene actually results in a performance drop of about 30 percent.

To recap, shadow volume culling basically involves two steps. In the first, the set of possible shadow receivers (PSR) is determined by performing visibility tests from

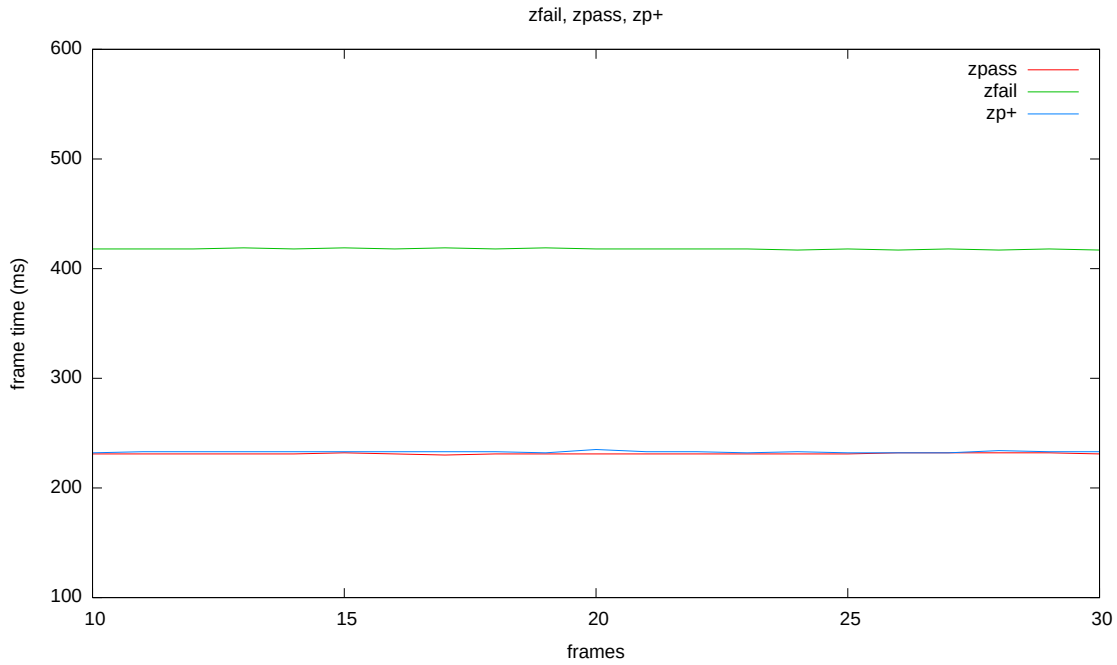

Figure 5.3: The city scene rendered with standard shadow volumes, but restricted to silhouette edges. The plot depicts frame times for each of the zfail, zpass, and zp+ algorithms. For this view, zpass is about twice as fast as zfail because of the additional cap geometry needed for zfail.

the viewpoint. The second step is similar to the first, but performs visibility tests from the viewpoint of the light to determine the set of possible shadow casters (PSR) as those objects that are visible from the light. The exact determination technique for PSC and PSR as described by Lloyd et al. involves $2 + 3i$ render passes, where i denotes the number of light sources in the scene. Even a single light source requires 5 additional passes, which is a lot and expensive for complex scenes with hundreds of thousands of triangles.

The first culling step - determining the PSR - is clearly view-dependant, because the PSR vary from viewpoint to viewpoint. In contrast, the visibility from the light source changes only if the position of scene objects and the light source changes, which is not the case for static scenes. However, the computation of the PSC depends on the PSR because the PSC should contain only objects that actually occlude any object $\in$ PSR, and since PSR changes with the viewpoint, PSC changes as well. A possible way to reduce the number of rendering passes in the culling step is by removing the dependency of PSC on PSR. Then, it is necessary to perform the culling step only if the scene changes, but the set of PSC will contain some intrinsically irrelevant shadow casters that do not occlude any object in PSR. In the city scene, this simplified culling technique noticeably outperforms the exact culling technique, even though more shadow volumes are rasterized. Figure 5.5 compares the performance for different shadow volume culling algorithms.

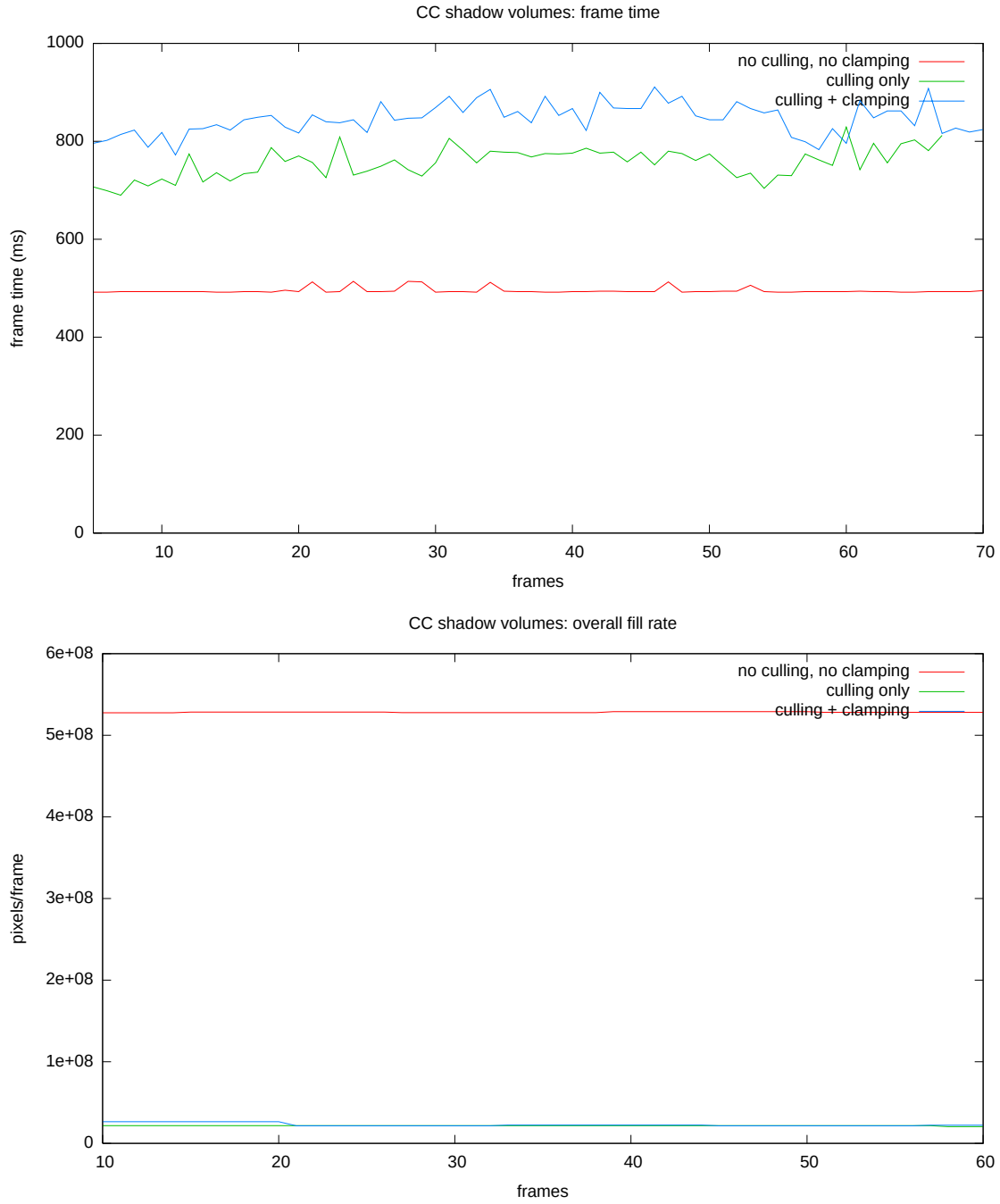

Figure 5.4: Performance plots for CC shadow volumes [69]. As we can see in the bottom plot, culling and clamping indeed cuts fill costs. However, in complex scenes, the added overhead of the culling and clamping steps (culling has to be performed on the CPU) results in an actually lower performance than standard shadow volumes.

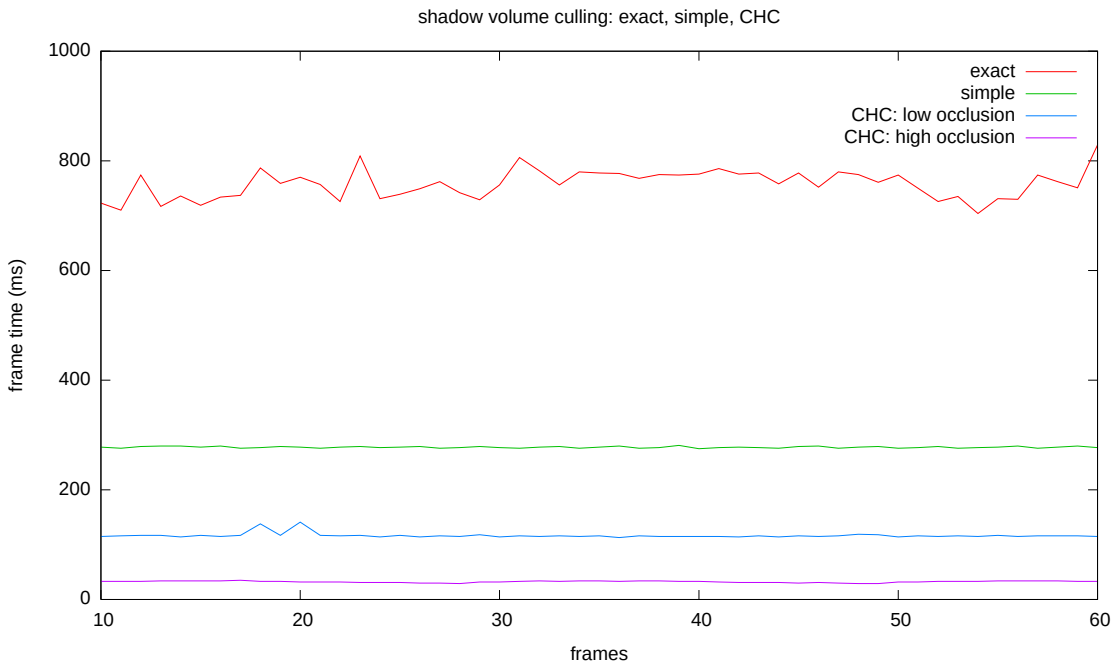

Figure 5.5: A comparison of different shadow volume culling techniques. All plots were sampled using the zfail algorithm. CHC is not only superior in performance to both exact and simple culling, but also preserves the dependency of PSC on PSR, which is not the case with the culling algorithm used in [69].

As it turns out, the use of coherent hierarchical culling (subsection 2.5.1) makes the above optimizations obsolete (Figure 5.5). Performance-wise, CHC is much preferrable over the above techniques to resolve visibility, even while keeping the dependency of PSC on PSR. Performance plots for *CHC shadow volume culling* in viewpoints with high and low occlusion are shown in Figure 5.6. As can be seen, using CHC shadow volume culling with zp+ algorithm gives an average framerate of ˜ 55 fps (800x600) in scenes with high occlusion, and ˜ 10 fps in scenes with low occlusion (both with a resolution of 800x600 pixels). Obviously, CHC improves the performance even for viewpoints with little occlusion (compare Figure 5.6, top, to Figure 5.3).

Generally speaking, for shadow volume culling to make sense, there must be some degree of *unnecessary* shadow volumes in the scene. With the complex self-shadowing of branches in a tree, there are only few shadow casters that are completely enclosed within the shadow of another object or do not cover any shadow receiver at all. Though this scene may be fill bound as well, CHC culling would provide only little performance gain, if any at all (note the increased overhead of the culling step itself). A slight drawback when CHC is used for shadow volume culling is that the framerate is highly dependent on the position of the viewpoint. If the position of the viewpoint makes up for a high occlusion, which results in CHC culling a large number of objects, the framerate can easily be ten times higher and more than for little occlusion (for instance a bird's eye view of the city).

Shadow volume clamping aims at reducing the surviving shadow volumes in size so that they tightly fit around scene geometry. The discrete clamping technique uses the graphics hardware to determine the slices in which shadow volumes have to be rasterized. How tight the resulting shadow volume segments fit around the shadow receiver depends on the number n of slicing planes that are used to partition the view frustum. With a field-of-view angle of 60 degrees, 10 slices are usually sufficient, resulting in a slice angle φ of 6 degrees. For every shadow caster in the scene, discrete clamping adds n additional rendering passes to determine occupied intervals. Therefore, in large scenes, discrete clamping makes sense only in conjunction with shadow volume culling to limit the number of shadow casters. Even if the culling step reduces the number of shadow casters to 10 (which would mean a culling rate of more than 99.9 % in the city scene), there is an overhead of 100 (10 slices for 10 occluders each) additional rendering passes. Also noteworthy is the fact that discrete clamping is view-dependant, because if the viewpoint changes, the slicing planes have to change as well. Therefore, the computations have to be performed in every frame. In practice, the overhead involved with discrete clamping is not feasible, because it eats up any fill reductions due to clamped shadow volumes.

In contrast to discrete clamping, continuous clamping is performed entirely on the CPU. The scene objects are transformed into light space, where overlap tests are performed to determine object intersections. The extents in the z dimension then dictate the intervals that must be occupied by shadow volumes. An efficient culling of

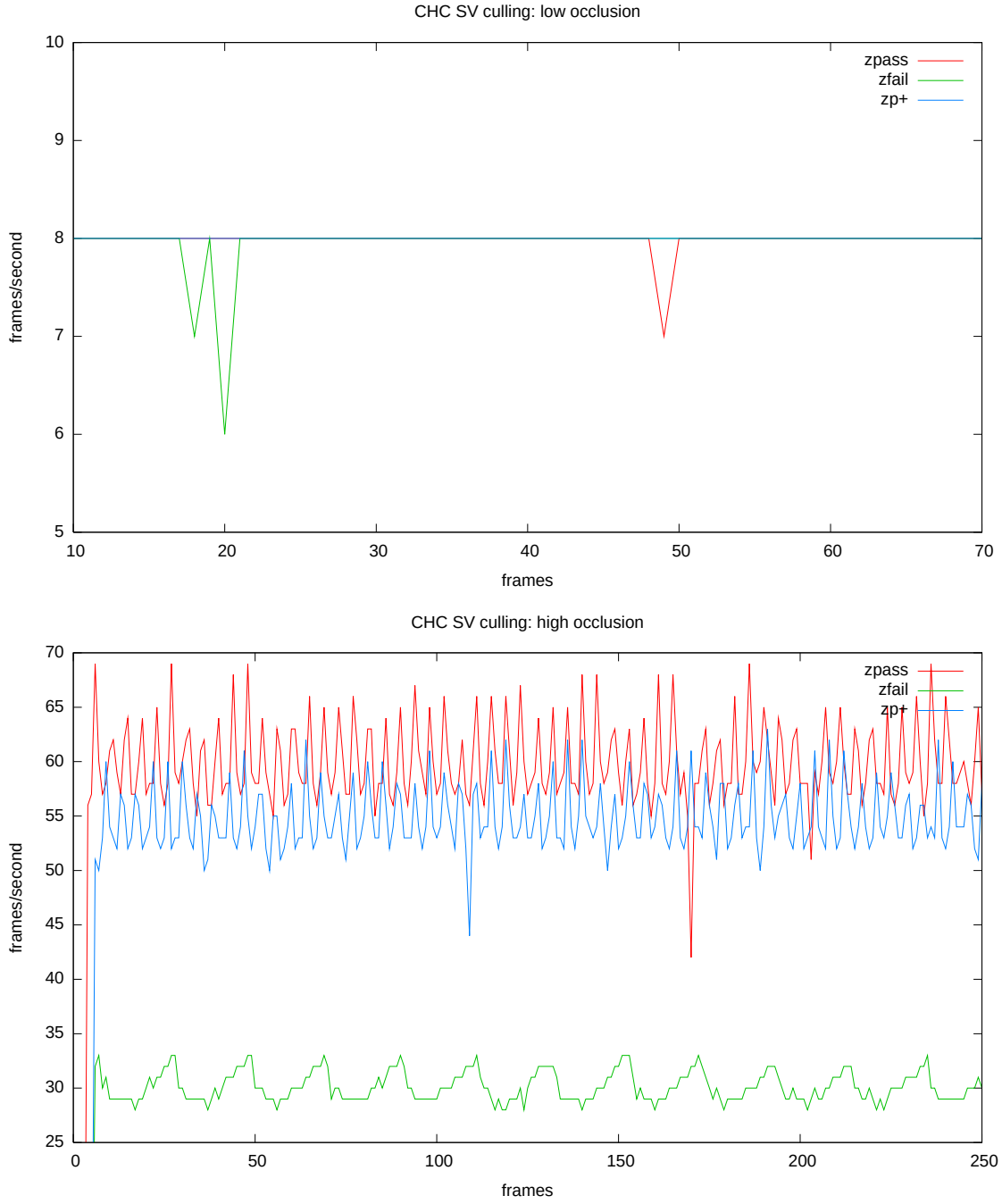

Figure 5.6: Performance plots for CHC in shadow volume culling. The top figure shows the plot for a viewpoint with very little occlusion (a bird's eye view over the city). Note how there is almost no difference in fps between zpass, zfail, and zp+. This is because the rasterizer is the bottleneck even with the reduced shadow volume polygon counts in the zpass and zp+ algorithms. A viewpoint with high occlusion is illustrated at the bottom.

the scene objects and tight PSR and PSC sets are crucial for continuous clamping, because the two-dimensional intersection tests have quadratic complexity $O(n^2)$, where n is the total number of objects in the scene. Exploiting temporal coherence, the expected time can be reduced to $O(n + k)$, with k being the actual number of overlapping objects. Even the sweep-and-prune technique, which benefits from temporal coherence and performs incremental operations, has to insert-sort the x and y lists initially, which can take a long time if there is a huge number of objects involved. Another problem with continuous clamping involves the transformation of the scene objects into light space. The CC shadow volumes algorithm accelerates this step by transforming axis-aligned bounding boxes instead of the real geometry, which makes sense in a scene that consists of only a few, but complex objects. Bounding boxes do not make much of a difference in the city scene, though, since the objects are already simple enough, and the main problem is the high number of objects, not their individual complexity.

Even in conjunction with CHC shadow volume culling, the performance with continuous clamping turned out to be worse than when shadow volume culling is performed alone. CHC shadow volume culling is a more conservative technique than the exact method described in subsection 3.2.5 and results in slightly larger sets of PSR and PSC. Obviously, the remaining objects in PSR and PSC are too numerous in the city scene, which results in a shift of the bottleneck from the GPU to the CPU.

Continuous clamping is view-independent. The depth intervals of the shadow volumes depend only on the relative position of the objects and the light source. Therefore, for static scenes, shadow volume clamping can be executed in a preprocessing step. However, since the shadow casting objects can change from frame to frame (due to shadow volume culling), the preprocessing step must process all objects in the scene, which takes a very long time for the 17.420 individual objects in the city scene (about 45 minutes on a Pentium 4 with 3.2 GHz and 1 GB RAM). As it turned out, the increase in performance with the 800x600 resolution was only about 10% with those precomputed shadow volume segments, clearly not worth the 45 minutes waiting time. The poor performance gain is mainly due to the culling step, which relieves the rasterizer sufficiently. With high resolutions (1600x1200), the performance gain is between 8 and 10 %.

Summarizing, we achieved the best result by using CHC to cull unnecessary shadow casters and then rendering standard shadow volumes without performing any shadow volume clamping. This results in a distinct reduction of the number of pixels that have to be rasterized every frame (Figure 5.7 compares fillrate and framerate at a resolution of 1600x1200 pixels for standard shadow volumes and CHC shadow volume culling). Clamping is an interesting technique to reduce fill cost, but relies on a small number of objects in the scene. Discrete clamping is executed on the GPU, which leaves the CPU free to perform other tasks, but introduces a lot of additional rendering passes, which in practice results in an even lower performance. Continuous clamping executes on the CPU and has quadratic complexity (due to

the sorting steps), therefore can be used only if the number of objects is sufficiently small. The shadow volumes are constructed every frame on-the-fly, using a vertex shader. This provides flexibility and works with clamping as well as without; the application only has to set the parameters accordingly. Figure 5.8 and Figure 5.9 show performance plots for a typical walkthrough in the city and PowerPlant scenes, using CHC shadow volume culling with zp+. By comparing Figure 5.2 and the bottom of Figure 5.6, we achieve an almost better average framerate when doing CHC shadow volume culling than using a standard renderer without any kind of occlusion culling.

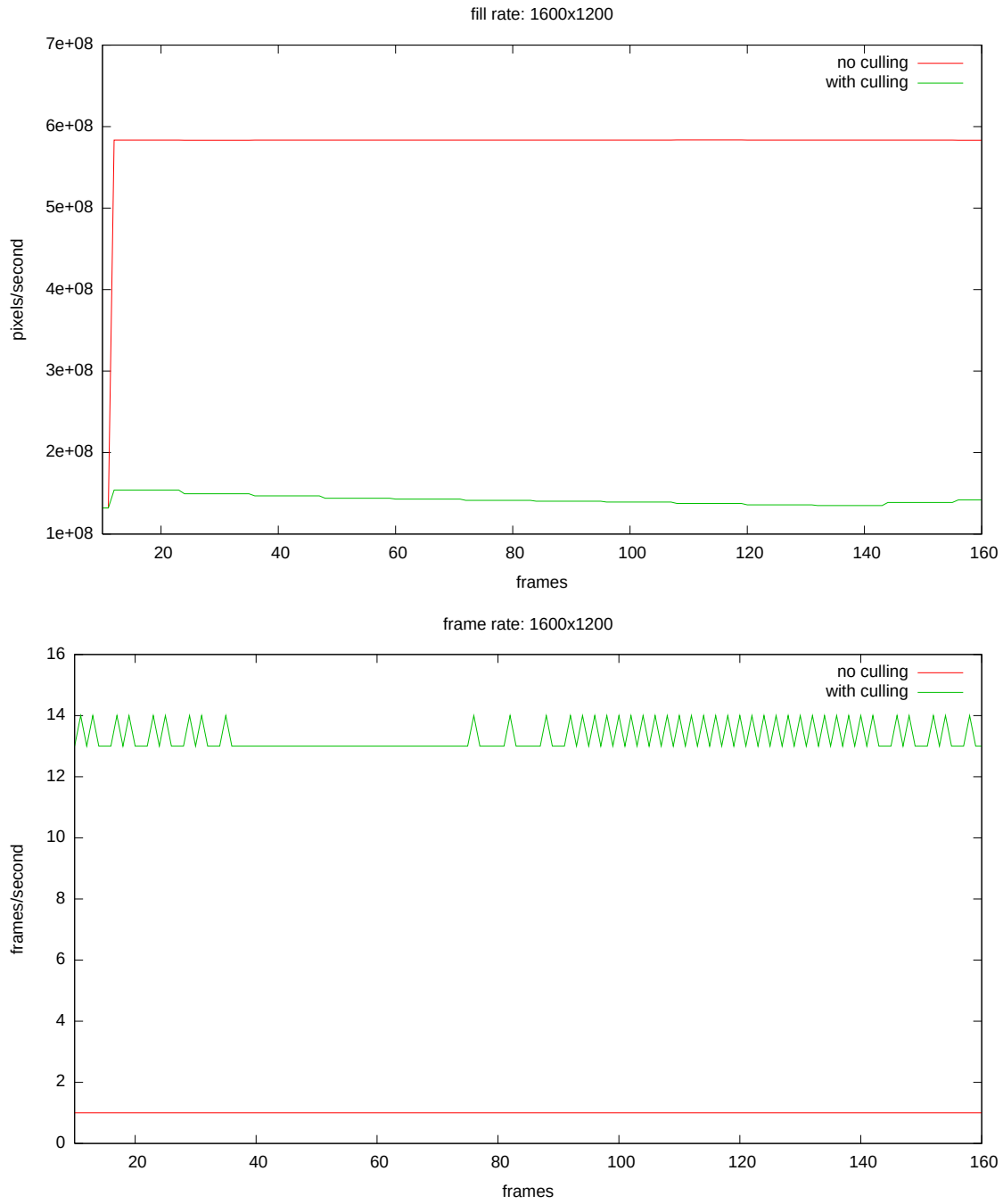

Figure 5.7: A comparison of standard shadow volumes and CHC shadow volume culling at a high resolution (1600x1200) from a static viewpoint. The top plot shows the number of pixels drawn every frame, which is four times higher without shadow volume culling. This reduction leads to a framerate increase from 1 to about 13 at this resolution, as indicated in the bottom plot.

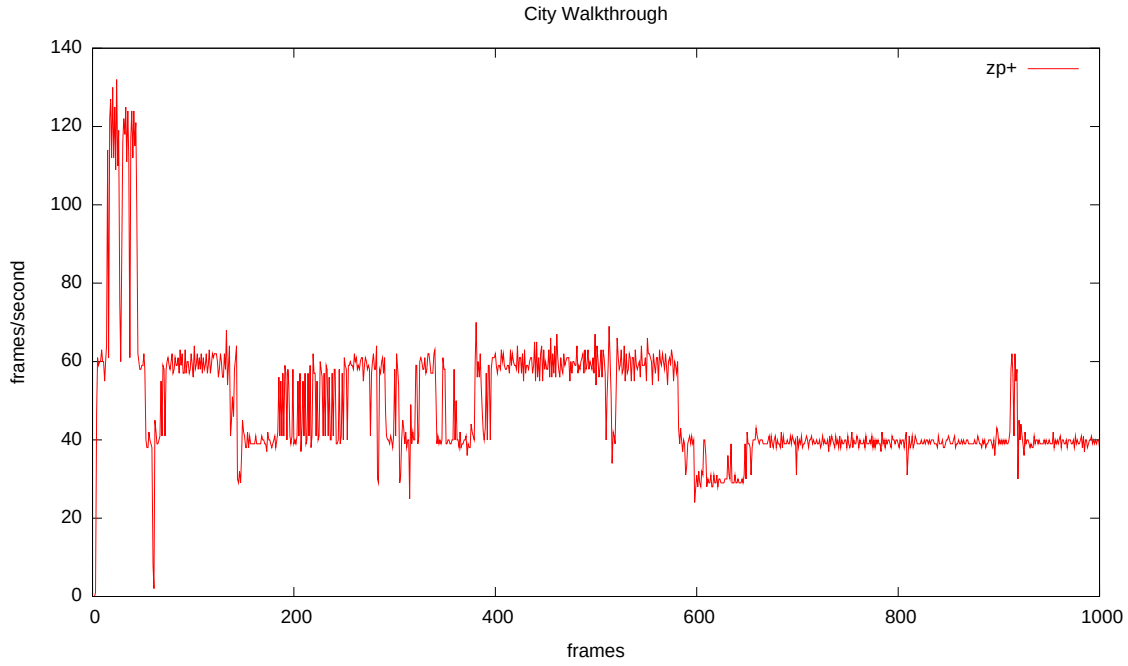

Figure 5.8: A performance plot of a typical walkthrough in the city scene, using standard shadow volumes, CHC culling and the zp+ algorithm.

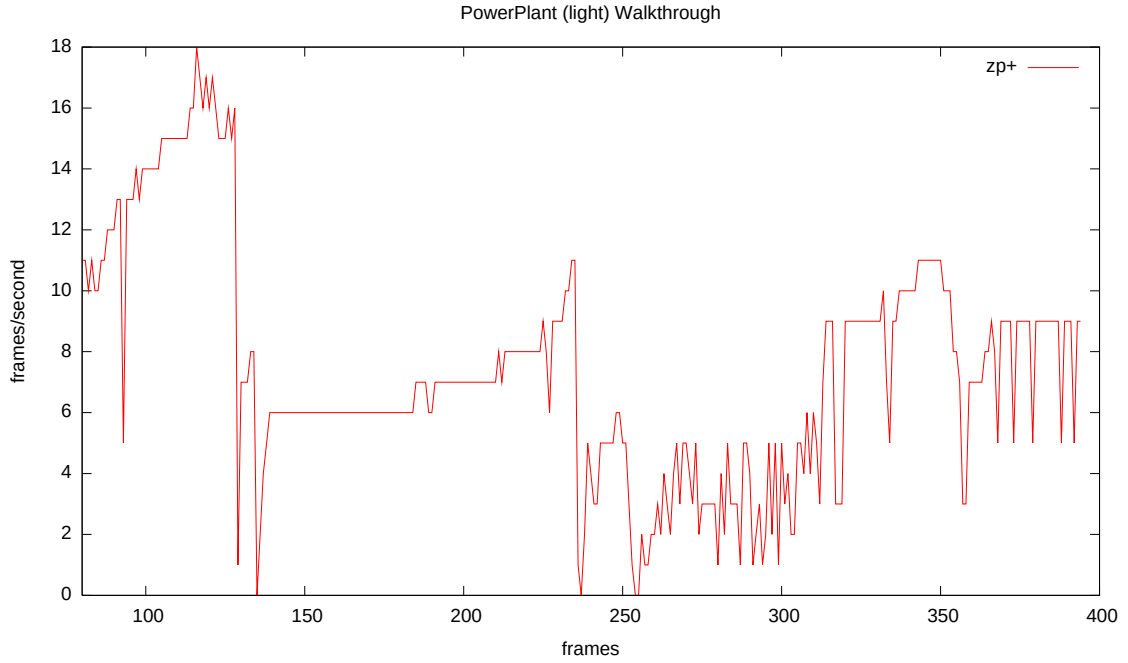

Figure 5.9: A performance plot of a walkthrough in the PowerPlant "light" scene, using standard shadow volumes, CHC culling and the zp+ algorithm. This plot was taken on a Dell Inspiron 9200 Laptop with 2 GB RAM and a NVIDIA GeForce 6800 GPU.

6 Conclusion

In this thesis, we first presented an overview over the current state-of-the-art in real-time shadowing algorithms, followed by a description of several techniques that try to address the deficiencies in the standard shadow volume technique. We described how shadow volumes can be constructed efficiently directly in the graphics hardware by using a vertex shader and how the number of shadow volume polygons can be reduced by extruding only those edges of an object that make up the silhouette with respect to the light source. Several different techniques can be used to identify silhouette edges, including exact and probabilistic ones. The zp+ algorithm is an extension to the zpass algorithm and corrects the robustness problems of the latter when shadow volume polygons are clipped by the viewer's near plane. Shadow volume culling utilizes standard visibility culling techniques to cull shadow casters that are completely enclosed within the shadow of another object or do not cast a shadow onto any visible objects. Shadow volume clamping tightly fits a shadow volume around a shadow receiver, preventing shadow volumes from covering large regions of empty space and reducing fill cost, but requiring a sufficiently small number of objects in the scene to provide a performance gain.

The main contribution of this work is the analysis of the shortcomings of several existing optimization techniques that do not work well in our target scenes. To overcome these shortcomings, we developed a number of optimized algorithms that work well together and result in acceptable performance.

Even though the shadow volume culling algorithm presented in [69] was designed for scenes with a large amount of triangles, in fact it makes sense only if the scene consists of only a few number of individual objects. This is due to the fact that the algorithm makes heavy use of bounding volumes that reduce the complexity of the objects but don't yield a performance gain if the scene contains many but simple individual objects. Therefore, we present a new technique for shadow volume culling based on the CHC visibility determination algorithm [13]. This new technique is designed to work in all scenes, no matter how the triangles are distributed and how many distinct objects there are. To improve culling accuracy, we additionally apply a technique to focus the lights' view on the relevant parts of the scene. Shadow volume clamping [69] tightly fits shadow volumes around shadow-receiving objects and avoids shadow volumes that cover large portions of empty space, but imposes a computational overhead, since the individual shadow volume segments must be identified and recomputed every frame. We improve the occupied-slices determination by using the stencil buffer once instead of rendering the shadow casters

multiple times (once for each slice) and also present an optimized way of rendering the clamped volumes using a vertex shader that completely frees the CPU from the task of creating shadow volume segments.

Not all of these techniques are equally suited for the use in our target scenes. For example, performing shadow volume clamping (even with our optimized technique) did not yield a performance gain because it relies heavily on a small number of shadow casters. For the city scene that was the basis for this thesis, we achieved the best result with a combination of the zp+ algorithm, shadow volume culling using the coherent hierarchical culling technique, and creating shadow volumes directly on the graphics hardware, without doing shadow volume clamping. By these means we achieved a significant performance gain with the city scene from several seconds per frame with standard shadow volumes up to approximately 60 frames per second with a screen resolution of 800x600 pixels and a sufficient degree of occlusion (with CHC SV culling, the actual speed depends on the degree of occlusion in the scene). Compared to about 200 fps that can be achieved by rendering without any shadows at all, this amounts to a performance overhead of a factor of only 4.

6.1 Further Improvements

There are several directions for further improvements. The main focus of this work was to reduce the rasterization cost of the shadow volumes. The described clamping techniques are not very well suited for large scenes, mainly because of their computational overhead both on the GPU and the CPU. More sophisticated techniques might be found that reduce the overhead of both continuous and discrete clamping by better exploiting temporal coherence.

Second, the data structures associated with the algorithm can become quite large with complex scenes. This is mainly due to the caching of intermediate data to speed up processing time on the CPU. Basically, we improve CPU load at the expense of main memory (for the city scene, a minimum of 512 MB are required to avoid swapping). It may be possible to reduce the RAM requirements by using smarter data structures and caching strategies.

Finally, all described techniques are tailored for hard shadows, e. g., shadows cast by point light sources. Point light sources are less realistic because they don't exist in the real world, but have been used in computer graphics for a long time because of their simplicity. It might be desirable to integrate some of the presented optimizations with existing soft-shadowing algorithms.

A Implementation Details

A.1 The Engine

This thesis was implemented into the YARE graphics engine (Yet Another Rendering Engine), which was developed at the Institute for Computer Graphics and Algorithms of the Vienna University of Technology. It is originally part of the *UrbanViz* project which aims at providing a platform for modelling and real-time visualizing of large urban environments. However, its capabilities are not restricted to this kind of scene. YARE is designed as a general-purpose graphics engine that also serves as a framework for the implementation and assessment of new techniques like occlusion culling or image-based rendering.

The API that is provided by the engine is similar to Java3D. Internally, OpenGL is used as the underlying graphics library. YARE organizes the scene in the form of a scene graph, using most of the concepts described in section 2.4 like internal transformation nodes and node instancing. Scene graph traversal is implemented after the *Visitor* design pattern [35], which provides enough flexibility to implement many different types of traversals, including the multiple passes that are necessary for the shadow volume algorithm.

A.2 Non-manifold objects

The standard shadow volume algorithm works with any kind of objects, because every single triangle casts its own shadow volume. To optimize shadow volumes by extruding only silhouette edges, the objects are required to be *manifold* to produce correct results. Manifold objects are closed, i. e., none of the triangles' backfaces are visible from the outside. Furthermore, every edge must connect exactly two triangles. For manifold objects, the silhouette is guaranteed to be a closed loop outlining the object.

In the city scene, not all objects fulfill this requirement. There are many objects that are not closed, but especially problematic for most silhouette detection algorithms are those objects that contain T-junctions at edges, i. e., edges where three or more triangles connect (Figure A.1). The edge elimination algorithm for silhouette determination fails with such edges, because it expects every edge to be part of exactly

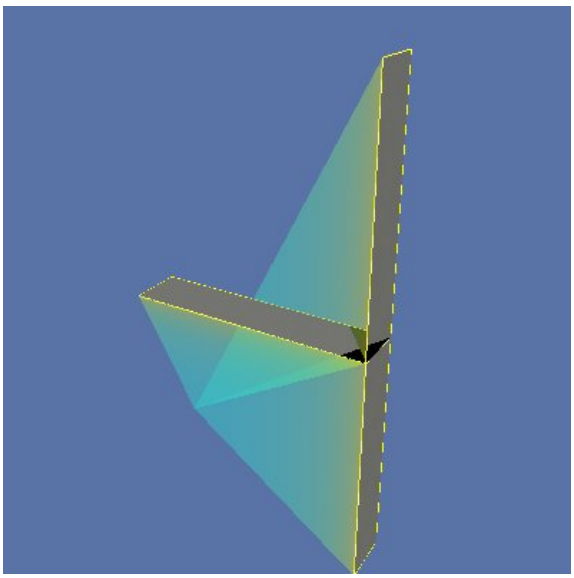

Figure A.1: This figure shows an edge that connects three triangles with incompatible orientations.

two triangles. Even worse, depending on the order in which the adjacent triangles are processed, the generated results may be correct or not.

To circumvent this problem, edges that connect more than two triangles must be duplicated. The original data structure includes an edge table that contains all edges. Initially, all edges contained in this list are unique, shared edges are therefore referenced by multiple triangles. By duplicating shared edges that are part of incompatible triangles, they are regarded as individual edges by the silhouette determination algorithm. For example, consider an edge that connects three triangles. The edge is duplicated. Now, there are two individual edges (with identical vertices). The first connects two triangles, and the second is part of only one triangle. Edges that are part of only one triangle are not a problem because they are put into the edge stack when they are first encountered, and stay there until the algorithm terminates. Therefore, ultimately all "dangling" edges are considered part of the silhouette, which leads to a correct result.

A second problem is the orientation of the triangles. With manifold objects, adjacent triangles must have a compatible orientation. Refer to Figure A.2 for an illustration. This problem can also be solved with the edge duplication method mentioned above. However, care has to be taken that the right triangles are chosen for the edge that connects two triangles.

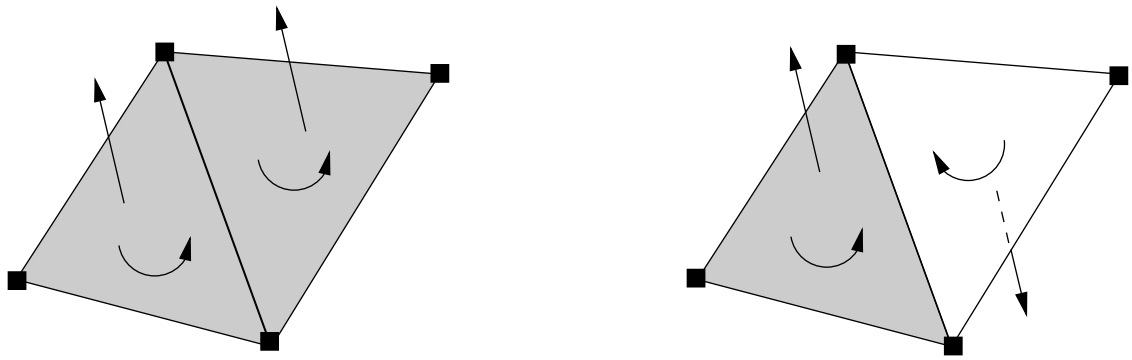

Figure A.2: The *vertex winding* determines the triangle's normal, which in turn determines the diretion the triangle faces. Adjacent triangles must have a consistent winding, otherwise they face opposite directions, which results in a hole in the object.

A.3 Vertex Buffer Objects

Many real-time rendering packages, including OpenGL and DirectX, provide two different approaches for rendering geometric data: *immediate mode* and *retained mode*. With immediate mode, the application sends in every frame all the data to the GPU. This can be advantageous in situations like modeling and animating, where the geometric data is prone to change frequently. However, because immediate mode transfers all data as individual elements, such as a single vertex or normal, it typically creates significant traffic to and from system memory and over the bus to the graphics hardware. This is additionally augmented by an increased CPU overhead because of the high number of API function calls. Optimized rendering primitives such as strips or fans attempt to mitigate some of the necessary data transfer by allowing for shared vertices. Nonetheless, rendering in immediate mode can often cause data transfer and CPU bottlenecks, which inhibits overall performance.

Retained mode is an alternative to immediate mode. OpenGL implements retained mode rendering in the form of *display lists*, which enable a series of rendering commands to be compiled into an optimized form that can be stored and later executed with a single call. Display lists can be kept in video memory, to avoid traffic over the graphics bus. Moreover, they make it attractive for OpenGL implementations to allow GPUs to pull data directly from system memory using DMA transfers. While this still requires the data to move over the graphics bus, applications usually benefit from the reduced CPU cycle count and front-side bus traffic.

Despite those benefits, display lists do have some disadvantages. Geometric data that is stored in a display list can not as easily be modified as with immediate mode. Depending on the frequency with which the data changes, the overhead of creating and managing display lists may outweigh the performance gains. Similarly, for best performance, OpenGL assumes that some states will not change within the display

list. Therefore, not all commands can be put into a display list, because some may prevent the block processing and still require CPU intervention. Moreover, display lists are initially issued to the OpenGL client. Ultimately, though, they are processed by the GPU from a copy stored on the server, which creates a doubling of data as compared to immediate mode. Also, the size of the display list's server copy is not visible to the application, possibly causing problems when memory size is limited.

As an alternative to display lists, OpenGL also offers *vertex arrays*. These allow vertex data (position, normal, color, texture coordinates) to be grouped and treated as a block, providing for a similar data transfer efficiency as display lists. The data contained in vertex arrays may also be interleaved, which means storing every vertex' attributes in a contingent block. This can be convenient for referencing the data. In contrast to display lists, however, vertex arrays do not assume that any individual piece of data will not change. Hence, when drawing an object using vertex arrays, the data in the array must be validated each time it is referenced, which adds to the overhead in data transfer. However, vertex arrays do not need to store two separate copies of all data.

Vertex Buffer Objects (VBOs) are intended to enhance OpenGL's capabilities by providing a combination of the benefits from immediate mode, display lists, and vertex arrays, while avoiding some of the limitations. VBOs allow data to be grouped and stored with the efficiency of vertex arrays. They are an improvement to a previous extension, `NV_vertex_array_range`, which provided similar functions, but was much more complex and incompatible to OpenGL's client-server architecture.

VBOs provide the possibility for applications to give hints about the data usage so that the OpenGL implementation can make decisions about the form in which the data should be stored, as well as its final location. This results in the possibility to modify data without causing transfer overhead due to validation. Finally, in combination with programmable shaders, VBOs enable the application to modify vertex data with previously rendered pixel data, and provide the possibility to render directly into a vertex array.

The main idea behind VBOs is to provide contingent memory regions (buffers) that are accessible from the application through identifiers. A buffer is made *active* by *binding* its identifier, similar to other OpenGL entities like texture objects or display lists. By binding a VBO, every client-state pointer is converted into an offset relative to the currently bound buffer. As a result, the bind operation turns a client-state function into a server-state function. This makes it possible to share VBO data among various clients, which is not possible for data that is contained in client-state. As a result, OpenGL clients are able to bind common buffers in the same way as textures or display lists. The following enumeration gives an outline of the most important steps involved in using VBOs:

1. Bind a VBO. This allows to use binding buffers, as opposed to work in absolute client-side memory.

2. Manage buffer size, provide usage hints, and copy data to the buffer.
3. Map or unmap the buffer, which returns a pointer into client-state absolute memory. This should be done only for short operations, and the result pointer is not persistent.

In short, VBOs are a great asset to complex scenes because they can move parts of the geometric data directly into graphics memory, avoiding transfer overheads.

A.4 The project shader

```
struct appdata
{
  float4 position : POSITION;
};

struct vfconn
{
  float4 hPos: POSITION;
  float4 col0: COLOR0;
};

vfconn main(appdata IN,
            uniform float4 l,
            uniform float4 lower,
            uniform float4 upper,
            uniform float4x4 modelViewProj)
{
  vfconn OUT;

  // l: light position (w == 1) or direction (w == 0) (world space)
  // lower: lower projection plane (world space)
  // upper: upper projection plane (world space)
  // IN.position: vertex (world space)

  // vertex
  float4 p = IN.position;

  // light-to-vertex vector
  float3 v = (l.w == 0) ? l.xyz : p.xyz - l.xyz;

  // if IN.position.w == 1 -> project to upper plane
  // if IN.position.w == 0 -> project to lower plane
```

```
  // plane parameters
  float3 n;
  float d;

  if (p.w == 1) {
    n = upper.xyz;
    d = upper.w;
    OUT.col0 = float4(0.8, 0.8, 0.2, .7);
  } else if (p.w == 0) {
    n = lower.xyz;
    d = lower.w;
    OUT.col0 = float4(1.0, 0, 0, 0.7);
  }

  // compute translation amount lambda
  float lambda = -(dot(n,p.xyz)+d) / dot(n,v);

  if (lambda > 0) {
    p.xyz = p.xyz + lambda*v;
  }
  p.w = 1;

  OUT.hPos = mul(modelViewProj, p);

  return OUT;
}
```

A.5 NVPerfKit

NVPerfKit is a tool developed by NVIDIA that gives an application direct access to low-level performance counters inside the driver as well as hardware counters inside the graphics hardware itself. These counters can be used to analyse the usage of the GPU, including the determination of bottlenecks. For NVPerfKit to work properly, the system must use a specially *instrumented* driver that provides the counters. NVIDIA's *Developer Control Panel* can then be used to select which signals are reported to the application. Counter querying is implemented using Windows' Management Instrumentation Performance Data Helper interface (PDH). The results can be obtained either from Windows' performance monitoring utility (PerfMon) which takes one sample per second, or directly within the application. The latter is the only way to perform per-frame sampling.

A code snippet for setting up PDH is listed below.

```
// Setup
PDH_HQUERY hQuery;
PDH_COUNTER hCounter;
PDH_STATUS status = PdhOpenQuery(0,0,&hQuery);
PdhAddCounter(hQuery,
              "\\NVIDIA GPU Performance(GPU0/% gpu_idle)\\GPUCounter Value",
              0,
              &hCounter));

// Periodically...
PDH_STATUS status = PdhCollectQueryData(hQuery);
PDH_FMT_COUNTERVALUE cvValue;
PdhGetFormattedCounterValue(hCounter,
                            PDH_FMT_DOUBLE|PDH_FMT_NOCAP100|PDH_FMT_NOSCALE,
                            0,
                            &cvValue);
double dCounterValue = cvValue.doubleValue;
float fCounterValue = cvValue.floatValue;
```

Two types of counters are available through NVPerfKit: *hardware counters* and *software counters*. Hardware counters are incorporated in various points directly on the GPU. Their results are accumulated from the last sampling point. For instance, `triangle_count` returns the number of triangles that have been processed since the last sample was taken. Therefore, hardware counters are somewhat cumbersome to use with PerfMon, because to get a per-frame average, the sample has to be divided by the average framerate of the application.

Software counters are integrated within the OpenGL and Direct3D drivers. In contrast to hardware counters, they accumulated and updated every frame, therefore representing a per-frame accounting. Sampling software counters at a sub-framerate frequency will result in identical values (from the previous frame).

Another way to distinguish counters is their method of reporting. *Raw counters* give an absolute count of something (triangles, milliseconds, pixels, ...), whereas *percentage counters* are based on the GPU clock rate and automatically divided by the number of GPU clock cycles since the last sample was taken. An example for a percentage counter is `gpu_idle`, because it counts the number of clock ticks the GPU has been idle since the last call, which is then divided by the number of clock ticks, yielding a percentage of time that the GPU was idle.

[77] gives a thorough explanation of the available counters and shows further examples how to integrate NVPerfKit into an application.

List of Figures

Bibliography

[1] Aila, Timo, and Ville Miettinen, *Umbra Reference Manual*, Hybrid Holding Ltd., Helsinki, Finland, 2000. http://www.hybrid.fi/umbra/download.html

[2] Airey, John M., *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*, Ph.D. Thesis, Technical Report TR90-027, Department of Computer Science, University of North Carolina at Chapel Hill, 1990.

[3] Airey, John M., John H. Rohlf, and Frederick P. Brooks Jr., *Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments*, Computer Graphics (1990 Symposium on Interactive 3D Graphics), vol. 24, no. 2, pp. 41-50, 1990.

[4] Akeley, Kurt, *The Silicon Graphics 4d/240GTX Superworkstation*, IEEE Computer Graphics and Applications, vol. 9, no. 4, pp. 71-83, 1989.

[5] Akeley, Kurt, and Thomas Jermoluk, *High-Performance Polygon Rendering*, Computer Graphics (SIGGRAPH '88 Proceedings), pp. 239-246, 1988.

[6] Akenine-Möller, Tomas, and Eric Haines, *Real-Time Rendering*, Second Edition, A K Peters, Ltd., 2002.

[7] Atherton, Peter, and Kevin Weiler, *Hidden Surface Removal using Polygon Area Sorting*, Computer Graphics (SIGGRAPH '77 Proceedings), vol. 11, pp. 214-222, 1977.

[8] Atherton, Peter, Kevin Weiler, and Donald Greenberg, *Polygon Shaow Generation*, Computer Graphics (SIGGRAPH '78 Proceedings), vol. 12, pp. 275-281, 1978.

[9] Benichou, Fabien, and Gershon Elber, *Output Sensitive Extraction of Silhouettes from Polygonal Geometry*, In Proceedings of the 7th Pacific Conference on Computer Graphics and Applications, Seoul, Korea, pp. 60-69, 1999.

[10] Bestimt, Jason, and Bryant Freitag, *Real-Time Shadow Casting Using Shadow Volumes*, Gamasutra, 1999. http://www.gamasutra.com/features/19991115/bestimt_freitag_03.htm

[11] Bilodeau, Bill, and Mike Songy, *Real-Time Shadows*, Creativity '99, Creative Labs Inc. sponsored game developer conferences, Los Angeles, California, and Surrey, England, 1999.

[12] 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, 2001.

[13] Bittner, Jiří, Michael Wimmer, Harald Piringer, and Werner Purgathofer, *Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful*, Proceedings of Eurographics 2004, pp. 615-624, September 2004.

[14] Bittner, Jiří, and Peter Wonka, *Visibility in Computer Graphics*, Environment and Planning B: Planning and Design, vol. 30, no. 5, pp. 729-756, 2003.

[15] Bittner, Jiří, Peter Wonka, and Michael Wimmer, *Visibility Preprocessing for Urban Scenes using Line Space Subdivision*, Pacific Graphics 2001, pp. 276-284, 2001.

[16] Blinn, Jim, *Me and My (Fake) Shadow*, IEEE Computer Graphics and Applications, vol. 8, no. 1, pp. 82-86, 1988.

[17] Brabec, Stefan, Thomas Annen, and Hans-Peter Seidel, *Hardware-Accelerated Rendering of Antialiased Shadows with Shadow Maps*, Computer Graphics International Proceedings, 2001.

[18] Brabec, Stefan, Thomas Annen, and Hans-Peter Seidel, *Shadow Mapping for Hemispherical and Omnidirectional Light Sources*, Computer Graphics International Proceedings, Bradford, 2002.

[19] Brennan, Chris, *Shadow Volume Extrusion Using a Vertex Shader*, in Engel, Wolfgang, ed., Shader X, Wordware, May 2002. http://www.shaderx.com/

[20] Buchanan, John W., and Mario C. Sousa, *The Edge Buffer: A Data Structure for Easy Silhouette Rendering*, Proceedings of the First International Symposium on Non-photorealistic Animation and Rendering (NPAR), pp. 39-42, 2000. http://www.red3d.com/cwr/npr/

[21] Chan, Eric, and Frëdo Durand, *An Efficient Hybrid Shadow Rendering Algorithm* Proceedings of the Eurographics Symposium on Rendering, pp. 185-195, 2004.

[22] Clark, James H., *Hierarchical Geometric Models for Visible Surface Algorithms*, Communications of the ACM, vol. 19, no. 10, pp. 547-554, 1976.

[23] Cohen, Jonathan D., Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi, *I-COLLIDE: An Interactive and Exact Collision Detection System for Large Scale Environments*, Proceedings of the 1995 Symposium on Interactive 3D Graphics, pp. 189-218, 1995.

[24] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Livest, and Cliff Stein, *Introduction to Algorithms*, Second Edition, MIT Press, Inc., Cambridge, Massachusetts, 2001.

[25] Crow, Franklin C., *Shadow Algorithms for Computer Graphics*, Computer Graphics (SIGGRAPH '77 Proceedings), pp. 242-248, 1977.

[26] Cunnif, R., *Visualize fx Graphics Scalable Architecture*, Hot3D Proceedings, ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, Switzerland, 2000.

[27] Davis, Tom, Jackie Neider, Dave Shreiner, and Mason Woo, *OpenGL Programming Guide*, Third Edition, Addison-Wesley, 2000.

[28] Downs, Laura, Tomas Möller, and Carlo Séquin, *Occlusion Horizons for Driving through Urban Scenery*, Proceedings 2001 Symposium on Interactive 3D Graphics, pp. 121-124, March 2001.

[29] Drettakis, George, and Marc Stamminger, *Perspective Shadow Maps*, Transactions on Graphics 21(3) (SIGGRAPH '02 Proceedings), pp. 557-563, 2002.

[30] Eberly, David, *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*, Morgan Kaufmann Publishers Inc., San Francisco, 2000. http://www.magic-software.com/

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

[32] Engel, Wolfgang, ed., *ShaderX*, Wordware, 2002. http://www.shaderx.com/

[33] Everitt, Cass, and Mark Kilgard, *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*, NVIDIA White Paper, 2002. http://developer.nvidia.com/

[34] Fernando, Randima, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg, *Adaptive Shadow Maps*, Computer Graphics (SIGGRAPH '01 Proceedings), pp. 387-390, 2001. http://www.graphics.cornell.edu/pubs/2001/FFBG01.html

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

[36] Gooch, Bruce., Peter-Pike J. Sloan, Amy Gooch, Peter Shirley, and Rich Riesenfeld, *Interactive Technical Illustration*, ACM Symposium on Interactive 3D Graphics, April 1999. http://research.microsoft.com/~ppsloan/

[37] Gouraud, Henry, *Continuous Shading of Curved Surfaces*, IEEE Transactions on Computers, vol. C-20, pp. 623-629, 1971.

[38] Govindaraju, Naga K., Avneesh Sud, Sung-Eui Yoon, and Dinesh Manocha, *Interactive Visibility Culling in Complex Environments using Occlusion-Switches*, Proceedings of the 2003 symposium on Interactive 3D graphics, pp. 103-112, 2003.

[39] Govindaraju, Naga K., Brandon Lloyd, Sung-Eui Yoon, Avneesh Sud, and Dinesh Manocha, *Interactive Shadow Generation in Complex Environments*, ACM Transactions on Graphics (SIGGRAPH 2003 Proceedings), vol. 22, no. 3, pp. 501-510, 2003. http://gamma.cs.unc.edu/shadow/

[40] Govindaraju, Naga K., Ming C. Lin, and Dinesh Manocha, *Fast and Reliable Collision Detection Using Graphics Hardware*, Technical Report, Department of Computer Science, University of North Carolina at Chapel Hill, 2004.

[41] Greene, Ned, *Environment Mapping and Other Applications of World Projections*, IEEE Computer Graphics and Applications, vol. 6, no. 11, pp. 21-29, 1986.

[42] Greene, Ned, *Hierarchical Rendering of Complex Environments*, Ph.D. Thesis, University of Carolina at Santa Cruz, June 1995.

[43] Greene, Ned, *Occlusion Culling with Optimized Hierarchical Z-Buffering*, SIGGRAPH 2001 course notes: Visibility, Problems, Techniques and Applications, 2001.

[44] Greene, Ned, Michael Kass, and Gavin Miller, *Hierarchical Z-Buffer Visibility*, Computer Graphics (SIGGRAPH '93 Proceedings), pp. 231-238, 1993.

[45] Hart, Evan, Dave Gosselin, and John Isidoro, *Vertex Shading with Direct3D and OpenGL*, Game Developers Conference, San Jose, March 2001. http://www.ati.com/na/pages/resource_centre/dev_rel/techpapers.html

[46] Hartner, Ashley, Mark Hartner, Elaine Cohen, and Bruce Gooch, *Object space silhouette algorithms*, Submitted for publication, 2003.

[47] Hearn, Donald, and Pauline Baker, *Computer Graphics, C Version*, Second Edition, Prentice Hall, Inc, 1997.

[48] Heckbert, Paul, *Survey of Texture Mapping*, IEEE Computer Graphics and Applications, pp. 5667, November 1986.

[49] Heckbert, Paul S., and Michael Herf, *Simulating Soft Shadows with Graphics Hardware*, Tech Report CMU-CS-97-104, Carnegie Mellon University, 1997. http://www.cs.cmu.edu/~ph/shadow.html

[50] Heidmann, Tim, *Real Shadows, Real Time*, Iris Universe, no. 18, pp. 23-31, Silicon Graphics Inc., 1991

[51] Heidrich, Wolfgang, *High-Quality Shading and Lighting for Hardware-Accelerated Rendering*, PhD Thesis, Universität Erlangen, 1999. http://www.cs.ubc.ca/~heidrich/Papers/phd.pdf

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

[53] Herf, Michael, and Paul S. Heckbert, *Fast Soft Shadows*, Visual Proceedings (SIGGRAPH '96), p. 145, 1996.

[54] Hertzmann, Aaron, and Denis Zorin, *Illustrating Smooth Surfaces*, SIGGRAPH 2000 Conference Proceedings, New Orleans, Louisiana, pp. 517-526, 2000. http://www.mrl.nyu.edu/publications/illustrating-smooth/

[55] Hoffman, Naty, and Kenny Mitchell, *Photorealistic Terrain Lighting in Real-Time*, Game Developer, vol. 8, no. 7, pp. 32-41, 2001. http://www.gdconf.com/archives/proceedings/2001/prog_papers.html

[56] Hornus, Samuel, Jared Hoberock, Sylvain Lefebvre, and John C. Hart, *ZP+: Correct Z-Pass Stencil Shadows*, ACM Symposium on Interactive 3D Graphics and Games, April 2005. http://artis.inrialpes.fr/Publications/2005/HHLH05

[57] Kay, Timothy L., and James T. Kajiya, *Raytracing Complex Scenes*, ACM SIGGRAPH 1986, pp. 269-278.

[58] Kilgard, Mark J., *More Advanced Hardware Rendering Techniques*, Game Developers Conference, 2001. http://developer.nvidia.com/

[59] Kilgard, Mark J., *Shadow Mapping with Today's OpenGL Hardware*, Game Developer's Conference, 2000. http://developer.nvidia.com/

[60] Kilgard, Mark J., and Cass Everitt, *Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering*, Technical Report, NVIDIA Corporation, 2002. http://developer.nvidia.com/

[61] Kilgard, Mark J., and Cass Everitt, *Optimized Stencil Shadow Volumes*, Technical Report, NVIDIA Corporation, 2003. http://developer.nvidia.com/

[62] King, Yossarian, *Ground-Plane Shadows*, Game Programming Gems, Charles River Media, pp. 432-438, 2000.

[63] Klosowski, James T., Martin Held, Joseph S. B. Mitchell, Henry Sowrizal, and Karel Zikan, *Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs*, IEEE Transactions on Visualization and Computer Graphics, vol. 4, no. 1, 1998.

[64] Knuth, Donald E., *The Art of Computer Programming, Volume 3*, Addison-Wesley, 1973.

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

[66] Koltun, Vladlen, Yiorgos Chrysanthou, and Daniel Cohen-Or, *Hardware-Accelerated From-Region Visibility using a Dual Ray Space*, 12th Eurographics Workshop on Rendering, pp. 204-214, 2001. http://www.math.tau.ac.il/~vladlen/

[67] Laine, Samuli, *Split-Plane Shadow Volumes*, in Proceedings of Graphics Hardware 2005, pp. 23-32, 2005.

[68] Lander, Jeff, *Images from Deep in the Programmer's Cave*, Game Developer, vol. 8, no. 5, pp. 23-28, 2001. http://www.gdmag.com/code.htm

[69] Lloyd, Brandon, Jeremy Wendt, Naga K. Govindaraju, and Dinesh Manocha, *CC Shadow Volumes*, Eurographics Symposium on Rendering Proceedings, 2004. http://gamma.cs.unc.edu/ccsv

[70] Markosian, Lee, Michael A. Kowalski, Samuel J. Trychin, Lubomir D. Bourdev, Daniel Goldstein, and John F. Hughes, *Real-Time Non-photorealistic Rendering*, Computer Graphics (SIGGRAPH '97 Proceedings), pp. 415-520, 1997. http://www.cs.brown.edu/research/graphics/research/npr/home.html

[71] Marshall, Carl S., *Cartoon Rendering: Real-Time Silhouette Edge Detection and Rendering*, in Mark DeLoura, ed., Game Programming Gems 2, Charles River Media, pp. 436-443, 2001.

[72] Maughan, Chris, and Matthias Wloka, *Vertex Shader Introduction*, NVIDIA White Paper, May 2001. http://developer.nvidia.com

[73] McCool, Michael D., *Shadow Volume Reconstruction from Depth Maps*, ACM Transactions on Graphics, vol. 19, pp. 1-26, 2000.

[74] McReynolds, Tom, David Blythe, Brad Grantham, and Scott Nelson, *Advanced Graphics Programming Techniques Using OpenGL*, SIGGRAPH '99 Course Notes, 1999. http://www.opengo.org/developers/code/sig99/index.html

[75] Nadeau, David R., Michael J. Bailey, and Michael F. Deering, *Introduction to Programming with Java3D*, SIGGRAPH '98 Lectures, 1998. http://java.sun.com/products/java-media/3D/collateral/class_notes/java3d.htm

[76] NVIDIA, Inc., *Using Vertex Buffer Objects*, NVIDIA White Paper WP-01015-001_v01, October 2003.

[77] NVIDIA, Inc., *NVPerfKit User's Guide*, published online at http://developer.nvidia.com/, 2005.

[78] Pop, Mihai, Christian A. Duncan, Gill Barequet, Michael T. Goodrich, Wenjing Huang, and Subodh Kumar, *Efficient Perspective-Accurate Silhouette Computation*, ACM Computational Geometry, pp. 60-68, 2001. http://www.cs.jhu.edu/~subodh/research/papers/

[79] Reeves, William T., David H. Salesin, and Robert L. Cook, *Rendering Antialiased Shadows With Depth Maps*, Computer Graphics (SIGGRAPH '87 Proceedings), vol. 21, pp. 283291, 1987.

[80] Roettger, Stefan, Alexander Irion, and Thomas Ertl, *Shadow Volumes Revisited*, 10th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG), pp. 373-379, 2002. http://vis.informatik.uni-stuttgart.de/ger/research/pub/pub2002/roettgerWSCG02.pdf

[81] Rogers, David F., *Procedural Elements for Computer Graphics*, Second Edition, McGraw-Hill, 1998.

[82] Sander, Pedro V., Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder, *Silhouette Clipping*, in Computer Graphics (SIGGRAPH '00 Proceedings), pp. 327-334, 2000.

[83] Schaufler, Gernot, Julie Dorsey, Xavier Decoret, and François Sillion, *Conservative Volumetric Visibility with Occluder Fusion*, Computer Graphics (SIGGRAPH 2000 Proceedings), pp. 229-238, 2000.

[84] Scott, Noel D., Daniel M. Olsen, Ethan W. Gannett, *An Overview of the VISUALIZE fx Graphics Accelerator Hardware*, Hewlett-Packard Journal, pp. 28-34, 1998. http://www.hp.com/hpj/98may/ma98a4.htm

[85] Segal, Mark, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli, *Fast Shadows and Lighting Effects Using Texture Mapping*, Computer Graphics (SIGGRAPH '92 Proceedings), pp. 249-252, 1992.

[86] Sen, Pradeep, Mike Cammarano, and Pat Hanrahan, *Shadow Silhouette Maps*, ACM Transactions on Graphics, vol. 22, pp. 521-526, 2003.

[87] Teller, Seth J., *Visibility Computations in Densely Occluded Polyhedral Environments*, Ph.D. Thesis, Department of Computer Science, University of Berkeley, 1992.

[88] Teller, Seth J., and Carlo H. Séquin, *Visibility Preprocessing For Interactive Walkthroughs*, Computer Graphics (SIGGRAPH '91 Proceedings), pp. 61-69, July 1991.

[89] Teller. Set J., and Pat Hanrahan, *Global Visibility Algorithms for Illumination Computations*, Computer Graphics (SIGGRAPH '94 Proceedings), pp. 443-450, July 1994.

[90] Tessman, Thant, *Casting Shadows on Flat Surfaces*, Iris Universe, pp. 16-19, 1989.

[91] Wanger, Leonard, *The effect of shadow quality on the perception of spatial relationships in computer generated imagery*, Computer Graphics (1992 Symposium on Interactive 3D Graphics), vol. 25, pp. 39-42.

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

[93] Williams, Lance, *Casting Curved Shadows on Curved Surfaces*, Computer Graphics SIGGRAPH '78 Proceedings, pp. 270-274, 1978.

[94] Wimmer, Michael, Daniel Scherzer, and Werner Purgathofer, *Light Space Perspective Shadow Maps*, 15th Eurographics Symposium on Rendering, 2004.

[95] Wimmer, Michael, Markus Giegl, and Dieter Schmalstieg, *Fast Walkthroughs with Image Caches and Ray Casting*, Computers & Graphics, vol. 23, no. 6, pp. 831-838, 1999.

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

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

[98] Wonka, Peter, Michael Wimmer, and François X. Sillion, *Instant Visibility*, Proceedings of Eurographics 2001, vol. 20, no. 3, pp. 411-421, 2001.

[99] 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, 1998.

[100] Zhang, Hansong, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff III, *Visibility Culling using Hierarchical Occlusion Maps*, Computer Graphics (SIGGRAPH '97 Proceedings), pp. 77-88, 1997. http://www.cs.unc.edu/~zhangh/hom.html