Visibility and Occlusion: Rendering Acceleration and Shadow Computation

Yiorgos Chrysanthou  Daniel Cohen-Or
Frédo Durand       Cláudio T. Silva
Andrew Woo         Pierre Poulin
TABLE OF CONTENTS

Introduction 5

I Basics 7

1 A Visibility Starter 9
2 Important Concepts and Issues 21
3 Analytical visibility 33
4 Basic Tools 61
5 Robustness 91
6 Visible surface determination 93
7 Graphics Hardware 103

II Point-based occlusion culling 115

8 View volume and back face culling 117
9 Image-Space From-point Occlusion Culling 119
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Object-Space From-point Occlusion Culling</td>
<td>121</td>
</tr>
<tr>
<td>11</td>
<td>Occlusion culling in entertainment industry</td>
<td>123</td>
</tr>
<tr>
<td>III</td>
<td>From-region Culling</td>
<td>125</td>
</tr>
<tr>
<td>12</td>
<td>Architectural interiors visibility</td>
<td>127</td>
</tr>
<tr>
<td>13</td>
<td>Generic From-Region Visibility</td>
<td>131</td>
</tr>
<tr>
<td>14</td>
<td>Advanced From-Region Visibility Techniques</td>
<td>151</td>
</tr>
<tr>
<td>IV</td>
<td>Misc. occlusion culling</td>
<td>167</td>
</tr>
<tr>
<td>15</td>
<td>Occluder Selection and Synthesis</td>
<td>169</td>
</tr>
<tr>
<td>16</td>
<td>Integration with other acceleration techniques</td>
<td>187</td>
</tr>
<tr>
<td>V</td>
<td>Shadows</td>
<td>189</td>
</tr>
<tr>
<td>17</td>
<td>Preliminaries of Shadows</td>
<td>191</td>
</tr>
<tr>
<td>18</td>
<td>Hard Shadows</td>
<td>209</td>
</tr>
<tr>
<td>19</td>
<td>Soft Shadows</td>
<td>241</td>
</tr>
<tr>
<td>VI</td>
<td>Misc.</td>
<td>263</td>
</tr>
<tr>
<td>20</td>
<td>Other</td>
<td>265</td>
</tr>
<tr>
<td>21</td>
<td>Conclusions</td>
<td>267</td>
</tr>
<tr>
<td></td>
<td>Contents</td>
<td>269</td>
</tr>
<tr>
<td></td>
<td>List of Figures</td>
<td>277</td>
</tr>
</tbody>
</table>
ESPITE the dramatic achievements of computer graphics over the last decades, the motto remains the same: faster and better. The power of graphics hardware grows at an amazing pace, but the greed of conceptors and users grows even faster. The race towards smarter acceleration algorithms and nicer rendering is more important than ever.

Interestingly, the same class of problems is at the heart of these two goals. Visibility is crucial both for faster display and for better shadow computation. Indeed, an obvious way to speed up rendering consists in avoiding to waste time drawing invisible objects. And shadows correspond to points that are not visible from the light source. The techniques developed to solve these issues therefore share many similarities. blah blah

Visibility is an exciting field that involves both complex geometric interactions and very practical rendering issues. The last decade has been particularly fruitful, and has seen the development of a wealth of efficient techniques. Due to their novelty, these method have unfortunately received a scarce coverage at best.

The present book attempts to fill this gap. It surveys occlusion culling and shadow generation algorithms, focusing on recent techniques and practical solutions. Computer graphics is an ever-evolving field, and visibility will certainly remain an active area of research. We however believe that the state of the art of visibility computation has reached a point where many engineers from e.g. the game industry, production rendering, or from graphics hardware architecture can benefit from the most recent techniques. We hope that this book will contribute to expose these ideas to a broader audience, allowing them to produce faster and better images.
1 A historical perspective

Clark, Jones
   Teller, Airey
Greene
Quake, SGI optimizer, HP flag, ATI hyper-z
Failure of Fahrenheit

2 Audience

For whom: academia and industry
   Academia: computer graphics, game curriculum.
   Industry: games, production, hardware
   practical (hopefully)
   prerequisites basic CG, z-buffer, linear algebra, dot and cross product

3 Overview of the book

4 Acknowledgments
Part I
Basics
CHAPTER 1

A Visibility Starter

[ADD BIBLIO REFERENCES]

In order to brush the context of visibility techniques and introduce important notions and issues, we start with an informal overview of two simple visibility algorithms. In this chapter, we present an occlusion technique to accelerate rendering for urban walkthroughs. It eliminates objects hidden by large façades, which reduces the load of the graphics hardware. We then show that a very similar method can be used to compute cast shadows. These algorithms will be described further in the rest of the book. In the meanwhile, we use them as an introduction and as a concrete reference for the discussion and classification described in the next chapter.

Interactive rendering of large datasets is a fundamental issue for various applications in computer graphics. Although graphics-processing power is increasing every day, its performance has not been able to keep up with the rapid increase in data set complexity. To address this shortcoming, techniques, such as occlusion culling and level-of-detail rendering, are being developed to reduce the amount of geometry that needs to be rendered, while preserving image accuracy. Levels of details reduce the number of rendered triangles by simplifying the geometry of the objects. Coarser polygonal approximations are used for distant objects. In contrast, occlusion culling reduces the size of the rendered-polygons set by culling invisible objects.
1 Urban walkthrough

Imagine developing an urban walkthrough application such as a driving simulator or a game. Modern graphics hardware allows the application to display hundreds of thousands of polygons with complex texturing and shading in real time. But cities are composed of several thousand buildings and contain many polygon-intensive objects such as trees, moving cars, or pedestrian. This raises the size of the model to several millions of polygons to say the least, which clearly exceeds the capacity of the graphics hardware.

But these zillions of buildings, pedestrians and other complex models are usually not all visible at the same time. From the point of view of a car in a street, most objects are hidden by the nearby buildings. Trying to render them is a waste of resources, since they eventually do not contribute to the final image. This problem is known as overdraw. Ideally, we would like to spend resources only for primitives that are visible in the final image.

This is the goal of visibility culling. Visibility culling attempts to eliminate objects that are “obviously” not visible in the image. The specific definition of “obviously hidden” primitives and the quest for fast ways to detect them lie at the heart of visibility techniques.

2 Basic culling

Before presenting occlusion culling, we quickly review two more classical culling techniques: backface culling and view-frustum culling.

2.1 Backface culling

When displaying a solid object, it is not necessary to draw the polygons that face away from the viewer. For example, when rendering a cube, the back faces are always hidden by the rest of the object (Figure 1.1(a)). Backface culling consists in eliminating all the polygons that face away from the viewer. Back-facing polygons can be identified with a simple dot product, since their normal points away from the viewpoint (Figure 1.1(b)). On average we expect half of the scene polygons to be back-facing.

Backface culling is usually performed by the graphics API, by testing each polygon. It can only be used if objects are solid, and if the normals of the polygons are consistently oriented. If no normal is provided, the graphics API deduces them from the vertex ordering. The programmer has the option to turn backface culling on or off, depending if the current objects are solid or not, and to change the relation between vertex ordering and normals (clockwise or counter-clockwise). We will see in chapter 8 that a hierarchical approach can obviate the need to test each polygon.

2.2 View-frustum culling

View-frustum culling eliminates the objects that are outside the field of view. It is a simple geometric test, that checks if an object or primitive lies inside the viewing
3. OCCLUSION CULLING USING LARGE POLYGONS

Figure 1.1: Backface culling. The faces facing away from the viewer are eliminated. The dot product between their normal and a vector reaching the viewpoint is negative. (a) 3D example: Face B can be culled. (b) 2D example: Face B can be culled.

Figure 1.2: Three types of visibility culling techniques: (i) view frustum culling, (ii) back-face culling and (iii) occlusion culling.

frustum. The viewing frustum is the pyramid defined by the viewpoint and the image plane. It is usually truncated by a near and a far plane: Objects closer than the near plane or farther than the far plane are not displayed (Figure 1.2).

View-frustum culling could be performed by the hardware, but this would waste bandwidth, and it is advised to perform it as early as possible in object space. It can be performed by testing if each object or polygon is inside the volume. In chapter 8, we will discuss more efficient view frustum culling approaches.

3 Occlusion culling using large polygons

So far, we have reviewed backface and view frustum culling, which are simple tests that can be performed individually on each object or primitive. We now introduce occlusion culling, which eliminates objects that are hidden by other objects. See Figure 1.2 for a comparison of the three sorts of culling. Because any object of the scene can hide any
other object, occlusion culling is a more complex problem. In this chapter, we present a simple method for the special case of urban models, where large objects (buildings) cause most of the occlusion.

### 3.1 Basic method

We assume that the façades of buildings are modeled as large vertical rectangles. An object is hidden by a given façade from the current viewpoint if it lies inside the truncated pyramid defined by the rectangle and the viewpoint (Figure 1.3). Such a truncated pyramid is called a shadow frustum. It is defined by five planes $P_i$. Four of these planes go through the viewpoint $V$ and two vertices of the rectangle $R$, and the fifth plane is the plane of the rectangle $R$. The planes are defined such that the normals point outside the frustum (Figure 1.4).

In fact, the present occlusion culling method can be seen as the inverse of view-frustum culling. We reject objects that are inside the shadow frustum, while view-frustum culling rejects objects that are outside the view frustum.

Since the frustum is convex, a polygon lies completely inside it if all its vertices lie inside it. And we can test if a point $p$ is inside the frustum by computing the equations of the $P_i$ for $p$. If they are all positive, the point is inside the frustum. See Figure 1.5 for the pseudo code.

The display algorithm with occlusion culling is then very simple, as summarized in Figure 1.6. For each polygon, it simply tests if it is hidden by one of the rectangles, and if this is not the case, the polygon is actually rendered. This method culls the polygon that are hidden by one large rectangle. This relieves the graphics hardware from unnecessary computation, at the expense of some CPU visibility computation.
3. OCCLUSION CULLING USING LARGE POLYGONS

Figure 1.4: Shadow frustum of \( R \) with respect to \( V \).

<table>
<thead>
<tr>
<th>IsInsideFrustum (Rectangle R, Viewpoint V, Polygon Q)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute coefficients of planes ( P_i );</td>
</tr>
<tr>
<td>for each vertex ( p ) of ( Q )</td>
</tr>
<tr>
<td>for each plane ( P_i )</td>
</tr>
<tr>
<td>if ((P_i(p) &gt; 0)) return ( false );</td>
</tr>
<tr>
<td>return ( true );</td>
</tr>
</tbody>
</table>

Figure 1.5: Testing if an object lies inside a frustum.

3.2 Some important notions

This method can be improved in many ways, as we will see after discussing some important aspects of occlusion culling.

The major difference between occlusion culling and the classical hidden-surface removal used to compute a view is that occlusion culling does not need to compute the precise visible portion of the primitives. This is handled later by the final hidden-surface removal (usually a \( z \)-buffer). Occlusion culling only needs to classify an object as completely hidden or not, which is often called weak visibility. This permits simpler and faster computation.

Moreover, an occlusion culling algorithm can overlook hidden objects. For example, in Figure 1.3, if we fail to classify some polygons of the tree as hidden, the view is

<table>
<thead>
<tr>
<th>Display (List of rectangles ( R_i ), Viewpoint ( V ), List of polygons ( Q_i ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>for each polygon ( Q_i )</td>
</tr>
<tr>
<td>hidden=false;</td>
</tr>
<tr>
<td>for each rectangle ( R_i )</td>
</tr>
<tr>
<td>if ((IsInsideFrustum(Ri,V,Qi))) hidden=true;</td>
</tr>
<tr>
<td>if (not hidden) render(Qi);</td>
</tr>
</tbody>
</table>

Figure 1.6: Pseudo code of the basic main display routine.
still correct. It only takes longer to render. It means that the method has a very important property: It is conservative. It never classifies as hidden an object that is actually visible. However the opposite may happen: It may fail to identify an object as hidden. Conservativeness is crucial because it ensures that the final image that we compute is exactly the same as without occlusion culling.

The objects that are not declared hidden and that are sent to the graphics hardware are called potentially visible. This defines the potentially visible set or PVS as the set of objects that might contribute to the image. The set of all the polygons that actually contribute to the image is called the visible set. Ideally, we would like the PVS output by an occlusion culling method to be exactly the visible set. This is however rarely the case, and the PVS usually includes the visible sets as well as additional polygons that are not visible but that the algorithm has failed to determine hidden. These polygons are eventually be handled by the z-buffer.

Our method makes a conceptual distinction between entities that hide, called occluders, and entities that are hidden, occludees. The large rectangles are our occluders, while the polygons are occludees. Occluders and occludees can correspond to the same geometric object. For example the façades of the buildings are used both as occluders and occludees. They are tested for occlusion as well, to avoid rendering them if they are hidden by other façades.

4 Accelerating things a little bit

The previous occlusion culling method incurs a large CPU overhead because it tests every vertex of each polygon against every rectangle. We first avoid testing all the vertices of a polygon, and we then reduce the number of rectangles considered.

4.1 Bounding boxes

In order to avoid testing all the vertices of all polygons, we group them and perform the test on a simpler surrogate. We group the polygons into objects (e.g., trees, cars, buildings). How polygons are actually grouped is beyond the scope of this chapter, it can be information from the modeling phase or come from automatic clustering. This will be discussed in chapter ??.

Instead of performing the test for each polygon, we use boxes completely enclosing each object: bounding boxes. An axis-aligned bounding box can be easily computed for an object by taking for each axis the minimum and maximum coordinates of all its vertices (see Figure 1.7).

If the bounding box is completely inside a frustum, the corresponding object is also inside. It means that we can safely replace the test on the object by the test on the bounding box. This can result in a large speedup, since the bounding box has only 8 vertices, while the enclosed object might have thousands of vertices.

However, note that more polygons might be sent to the graphics hardware, because some individual polygons might be hidden while the whole object is not. Consider the example in Figure 1.8. The whole tree is not completely occluded, but most of the individual polygons modeling the leaves are. This is a typical issue of granularity, and
speed of visibility computation vs. quality of the result. Our new occlusion culling algorithm is faster, but it can miss more hidden polygons. The balance depends on the respective available CPU resources and graphics hardware.

One solution would be to refine the computation if the bounding box is partially hidden by the rectangle. Such techniques and other spatial acceleration data structures are fundamental for efficient visibility acceleration and will be covered in chapter ??.

4.2 Occluder selection

Our algorithm is still costly because we have to test each bounding box against each vertical façade. But in an urban environment, most occlusion is caused by the closest buildings. We thus decide to use a small subset of rectangles. We present a simple strategy: We simply pick the $k$ façades closest to the viewpoint. We hope that they are the most important occluders for the current viewpoint, and that this heuristic does not miss too much occlusion. The new algorithm is summarized Figure 1.9.

Of course, we might fail to identify hidden objects because we overlook some distant rectangles. This new acceleration strategy has again traded the speed of the occlusion computation for the number of polygons declared hidden. The number $k$ of
DisplayFaster ( List of rectangles $R_i$, Viewpoint $V$, List of objects $O_i$ )

Compute the $k$ closest rectangles $R_j$;

for each object $O_i$
    hidden = false;
    for each or the $k$ rectangles $R_j$
        if (IsInsideFrustum($R_j$, $V$, BoundingBox($O_i$))) hidden = true;
        if (not hidden) render($O_i$);

Figure 1.9: Pseudo code of the main display routine with occluder selection and bounding volumes.

Figure 1.10: Occluder selection. Occluder $C$ is not effective because it is occluded by $A$ and $B$. Occluder $D$ is small and does not cause a lot of occlusion. $E$ is distant but the configuration of the scene makes it effective.

rectangles considered is in particular crucial to control this tradeoff. Moreover, this simple strategy does not necessarily result in the optimal $k$ occluders, because more distant occluders can hide a large number of primitives. The visual angle of the occluder and the configuration of the scene have to be taken into account (Figure 1.10). More advanced strategies to select a relevant subset of occluders will be studied in chapter ??.
4.3 Other display acceleration techniques

One of the key advantages of occlusion culling is that it accelerates rendering without affecting the final result. However, there are cases where even a perfect occlusion culling cannot significantly speed up the display. In our city example, if the viewer is on top of a high building, most of the town are visible. Other acceleration techniques are then required. As opposed to occlusion culling, these techniques usually result in a degradation of the final image (lossy vs. lossless acceleration). However, they usually attempt to bound or minimize the error. We outline the two classical techniques, and their integration with occlusion culling will be described in Chapter 16.

**Level of Detail** consist in rendering more distant objects using fewer polygons. In its simplest form, each object is modeled at different precision, using fewer and fewer triangles, and the relevant precision is selected for each frame, usually depending on the screen size of the object. Algorithms have been developed to decimate a highly tesselated model and obtain faithful coarser versions. Another strategy is to build a data structure that permits the decimation the object continuously at run-time. These methods work best on finely subdivided meshes, and are typically less efficient for complex objects such as trees.

**Image-based acceleration** takes advantage of the limited parallax observed in the distance. It replaces distant portions of the scene with rendered images placed either on billboards or on simple meshes. These are called impostors. Image-based acceleration is better at simplifying trees or other collections of complex objects.

5 Shadow computation

In order to improve the realism of our images, we might want to render the shadows cast by the buildings. For simplicity, we model the Sun as an infinitely small point light source. Such a light-source is said to cast hard shadows, because a point of the scene is either fully illuminated or completely in shadow.

We show how the previous technique can be modified to compute the shadows. Note that historically, the shadow generation method was developed first and then later adapted to occlusion culling. It suffices to remark that shadows correspond to the parts of the scene that are not visible from the point light source.

A point is in the shadow of a building if it is inside the shadow frustum of the corresponding rectangle (Figure 1.11). We can check if a point \( p \) of the scene is in the shadow of a rectangle \( R \) by using our IsInsideFrustum method, but replacing the viewpoint \( V \) by the point light source \( L \).

Shadow computation requires a finer-grain visibility classification than occlusion culling: We want to compute precisely which points are in shadow, not just to know which polygons lie completely in shadow. Shadow computation is more similar to classical hidden surface removal. In particular, if an object is partially hidden by the light source, it means that there is a shadow boundary crossing it, and accurately rendering it is crucial for visual quality. This is why we need to perform the shadow test for each pixel of the image, and not for whole polygons. See Figure 1.12 for the pseudo code.

We will see in chapter ?? that testing if a point lies inside the shadow volume can
be implemented on current graphics hardware, making this technique very efficient for real-time rendering of shadows. It can be extended to the computation of shadow cast by arbitrary objects, not only by rectangle, by changing the definition of the frustum. We then talk of shadow volumes.

The method can also easily be extended to a directional light source, which is a better approximation of the Sun. The definition of the shadow volume has to be changed: The planes are then defined by the direction $\vec{L}$ of the light source and two vertices of the object (Figure 1.13).
6. **SUMMARY**

In this chapter, we have presented a simple visibility calculation technique. It computes the occlusion caused by vertical facades of buildings. The method is based on the notion of shadow frusta. These are truncated pyramids that describe the volume of space hidden by a polygon. They are defined by the viewpoint or light source and the polygon. A point inside the shadow frustum is hidden from the viewpoint or light source, and this can be tested easily using dot products. The technique can be used for display acceleration and shadow computation.

For display acceleration, it permits occlusion culling, that is, the elimination of objects that are completely hidden from the viewpoint. The method is conservative, which means that it never wrongly classifies as hidden an object that contributes to the image. However, it may fail to identify an hidden object. We have seen that the accuracy of the technique can be traded for speed.

Shadows can be obtained by computing for each point of the scene if they lie inside a shadow frustum. The computation is more involved because we need to determine precisely which parts are in shadow. Typically, we compute this for the 3D point corresponding to each pixel in the final image. In the rest of this book, we introduce many more techniques for occlusion culling and shadow computation. We will introduce more efficient methods, we will compute visibility with respect to volumes, we will compute compelling shadows including penumbra effects, etc. Before this, the next few chapters introduce important basic notions that are useful for all visibility techniques, and often for a variety of computer graphics algorithms.
In the previous chapter, we have sketched two algorithms for occlusion culling and shadow computation. They are simple but they are not always the most efficient techniques, and different strategies might be more relevant depending on the context. In this chapter, we introduce important issues and elements of classification that will help us find our way among the variety of techniques that have been developed.

1 Occlusion culling

1.1 Image and object precision

Occlusion culling techniques can be classified similarly to hidden-surface removal, in terms of object vs. image precision [?]. The techniques described in the previous chapter perform the computation in the 3D space where the objects are defined, at object precision. In contrast, some methods perform computation in the discrete pixel-based representation of the final image, at image precision. This means that their accuracy is limited by the resolution of the image.

We give a simple example of image-precision method working in conjunction with a z-buffer. As the image is rendered, objects are tested against the current values in the z-buffer. If the test determines that an object is completely hidden, it is not drawn. Consider the example in Figure 2.1. The buildings have already been rendered into the z-buffer. Before rendering the tree $O_1$, we test its bounding box against the z-buffer.
Figure 2.1: Image-precision occlusion culling. (a) Object space. (b) Z-buffer from \( V \). The bounding box of occludee \( O_1 \) is tested against the current z-buffer.

That is, we try to render the bounding box without modifying the color- nor the z-buffer. If for each pixel of the projection of the bounding box, the current value of the z-buffer is closer, then the bounding box is completely hidden and so is the tree.

These methods are more effective if the scene is rendered roughly from front to back. This way, nearby objects are rendered first (the buildings in our example), and their occlusion of more distant objects is captured. Strict front-to-back order is not required, the result will still be conservative, but some occlusions may not be detected.

Image-precision methods are usually easy to implement, quite robust, and effective. However, they often require feedback from the graphics hardware. In our example, we need to be able to test if the bounding box passes the z-buffer test. Unfortunately, this kind of feedback is not always available on graphics cards, which are optimized for CPU-to-graphics transfers, and not in the other direction. However, we will see in chapter ?? that some form of occlusion culling can be implemented completely in hardware.

An often overlooked assumption of image-precision techniques is that the CPU and the graphics card are working on the same frame. This is not always the case in modern real-time applications, where a pipeline acceleration can be used, especially if multiple CPUs are available. While the graphics hardware displays the current frame, the CPU is managing the database and performing various culling operations on the following frame. This means that when the CPU is managing primitives, it cannot rely on the hardware’s buffer because they are not working on the same frame.

In contrast, object-precision methods are usually implemented on the CPU only. This relieves the graphics hardware, and makes them simpler to implement in a pipelined application. Unfortunately, as we will discuss shortly, object-precision techniques are usually less efficient at handling the combined occlusion due to multiple occluders. As a result, they might cull less objects, i.e., provide a less tight estimate of the visibility test.
1. OCCLUSION CULLING

1.2 From-region visibility

The algorithms presented so far perform occlusion culling from the current viewpoint. This is called from-point occlusion culling. This means that the calculation has to be done for each frame. This is called online occlusion culling, because it is performed at run time.

Another strategy consists in preprocessing visibility. In order to be able to encode visibility for any possible viewpoint, the space of possible viewpoints is subdivided into volumetric cells (Figure 2.2 shows three box-shaped cells). A PVS is computed and stored for each cell.

Precomputed from-region visibility is then simple and efficient to use for display. For the current viewpoint, we find the corresponding viewing cell, and read the stored PVS. Then, only objects in the PVS are sent to the graphics hardware. Almost no cost is incurred at run time.

In order to be conservative, we can declare an object hidden from a viewing cell only if it is hidden from all the viewpoints inside the cell. Consider for example the computation of the PVS with respect to the cell VC$_1$ in Figure 2.2. Only $O_1$ can be eliminated. $O_2$ cannot be declared hidden because it is visible from e.g. viewpoint $V_2$.

The PVS from a region is usually larger than the PVS with respect to a point. For example, in Figure 2.2, $O_2$ is contained in the PVS of VC$_1$ but could be culled from the PVS of $V_1$. Moreover, preprocessing visibility comes with a storage cost that can be significant. The from-region strategy also has constraints. It assumes that the scene is mainly static. As we will discuss shortly, this might rule out preprocessing the scene is often modified, such as for interactive modeling. However, the static-scene assumption...
is acceptable in many situations. Consider the example of a driving game. The city usually does not change during the game, only cars do. In consequence, most of the visibility can be preprocessed and included as data on the CD-ROM, saving precious computational resources when the user plays.

The key property of from-region visibility is its prediction capability. In particular, if the scene is too big to fit into main memory, or if it is transmitted through a network, advanced scene management and pre-fetching are necessary to avoid latency when new geometry becomes visible and has to be loaded. If visibility has been preprocessed, the information for neighbouring view cells is available and can be used for pre-fetching. For example, in Figure 2.2, when the user is in cell VC1, we consider the neighbouring cells, including VC2. We can thus detect that O1 might become visible and has to be pre-fetched.

Recently, online from-region techniques have appeared. They compute visibility for a region of space around the current viewpoint. The goal is to benefit from the predictive capabilities of from-region visibility without incurring the preprocessing and storage cost. It also permits asynchronous visibility computation in the case where the CPU and the frame buffer are not working on the same frame.

1.3 Architectural scenes and cell/portal structure

Historically, from-region methods have first been developed for the special case of architectural interiors. These scenes can naturally be decomposed into rooms, which are used as viewing cells. These cells are connected by openings such as doors or windows. Different rooms are visible only from the openings, which are called portals in this context. The PVS of a room can be computed by propagating visibility through portals. For example, in Figure 2.3, the PVS of cell A is computed by propagating to C. Then, from C to D, then from C to E, and then from E to F.

This approach can be seen as the dual of the algorithms presented so far. In these methods, all objects were assumed a priori visible until an occlusion test determined them hidden. In contrast, cell-and-portal methods assume all objects hidden, and declare visible only those visible through a sequence of portals.

1.4 2D vs. 3D Visibility

Many scenes have a strong two-dimensional nature. For example, a floor of a building corresponds to the elevation of a 2D floorplan. This permits the reduction of occlusion culling to simpler 2D computation. We will discuss in Chapter 3 the fundamental reasons that make 2D visibility a much simpler problem than its 3D counterpart. The 2D plane is often referred to as flatland, after the novel by Abbott [?

Terrains or height fields represent an intermediate case. They are mostly two-dimensional, but the elevation is a crucial factor. The loose term 2.5D is sometimes used to describe them. These scenes are also amenable to simpler algorithms. Cities are a good example where 2.5D visibility can be particularly efficient. For example, the occlusion culling algorithm presented in the previous chapter can be simplified if all the buildings touch the ground. We then do not need to test the bottom plane $P_3$ in $IsInsideFrustum$ (see Figure 1.4). We will see that some algorithms depend even
Figure 2.3: Cell and portal structure of an architectural interior. The PVS of room A includes rooms \( \{ C, D, E, F \} \). The sequence of portals through which room F is visible is highlighted.

more heavily on the terrain nature of the scene. Often, only the occluders need to be a terrain. In our example, we can test non-terrain occludees such as cars or street lamps against the buildings, and the test without plane \( P_3 \) remains valid.

In the rest of the book, most algorithms will be illustrated with 2D figures for simplicity, when the extension to 3D is trivial. Note however that some 2D visibility property do not apply to 3D. They will be indicated when necessary. As a general rule, and as discussed in Chapter 3, one has to be extremely cautious with the generalization from 2D visibility to 3D visibility.

1.5 Exact, conservative, approximate and aggressive culling

We have seen that the visible set is the set of all the primitives that are at least partially visible. The potentially visible set (PVS) is the set of primitives declared visible by an occlusion culling method. The occlusion culling method presented in the previous chapter is conservative. This means that it never wrongly classifies a visible object as occluded. It also means that the PVS is a superset of the visibility set (Figure 2.4). When the PVS contains only the visible set, the algorithm is said to be exact.

The tightness of the PVS is a crucial measure of the efficiency of an occlusion culling algorithm. The closer the PVS is to the visibility set, the less resources are wasted during rendering. However, as we have seen, the tightness of the PVS might sometimes need to be traded for faster occlusion calculation.

In contrast, some method are not conservative but approximate. This means that some objects that actually contribute to the image might be wrongly classified as hidden (see Figure 2.4), potentially causing artifacts in the final image. However, the probability is hopefully small, and there are advantages that compensate for this potential drawback. Consider the example of from-region visibility. It is challenging to compute
visibility properties with respect to a region of space. We can however easily use a from-point method for a certain number of sample viewpoint, e.g. the vertices defining the region and its center (Figure 2.5). An object is then declared hidden with respect to the region if it is hidden from all sample viewpoints. Unfortunately, it is possible that the object is visible from a viewpoint that has not been sampled.

We see that approximate methods can be simple to implement, especially for from-region visibility. And the hope is also that the PVS will be tighter than with conservative strategies. Approximate methods are usually based on probabilistic sampling approaches (but not necessarily at the viewpoint level).

Approximate methods can be derived from conservative techniques, but choose to trade conservativeness for a tighter PVS. For example, we can modify the image-precision occlusion culling presented in Section 1.1 by declaring that an object is hidden if less than e.g. 2% of the pixels of the bounding box is visible. We can see in Figure 2.6 that this strategy permits the detection of more occlusion, at the price of potential artifacts.

Finally, aggressive methods can be seen as the opposite of conservative techniques. They compute a subset of the visibility set (Figure 2.4). An object can never be miss-classified as visible. All invisible objects are eliminated, and some visible objects might be culled as well.
1. OCCLUSION CULLING

Figure 2.6: Aggressive occlusion culling. An approximate strategy might rightly determine $O_1$ as hidden, but miss-classify $O_2$ as hidden too.

Figure 2.7: The tree $O$ is occluded by the conjunction of $R_1$ and $R_2$.

1.6 Occluder fusion

The shadow-frustum occlusion culling method exposed in the previous chapter has a major limitation: It declares an object hidden only if it is fully occluded by a single façade. Yet, objects are often hidden not by a single occluder, but by the cumulative effect of multiple occluders. In Figure 2.7, the tree $O$ is occluded half by $R_1$ and half by $R_2$. This is a simple case where two connected occluders together hide an object that none of them completely hides.

And there are more complex cases where two disjoint occluders interact to hide one object. This is the case in Figure 2.8 where $A$ and $A'$ together hide $C$ while none of them hides it alone. The capability to handle the cumulative occlusion due to multiple occluders is called occluder fusion. It is crucial to obtain tight PVS. From-point image-precision methods usually handle occluder fusion very well. Occluder fusion is particularly challenging to treat in the context of from-region visibility.
Figure 2.8: Occluder fusion. Neither A nor b completely occludes C, but their combined effect occludes it.

1.7 Occluder limitations and synthesis

Some methods have limitations on the kind of occluders they can handle. For example, the shadow-frustum technique presented in the previous chapter is limited to convex polygons. Convexity is the most common constraint, because it can reduce complex geometric computations on polygons into simpler computations on their vertices.

Moreover, some methods treat occlusion with respect to all objects in the scenes, while other handle only a small subset, usually for efficiency reasons: treating all the objects as occluders can be very costly, as we have seen in the previous chapter.

Another problem is that appropriate occluders might not be present in the input model. Consider our shadow frustum technique for large occluders. If the buildings are finely tessellated, with exquisite details for the windows, balconies, and façade, no large polygon can be present in the input scene to act as large occluder. To cope with these issues, methods have been developed to synthesize proper occluders. They will be covered in chapter ??.

1.8 Spatial acceleration

We have seen that testing every single polygon for occlusion is very costly. In the previous chapter, we have introduced bounding boxes to group polygons and perform the visibility test on a simpler surrogate. Bounding boxes are one example of spatial acceleration data structures. Spatial acceleration usually consists in organizing the scene into sets of polygons and providing a simple convex volume enclosing each set. The organization can be hierarchical: Bounding volumes are grouped and bounded by other volumes. The use of efficient spatial acceleration is crucial for occlusion culling, since it greatly speeds up the visibility tests by quickly rejecting large portions of the scene.
Important criteria include the tightness of the bounding volumes, the cost of the traversal of the structure, the memory cost, and if the structure imposes to split polygons or not. Spatial acceleration will be treated in Chapter ??.

1.9 Dynamic scenes

We call dynamic scenes, scenes in which objects are moving. They introduce more difficulty because everything has to be recomputed for each frame. Dynamic scenes make the use of preprocessing very hard. One has to be careful that even methods that perform the visibility calculation for each frame might still prove difficult to use with dynamic scenes. In particular, maintaining the spatial acceleration structure can be costly.

And obviously, pre-computation using from-region techniques can become much less attractive, unless some a-priori knowledge of the motion is available. In fact, the situation is not as bad as it seems. Scenes often contain a mix of dynamic and static objects. In a city for example, buildings are usually static while cars and pedestrians are dynamic. The occlusion caused by buildings can be precomputed and motion volumes are used to represent the possible motion of moving objects.

We will see that some algorithms treat only static occluders, while others can handle moving occluders. The update of the spatial acceleration structure is however rarely avoidable. A typical solution is to use separate structures for dynamic and static objects.

2 Shadow computation

2.1 Hard and soft shadows

In chapter 1, we have discussed hard shadows, that is, shadows cast by a point light source. In this case, a point of the scene can either see the light source or not: it is either lit or in shadow. However, most real-world light sources are not infinitively small. They are called extended light sources, and cast soft shadows (see Figure 2.9). A point of the scene can see the extended light source completely ($P_1$), partially ($P_2$), or not at all ($P_3$), leading to the definition of the lit region, the penumbra and the umbra.

The case of points within the penumbra is particularly interesting. The amount of light they receive depends on the visible portion of the light source, that is, on the view of the light source from the point (see Figure 2.9(b)).

To this date, most practical application only use hard shadows, because of the cost and complexity of soft shadow computation. However, the last few years have seen the development of a variety of efficient and robust soft-shadow algorithms, which we present in this book.

2.2 Sampling vs. object precision

Two strategies can then be distinguished that are valid for both hard and soft shadows: sampling vs. computing shadow boundaries. The shadow volume method discussed
Fig. 2.9: (a) An extended light source \( L \) casts a soft shadow. (b) the view of the light source from point \( P_2 \).

Fig. 2.10: Soft shadow boundaries.

above involves sampling: for each point of the scene.
This distinction is actually very similar to the image vs. object precision categories for hidden-surface removal and occlusion culling.

### 2.3 Precision and aliasing

because not computed in the same frame as the final view, precision is not the same

### 2.4 Dependence on particular rendering algorithms

Does the shadow method require a specific rendering algorithm?
Does it naturally fit into a given rendering framework?
Can routines be reused?

### 2.5 Can all objects be blockers? Receivers?

Some algo work only for specific receivers (ground plane)
some don’t treat self-shadows
3. DISCUSSION

2.6 Accuracy

Fake shadows
Fake penumbra

3 Discussion

Summarize
Announce the organization of the book.
I have a problem with the references. Should I put them in the next, or should I have a section “further reading”.

The present chapter is certainly the most theoretical in the book. Our goal is to step back and discuss analytical tools that provide deeper insights about the fundamental nature of visibility problems and properties. In particular, we try to understand what could be a “perfect” visibility data structure and why it is not practically usable. The good news is that these analytical solutions can sometimes be simplified or applied to special problems in a very practical context. They can also help to understand the limits of practical methods, and they can prove important sources of inspiration for improvement or development of original solutions.

This chapter is oriented towards a morphological or structural study of visibility. We study the global properties of visibility in a 3D scene, and how they interact. We are not really interested in solving particular queries such as “is this object visible from this point of view?” or “is this point in umbra or penumbra?” In contrast, we want to extract the structure of these visibility properties, such as “what are all the points from which this object is visible?” or “where are the limits of umbra and penumbra?”.

A structural study involves an analysis of the basic elements of the structure, and how they are related. We introduce two different approaches that reduce visibility to a basic property and study where this property is constant and where it changes. The first approach considers the qualitative structure of the view from a viewpoint. The second approach studies the object seen from a given viewpoint along a particular direction, i.e. along a ray. In both cases, visibility changes at so-called visual events, which are
a crucial concept to study visibility. We conclude the chapter with a discussion of 2D vs. 3D visibility, and we show why 3D visibility is much more intricate.

1 Visual events

Visual events describe all the visibility changes in a scene. They correspond to the appearance or disappearance of objects when an observer moves within a scene, they delimit shadow boundaries, and they are central to many more visibility properties. In this section, we first study visual events in the simpler setting of two-dimensional visibility, before discussing the more complex 3D case.

1.1 2D visual events

Figure 3.1 represents a simple 2D scene composed of two triangles. When moving from viewpoint $V_1$ to $V_3$, triangle $B$ starts occluding triangle $A$. If we move further to $V_5$, $A$ becomes completely hidden. These are called qualitative changes, in the view. We are not dealing here with a simple quantitative change in the projected coordinates, but with a major change in the structure of the view: An objects becomes partially or completely occluded. This is called a visual event. The visibility set changes only when a visual event is crossed.

In 2D, visual events occur when the viewpoint is aligned with two vertices of the scene. For example, $A$ becomes hidden at viewpoint $V_4$ that is aligned with $a$ and $d$. The locus of the visual event is the line $\langle ad \rangle$: the visual event occurs when the viewpoint is on $\langle ad \rangle$. Similarly, $A$ becomes partially occluded at $V_2$ when the viewpoint crosses the line $\langle cd \rangle$. The latter is called a separating line because the two triangles lie on its opposite sides, while $\langle ad \rangle$ is called a supporting line because they lie on the same side.
1. VISUAL EVENTS

There is a third kind of visual event, when the viewpoint crosses the line of the edge of a polygon. When going from \( V_1 \) to \( V_0 \), face \([ed]\) becomes visible when the line \((ed)\) is crossed.

In 2D, all the visual events are separating, supporting or edge lines joining two vertices of the scene. Their number is thus \( O(n^2) \), where \( n \) is the number of vertices. If the polygons have a bounded number of vertices (which is usually the case), then the complexity is also quadratic with respect to the number of polygons.

The locus of a visual event is usually not the whole line joining the two generating vertices. Occlusion has to be taken into account. Consider the example of Figure 3.2. There is no visual event when going from viewpoint \( V_1 \) to \( V_3 \), although line \((bc)\) is crossed. This is because Object \( B \) hides the appearance of edge \([bc]\). The locus of the visual event \( bc \) is the half line \([c'b]\). In the general case, the locus of a visual event is a line, a half line or a segment.

1.2 3D visual events

We now turn to the 3D case. Consider the scene in Figure 3.3. From viewpoint \( V_1 \), the church is partially visible behind the house. As the viewpoint goes down, the house eventually completely hides the church: A visual event has occurred.

It is easy to understand why such changes are important. During a walkthrough, this is when the visibility set changes, when objects appear or disappear. If the scene is transmitted through a network, this is when new geometry has to be downloaded. And if we replace the church by a lighthouse (Figure 3.4), then the visual event corresponds to a limit of umbra, since the diamond-shape light becomes completely blocked by the house.

In the above examples, the disappearance occurs when the apex \( v \) of the church tower or of the light is visually superimposed with the edge \( e \) of the roof of the house. This is why it is called an \( EV \) event. The locus of such changes is the wedge defined by \( e \) and \( v \) (see Figure 3.3 and 3.4). \( EV \) events are the first kind of visual events that we study in polygonal scenes. Assuming that polygons have a bounded number of vertices, there can be \( O(n^2) \) \( EV \) events in a scene of \( n \) polygons, because any edge-vertex pair can generate an event.

We can see that the disappearance \( EV \) event in Figure 3.3 and 3.4 corresponds to
Figure 3.3: Visual event. As the viewpoint goes down, the church disappears behind the house. This occurs when the apex of the church tower and the edge of the roof are superimposed in the view.

Figure 3.4: The visual event $e^v$ correspond to the limit of umbra.

a supporting plane between the church and the house. Similarly, a separating plane would correspond to the beginning of the partial occlusion of one object by another. This is similar to the importance of separating and supporting lines in 2D.

Visual events do not only correspond to appearance and disappearance of polygons. They also occur when the structure of the view is modified. Consider the example in Figure 3.5. When the viewpoint goes down from $V_1$ to $V_2$, the structure of the visible part of house $B$ changes, since edge $e'$ becomes hidden.

Formally, visual events are defined as changes in the structure of the Image Structure Graph or ISG. The image structure graph is the labeled graph defined by a view (Figure 3.6). For polygonal scenes, arcs of the graph correspond to visible portions of edge. Nodes correspond to visible vertices of the scene, or to apparent vertices.
1. **Visual Events**

![Figure 3.5](image)

**Figure 3.5**: Visual event. As the viewpoint goes down, the structure of the visible part of house B changes. The view from $V_0$ is not represented.

Apparent vertices are the intersections of two visible edges, and they are also called *t-vertices*. See Figure 3.6 for the image structure graphs of the previous scene. The graphs from $V_0$ and $V_1$ are isomorphic, they have the same structure although the quantitative coordinates of the vertices are not the same. In contrast, the graph from $V_3$ has a structure because edge $e'$ has disappeared. A class of equivalent ISG is called an *aspect*. The aspect changes at visual events.

### 1.3 Triple-edge events

$EV$ events are not the only type of visual events for polygonal scene. Qualitative visibility changes can be caused by the interaction of three edges: the so-called *triple-edge events*, or $EEE$. In Figure 3.7, the roof of the church disappears behind the two houses when the three edges $e_1$, $e_2$, and $e_3$ meet visually. $EEE$ events are more intricate be-
Figure 3.7: EEE event. (a) the roof of the church C is visible. (b) When the viewpoint goes down, the roof disappears behind the roofs of A and B. (c) The EEE visual event corresponds to a ruled quadrics.

Figure 3.8: EEE event in an axis-aligned scene. Box C will disappear when $e_1$, $e_2$ and $e_3$ meet.

cause their locus is not a planar surface, but a ruled quadrics (Figure 3.7(c)). Moreover, their number can be as high as $O(n^3)$ because any triple of edges can potentially generate a visual event.

The other bad news is that, contrary to a common belief, triple-edge events occur also in simple scenes composed of axis-aligned bounding boxes, as shown in Figure 3.8. The bottomline is: Triple-edge events are the nightmare of analytical visibility, because they are hard to treat robustly and because they cause a combinatorial explosion.

1.4 Visual events of smooth objects

We simply mention here that the definition of visual events are not restricted to polygonal scenes. In Figure 3.9 we show a simple example composed of two spheres. The
second sphere disappears when the two spheres share a tangent plane. This is called a tangent crossing event.

![Figure 3.9: Tangent crossing singularity. As the viewpoint moves downwards, the back sphere becomes hidden by the front-most one. At viewpoint (b) a visual event occurs (highlighted with a point): the two spheres are visually tangent.](image)

Another visual event for scenes of smooth objects is shown in Figure 3.10. It corresponds to the disappearance of an end junction and of a t-vertex. In this specific example, two visual events appear at the same time because of symmetry.

![Figure 3.10: Swallowtail visual event for a rotating torus. At viewpoint b, the end junctions and the t-vertices disappear.](image)

Visual events of curved objects can be studied using singularity theory. They depend on differential properties of the surface. Catalogues have been developed for smooth and piecewise smooth objects.

## 2 The aspect graph

Now that we are familiar with visual events and qualitative changes of visibility, we can introduce the aspect graph. It has been developed in computer vision in the context of model-based object recognition. The aspect graph proposes to enumerate and classify all the possible views of an object. The principle is to group all qualitatively equivalent views.

### 2.1 2D aspect graph

We have seen that a view changes qualitatively when the viewpoint crosses a visual event. Visual events partition the scene into cells where the aspect of the scene is
equivalent, that is, the Image Structure Graph (ISG) has the same structure. This partition is called the viewing space partition. Consider the example in Figure 3.11(a). The supporting and separating lines partition the plane into 18 different regions where the view remains qualitatively invariant. For example, in region VC, A is always partially occluded by B.

![Figure 3.11](image)

**Figure 3.11**: (a) Viewing space partition of a 2D scene composed of two segments. (b) Corresponding aspect graph.

The boundaries of the cells of the viewing space partition correspond to visual events. The vertices of these cells, that is, the intersection of two visual event, correspond to viewpoint where two qualitative changes of visibility occur at the same time. For example, from viewpoint V in Figure 3.11(a), both a and d, and c and e are visually superimposed. This yields a very unstable view, since in the neighborhood of V, B can be completely hidden, partially hidden on the left, partially hidden on the right, or partially hidden in the middle of A.

The *aspect graph* is the dual graph of the viewing space partition (Figure 3.11(b)):

- Each node corresponds to an aspect. A node is the dual of a cells of the viewing space partition.
- Each arc corresponds to a portion of a visual event that separates two cells of the viewing space partition.

The terms aspect graph and viewing space partition are often used interchangeably, although they refer to dual objects.

When the viewpoint goes from one cell of the viewing space partition to an adjacent cell, it crosses a visual event. This corresponds to going from one node of the aspect graph to another through the arc joining them.
The number of regions of the viewing space partition, and hence the number of nodes of the aspect graph, grows very fast with the number of objects, as can be seen in Figure 3.12. The size of the viewing space partition in 2D can be shown to be $O(n^4)$, where $n$ is the number of vertices. This is because the $O(n^2)$ visual events interact to form the arrangement. Each pair of visual events can intersect and generate a vertex of the partition. This means that if the number of vertices is multiplied by ten, the size of the aspect graph can be multiplied by 10,000.

![Figure 3.12: (a) viewing space partition of a 2D scene composed of two triangles. (b) Corresponding aspect graph. (c) Viewing space partition for three triangles.](image)

**2.2 3D aspect graph**

Before introducing the 3D aspect graph, we need to discuss the different projections from 3D to 2D used to define a view. In particular, it is important to understand the difference between perspective and orthographic projection, because they result in different aspect graphs. Some aspects that can be observed under perspective projection never occur in orthographic views, or have a null probability. For example, the view of a cube showing only two vertical faces is common under perspective projection. In contrast, it can be observed under orthographic projection only when the viewing direction is parallel to the top face, which corresponds to a visual event (Figure 3.13). A view showing only one or two faces of a cube has a null probability under orthographic projection, in contrast to perspective projection. Orthographic projection yields fewer aspects and is thus simpler. This explains that it has been more studied in computer vision than the perspective case.

Formally, the aspect graph is defined with respect to a given projection, (perspective or orthographic). The field of view is not taken into account, we consider full spherical views. The viewpoint space of the projection corresponds to the parameters of the projection: a point of view for perspective projection, and a direction for orthographic projection. The viewpoint space of perspective projection is the usual 3D space $\mathbb{R}^3$, while the viewpoint space of orthographic projection is the sphere of directions $S^2$, often called the viewing sphere.

We have seen that the viewing space partition is the partition of the viewing space
CHAPTER 3. ANALYTICAL VISIBILITY

Figure 3.13: Aspect stable under perspective projection but not under orthographics projection. (a) set of perspective viewpoint that see face 2 and 3. (b) Representative view. (c) An orthographic view shows 2 and 3 only when the view direction is parallel to face 1 and 6. (d) corresponding visual visual event under orthographic projection.

Figure 3.13: Aspect stable under perspective projection but not under orthographics projection. (a) set of perspective viewpoint that see face 2 and 3. (b) Representative view. (c) An orthographic view shows 2 and 3 only when the view direction is parallel to face 1 and 6. (d) corresponding visual visual event under orthographic projection.

2.3 Implications and applications

The implications of the aspect graph are parallel to those of visual events. A cell of the viewing space partition corresponds to a connected region of space where the visibility set if constant. In fact, it is event stronger, since the structure of the visible portions of objects is unchanged. This means that within such a cell, the view can be updated with no additional hidden-part removal, only the projected coordinates need recomputation.

As we have seen, visual events do not always correspond to the disappearance or appearance of objects. The aspect graph operates at a finer level than the visibility set. Different cells can have the same set of visible polygons. However, one can imagine post-processing the aspect graph to merge cells/nodes with the same visibility set. This would result in an exact and optimal from-region occlusion culling method, where the viewing cells are maximal connected regions of space with the same visibility set. This can thus be considered as the “perfect” from-region algorithm: The tightness of the PVS is always optimal. Unfortunately, because of the combinatorial explosion of visual events, the storage cost and computation time would be prohibitive.

We will see in Chapter 10 that researchers have adapted the concepts involved in
2. THE ASPECT GRAPH

Figure 3.14: Aspect graph of a convex cube. (a) Initial cube with numbered faces. (b) and (c) Partition of the viewpoint space for perspective and orthographic projection with some representative aspects. (d) and (e) Corresponding aspect graphs. Some aspects are present in perspective projection but not in orthographic projection, for example when only two faces are visible. The cells of the perspective viewpoint space partition have infinite extent.

the aspect graph to from-point occlusion culling. They obviate the need to treat EEE event and only maintain visual events in the neighborhood of the viewpoint. Their method can be seen as a linearized local version of the aspect graph.

The concept of aspect graph can also be applied to soft shadow computation. We explained in Chapter 2 that the amount of light reaching a point in penumbra depends on the visible part of the light source. We can restrict the Image Structure Graph and the notion of aspect to the extended light source. The Viewpoint Space partition then corresponds to a partition of the scene where the view of the light source is qualitatively equivalent. The visual events contain all shadow boundaries, including the limits of umbra and penumbra.

Consider the example in Figure 3.16. The ground plane is subdivided by the visual events generated by the light source $L$ and the blocker $A$. This subdivision is basically a viewing space partition, where the viewing space is restricted to the ground. In each cell
Figure 3.15: Aspect graph of a L-shaped polyhedron under orthographic projection (adapted from [?]). (a) Partition of the viewing sphere and representative views. (b) Aspect graph.

Figure 3.16: Viewing space partition and soft shadow. We show an EV visual event corresponding to an umba boundary. The aspect of the light source for a sample cell is represented in the inset.

of this subdivision, the aspect of the light source is constant. In particular, the aspect where the source is completely visible corresponds to the lit region, while the null aspect corresponds to the umbra. The other cell boundaries of the penumbra correspond
to second-order discontinuities of the illuminance function, that is, the derivative is continuous, but the second order derivative is not. We will discuss related approaches in Chapter ??.

2.4 Complexity

As we have seen, the aspect graph is a very powerful data structure. Unfortunately, its complexity grows very rapidly. Consider the example of the cube shown in Figure 3.14. For orthographic projection, the scene contains 3 visual events, corresponding to the appearance/disappearance of each pair of faces. The viewpoint space partition is generated by the arrangement of these visual events on the viewing sphere. It contains 8 cells, because of the intersections between these visual events. In general, the size of the aspect graph under orthographic projection grows as the square of the number of visual events, that is, \( O(n^6) \) for polygonal scenes because of the \( O(n^3) \) EEE events. This means that if the number of polygons of the scene is multiplied by 10, the size of the aspect graph may be multiplied by a million!

The situation is even worse for perspective projection. Because the visual events interact in the 3-dimensional viewpoint space, their arrangement grows cubically with respect to the number of visual events. Intuitively, consider the simpler example of a set of planes in 3D space. The intersection of each triple of planes defines a vertex of the arrangement of these planes, resulting in a cubic growth of the number of vertices. Since a 3D polygonal scene can contain \( O(n^3) \) visual events, this results in a prohibitive bound of \( O(n^9) \) for the size of the aspect graph.

The cells of the viewing space partition can be complex. They are not necessarily convex. This is obvious when they are bounded by the ruled quadrics defined by EEE events. Note however that this can also happen for cells bounded only by EV events.

These pessimistic results should not be taken as a failure of analytical approaches, but rather as a strong indication of the complexity of visibility itself.

2.5 Implementation

The aspect graph has been implemented and tested on objects composed of a small number of polygons. Most implementations deal with orthographic projection only. As expected, the size of the structure grows quite rapidly, but not as rapidly as the \( O(n^6) \) worst-case bound. A probabilistic study has estimated that for “normal scenes”, the number of EEE events is closer to quadratic than cubic. However, an \( n^4 \) or \( n^6 \) growth is still not practical.

The aspect graph is usually constructed by enumerating all the visual events, using nested loops on the edges and vertices. These events are then projected onto the viewing sphere, where their intersections are computed.

The aspect graph of smooth objects has been studied too, and implementations exist for algebraic objects defined as implicit surface (zero set of an algebraic polynomial function). See Figure 3.17 for an example.
3 Line-space visibility

We have studied the aspect graph, which encodes the qualitative properties of views. We now focus on a more atomic visibility property and study visibility in line space. Visibility can intuitively be defined in terms of lines: Two points $A$ and $B$ are mutually visible if no object intersects line $(AB)$ between them. It is thus natural to describe visibility problems in line space. For example, the set of lines going through a point describes the view from this point. This is illustrated by the popular ray-tracing technique, where a view is computed by computing the intersection of rays leaving the eye with the objects in the scene (Figure 3.18).

More precisely, many visibility queries are expressed in terms of rays and not lines. The ray-shooting query computes the first object seen from a point in a given direction. Mathematically, such a ray is a half line.

In the context of occlusion culling, an object is hidden from a given viewpoint $V$ if all rays from $V$ to the object are blocked. The definition is easily extended to from-region visibility: An object is hidden from a region of space if all rays leaving the region and going to the object are blocked. Some methods use a sampling of rays to perform occlusion culling, as will be discussed in Chapter 26.

Shadows are also naturally expressed as properties on rays. A point $p$ is in umbra if all rays from the light source to $p$ are blocked. It is fully lit if no such ray is blocked. In the case of an extended light source, if only a portion of the rays joining a point to the light, then it lies in the penumbra. We will see that soft shadows can be computed by sampling rays to the light source.

As we have seen, rays and lines are natural atomic objects to describe visibility.
3. LINE-SPACE VISIBILITY

Figure 3.18: A view is defined by the set of lines going through the viewpoint, as illustrated by the ray-tracing technique.

In this section, we focus on a qualitative morphological study of lines and rays. Since visibility is expressed in terms of line- or ray-object intersections, the basic idea is to group all the rays that hit the same object. Not surprisingly, visual events come into play to describe where the object hit by a ray changes. And because sets of rays are complex to study, we will introduce the concepts of dual space and parameterizations, which embeds the issue into simpler spaces.

3.1 2D dual space and the Visibility Complex

Sets of lines are complex objects that are hard to comprehend and hard to compute. In 2D, we can use the duality between points and lines, an elegant and efficient tool to study properties of lines. Duality permits the reduction of problems on lines to simpler problems on points. It is based on mapping lines onto points. We call dual space the 2D plane where lines have been mapped to points. We call primal space the original 2D plane. As we will see, the power of duality comes from the fact that reciprocally, points are mapped to lines.

Consider the following mapping. A non-vertical line $y = ax + b$ is mapped to the point $(a, -b)$ in the dual space. In Figure 3.19, we see that line $L_1 : y = x - 1$ is mapped to the point $L_1^* (1, 1)$. The set of lines going through a point in the primal space is mapped to a line in the dual space. For example, point $P$ is mapped to the line $P^*$ in the dual space. And since $P$ is the intersection of $L_1$ and $L_2$, $P^*$ is the line joining their dual points $L_1^*$ and $L_2^*$. And the transformation from the dual space back to the primal space is exactly the same. A line $b = xa + y$ in the dual space represents the set of lines going through the point $(x, -y)$ in the primal space.

Any property on lines in the primal space corresponds to a property on points in the dual space, and reciprocally. For example, the line joining two points in the primal is the intersection point of intersection of two lines in the dual: the line $L_1$ joining $P$ and $Q$ corresponds to the point $L_1^*$ of intersection between $P^*$ and $Q^*$.

Intuitively, these properties are valid because lines in 2D can be parameterized using two parameters $a$ and $b$, and because we manipulate linear expression. However,
vertical lines are not handled by this mapping. This limitation can be alleviated by considering the projective 2D plane, but we will not discuss this in the book. We refer the interested reader to any classical book on projective geometry, e.g. [? , ?]. In fact, the principle of duality is independent of the precise mapping. It simply states that any property or theorem on points can be translated for lines. We refer the interested reader to any textbook on computational geometry, e.g. [? , ? , ?]

Under this mapping, parallel lines are mapped to vertically-aligned points (Figure 3.20). The set of lines going through a segment is a double-wedge in the dual space. The apex of the wedge is the line of the segment, and the two boundaries are the lines dual of the two end points of the segment (Figure 3.20). The set of lines piercing two segments is the intersection of the lines intersecting each segment. In the example shown in Figure 3.21, it is a quadrilateral in the dual space. It can also be the union of two infinite regions (Figure 3.22). This occurs when a vertical line pierces the two segments.

Recall that in 2D, visual events correspond to supporting and separating lines. That is, they correspond to lines joining two vertices of the scene. In the dual space, visual
Figure 3.21: The set of lines going through two segments is a quadrilateral. The four dual vertices correspond to the four lines joining pairs of segment endpoints in the primal.

Figure 3.22: The set of lines going through two segments can be the union of two regions in the dual space (which can be seen as a quadrilateral that spans infinity).

events are the dual points of these supporting and separating lines. The dual point of a visual event is the intersections of the dual lines of the vertices of the scene. For example, in Figure 3.21, the supporting and separating lines of $[RS]$ and $[PQ]$ are mapped to the vertices of the quadrilateral describing the set of lines intersecting $[RS]$ and $[PQ]$.

Figure 3.23 represents occlusion in the dual space. The additional segment $[TU]$ causes occlusion between $[PQ]$ and $[RS]$. In the dual space, the dual of $[TU]$ overlaps the quadrilateral formed by the dual of $[PQ]$ and $[RS]$. If the quadrilateral is completely overlapped, there is full occlusion. This property has been exploited to perform from-region occlusion culling in 2D, as will be described in chapter ???. This reduces the problem of occluder fusion into a simpler overlap problem on 2D regions.
The dual arrangement

The dual lines of the vertices of the scene define the dual arrangement. The dual arrangement is the partition of the dual space along the dual lines of the vertices of the scene. A cell of the dual arrangement correspond to a set of lines in the primal space that intersect the same set of objects. For example, in Figure 3.24, all the lines whose dual is in cell $A$ intersect $[RS]$ and $[PQ]$. All lines in $C$ do not intersect any object.

The dual arrangement is a very useful concept because it captures coherence among rays or lines. It realizes our goal of grouping all the lines that "see" the same object. Note that the vertices of the dual arrangement are visual events. However, the dual arrangement describes all the intersections of a line, while most visibility queries are only interested in the first object intersected by a ray. And recall that we have seen that not all lines that join two vertices of the scene are visual events, because of occlusion. Similarly, we need to restrict the information present in the dual arrangement.

The 2D visibility complex

Pocchiola and Vegter [?, ?] have developed the 2D visibility complex which is a topological structure encoding the visibility of a 2D scene. It builds upon the dual arrangement but obviates its limitations. Since we are not interested in all the intersections
3. **LINE-SPACE VISIBILITY**

of a line, we decompose it into multiple segments separated by the intersection points. This way, we can “stop” at each intersection. However, this makes visualization in a dual space more complicated, since each line corresponds to a discrete set of segments. This will require the use of an additional pseudo-dimension to distinguish the different segments.

More formally, we deal with maximal free segments. These are segments of maximal length that do not intersect the interior of the objects of the scene. A maximal free segment has its extremities on the boundary of objects, it may be tangent to objects but does not cross them. A line is divided in many maximal free segment by the objects it intersects. For example, segments $L_1$ and $L_2$ in Figure 3.25 correspond to the same line, but are separated by segment $[PQ]$. A maximal free segment represents a group of colinear rays that see the same objects. The set of 2D maximal free segments is two-dimensional nearly everywhere, except at certain branchings corresponding to tangents of the scene. A tangent segment has neighbours on both sides of the object and below the object, as illustrated in Figure 3.25, where a branching occurs in the dual space at the dual of the two tangent segments $T_1$ and $T_2$.

![Figure 3.25](image)

**Figure 3.25:** Topology of maximal free segments. (a) In the scene. (b) In a dual space where lines are mapped into points (the polar parameterization of line is used).

The visibility complex is then the partition of maximal free segments according to the objects at their extremities. A face of the visibility complex contains a set of segments that touch the same objects. It is bounded by chains of segments tangent to one object (see Figure 3.26). The vertices of the face correspond to visual events. In contrast to the dual arrangement, the visibility complex takes occlusion into account, and all vertices of the complex correspond to visual events.

The 2D visibility complex has been applied to the 2D equivalent of lighting simulation by Orti et al. [?, ?]. The form factor between two objects corresponds to the face of the complex grouping the segments between these two objects. The limits of umbra and penumbra are the vertices (bitangents) of the visibility complex.
3.2 Lines parameterization in 3D

Unfortunately, 3D visibility cannot be expressed in such an elegant dual space. In 3D space, the duality principle applies to points and planes, not to lines. In 3D, lines have 4 dimensions. They can for example be parameterized by their direction \((\theta, \phi)\) and by the intersection \((u, v)\) on an orthogonal plane (Figure 3.27(a)). This parameterization has nice properties, but the use of spherical coordinates for the direction makes it non-linear. And it has a singularity for vertical lines, corresponding to the poles of the spherical coordinates.

Line can also be parameterized by their intersection with two planes (Figure 3.27(b)). Unfortunately, this parameterization is not as linear as one would expect. For example, the set of lines piercing a line is described by a bilinear function \([2]\). And the singularity that occurs for the lines parallel to the two planes defining the parameterization
3. LINE-SPACE VISIBILITY

cannot be handled using a projective space as in 2D. In fact, Lines in 3D cannot be parameterized in \( \mathbb{R}^4 \) without a singularity, which motivates the next section.

**Plücker coordinates**

While they cannot be easily parameterized in \( \mathbb{R}^4 \), adding one dimension can surprisingly make things easier. This is the principle of Plücker coordinates [?]. The equations are given in the appendix. Here we only outline the principles. One nice property of Plücker coordinates is that the set of lines which intersect a given line \( a \) is a hyperplane in Plücker space (its dual \( \Pi_a \); The same notation is usually used for the dual of a line and the corresponding hyperplane). It separates Plücker space into oriented lines that turn around \( \ell \) clockwise or counterclockwise (see Figure 3.28).

![Figure 3.28](image)

**Figure 3.28**: In Plücker space the hyperplane corresponding to a line \( a \) separates lines which turn clockwise and counterclockwise around \( a \). (The hyperplane is represented as a plane because a five-dimensional space is hard to illustrate, but note that the hyperplane is actually 4D).

Not all 5-uples of coordinates in Plücker space correspond to a real line. The set of lines in this parameterization lie on a quadric called the *Plücker hypersurface* or *Grassman manifold* or *Klein quadric*.

Now consider a triangle in 3D space. All the lines intersecting it have the same orientation with respect to the three lines going through its edges (see Figure 3.29). This makes stabbing computations very elegant in Plücker space. Linear calculations are performed using the hyperplanes corresponding to the edges of the scene, and the intersection of the result with the Plücker hypersurface is then computed to obtain real lines. In fact, the use of Plücker coordinates is one of the most efficient practical methods to compute ray-triangle intersections in ray tracing [?].

Let us give a last example of the power of Plücker duality. Consider three lines in 3D space. The lines stabbing each line lie on its (4D) hyperplanes in Plücker space. The intersection of the three hyperplane is a 2D plane in Plücker space which can be
CHAPTER 3. ANALYTICAL VISIBILITY

Figure 3.29: Lines stabbing a triangle. In 3D space, if the edges are well oriented, all stabbers rotate around the edges counterclockwise. In Plücker space this corresponds to the intersection of half spaces. To obtain real lines, the intersection with the Plücker hypersurface must be considered. (In fact the hyperplanes are tangent to the Plücker hypersurface)

computed easily. Once intersected with the Plücker hypersurface, we obtain the critical line set as illustrated Figure 3.30.

Figure 3.30: Three lines define a critical line set in 3D space. This corresponds to the intersection of hyperplanes (not halfspaces) in Plücker space. Note that hyperplanes are 4D while their intersection is 2D. Unfortunately they are represented similarly because of the lack of dimensions of this sheet of paper. (adapted from [?]).

More detailed introductions to Plücker coordinates and their use can be found in the books by Sommerville [?] or Stolfi [?] and in the thesis by Teller [?].
3. Visual events, critical sets and extremal stabbing lines

As we have seen, lines in 3D are more complex than their 2D counterpart. Before studying the extension of the visibility complex to 3D, we need to understand how visual events can be expressed in terms of lines.

We have seen that visual events occur when different features of the scene are visually superimposed. This means that there is a ray going from the viewpoint to these features. For example, when an \( EV \) event occurs, there is a line going through the viewpoint, the vertex and the edge (Figure 3.31). The wedge locus of an \( EV \) events can also be seen as the set of lines going through the edge and the vertex. Similarly, an \( EEE \) event occurs when the viewpoint is on a line going through the three edges.

The lines corresponding to a visual event have one degree of freedom. For example, the set of lines of an \( EV \) event can be parameterized by their intercept \( t \) on the edge (Figure 3.31). We say that this is a 1-dimensional set of lines, called a critical line set. The terms critical line sets and visual events are often used interchangeably, since they both describe the locus of qualitative changes of visibility.

The two lines at the extremities of such a 1D line set are called extremal stabbing lines. For example, in Figure 3.31, the line joining \( v \) and \( v' \) is the last line stabbing \( e \) and \( v \). A line going through two vertices has no degrees of freedom, we can say that it is a zero-dimensional set of lines.

The line \( vv' \) is also the extremity of an other critical line set, corresponding to the visual event \( e'v \). We say that the two visual events are adjacent at this extremal stabbing line. This allows us to understand how different visual events in the scene can be related, and provides a clue on the structure of visibility properties.

\( VV \) extremal stabbing lines are the simplest. Extremal stabbing line can also be generated by the interaction of a vertex and two edges. And the most general sort of stabbing line is generated by the interaction of four edges. There are two lines going through four lines in 3D (Figure 3.32). If you want to convince yourself, remember that the set of lines going through three lines is a ruled quadric. If we introduce a fourth line, it pierces the quadric at two points, corresponding to the two lines.
3.4 The 3D Visibility Complex and Skeleton

The visibility skeleton

We have seen that visual events can be seen as 1D critical line sets, and that they are adjacent to 0D extremal stabbing lines. This naturally defines a graph structure in line space, called the visibility skeleton. The visibility skeleton encodes all the visual events of a 3D scene and their adjacencies:

- A node of the skeleton corresponds to an extremal stabbing line
- An arc of the skeleton corresponds to a critical line set or visual event.

The graph of the visibility skeleton is different from the aspect graph, although the arcs correspond to the same entities. In fact, it is more relevant to relate the visibility skeleton to the viewing space partition. As we have seen, the viewing space partition is the arrangement generated by the visual events. A vertex of the viewing space partition is either the projection of a node of the skeleton or the intersection of the projection of two arcs (Figure 3.33).

This explains the larger size of the viewing space partition, \(O(n^9)\) for perspective projection, while the visibility skeleton is “only” \(O(n^4)\), because of the extremal stabbing lines generated by quadruples of edges. We can say that the visibility skeleton implicitly encodes the viewing space partition, without the actual arrangement of the visual events.

The visibility skeleton does not have a meaningful 2D equivalent. In 2D, visual events are single lines, and studying their adjacencies results in very different properties, as we will discuss shortly.
3. LINE-SPACE VISIBILITY

![Diagram of line-space visibility](image)

**Figure 3.33:** Viewing space partition and visibility skeleton. (a) Vertex $v_1$ of the viewing space partition corresponds to node $vv'$ of the visibility skeleton, while $v_2$ does not correspond to any node. (a) Viewing space partition. (b) Abstract embedding of the visibility skeleton.

### The 3D visibility complex

We give a brief description of a more comprehensive structure, the *visibility complex*. The visibility complex partitions all rays of space according to the object they hit. The boundaries of this partition include visual events, and describe the structure of visibility in line space.

A face of the visibility complex is a maximal connected set of rays that hit the same objects. Ray shooting can thus be thought of as finding which face of the complex a ray belongs to. Unfortunately, this is far from a trivial operation.

The object hit by a ray changes when the ray becomes tangent to an object. The boundaries of the faces of the visibility complex are thus sets of tangent rays. These sets of tangent rays have intersections, corresponding to rays tangent to two or more objects.

The visibility skeleton is in fact a subset of the visibility complex (hence its name). It corresponds to faces of dimension 0 or 1 (in terms of lines) or 1 or 2 in terms of rays.

### Implementation

The visibility skeleton has been implemented and used for lighting simulation. The skeleton is built by computing the extremal stabbing lines and deducing the critical sets (and thus visual events) using a catalogue of adjacencies. Potential extremal stabbing lines are generated by enumerating their generators (edges, vertices).

Different acceleration techniques are used in order to avoid enumerating all quadruple of edges. Durand et al. have tested scenes up to 1,500 polygons, and observed a running time growth slightly more than quadratic. This is better than the theoretical
58 CHAPTER 3. ANALYTICAL VISIBILITY

\(O(n^5)\), but it is still significant. Similarly, the size of the skeleton on these scenes grows quadratically.

The visibility skeleton has been used for lighting simulation using the hierarchical radiosity technique [2]. Radiosity subdivides the scene into patches and simulate the light transfer between all pairs of patches. Visibility has to be taken into account because objects might block the light transfer between two patches. The visibility skeleton permits the subdivision of the scene along shadow boundary, resulting in finer shadows. It also permits a more accurate calculation of transfer coefficients (called form factors).

![Figure 3.34: Images computed using erarchical radiosity and the Visibility Skeleton.](image)

(a) A scene with multiple sources. The skeleton construction took 2min 23s and the lighting simulation 8min. (b) A scene mainly lit by indirect light. The skeleton construction took 4min 12s and the lighting simulation 6min 58s. Note the shadows caused by indirect illumination, cast by the books on the back wall.

Unfortunately, the size and running time growth of the visibility skeleton, as well as robustness issues are challenging issues that currently obviate its practical use.

In chapter 12, we will see that the concepts of the visibility skeleton have been used to perform from-region occlusion culling in architectural environment.

4 2D vs. 3D visibility

Geometric problems are always harder in 3D than in 2D. Visibility is no exception, and it is actually worse than most cases. In this section, we outline the fundamental reasons that make the step from 2D to 3D visibility much harder than would be expected from an increment of 1 dimension.

4.1 A line is not a hyperplane in 3D

As we have seen, visibility is atomically tied to lines. Lines in 2D have very different properties than lines in 3D, and in particular they are not hyperplanes. A hyperplane is
4. 2D VS. 3D VISIBILITY

the generalization of the notion of plane. In 2D, hyperplanes are lines, in 3D planes, etc.
A hyperplane in dimension $n$ is a linear subspace of dimension $n - 1$. The hyperplanes
in dimension 3 are planes, not lines. And unfortunately, visibility can hardly be reduced
to properties on planes, it is fundamentally a notion related to lines.

Hyperplanes have interesting characteristics that make them very amenable to geo-
metric reasoning and calculations. They can be parameterized using the same number
of parameters as points: a line in 2D needs two parameters (e.g. its angle $\theta$ and distance
to the origin $\rho$), a plane can be parameterized using 3 numbers (e.g. two angles for its
normal and its distance to the origin). In contrast, we have seen that lines in 3D require
4 parameters. This corresponds to an increment of 2 dimensions compared to lines in
2D.

In fact, hyperplanes can be dualized with points. Neglecting vertical lines, a line
$y = ax + b$ can be mapped to the point $(a,b)$, and a non-vertical plane $z = ax + by + c$
can be mapped to the point $a,b,c$. We discuss here the 2D line/point duality since it can
be used for visibility computation. This mapping is linear and thus trivial to compute.
It maps a line to a point. The set of lines going through a point is mapped to a line in the
dual plane. We will study 2D visibility algorithms that take advantage of this duality.
Unfortunately, lines in 3D space cannot be dualized, and no linear parameterization
is possible in $\mathbb{R}^4$. Simple computation over 3D lines can be performed using Plücker
coordinates, but this requires to use a 6 dimensional projective space, and to project
back to the real line space.

But the more important property of hyperplanes for our purpose is separability. A
hyperplane separates the space into two halfspace. This property is very important for
example for visual events. A point can be only on one side or the other of a visual
event.

4.2 Complexity of the visual events

In 2D, visual events are portions of lines defined by the interaction between two scene
vertices. In 3D, even the simplest visual events, the EV events, are more complex
portions of planes in the shape of wedges. And we have seen that the locus of a EEE
event is a ruled quadric.

The cells of the viewing space partition or the cells of the visibility complex are
not convex in 3D. This mostly comes from the absence of the separability property. It
makes many algorithms much more involved, because one cannot rely on convexity to
simplify computation.

4.3 combinatorial explosion

While the jump from 2D to 3D adds only one dimension, the combinatorial of ana-
lytical visibility undergoes a significant leap. First, lines jump from 2D to 4D. Visual
events for polygonal scenes grow from $O(n^2)$ to $O(n^3)$

From $n^2$ to $n^3$, or from $n^4$ to $n^9$.

Blah blah
5 Summary

Summarize visual events, aspect graph and visibility skeleton

Bottomline: analytical visibility is costly
Bottomline 2: Visibility is complicated
announce where this material will be used
This chapter describes some basic tools that are fundamental building blocks for a variety of visibility techniques.

If we take a typical visibility method we can see it as being composed of multiple applications of individual visibility tests against the scene geometry. For example in the basic method described in Section (1.3), the shadow volume, representing the occlusion of a specific occluder, is tested against the scene polygons. This is repeated for all the different occluders as required. As testing every single polygon would be too wasteful, in Section ?? we suggest using a bounding volume (Figure 1.7) to enclose all the object polygons so they can be tested together thus accelerating the culling process.

From the above we can see that there are three main components required in implementing such a visibility algorithm. First we need to decide on the type of the basic visibility query that we will use as the building block and the data structure that will be used to represent it, such as the shadow volume used here, (Figure 1.4). Then we need some kind of acceleration data structures over the geometry of the scene, otherwise the method will never perform well enough. Finally we need a method for intersecting the query structures with the scene 1.5. Of course different algorithms and different application domains might favor different query and acceleration data structures. We will try to look at a broad enough spectrum to give a good feel of the available options.

In the rest of this chapter we will look at the three components above in turn. We will start by examining the different visibility queries and their associated data structures. They can be classified into three types: point-to-point, point-to-region and region-to-region. Then we will look at various acceleration data structures such as
bounding volumes, hierarchies and space partitioning trees followed by algorithms for intersecting the query with the scene. In the final section of this Chapter we present a collection of various acceleration techniques often found scattered in literature that could be helpful for a faster implementation, or even just provide some inspiration.

1 Visibility Queries

The kind of queries we most often need to answer in a visibility algorithm, whether that be a culling or a shadow method, fall in three categories: point-to-point, point-to-region and region-to-region. It is interesting to note that the three different cases, when seen in a dual line space, correspond to problems of different dimensionality. The first one is visibility along a line which has a dual of a single point and thus is zero-dimensional. In the second the origin is fixed but we have two degrees of freedom on the direction of the rays emanating from it, even if they are restricted and don’t span the whole sphere. In the dual space this corresponds to a line, two-dimensional. Finally the region-to-region is a four dimensional problem [?].

1.1 Point-to-Point

This occurs whenever we want to decide on the visibility between two distinct points in space (Figure 4.1), or find out what is visible from a point in a certain direction (Figure 4.2). A typical example is when we use ray casting from a specific viewpoint, however, it might also be employed when solving any type of from-point or from-region method where we want to sample the visibility. The answer to this query is often binary: visible or not, and possibly the id of the first object being intersected.

![Figure 4.1: Visibility between two points.](image)

Let A = (x,y,z) and B = (x,y,z) be our two candidate points, if the line segment AB, connecting A and B, does not intersect any object in the scene, then A and B are said to be mutually visible, Figure 4.1. Here the line segment plays the role of an infinitesimal ray of light passing between the two points. Indeed, it is more common in computer graphics to regard this line segment as a ray which is a directed line segment. The ray
1. VISIBILITY QUERIES

Figure 4.2: Visibility from a point along a certain direction.

has an origin and direction. It thus treats the visibility problem asymmetrically, where, say A is the origin and the vector B-A is the direction, Figure 4.2.

The parametric representation of the ray, which describes the locus of the points P along the ray is:

\[ P(t) = O + t \cdot D \]

Where O and D \( \in \mathbb{R}^3 \) are the origin and the vector direction, respectively, and the parameter \( t \) is commonly treated as time. A given point along the ray is associated to unique time that is proportional to its distance to the origin (note that \( t \) can be also negative). The ray emanating from A towards B can be written in a parametric representation as:

\[ P(t) = A + t \cdot (B - A) \]

Where the ray is normalized to the line segment AB with \( P(t = 0) = A \) and \( P(t = 1) = B \).

The above reformulation of the mutual point visibility problem as an asymmetric problem is the basis of the so called ray-shooting problem: Given a scene and two points A and B, determine whether the ray connecting A and B intersects any object of the given scene. In case there is an intersection, it is in many cases of most importance to determine the closest intersection point to the origin, which the point visible from the origin when looking toward B (or along the vector B-A).

In some algorithms the ray going through a pixel might not be constructed in the parametric form. For example if want to perform many ray queries that all start from the same viewpoint or go parallel to one direction then it is possible to use the graphics hardware to rasterizing the scene and read the closest intersection object from the contents of the pixels. In some other methods the rays are examined in the dual ray space (Section ??).
1.2 Point-to-Region

We have two cases falling in this category, both involving a view-point and a target region. The region might be any volume in space, although often it is represented by a single convex polygon. The first case is when we want to determine if a region is occluded or visible, when seen from the given viewpoint. The query is defined as a pyramid with tip at the view-point and base at the polygon, region A in Figure 4.3. The second case is when the region corresponds to an occluder or a shadow casting object and we want to know what is hidden behind it. The structure we use in this case is a truncated semi-infinite pyramid, shown as region B in the figure.

In both cases the sides of the pyramid are defined by the set of planes that go through the viewpoint and each edge of the target polygon. They are referred to as shadow planes and face outwards, away from the enclosed volume, as seen in Figure 4.4. To represent the planes we use the implicit form of the equation:

\[ax + by + cz + d = 0\]

The coefficients \(a, b, c\) in the equation above correspond to the normal \(n\) to the plane. Given three points \((v, v_i, v_{i+1})\) defining a shadow plane, we can obtain the normal by taking the cross product of any two vectors defined by the three points:

\[n = (v - v_i) \times (v - v_{i+1})\]

However, since we want the planes to face outwards, we need to be careful to do the cross product in the right order, counterclockwise. The remaining value \(d\) is then found as \(d = -np_i\), where \(p_i\) is any one of the three points.

To test if any point from the scene geometry lies in-front, behind or on the plane, suffices to substitute into the plane equation and check if it evaluates to \(> 0\), \(< 0\) or \(= 0\).

If we are creating the shadow frustum of an occluder then we can use the plane of the occluder to truncate it. This is not necessary though if we are processing the scene in a front-to-back order.

If the target region is a more complex object, rather than just a polygon, then either we do a separate pyramid for each face of the object or we use the edges of its silhouette (see Section 4.3) to form the shadow planes. In the latter case, the plane that truncates region B in Figure 4.3, is now defined by the furthest point of the silhouette, and it
has a normal vector from that point towards the viewpoint \([?]\). On the other hand, the base of pyramid A is defined by the closest point of the target region. (I will draw a picture and explain this better). These are of course conservatives structures. **Hmm, how about the case where we have big concavities, for B?**

### 1.3 Region-to-Region

Define what is a shaft between two axis aligned regions and how to compute it.

Another version of this also occurs in the case of shadows where we want to find the penumbra and umbra volumes extending behind an occluder for a given view area. These of course correspond to the partially and fully occluded regions.

### 2 Scene Data Structures

Acceleration data structures are an essential ingredient of any visibility culling method. In fact the very definition of culling is to be able to quickly reject large chucks of the scene without having to go down to testing the individual polygons. In all methods, the 3D geometry of the scene is arranged into an acceleration structure, typically a hierarchy. Many methods also build a hierarchy on the occlusion information to speed
up things further. For example in the Hierarchical Occlusion Map [?] or z-buffer [?] we have a 2D hierarchy in image space while in the occlusion trees of Bittner we have an occlusion hierarchy in 3D object space [?]. In this section we will look mainly at the scene structures.

It is very important to have an appropriate and well built hierarchy. In fact one can argue that the goodness of the hierarchy is in general more important in the case of culling than in, say, ray tracing. In the latter everything is done in software and the cost of intersecting a ray with a hierarchy node is small compared to the rest of the operations (ray-object intersection and shading calculations). It is worth doing an extra hierarchy test rather than a ray-object test. While for culling with today’s technology, where we can render millions of polygons per second on standard graphics cards, the cost of testing the extra node can be in some cases be higher than just rendering the objects with in it.

When deciding on our acceleration data structures and their parameters we should remember what we are trying to achieve. For example in a real-time application where the visibility is computed on-line, we want to maximize the frame rate and not necessarily maximize the culling rate. By having a finer spatial hierarchy going down to individual polygons, or even simple objects we can reduce the conservativeness of the culling algorithm, however the extra work required for classifying the nodes of the individual polygons will probably be more than that involved in actually rendering them.

Other factors to consider are the platform and the target scenes. A hardware accelerated method where the test of the nodes or bounding volumes is done on the card and passed back to the CPU could make the ”false positives” very expensive. Better fitting volumes might be preferable, even if they are more complex [?].

The basic step in all cases is to place each object into a bounding volume, so however complex it might be, it can then be tested in a few operations. From there on we have two options, either an object centered structure (a bounding volume hierarchy), or a space partitioning structure. In a bounding volume hierarchy each node entirely encloses the objects with in it but at any level siblings might (and often do) overlap in space. On the other hand in the space partitioning methods space is subdivided into disjoint convex regions, but any object might overlap more than one region. In either of the two approaches usually the bounding volumes and are the basic units used for the construction. We will consider first the different types of bounding volumes, then the look at the BV hierarchies and finally the space partitioning.

### 2.1 Bounding Volumes

The most fundamental concept is to group nearby elements and enclose them in some bounding volume. The bounding volume is first tested, which for that matter represents the set of $k$ elements that it bounds. The idea is that if the bounding volume is missed, that avoids $k$ intersections. This is a quick reject strategy.

There are many volumes we can use, some of the main ones are shown in Figure 4.6. An efficient bounding volume has two primary properties: (i) it tightly encloses its elements and (ii) it is simple enough such that it can be intersected with low computational cost. A somewhat less important property is that it can be constructed fast.
Depending on the visibility method and hardware configuration we are using one type of volume might be preferable over another.

In the context of Ray Tracing, Weghorst et al. [?] suggested the following formula for measuring the efficiency of a bounding volume hierarchy:

\[ C_t = n_v C_v + n_p C_p \]  

(4.1)

Where

- \( n_v \) is the number of ray-bounding volume tests
- \( C_v \) is the cost of each ray-bounding volume test
- \( n_p \) is the number of ray-primitive tests
- \( C_p \) is the cost of each ray-primitive test

This formula suggests that to minimize the cost we need to carefully select the type of volume and the depth of the hierarchy. Choosing a more complex but better fitting volume will reduce the \( n_v \) and \( n_p \) but it will increase \( C_v \). Having a finer hierarchy will reduce \( n_p \) but increase \( n_v \). An extra term can be added to the above to take in account possible dynamic changes to the hierarchy. For non-ray based occlusion culling this formula is still relevant if we interpret the first term as referring to the cost of query-bounding volume tests and the second term to the number and cost of primitives being classified visible and thus rendered.

Let us look with some more detail the most popular of the bounding volumes.

**Bounding Boxes**

Figure 4.6(a) shows an *axis-aligned bounding box* (AABB) enclosing a complex shape. The AABB is a parallelepiped with six faces, where each two faces are perpendicular.
to one of the main axes. Commonly, by a bounding box, one implicitly means, a minimum bounding box that enclose a given set. The minimum is in the sense that the bounding box cannot be further scaled down while still bounding all the elements. However, it might be that the bounding box could be rotated to a different orientation in which a smaller bounding box exists (see Figure 4.6(b)).

One the reasons that axis-aligned bounding boxes are very popular is simply because they can easily be constructed. An AABB is simply defined by the minimum and maximum coordinates among all the elements of the cluster.

A non axis-aligned bounding box, called oriented bounding box (OBB) or arbitrary bounding box, encloses a set of primitives of a general position more tightly. A tight bounding box, or a bounding volume in general, increases the chances that when the bounding volume is intersected so is also one of it enclosed elements. It is however, more expensive to test.

Brief description of how they are constructed.

**Spheres and Ellipsoids**

Description of the sphere and how to compute the best bounding sphere for an object.

Very efficient methods exist for testing a sphere both against a ray and against a frustum (Section 3) This can lead us to believe that spheres are the best bounding volumes. However, typically a bounding sphere does not provide a tight bounding of most typical objects or a group of objects. This claim is also holds for cubic bounding boxes, since usually the distribution of elements is not isotropic. Just like oriented bounding boxes are more tight so are oriented ellipsoids. The good news is that oriented ellipsoids can be easily tested for intersection with a ray, almost as easily as a sphere does. Moreover, defining an oriented bounding ellipsoid is as easy as defining an oriented bounding box, and it practice it would be more tight than an oriented bounding box. This because usually is defined to bound the ellipsoid.

explain how to define an oriented bounding ellipsoid.
2. SCENE DATA STRUCTURES

**k-DOPs**

These can be seen as an extension of the oriented bounding boxes. The OBB uses three pairs of parallel planes, arranged at right angles between them, to define the enclosing volume. In each pair (or *slab*) the two planes are facing away from each other (antiparallel) and their position along the normal is defined by the extreme point of the object in the normal’s direction. For the k-DOP (k Discreet Orientation Polytope), this idea is extended from 3 to an arbitrary $k$ number of slabs aiming to shave off more space than the OBB. These $k$ directions are usually fixed for all objects to make the testing more efficient.

2.2 Bounding Volume Hierarchies

Of course a typical scene, where culling will be applied, might consist of hundreds or thousands of components. Testing each one of them individually would still be too costly. To improve on that we often arrange the BVs into a hierarchy by grouping nearby elements together, thus we can apply the quick reject test to a larger part of the scene at once. At the other end, sometimes an object might consist of too many polygons, therefore we don’t need to restrict our selves to the BV of the object as the smallest component in the hierarchy. We can make subsets of the object polygons and continue the hierarchy within it, thus minimizing the problems mentioned in Figure 1.8.

In such a hierarchy nearby elements are grouped into clusters and enclosed by larger bounding volumes. This constructs a hierarchy tree of bounding volume, where each node of the tree bounds its descendants. Quite often, in practice the designer of the geometric data already provides their organization. However, we can also construct a BVH automatically. There are two approaches that can be take. Top-down or bottom-up. Top-down starts with all the objects in one group and splits them into smaller and smaller groups, while bottom-up starts with the individual objects and groups them together. Top-down methods are better at creating balanced hierarchies with smaller overlap between siblings.

There are a number of properties, desired in the final hierarchy, that can guide us during the construction. In any given subtree the objects should be near each other. This nearness is relative, the lower down the subtree the closer the objects should be. This helps to keep the volume of the nodes small and thus avoid false positives. The overall sum of all the bounding volumes should also be minimal. Finally it is more important that the top nodes are ”good” than it is for the nodes closer to the leaves [?].

**Pseudocode for the median cut method.**

One of the simplest methods is the median cut proposed by [?]. It’s a recursive top-down method and constructs a binary tree. At each level the objects are sorted along one of the main coordinate axis and they are split in two groups along their median. The recursion continues with the two new groups by subdividing along alternating axis. To make it easier one can use the centroid of the objects for the decision on each axis.

Klosowski et al [?] later suggested that for deciding which axis to use at each step of the subdivision we can use criteria such as the splatter - project the centroids on the three axis and find the variance on each, selecting the one with the greatest area or the
Goldsmith and Salmon in [?] suggested an iterative method where the BV tree is constructed incrementally, trying to keep to a minimum the cost of intersecting it with a random ray. This is done in the context of ray tracing. This costs depends on the geometric probability that a node will be hit given the ray already intersects its parent, plus the cost of traversing its children. This means that the probability of the nodes being hit needs to be minimised and since this probability depends on their surface area, that needs to be minimised (see Section 2.4).

To avoid an $N^2$ algorithm that would compare each object against each other, they add the objects into the BV tree one at a time. At each step they select the subtree that will have the smallest increase in surface area if the object were to be inserted as a child of it. They also give a measure of the conditional probability of a ray hitting the BV of a child node given that we know it already intersects the parent BV. This is an approximation and is computed as the surface area of the child divided by the surface area of the parent node. If at a certain node the object gives the same increase in two or more of the children then it can be inserted in both and the best one will be selected in the end.

Another method is to construct first a space subdivision, such as a kd-tree and then start from the leaves upwards clustering the objects.

The maximum number of children of a node in a hierarchy is called the degree of the tree denoted by $d$. If we assume that the number of objects is $n$, then the total number of nodes in a tree is at most $2n - 1$, with the maximum in the case of a binary hierarchy. The height of the tree is at least $\log_d n$.

### 2.3 Space Partitioning

The hierarchical bounding volumes described above are a data structure that provides information on the spatial relationship among the objects of the scene. Each volume is defined aiming to bind tightly its content so as to minimize false positive tests. An alternative approach is offered by the spatial subdivision schemes.
Uniform Grids

Assuming the scene is finite and can be bounded by a large (axis-aligned) bounding box. The space is subdivided into small rectangular regions called cells (they are also called voxels since by many means they are the 3D counterparts of the 2D pixels). These cells form a regular grid of axis-aligned rectangular regions, which are disjoint and tessellate the entire space (see Figure 4.9). Each cell contains a list of objects (or bounding volumes) that intersect the spatial region covered by it. Note that some objects can intersect more than one cell, and thus different cells may contain pointers to the same object. This spatial subdivision is an effective scheme for accelerating ray shooting due to the existence of very efficient ray traversal algorithms but they have found only the occasional use in non-ray based culling methods [?]. They are often used for representing the viewspace where their regularity makes it convenient to store information and trace the viewpoint through.

![Figure 4.9: A regular grid is more suitable when the objects are fairly evenly distributed](image)

There is an interesting question here. What resolution should be used for the space subdivision? There is no clear answer to that since it depends on many unknowns, like the scene complexity and density, the distribution of objects in space and it depends very much on the computer architecture used. However, it is important to understand the tradeoffs (see Figures 4.10 and 4.11). A fine resolution means: (i) less false positive tests since the space of each cell tends to be more occupied by the objects, and (ii) less intersections in general since each ray can potentially test only objects that are closer to the ray. However, a fine resolution also means (i) a larger spatial data structure, (ii) many small (redundant) empty cells and (iii) more time spent on the ray traversal.

Using a proper resolution makes the spatial subdivision data structure a very effective tool to accelerate ray shooting since it allows the ray to focus only on the object which are likely to be intersected by the ray. When the scene is dense, the inherent front-to-back order of the traversal provide an early termination and in fact a very effective occlusion mechanism.

Non-Uniform Spatial Subdivision

The above regular spatial subdivision is a uniform subdivision since the initial cell is partitioned into a set of equal sized cells by axis-aligned splitting planes. The partition
and thus the location of the splitting planes are done independently of the distribution of objects in the scene. The uniform partition assumes that the objects are uniformly distributed and evenly occupying the space. When the scene is unevenly occupied and there are large regions that are sparser than others. In such case, the objects are said to have a non-uniform distribution and a uniform subdivision produces many empty cells, see Figure 4.9 on the right. This suggests the use of non-uniform spatial subdivision schemes.

There are different possible forms of non-uniform regular subdivisions. A non-uniform grid can be defined by axis-aligned splitting places, which are positioned arbitrarily non-uniformly to better adapt to the distribution of the objects in space [?]. Dense regions are subdivided more than sparse region (do we need an example figure...
2. SCENE DATA STRUCTURES

Figure 4.12: A partition by a quadtree and its corresponding graph. A quadtree is the 2D counterpart of the 3D octree.

Other types of non-uniform grids can be defined using a hierarchical grid in which some cells of a uniform grid are recursively partitioned into another uniform grids. The common feature of such non-uniform subdivision is that they define cells that are always disjoint. While a non-uniform grid can fit better to the geometry, its traversal is more involved and more time consuming than the simple traversal of a uniform subdivision.

Other recursive non-uniform subdivision schemes are space partitioning trees such as the Octrees, k-d trees and BSP trees.

Octrees

This is the simplest of the partitioning trees. Assume a large cubic cell bounds the space. The cell is then recursively subdivided uniformly into eight smaller subcells. Each cell that is still too complex is subdivided independently to the other cells. This is an adaptive subdivision, where the size of the cells is adapted and fit to the density and complexity of the objects (see Figure 4.15). Such an adaptive subdivision scheme where at each step a cell is partitioned by three axis-aligned splitting planes is call an octree. This top-down subdivision can be represented as a tree where each node represents a cell and has eight sub-trees. A cell that is not subdivided is leaf of the tree, and the initial bounding cell is the root of the tree. Since each cell is partitioned uniformly into equal sized sub-cells, octrees have a simple structure and thus quite popular. On the other hand, the fact that the position of their splitting planes is fixed reduces the effectiveness of their fitness to the scene. Also the fact that each cell is by definition subdivided into eight sub-cells, they impose redundancy, Figure 4.13.

This kind of data structure is fairly common in culling methods [?] because of its simplicity and its reasonable flexibility.

Binary Partitioning Trees

A Binary Space Partition (BSP) tree is a recursive subdivisions scheme in which the splitting planes can have an arbitrary orientation and they split a cell into two sub-cells.
Like in the Octree, each subdivision is represented by a node in a tree. Here, however, the initial cell does have to be bounded and the initial splitting plane partition the space into two sub- spaces separated by the plane. Each sub-space contains the list of objects that are not disjoint with the sub-space (their intersection is not empty). Recursively, each sub-space is then further subdivided as long that it is too complex, where commonly, the number of objects in the sub-space defines the complexity, although the volume of space might also be a factor. Also the recursion may terminates if a maximum leaf depth criterion is meet. Figures x11 and x12 show a scene partitioned and the associated BSP-tree. It is important to note that the cells of this subdivision are always convex, and that when the initial scene is not bounded, so does the external cells which remain unbounded.
2. SCENE DATA STRUCTURES

BSP-trees have been extensively used for various applications due to their two important properties. First, they provide an effective space partition which fits better to the distribution of objects in the scene. Second, the tree associate with is binary and the algorithms associated with it are fairly simple. For example, to locate a given point, that is, to find the cell in which the point is located, requires a simple top-down traversal of the tree. At each node the point is tested with the partition plane associated with it, to determine the sub-space to continue the search. This is achieved by testing on which side of the partition plane the point is located. This is done simply plugging the point coordinates into the plane implicit equation and testing its sign.

The problem of creating an efficient BSP tree in fact is not at all trivial. Not only due to the \( n! \) different trees we have to select from but also because we don’t know exactly what is an efficient tree.

For a set of \( n \) initial polygons the upper bound for space and time complexity of the BSP tree is \( O(n^2) \) but the expected case is between \( O(n) \) and \( O(n \log n) \). There can be a great variation depending on the partitioning polygon selected at each node. Traditionally to select the root at each subtree a few candidate polygons are selected and evaluated by comparing them against the rest of the polygons in the subspace and computing some function.

There are two qualities to usually measured, size (number of resulting splits) and distribution (how equal the resulting subsets are). Using a weighted sum of these we can get the ”goodness” of the tree. This however depends on the application. For visible surface determination the balance of the tree is irrelevant since every node is visited once. Size is however important. For ray tracing, or any algorithm involving classifications, balance maybe more important than size. At the same time balanced trees are faster to build, even though this doesn’t reflect on the online performance. A different measure of efficiency is presented in [?]. Based on probability models. The idea is to keep the largest, more probable to visit cells on short paths.

An often tricky part to implement in the BSP building code, is the partitioning of a polygon by a plane. Special care must be paid to avoid problems due to limited machine precision. See [12][?] for a description and code on how to do this correctly.

The BSP tree can be built either by using partitioning planes defined by the scene polygons themselves or using any arbitrary plane. For example if we will have a BSP tree where the leaves contain objects, then we can do that, see [?]

Kd-tree

In most occlusion culling methods, whenever a space hierarchy of the scene is used, Kd trees seem to be the hierarchy of choice. In terms of effectiveness and complexity they sit in the middle of the hierarchy spectrum. They are less flexible than a full BSP tree but they are much faster and simpler to build, and provide faster access at run time. On the other hand octrees are even simpler to build and test but they are much more limiting than kd-trees. We sometimes see in the literature the Kd-tree being referred to as a BSP tree. Although technically correct, since it does actually perform a binary partitioning of space with axis aligned planes, we will reserve the word BSP for the general non-axis aligned case.

The process of constructing a Kd-tree is conceptually straight forward. Typically
With a quick comparison with Figure 4.15 we can see that a kdtree can produce a much more compact subdivision.

\[
\text{Kdt ConstructKdt} \ (\text{Object List } ol, \text{ Recursion Depth } d) \\
\{ \\
\quad \textbf{if} (\text{number of objects in } ol \leq N_{\text{max}} \textbf{ or } d == D_{\text{max}} \textbf{ or } \text{some other termination criterion is met}) \\
\quad \hspace{1em} \textbf{return NewLeaf}(ol); \\
\quad \textbf{else} \{ \\
\quad\hspace{1em} (axis, t) = \text{SelectPartition}(ol); \\
\quad\hspace{1em} (ol_l, ol_h) = \text{PartitionObjectList}(ol, axis, t); \\
\quad\hspace{1em} n = \text{NewNode}(axis, t); \\
\quad\hspace{1em} n->low = \text{ConstructKdt}(ol_l, d+1); \\
\quad\hspace{1em} n->high= \text{ConstructKdt}(ol_h, d+1); \\
\quad\hspace{1em} \textbf{return } n; \\
\quad \} \\
\}
\]

Figure 4.16: Outline of the kd-tree construction

it is a top-down approach starting with the whole space and subdividing recursively. Although bottom-up based on clustering of objects could also be possible and maybe even desirable for scenes with a certain configuration. For occlusion culling we tend to use objects as the primitives rather than going down to individual polygons. In fact we tend to deal entirely with the axis aligned bounding boxes of the objects, basing all our decisions on them. We start with the set of all objects and we recursively split them using axis aligned partitioning planes in two sub-sets at each step until a certain criteria is met, see Figure 4.16. There are then only two non-trivial issues to consider, how to choose the partitioning plane and when to stop the partitioning. These two together can define the shape and size of the resulting tree, which can make a big difference in the efficiency of it’s traversal and as a result in the efficiency of the culling method. We’ll look at these two issues in more detial in the following sections. (I will revise this final part once Section 2.4 is written).
Dynamic Scenes

One of the drawbacks of hierarchical data structures, and in particular the space partitioning trees, is that a lot of pre-processing is invested in their construction and this can be lost when the objects in the scene move. An extreme example is the case of a BSP tree constructed using the supporting planes of the scene polygons as the partitions. In such case when an object moves the tree is actually invalid and needs to be restored, [2]. Of course in other types of trees, such as the octree or the kd-tree the damage is no so profound but it can still be costly to maintain or might lead to deterioration of the performance.

Quite often the greatest part of the scene is static and only some objects have the potential of moving. If these objects and the region in which they will move are known then a bounding box enclosing this region can be inserted in the hierarchy. This is a conservative estimate of the object at any given time ... This volume can also be computed at run time and be made large enough so as to enclose the object for a certain time period. Mention Sudarsky & Gotsman [20] and Fredo [21]

This bounding can also be inserted in the hierarchy in a lazy manner.

2.4 Building a better hierarchy

Here we will some general ideas but concentrate mainly on kd-trees since they are a very common hierarchy in visibility algorithms.

3 Intersecting the Query and Scene Data Structures

In the first two sections we have seen some of the data structures employed for implementing the queries and for holding the scene geometry. Here we will go on to give some tools that can be used for determining intersection between the two.

3.1 Ray intersections

This is a very well studied area with a large number of algorithms in existence, mainly coming from the Ray Tracing literature [20]. We will not attempt a thorough review but rather just look at some representative methods. It is important to remember that what we are interested in our case is to know whether the ray intersects a specific object/volume or not. We usually do not need the intersection point.

Ray box intersection

One of the advantages of the bounding boxes and in particular the axis-aligned (AABB) ones is that it is fairly simple to test the intersection between a bounding box and a ray. In Kay and Kajiya[20], the box is defined as the volume enclosed by a set of three slabs, where a slab is a pair of anti-parallel planes. For the base case of the AABB these slabs are aligned with the three axis. It is enough to test if the ray intervals defined by
the intersection of the ray with each slab overlap. If they do then they ray hits the box otherwise it misses it, Figure 4.17. This can also be applied to oriented bounding boxes.

![Figure 4.17: The ray intersects the box, if the intervals \((t_{\text{min}}, t_{\text{max}})\) and \((t_{\text{min}}, t_{\text{max}})\) overlap.](image)

Alternatively, when the bounding box is oriented, the ray can be first rotated to coordinate system defined by the three (orthogonal) axes of the box. Then the aligned algorithm can be applied, and the intersection point, if any, should then, be rotated back to the world coordinate system.

**Spheres and ellipsoids**

A sphere has a simple geometry and the ray-sphere calculation is easy. Usually, the intersection is applied using a parametric representation of a ray, where the coordinates are a function of \(t\), and substitute it into the equation of the sphere. This lead to a quadric equation in \(t\). If the determinate of the quadric is negative then the ray misses the sphere. Otherwise, either the ray touches the sphere only at a tangential point the determinate equal zero, or it intersects the sphere twice and the two roots of the quadric are real. Then the smallest positive value among the two \(t\)'s is the closest intersection.

While this calculation is simple, we can do much better since we only need to know whether the ray intersects the sphere or misses it. This is a binary decision and the calculation can be further simplified. All it requires is to calculate the distance \(d\) between the center of the sphere and the line equation of the ray, and compare it with \(R\) the radius of the sphere. If \(d \leq R\) there is an intersection and otherwise the ray misses the sphere. The distance between a point \(C\) and a ray with an origin at \(O\) with direction \(D\) is:

\[
d = \frac{|D \times (C - O)|}{|D|}
\]
Figure 4.18: On the left, if the determinant of the quadratic equation, resulting from the substitution of the ray into the sphere equation is positive then we have two points of intersection $t_1$ and $t_2$, while at if it zero we have one point. On the right, a faster approach, since we are not interested in the actual $t$ values, is to check if $d < R$. That is, the magnitude of the vector of the cross product between the vector $C - O$ connecting the sphere origin and the ray direction $D$. However, we can assume the ray direction $D$ is normalized, and instead of computing $d$, we can compute $d^2$ and compare this size with the $R^2$. Thus, we need to compute:

$$d^2 = |D \times (C - O)|^2$$

or explicitly:

$$d^2 = (D_x(C_y - C_z) - D_y(C_x - C_y))^2 + (D_z(C_x - C_z) - D_x(C_y - C_y))^2 + (D_y(C_x - C_z) - D_z(C_x - C_z))^2.$$

This calculation can be further simplified using the identity:

$$|a \times b|^2 = |a|^2 |b|^2 - (a \cdot b)^2$$

then the above cross product equation can be simplified to:

$$d^2 = |D \times (C - O)|^2 = |(C - O)|^2 - (D \cdot (C - O))^2$$

Note that this does not require any square root nor division calculations, and the ray parametric equation needs only to be normalized once when it is defined.

An oriented ellipsoid (see Figure x3(d)) can be defined as an affine transformation of a sphere. That is, it is a sphere scaled (stretched) and rotated. Thus, given an ellipsoid, we can compute the transformation $A$ that transforms the ellipsoid to a unit sphere. Now, given a ray $R$, say defined by two endpoints, we can apply $A$ on the
endpoints and test the intersection of the transformed ray with a unit sphere. If and only if the transformed ray misses the unit sphere so does the original ray and the oriented ellipsoid. This suggests that the oriented ellipsoid should be represented simply by the coefficients of the affine transformation that maps a unit sphere to its shape, so that the affine transformation is readily available for the intersection test.

**Ray intersection with a BV hierarchy**

The basic method for ray-hierarchy traversal is straightforward. First the root node is tested, if the ray misses it the traversal is finished, if not, then its children are tested. The recursion continues in a depth-first fashion. If a leaf node is reached then the geometry inside the bounding volume is checked.

A number of optimizations have been suggested over the basic approach, such as marking the objects as automatically hit or changing the order of the traversal, or flattening the hierarchy into an array to avoid the costly recursion.

**Ray traversal of a grid**

For a regular grid, the ray traversal mechanism proceeds from one cell to the next along the ray direction. In each step the ray visits a cell and test for an intersection with the list of objects pointed by the cell. If and only if the ray does not intersect an object in the current cell, it steps to the next adjacent cell. Thus, the gain is twofold. First, the ray avoids intersection tests with objects that are spatially away from the ray, and second the ray tests objects in a front to back order and as soon as it hits an object the rest of the traversal and tests are avoided.

A ray traversal algorithm steps along the ray path and visits the set of cells pierced by a continuous ray. By many means these algorithms are similar to 2D line rasterization algorithm. However, rasterization algorithm are meant to draw the line with a discrete set of pixels and they usually produce a set of pixels that do not cover all the continuous line (they skip some pixels that the continuous line pierces). This must be avoided for ray shooting since such skips might miss a cell that contains objects that the ray intersects. This implies that each time a ray leaves one cell, it is always to the adjacent cell that shares a cell face with the current cell. Specifically, a ray always leaves from one of three faces, either the front (back) face, left side (right) face or upper (lower) face, and this determines the face adjacent neighboring cell, and the sequence of cell adjacent cells along the ray traversal. Different traversal algorithms have been developed for ray traversal to accelerate ray tracing.

Let briefly presents one such line traversal algorithm which uses a parametric line representation \( \vec{x} = \vec{x}_0 + t \vec{V} \), or

\[
\begin{align*}
x_{0} + tV_{x} = y_{0} + tV_{y} = z_{0} + tV_{z}.
\end{align*}
\]

where parameter \( t \) is treated as time. The efficiency of the algorithm stems from the fact that the portion of time that it takes to cross consecutive grid lines along the \( k \) direction is constant denoted by \( \partial t_k \). Figure 4.19 shows the constant distance between the intersection points of the ray with vertical grid lines and with between horizontal grid
Figure 4.19: The red points are the intersection points of the ray with the vertical grid line, and the blue points with the horizontal lines. Note that the distances along the ray between consecutive points of the same color is the same. That sequence define the entry and leaving points between adjacent cells.

lines. Denote by \( e_k(i) \) the time when the line crosses the \( i^\text{th} \) \( k \)-grid line. Conceptually, we get three sequences of crossing times with the three main plains of the grid of cells:

\[
\begin{align*}
  e_x(0), e_x(1), & \ldots e_x(i), \ldots \\
  e_y(0), e_y(1), & \ldots e_y(i), \ldots \\
  e_z(0), e_z(1), & \ldots e_z(i), \ldots
\end{align*}
\]

Since \( e_k(i) - e_k(i + 1) = \partial t_k \), the sequences can be easily calculated incrementally. Merge-sorting the three sequences gives the order of crossing between adjacent voxels. Actually, the generation of the time sequences together with the merge-sorting are performed incrementally on the fly.

The line parametric algorithm is presented in Figure 4.20. Assuming fixed-point arithmetic, it requires two comparisons, two additions, and one auto-decrement-branch for each step.

Note that the initialization part of the parametric algorithm involves expensive computations (including square root and divisions) when generating the direction cosines (the normalization of the vector \((A, B, C)\)), and the \( \partial t_k \) values. The contribution of the initialization part of the algorithm becomes significant when many short lines (or rays) are generated.

Each time we step to the adjacent cell we have to check whether the step won’t move out of bound. To make the picture complete, we have to show how to initialize
while (1) {
  if there is an intersection in the current cell then break;
  if ($e_x > e_y$) {
    if ($e_x > e_z$) {
      $e_x += \partial t_x$;
      step to the X adjacent cell;
    } else {
      $e_z += \partial t_z$;
      step to the Z adjacent cell;
    }
  } else {
    if ($e_y > e_z$) {
      $e_y += \partial t_y$;
      step to the Y adjacent cell;
    } else {
      $e_z += \partial t_z$;
      step to the Z adjacent cell;
    }
  }
}

Figure 4.20: A parametric Ray Traversal Algorithm

the values of $e_k(0)$. Since the starting point is not necessarily aligned with the grid lines, $e_k(0)$ is the time when the line crosses the first $k$-grid line. For example, let $k$ be the $x$ dimension, and $x_0$ the $x$ coordinate of the ray origin, then $e_k(0) = (\lfloor x \rfloor + 1 - x)\partial t_x$, assuming the advances to the right.

**Ray traversal in a partitioning tree**

For hierarchical space subdivision we will look at the general case of the BSP tree. The Kd-tree, being also a binary tree is treated similarly although it results in much faster traversal times since the partitions are axis aligned and the intersection of the ray with them is much simpler. For the Octree there are also optimized routines [?].

Let’s assume that a large cell bound the scene. Given a ray with an origin at $O$ and that the ray intersects the bounding cell and leaves the cells at point $L$. Thus, the ray can be defined as directional line segment starting at point $O$ and ending at point $L$. The traversal of the binary space partition along the line segment is in fact a partition of that segment into an ordered sequence of sub-segments $S_1$ to $S_k$ where $S_i$ is the $i$th cell visited by the ray (see Figure 4.21). That sequence of segments is generated recursively through the insertion of the ray into the tree. At each step a given segment $S_j$ is associated with one of the cells of the hierarchy. That segment is tested with the
Figure 4.21: The initial ray $R$ is tested against the tree by recursively inserting it, starting from the root node. This is done in a top-down, near-to-far manner. Here we show how the ray is partitioned during the recursion with the "near" parts shown in red, and the "far" in blue. Note that in the bottom right figure, the blue part will be discarded since an intersection with the scene is found by the red piece.

plane that partitions that cell. The plane splits the segment into two parts $S_{j_1}$ and $S_{j_2}$ where one of them might be a void segment. These two segments are now associated with the two sub-trees of the BSP tree, or say, with two cells of the hierarchy. Then the two sub-segments $S_{j_1}$ and $S_{j_2}$ are recursively traversed in order. First, the sub-segment that is closer to the ray origin is traversed within the cell associated with it, and then the other sub-segment, if they are not void.

The recursive traversal in the front to back order is important since usually the traversal is searching for the closest intersection point of the ray with one of the objects of the scene. The above simple traversal scheme is a recursive top-down traversal, which has a very simple structure and it can be easily implemented. However, bottom-up traversals are also possible. Such traversals step from one cell to the next adjacent cell along the ray direction. Conceptually, such traversal is applied by stepping from one leaf of the tree to another. The ray leaves the current cell through one of the faces of the cell, in particular through $p$ the intersection point of the ray with that face. Stepping an $\epsilon$ distance from $p$ along the ray, yields the point $q$ which is now in the adjacent cell that shares a face with the current cell. By locating the cell that include $q$, the next cell along the traversal is found [?]. More sophisticated techniques, using neighbor links are also possible [?, ?].
RayIntersectBSP (BSP tree, Point O, Point L)
{
    if (tree is a leaf)
        return intersection with contained objects;

    if (O & L on same side of partitioning plane)
        return RayIntersectBSP(tree-
    else {
        pt = point of ray-partition intersection;
        hit = RayIntersectBSP(tree-¿near side, O, pt);
        if (hit = null) {
            (check geometry assigned to partition, if any);
            hit = RayIntersectBSP(tree-
        }
        return hit;
    }
}

Figure 4.22: Ray traversal of a binary partitioning tree

3.2 Shadow frustum tests

In a typical culling method where a frustum will be used we will need to know wether an object (or the bounding volume of it) lies totally inside, totally outside or it intersects the frustum boundary. We don’t really need to know the exact points of intersection. This makes for faster and simpler algorithms. For example a straight-forward way of testing an object against a frustum is to take the vertices of the object and test them in turn against the frustum planes. If all the vertices are found in front of a plane then we are done, the object is outside. If all of them are behind the plane then it could be inside but we need to test it against the other planes to make sure it is behind all before we can say for sure. Finally if some of the vertices are in front and some are behind the object could be intersecting but we need to go on testing to make sure it is not in front of any plane. It is of course easy to come up with more efficient methods than this, especially if they are tailored for specific types of volumes (boxes, spheres etc) 

Note that the above method only works if the frustum is a convex volume.

There are cases where we might need to find the exact part of the polygon that is inside or outside, for example in shadow algorithms. Although at first thought it might seem trivial to extend the above approach to find the intersection points when a polygon intersects a plane, in practice it is a bit more involved, with special cases and precision issues to be considered 

Frustum testing a hierarchy

Let us consider the case of a frustum to be tested against an object hierarchy or a space partitioning tree. Although both types of scene data structures are represented as trees (Figures 4.26 and 4.24), the testing takes a different approach. In the first we test
3. INTERSECTING THE QUERY AND SCENE DATA STRUCTURES

If we have a hierarchy of bounding volumes, say bounding boxes (BBs), and a frustum representing the view volume, then we do the following. Start from the root of the hierarchy and test the BB there against the frustum. The result can be one: Completely outside, in which case all the objects enclosed are outside and thus they can all be ignored for this frame, node A+B and node C in Figure 4.26. Completely inside in which case we don’t need to test any other of the volumes below the node, for example D+E in Figure 4.26. Or there is an intersection in which case we test each child of the root separately, e.g. the root or nodes C and F in Figure 4.26. If a leaf node gives an intersection then we need to render the enclosed geometry, nodes D and E in the figure.

If the scene is held in a hierarchical space subdivision, then the test is slightly different. Let’s take for example a binary tree such as a BSP tree or a KD-tree. Here at each node space is subdivided into two convex, disjoint subspaces by the partitioning plane. For the view volume culling, again we start traversing at the very top of the hierarchy. Once we establish that the overall subspace enclosing the scene is not entirely
outside or entirely inside we start the recursive descent. At each node we test the view volume against the partitioning plane. If the view volume lies entirely on one side, then we can safely ignore the subtree on the other side, for example the left children of \( H_1 \) and \( H_3 \) in Figure 4.24 are not traversed. If the plane intersects the view volume then we need to traverse both subtrees, as is the case with node \( H_4 \) in Figure 4.24.

**Mailbox**

In all the space subdivision techniques that we have seen, objects that lie in more than one cell are not split but rather pointers to the original object are stored in the relevant cells. This can be the source of problems if we are not careful. To avoid unnecessary work we need to be careful not to process the same object many times. If during the traversal of the hierarchy from a specific viewpoint, an object appears in several nodes that are currently visible, then we can render the object the first time we encounter it and then flag it so that it will be skipped from there on. At the next frame though, we want to render again the object the first time we encounter it, therefore just flagging it as rendered with one bit wouldn’t be enough unless we reset them at the beginning of each frame. To avoid the reset we can just mark it with the number of the current frame. Next time we encounter it if it is still in the same frame then we skip it, if it’s in a subsequent frame then we render it and mark it again with the new frame number.

The above issue is relevant both for ray casting and for frustum testing.

### 3.3 Shaft culling

### 4 Acceleration Methods (??)

#### 4.1 Ordering

It is quite useful to process the scene in front-to-back order. Where by front-to-back we mean that an element \( i \) that potentially blocks element \( j \) is considered in front of \( j \) and therefore comes earlier in the list. Note that this definition of front-to-back is not a strict distance measure but it is more related to a ray traversal. If we shoot a ray from the viewpoint in any direction then the elements that we will encounter will be in the same order (relative to each other) as they appear in the front-to-back order. Such an order might exist if it is relative to a viewpoint but it is much harder, if at all possible to get a full ordering with respect to a region.

This is relevant for the case where we have a space subdivision structure. For a bounding volume hierarchy, it’s more difficult [?]. We will look at the case of a binary tree, such as a BSP tree or a kd-tree. For the traversal of an octree see Sammet [?]

**Ordering from a point**

If the scene is built into a spatial subdivision down to the individual polygons then it is straightforward to order them. The code for the BSP case is given in Figure ???. However, this is not usually the case. In the applications that we will be discussing in this book it is common practice not to build a tree (whether that is a BSP, kd- or
void OrderTree(Tree node, 3DPoint viewpoint)
{
    if (node is a leaf)
        process the contained objects;
    else if (viewpoint in-front of node plane)
        OrderTree(front subtree, viewpoint);
        OrderTree(back subtree, viewpoint);
    else
        OrderTree(back subtree, viewpoint);
        OrderTree(front subtree, viewpoint);
    endif
}

Figure 4.25: Traversing the tree to get a front-to-back order

order-tree) using all the individual polygons in the scene but rather using the bounding volumes of the objects. In each cell of the space partitioning we might have several objects. The algorithm that follows gives an ordering of the nodes. If the nodes (eg the leaves) hold many primitives then we might need further processing to order within them, Figure 4.25.

The traversal is based on the fact that given a viewpoint and two sets of objects separated by a plane, the objects on the same (near) side as the viewpoint can obstruct but cannot be obstructed by objects on the other (far) side. The algorithm given in 4.25, compares the viewpoint against the root plane, traverse the near subtree first and then traverse the far subtree. The reverse, back-to-front, order can also be obtain with a simple modification of the algorithm.

This algorithm was first developed as a solution to the visible surface determination problem [?]. Given a BSP tree built with all the individual polygons, rather than the object bounding volumes, traverse it from any given viewpoint to get the back-to-front order of the polygons stored at the nodes. The polygons can then be displayed in that order using over-painting to cover hidden surfaces.

### Rough ordering

In the general case it is not possible to get an ordering of the objects with respect to a region. However, if we have a region that is not intersected by the plane of any of the partitions in the tree then we can.

Also possible to get a rough order.

### 4.2 Coherence and Lazy Evaluation

Here we will talk about Jiri’s [?] work on frame to frame coherence but also the trick mentioned in Muller and Heines on page 199-200 about storing the last plane it was outside.

Also say that you don’t need to test the nodes on the route to the nearest cell (not even the nearest cell) during the f-t-b, top to bottom traversal.
4.3 silhouettes

How do we compute the silhouette of an object as seen from a given viewpoint. What about a viewing area? Maybe even coherence?

4.4 Shadow Volume BSP tree

Although BSP trees are not usually the space partitioning of choice, for storing the scene in occlusion methods (there are exceptions [?]), they can be useful in representing the occlusion information. They can be also used as a shadow method. What we will see here works equivalently in the 2D image space as well as in the 3D object space.
5 Conclusion

In this chapter we reviewed some of the basic tools that are useful when implementing a culling method. Typically a scene which will employ culling will be composed of a very large number of primitives. As already described in Chapter [?], these are often clustered, using a bounding volume, into objects which can then be classified as visible or hidden. For culling we almost never go down to testing the individual object vertices, if a bounding volume is found to be partly visible then its contents are rendered.

In a scene where the number of objects is large, testing all the object bounding volumes will become unmanageable. Using a hierarchy reduces the complexity of the problem. There are two main classes of hierarchical data structures. Object subdivision (such as bounding volume hierarchies) clusters the objects that make up the scene, while space subdivision techniques subdivide space into non-overlapping regions, with each region holding the set of objects that intersect it.

In the visibility culling literature we can see a preference in using space subdivision techniques techniques rather than bounding volumes. This could be due to several reasons such as the fact that space subdivision techniques provide naturally a front-to-back or back-to-front visibility ordering which can help speeding up some of the algorithms. BV hierarchies can also be used to give a rough ordering in certain cases, eg when tracing a ray [??] but it is not as straight forward for culling and it is not a guaranteed correct order.
If any one of you is planning to write about robustness of geometric algorithms. Dani Halperin has a review which points to more reviews on this topic. See:


I hope this will help,

Danny

Mainly for shadows
1 Issues
1.1 Numerical precision
1.2 Degeneracy

2 Robust numerical computations
2.1 Perturbation
2.2 Interval arithmetic
Visible surface determination

maybe as an appendix

Philo, history, bottleneck, granularity

We briefly review the classical algorithms to solve the hidden surface removal problem. It is important to have these techniques in mind for a wider insight of visibility methods. We will however remain brief, since it is beyond the scope of this book to discuss all the technical details and variations of these algorithms. For a longer survey see [?], and for a longer and more educational introduction see e.g. [?].

Two categories of approaches have been distinguished by Sutherland et al. Image-precision algorithms solve the problem for a discrete (rasterized) image, visibility being sampled only at pixels; while object-precision algorithm solve the exact problem. The output of the latter category is often a visibility map, which is the planar map explain describing the view. The order in which we present the methods is not chronological and has been chosen for easier comparison.

Solutions to hidden surface removal have other applications that the strict determination of the objects visible from the viewpoint. As evoked earlier, hard shadows can be computed using a view from a point light source. Inversely, the amount of light arriving at a point in penumbra corresponds to the visible part of the source from this point. Interest for the application of exact view computation has thus recently been revived.
CHAPTER 6. VISIBLE SURFACE DETERMINATION

1 Hidden-Line Removal

The first visibility techniques were developed for hidden line removal in the sixties. These algorithms provide information only on the visibility of edges. Nothing is known on the interior of visible faces, preventing shading of the objects.

1.1 Robert

Robert [?] developed the first solution to the hidden line problem. He tests all the edges of the scene polygons for occlusion. He then computes the intersection of the wedge defined by the viewpoint and the edge and all objects in the scene using a parametric approach.

| figure | pseudocode |

1.2 Appel

Appel [?] has developed the notion of quantitative invisibility which is the number of objects which occlude a given point. Visible points are those with 0 quantitative invisibility. The quantitative invisibility of an edge of a view changes only when it crosses the projection of another edge. Appel thus computes the quantitative invisibility number of a vertex, and updates the quantitative invisibility at each visual edge-edge intersection.

| figure | how to compute intersections |

Markosian et al. [?] have used this algorithm to render the silhouette of objects in a non-photorealistic manner. When the viewpoint is moved, they use a probabilistic approach to detect new silhouettes which could appear because an unstable singularity is crossed.

1.3 Curved objects

Curved objects are harder to handle because their silhouette (or fold) first has to be computed (see section ?? of chapter ??). Elber and Cohen [?] compute the silhouette using adaptive subdivision of parametric surfaces. The surface is recursively subdivided as long as it may contain parts of the silhouette. An algorithm similar to Appel’s method is then used. Snyder [?] proposes the use of interval arithmetic for robust silhouette computation.

| figure | mesh case |

2 Image-precision methods

2.1 Scan-line

big family
Scan-line approaches produce rasterized images and consider one line of the image at a time. Their memory requirements are low, which explains why they have long been very popular. Wylie and his coauthors [?] proposed the first scan-line algorithms, and Bouknight [?] and Watkins [?] then proposed very similar methods.

The objects are sorted according to \( y \). For each scan-line, the objects are then sorted according to \( x \). Then for each \emph{span} (\( x \) interval on which the same objects project) the depths of the polygons are compared. See [?] for a discussion of efficient implementation. Another approach is to use a z-buffer for the current scan-line.

**Extensions**

Crocker [?] has improved this method to take better advantage of coherence.

Scan-line algorithms have been extended to handle curved objects. Some methods [?, ?, ?] use a subdivision scheme similar to Catmull’s algorithm presented in the next section while others [?, ?, ?] actually compute the intersection of the surface with the current scan-line. See also [?] page 417.

Sechrest and Greenberg [?] have extended the scanline method to compute \emph{object precision} (exact) views. They place scan-lines at each vertex or edge-edge intersection in the image.

Tanaka and Takahashi [?] have proposed an antialiased version of the scan-line method where the image is scanned both in \( x \) and \( y \). An adaptive scan is used in-between two \( y \) scan-lines. They have applied this scheme to soft shadow computation [?].

**2.2 Adaptive subdivision**

The method developed by Warnock [?] recursively subdivides the image until each region (called a \emph{window}) is declared homogeneous. A window is declared homogeneous if one face completely covers it and is in front of all other faces. Faces are classified against a window as intersecting or disjoint or surrounding (covering). This classification is passed to the subwindows during the recursion. The recursion is also stopped when pixel-size is reached.

The classical method considers quadtree subdivision. Variations however exist which use the vertices of the scene to guide the subdivision and which stop the recursion when only one edge covers the window.

Marks \emph{et al.} [?] presents an analysis of the cost of adaptive subdivision and proposes a heuristic toswitch between adaptive methods and brute-force z-buffer.

**2.3 The z-buffer**

**Z-buffer**

The z-buffer was developed by Catmull [?, ?]. It is now the most widespread view computation method.

A depth (or \emph{z-value}) is stored for each pixel of the image. As each object is scan-converted (or rasterized), the depth of each pixel it covers in the image is computed
and compared against the corresponding current z-value. The pixel is drawn only if it is closer to the viewpoint.

Z-buffer was developed to handle curved surfaces, which are recursively subdivided until a sub-patch covers only one pixel. See also [?] for improvements.

The z-buffer is simple, general and robust. The availability of cheap and fast memory has permitted very efficient hardware implementations at low costs, allowing today’s low-end computer to render thousands of shaded polygons in real-time. However, due to the rasterized nature of the produced image, aliasing artifacts occur.

The z-buffer is very simple and efficient because it solves the visibility problem at a pixel level, and reduces it to this atomic form: which object is closest for this pixel? Cycles issues are avoided, and the precise shape of the visible part is only implicitly handled. Moreover, this algorithm is very easy to implement in hardware, and the availability of cheap memory as made it a de facto standard for hidden-part removal. Scenes of millions of polygons can now be handled by commodity graphics hardware.

**A-buffer**

The A-buffer (antialiased averaged area accumulation buffer) is a high quality antialiased version of the z-buffer. A similar rasterization scheme is used. However, if a pixel is not completely covered by an object (typically at edges) a different treatment is performed. The list of object fragments which project on these non-simple pixels is stored instead of a color value (see Fig. 6.2). A pixel can be first classified non-simple because an edge projects on it, then simple because a closer object completely covers it. Once all objects have been projected, sub-pixel visibility is evaluated for non-simple pixels. 4*8 subpixels are usually used. Another advantage of the A-buffer is its treatment of transparency; Subpixel fragments can be sorted in front-to-back order for correct transparency computations.

The A-buffer can be credited to Carpenter [?], and Fiume et al. [?]. It is a simplification of the “ultimate” algorithm by Catmull [?] which used exact sub-pixel visibility (with a Weiler-Atherton clipping) instead of sub-sampling. A comprehensive introduction to the A-buffer and a discussion of implementation is given in the book by Watt.
Figure 6.2: A buffer. (a) The objects are scan-converted. The projection of the objects is dashed and non-simple pixels are represented in bold. (b) Close-up of a non-simple pixel with the depth sorted fragments (i.e., the polygons clipped to the pixel boundary). (c) The pixel is subsampled. (d) The resulting color is the average of the subsamples. (e) Resulting antialiased image.

and Watt [?].

The A-buffer is, with ray-tracing, the most popular high-quality rendering techniques. It is for example implemented in the commercial products Alias Wavefront Maya and Pixar Renderman [?] NO!! Renderman is REYES. Similar techniques are apparently present in the hardware of some recent flight simulator systems [?].

The A-buffer [?] was in fact originally developed in a scan-line system.

Most of the image-precision methods we present in chapter in this book are based on the z-buffer. A-buffer-like schemes could be explored when aliasing is too undesirable.

2.4 Ray-casting

The computation of visible objects using ray-casting was pioneered by Appel [?], the Mathematical Application Group Inc. [?] and Goldstein and Nagel [?] in the late sixties. The object visible at one pixel is determined by casting a ray through the scene. The ray is intersected with all objects. The closest intersection gives the visible object. Shadow rays are used to shade the objects. As for the z-buffer, Sutherland et al. [?] considered this approach brute force and thought it was not scalable. They are now the two most popular methods.

Whitted [?] and Kay [?] have extended ray-casting to ray-tracing which treats
transparency and reflection by recursively sending secondary rays from the visible points.

Ray tracing can handle any type of geometry (as soon as an intersection can be computed). Various methods have been developed to compute ray-surface intersections, e.g., [?, ?].

Ray-tracing is the most versatile rendering technique since it can also render any shading effect. Antialiasing can be performed with subsampling: many rays are sent through a pixel (see e.g. [?, ?]).

Ray-casting and ray-tracing send rays from the eye to the scene, which is the opposite of actual physical light propagation. However, this corresponds to the theory of scientists such as Aristote who think that “visual rays” go from the eye to the visible objects.

3 Object-precision hidden-surface removal

3.1 Depth order and the painter’s algorithm

The painter’s algorithm is a class of methods which consist in simply drawing the objects of the scene from back to front. This way, visible objects overwrite the hidden ones. This is similar to a painter who first draws a background then paints the foreground onto it. However, ordering objects according to their occlusion is not straightforward. Cycles may appear, as illustrated in Fig. 6.3(a).

![Figure 6.3](image)

**Figure 6.3**: (a) Classic example of a cycle in depth order. (b) Newell, Newell and Sancha split one of the polygons to break the cycle.

The inverse order (Front to Back) can also be used, but a flag has to be indicate whether a pixel has been written or not. This order allows shading and texturing computations only for the visible pixels.

**Newell Newell and Sancha**

In the method by Newell, Newell and Sancha [?] polygons are first sorted according to their minimum $z$ value. However this order may not be the occlusion order. A bubble sort like scheme is thus applied. Polygons with overlapping $z$ intervals are first
compared in the image for $xy$ overlap. If it is the case, their plane equation is used to test which occlude which. Cycles in occlusion are tested, in which case one of the polygons is split as shown in Fig. 6.3(b).

For new theoretical results on the problem of depth order, see the thesis by de Berg [?].

**Priority list preprocessing**

Schumacker [?] developed the concept of *a priori* depth order. An object is preprocessed and an order may be found which is valid from any viewpoint (if the backfacing faces are removed). See the example of Fig. 6.4.

![Figure 6.4: A priori depth order. (a) Lower number indicate higher priorities. (b) Graph of possible occlusions from any viewpoint. An arrow means that a face can occlude another one from a viewpoint. (c) Example of a view. Backfacing polygons are eliminated and other faces are drawn in the *a priori* order (faces with higher numbers are drawn first).](image)

These objects are then organised in clusters which are themselves depth-ordered. This technique is fundamental for flight simulators where real-time display is crucial and where cluttered scenes are rare. Moreover, antialiasing is easier with list-priority methods because the coverage of a pixel can be maintained more consistently. The survey by Yan [?] states that in 1985, all simulators were using depth order. It is only very recently that z-buffer has started to be used for flight simulators (see section below).

However, few objects can be *a priori* ordered, and the design of a suitable database had to be performed mainly by hand. Nevertheless, this work has led to the development of the BSP tree which we present in section ?? of chapter ??.

**Layer ordering for image-based rendering**

Recently, the organisation of scenes into layers for image-based rendering has revived the interest in depth-ordering *à la* Newell *et al.* Snyder and Lengyel [?] proposed the merging of layers which form an occlusion cycle, while Decoret *et al.* [?] try to group layers which cannot have occlusion relations to obtain better parallax effects.
3.2 Exact area-subdivision

Weiler-Atherton

Weiler and Atherton [?] developed the first object-precision method to compute a visibility map. Objects are preferably sorted according to their depth (but cycles do not have to be handled). The frontmost polygons are then used to clip the polygons behind them.

This method can also be very simply used for hard shadow generation, as shown by Atherton et al. [?]. A view is computed from the point light source, and the clipped polygons are added to the scene database as lit polygon parts.

The problem with Weiler and Atherton’s method, as for most of the object-precision methods, is that it requires robust geometric calculations. It is thus prone to numerical precision and degeneracy problems.

Curved objects

Krishnan and Manocha [?] propose an adaptation of Weiler and Atherton’s method for curved objects modeled with NURBS surfaces. They perform their computation in the parameter space of the surface. The silhouette corresponds to the points where the normal is orthogonal to the view-line, which defines a polynomial system. They use an algebraic marching method to solve it. These silhouettes are approximated by piecewise-linear curves and then projected on the parts of the surface below, which gives a partition of the surface where the quantitative invisibility is constant.

Mulmuley

Mulmuley [?] has proposed an improvement of exact area-subdivision methods. He inserts polygons in a randomized order (as in quick-sort) and maintains the visibility map. Since visibility maps can have complex boundaries (concave, with holes), he uses a trapezoidal decomposition [?]. Each trapezoid corresponds to a part of one (possibly temporary) visible face.

Each trapezoid of the map maintains a list of conflict polygons, that is, polygons which have not yet been projected and which are above the face of the trapezoid. As a face is chosen for projection, all trapezoids with which it is in conflict are updated. If a face is below the temporary visible scene, no computation has to be performed.

The complexity of this algorithm is very good, since the probability of a feature (vertex, part of edge) to induce computation is inversely proportional to its quantitative invisibility (the number of objects above it). It should be easy to implement and robust due to its randomized nature. However, no implementation has been reported to our knowledge.

4 Antialiasing

A buffer

supersampling
4. ANTIALIASING

different frequencies for shading and visibility
REYES
Distributed and or adaptive ray tracing (bias pb)
The purpose of this chapter is to present and review basic concepts of graphics hardware, which are used in the book. In particular, we are interested in explaining the impact of features that affect rendering performance from the point of view of optimizing visibility computations.

This chapter is not meant to be a replacement for an introduction to rendering, and the graphics pipeline. It assumes the reader has knowledge of the rendering process, and s/he is comfortable writing simple graphics programs.

1 Graphics Pipeline Overview

The graphics pipeline is a term that is used to identify the steps that are performed to generate pictures with a computer. The use of the term *pipeline* highlights the fact that the whole process can be naturally subdivided into disjoint functional units that can be executed in parallel.

One way to look at how the graphics pipeline works is to consider an application that generates a set of 3D primitives to be rendered. Conceptually, at each frame the calls to the underlying graphics API generates a model, which undergoes substantial processing (described below) to be rendered on the screen. Current graphics architecture (e.g., NVIDIA GeForceFX, ATI Radeon 9800) perform all the rendering operations completely in hardware. Recently, the term Graphics Processing Unit (GPU) has started being used for such architectures.
Figure 7.1 depicts different aspects of the graphics pipeline. (It is particularly helpful in establishing nomenclature.) A most important operation is rasterization, which plays a crucial role in the rendering process in that it turns continuous objects (e.g., triangles) into discrete ones. In a nutshell, rasterization is the process of sampling continuous objects and generating its pixelized parts, called fragments. Although this sounds simple, in fact, rasterization is quite a complex operation. One of the major problems is that the process of sampling is often plagued by artifacts (such as aliasing).

It is possible to classify most graphics computations in terms of whether they are performed in object or image-precision. Object-precision methods use the continuous representation in their computations. Image-precision methods on the other hand operate on the discrete representation of the objects when broken into fragments during the rasterization process. Depending on the properties of a given computation, it is often possible to take advantage of one representation versus the other. For instance, the z-buffer algorithm (described below) solves visibility at the fragment level, culling fragments that are invisible after the primitives that generated those fragments have been rasterized. It is also possible to perform visibility computations in object space, and other chapters in the book go into details on some such algorithms.

A detailed view of the architecture of a modern graphics pipeline is given in Figure 7.2. This figure is useful in understanding the relationships between the different data types supported by the hardware. In a high-level, one type of input is vertex data, which can be assembled into 3D primitives (e.g., triangles). As pointed out above, 3D primitives can be rasterized into fragments that undergo further processing in the per-fragment operations unit to be turned into pixels and are transferred to the frame buffer. A major feature of modern architectures is that the user can actually write programs that get executed for each vertex and fragment, instead of relying on a fixed-function pipeline, as those available in OpenGL 1.3 and DirectX 7. Such programmable capa-
2. THE GRAPHICS PIPELINE

It is useful to go back to Figure 7.1a, and explain some of the different boxes in detail. Since Clark’s 1982 seminar paper [?], graphics hardware architecture has undergone a tremendous evolution. Initially, the main purpose of the hardware was to accelerate rasterization operations. Then architectures evolved to the point that they could implement the complete rendering pipeline, including geometry transformation, clipping, rasterization operations, and visibility computations all in hardware. The architecture shown in Figure 7.2 actually goes a step beyond that, in that instead of fixed function geometry transformation, and fragment processing, modern GPUs actually introduce programmable functionality where the user can specify exactly what happens to the data as it is passed through the hardware.

2.1 Command Stage

As the application issue graphics commands, the first stage of the pipeline is supposed to keep track of them and properly generate 3D primitives. It is here that command buffering and interpretation, unpacking and format conversions, and several graphics state-related tasks are done.

The actual mechanism used for passing input to the GPU is extremely important. In fact, this is often a bottleneck on achieving top rendering speeds. With GPU getting faster, the cost of specifying what to do, and actually transferring the information to the GPU dominates the overall rendering time. Several mechanisms have been developed

![Figure 7.2: Block diagram of modern graphics architecture (based on the model used by NVIDIA Cg. In current architectures, users can write their own functions which are executed for each vertex and fragment as they are generated (see programmable boxes).](image)
for optimizing the way applications interact with the graphics pipeline, such as *display lists*, and *vertex arrays*.

Display lists can be seen as a way to specify a certain type of input once, which can then be used multiple times. Since display lists are transformed using the current matrix stack, it is useful in rendering objects that use a certain sub-piece multiple times (and it is possible to specify them hierarchically). When a call to execute a display list is sent, the actual geometry that constitutes the display list is passed down the pipeline. It is exactly the same as executing the set of commands that are part of it. One of the nice advantages of display lists (besides convenience) is that the graphics hardware will attempt to store them in its own memory (if enough memory is available), thus it is possible to considerably cut geometry transmission costs when using display lists.

Displays lists work quite well when geometry data can be reused across rendering calls. When this is not possible, and immediate-mode rendering is necessary, an efficient way to specify the input is to use vertex arrays. Here, instead of using function calls to specify each piece of input to the graphics board, the information can be collected in one or more arrays, and passed to the GPU with one (or few) function calls. This not only lowers the CPU cost associated with rendering, since this lowers the function call overhead, but it also enables the GPU to use more efficient ways to transfer data from main memory to its internal buffers.

### 2.2 Geometry Processing

As geometry trickles down the pipeline, it undergoes substantial processing in preparation for rasterization. On the traditional fixed-function pipeline, the processing is essentially as follows.

First, vertices (including their several components - note that besides space coordinates, vertices actually have lots of associated information, possibly including a normal, and potentially several texture coordinates) are transformed by the currently defined matrix stack. Lighting computations are also performed on the vertices to determine information (such as normalized texture coordinates and diffuse and specular components) to be used later in the pipeline. On modern GPUs, the operations performed on vertices are not actually fixed, but can be almost arbitrary. The simplest way to see the per-vertex operations stage is as a function that operates on its vertex input (the vertex information is simply a multi-value tuple) and returns an output vertex.

In general, for further processing, vertices are assembled into primitives such as triangles. Such primitives can undergo simple visibility operations. For instance, back-face culling can be performed at this stage. One of the advantages of performing it here is to avoid potentially expensive rasterization operations on invisible fragments. These primitives are often clipped to make sure they live inside the view frustum before further processing.

### 2.3 Rasterization

It is during rasterization (also called scan-conversion) that continuous primitives are converted into discrete representations. This is shown in Figure 7.4. The basic operation of the rasterization stage is to sample the geometry into fragments. In a nutshell,
2. THE GRAPHICS PIPELINE

Figure 7.3: Geometry processing (after Akeley and Hanrahan). Under a traditional rendering pipeline, it is here that geometry is transformed, clipped, and lighting computations are performed.

Figure 7.4: Rasterization (after Akeley and Hanrahan). Continuous primitives are broken up into discrete fragments, which are used throughout the rest of the rendering pipeline.

This is often done by linear interpolating the vertex coordinates (again, note that we mean all of them) across the higher-order primitive at the correct scaling given the current resolution and window sizes. Basically, the output of this operation is a set of fragments. Each fragment, which is associated to a given pixel coordinate of our image, is simply a tuple \( T \) of coordinates computed from the vertices of the scan-converted primitive. The tuple \( T \) includes color and depth as some of its components. Particularly important for visibility computations is the fact that fragments contain a depth value (see below). The computation of the fragments depends on hardware configuration flags, and they can be interpolated, or if flat shading is used, it corresponds to the color of the first vertex.

The average number of fragments generated for each triangle (primitive) is an important performance measure, and one that needs to be carefully considered when writing efficient code for graphics hardware. By considering the geometry-processing rate \( G \) and the rasterization rate \( R \), it is possible to compute the optimal size of an average of a triangle. On current architectures even though \( G \) is much smaller than \( R \), it is
Figure 7.5: Texturing (after Akeley and Hanrahan). Data in the fragments are used for computing, often through simple interpolation and table lookup, a final color and opacity for the fragment. Recent advancements allow complex computations where multiple texturing operations are performed.

quite easy to force the rasterization hardware to be the bottleneck and limit rendering performance. This can be accomplished by a few very large triangles. (For example, rendering a single triangle that covers a 1024 by 1024 window requires one four vertices. For each four vertices worth of work, the fragment stage needs to handle over one million pixels.) This has interesting implications, e.g., level-of-detail techniques generally improve the utilization of the geometry processing units by increasing the average triangle area. But they do not (necessarily) decrease the load on the rasterization engine, since, in general, the overall area of the rendered surfaces stays essentially same.

2.4 Texturing

After rasterization, all the operations now act only on fragments. Traditional texturing provides functionality for computing the color of a given fragment. The operation is akin to a table lookup (see Figure 7.5), and it normally uses the texture coordinates of the fragment as an index into a user-specified color table called a texture map. Recently, texturing has become more flexible by the introduction of programmability into the pipeline through the use of pixel shaders (also called fragment programs). At this time, such functionality lacks straightforward use in visibility computations.

Texturing operations are powerful. Particularly important is the notion of texturing as a general table lookups, i.e., it is possible to use the texturing functionality to define maps and the composition of maps. The basic approach for this is to use between one to three components of the variables defined in a fragment as a texture lookup that in turn returns a tuple with several (between one to four) components. This effectively defines functions with the following signatures $f : (x, y, z) \rightarrow (x, y, z, w)$. Furthermore, it is possible to use a technique called dependent textures for doing the composition of functions.

An important feature of the texturing functionality is the fact that it is possible to
Figure 7.6: Per-fragment operations (after Segal and Akeley). The way fragments contribute to the framebuffer is a complex set of steps. First, a fragment undergoes multiple culling steps, which can reject or accept a fragment based on a set of individual test. Then, logical and blending operations can possibly be applied to combine the fragments color contribution with existing framebuffer data.

It is important to note that although fragments and pixels are related, they are not one and the same. Each fragment is associated with a pixel, while each pixel might have several fragments that contribute to it. Computing a fragment’s contribution to a given pixel is affected by several factors, including the relative depth of the pixels and the order that the fragments are generated.

In order to understand exactly how fragments and pixel are related, it is important to understand the concept of OpenGL buffers. This concept generalizes the intuitive notion of a framebuffer. A framebuffer only needs to store a color per pixel in order to satisfy the minimal requirements needed to implement a simple display system. The piece of memory it resides needs to contain a raster image that is continuously painted on the screen at video rates.

OpenGL buffers generalize this concept by saving extra information on a per pixel basis. First of all, an OpenGL color buffer, not only stores color, but it also stores an alpha value per pixel. The alpha value is useful for representing transparent objects. On top of the color buffer, there are several other buffers, including a depth buffer, stencil buffer, and an accumulation buffer. We explain each of these in turn. The depth buffer, as the name implies, stores for each pixel a depth value. The stencil buffer stores a multi-purpose integer value that can be used for some tests (explained below). The
accumulation buffer can be seen as a specialized color buffer that can be used to save intermediate and accumulate intermediate result. Because of its use, it tends to have better precision (in the form of more bits per channel) than the actual color buffer.

Upon encountering a fragment, the fragment-processing unit (FPU) performs a series of tests and updates (on the respective buffers) before actually computing the fragment’s contribution to the corresponding pixel of a given color or accumulation buffer. Note that the FPU is not programmable, and only a fixed set of test and update operations can be specified. We refer the reader to the OpenGL reference manual contains a detailed list of allowable operations and their side effects. The tests are based on a fragment’s alpha and depth values and the content of the depth and stencil buffers. In general, the contents of the depth and stencil buffer can be modified in very simple ways. In any case, this functionality can be quite powerful.

**Fragment Operation: alpha test**

The alpha test is the simplest of all fragment operation. Basically, the hardware can be programmed to discard a fragment based on comparing its alpha value to a constant reference value. Allowable comparisons include testing for equality, less than, greater than, etc. No updates of any kind are performed here.

**Fragment Operation: depth test and update**

The depth test works by comparing a fragment depth value with the depth value currently stored on the depth buffer. As in the case of the alpha test, standard comparison operations are allowed. In the case that the fragment passes the depth test, its depth value can be used to replace the value stored in the depth value. (The decision whether to replace or not the stored value depends on a user-defined state variable.)

The usefulness of the depth test is obvious. By initially clearing the depth value of all pixels to infinity (a large depth), and setting the comparison function to *less than*, only the fragments closest to the viewer will survive after all the primitives have been rendered thus leading to the correct picture. This is the well-known z-buffer algorithm.

**Fragment Operation: stencil test and update**

The operations that affect the stencil buffer are similar to the ones available with the depth buffer, but slightly more complicated. One difference is that it is possible to use the result of the depth test as input to the action to the taken. Also, the update operation is more general and it is possible to do some simple arithmetic with the stencil value. In contrast, the one for the depth buffer is limited to replacement of the current value with the new depth value. This makes it possible to perform some very useful computations using combinations of depth and stencil buffers. (See next section for some examples.)

The stencil test works by comparing a fragment’s stencil value with the corresponding one stored in the stencil buffer. The comparisons are similar to the ones supported for the depth test. In case of the stencil buffer, it is possible to specify a function to be used for updating the stencil buffer depending on the result of the tests. It is possible to specify updates for three different cases: the fragment *fails* the stencil test; the
fragment passes the stencil test but fails the depth test; and the fragment passes both tests. The updates are simple operations, such as increment the stencil value with the fragment's one. For details see the documentation for `glStencilOp`.

### 2.6 Advanced Topics and Examples

In order to make the material more concrete and useful, this section discusses some practical issues related to visibility computations. For concreteness, we use OpenGL as the API for describing the algorithms.

Full running code for each example is available from the book website.

#### Enumerating Visible Triangles

A very useful computation is to actually be able to enumerate the visible triangles in a given scene. Simply drawing them with the z-buffer enabled will generate the correct picture, but for benchmarking (and other) purposes, it is sometimes useful to actually compute the visible triangles. A simple technique for doing this is to color all the triangles with different colors, then perform the rendering in the normal way. After the rendering is done, reading back and scanning the framebuffer gives us the visible set.

When developing occlusion-culling techniques, one would like to minimize the amount of non-visible geometry that gets rendered. By using the procedure described above, and comparing the number of visible triangles with the rendered ones, it is possible to determine how far from optimal a given technique is.

#### Computing Scene Depth Complexity

Another very useful computation is to find the depth complexity of a given scene from a given viewpoint. This is somewhat of a non-trivial computation, but with the help of the graphics hardware, this can be performed quite trivially. This is a particularly nice application of the stencil buffer.

The basic idea is to enable the stencil buffer, and to configure it to increase its values any time a triangle would project into a given pixel. In OpenGL, the stencil buffer can be configured as such:

```c
glStencilFunc(GL_ALWAYS, ~0, ~0);
glStencilOp(GL_KEEP, GL_INCR, GL_INCR);
```

Now, by simply drawing the triangles, the stencil buffer (on a per-pixel basis) will get populated with the depth complexity of each individual pixel. In order to retrieve the actual values, one can simply read back the stencil buffer from the GPU memory by issuing a `glReadPixels` call.

#### Occlusion Culling Extensions

One technique for performing the visibility queries is to use the HP occlusion culling extension, which is implemented in their fx series of graphics accelerators. This feature, which actually seems quite similar to the capabilities of the Kubota Pacific Titan
3000 reported by Greene et al.\[4\], makes it possible to determine the visibility of objects as compared to the current values in the z-buffer. The idea is to add a feedback loop in the hardware which is able to check if changes would have been made to the z-buffer when scan-converting geometric primitives. Basically, by simply adding instrumentation capabilities to the hardware that are able to count the fragments which pass the depth test, any architecture can be efficiently augmented with such occlusion culling capabilities. Since the functionality proposed by the different vendors is similar, in the rest of this paper, we concentrate on the HP implementation of such occlusion culling tests.

One possible use of this hardware feature is to avoid rendering a very complex object by first checking if it is potentially visible. This can be done by checking whether a bounding volume $bv$, usually the bounding box of the object, is visible and only rendering the actual object if $bv$ is visible. This can be done using the following fragment of C++ code:

```cpp
glEnable(GL_OCCLUSION_TEST_HP);
glDepthMask(GL_FALSE);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
DrawBoundingBoxOfObject();
bool isVisible;
.glGetBooleanv(GL_OCCLUSION_RESULT_HP, &isVisible);
glDisable(GL_OCCLUSION_TEST_HP);
glDepthMask(GL_TRUE);
.glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
if (isVisible)
    DrawGeometryOfObject();
```

The HP occlusion-culling feature is implemented in several of their graphics accelerators, for example, the HP fx6 boards. Although performing our visibility queries using the HP hardware is very easy, the HP occlusion-culling test is not cheap. In an HP white paper\[5\], it is estimated that performing an occlusion query with a bounding box of an object on the fx6 is equivalent to rendering about 190 25-pixel triangles. Our own experiments on an HP Kayak with an fx6 estimates the cost of each query being higher. Depending upon the size of the bounding box, it could require anywhere between 0.1 milliseconds (ms) to 1 ms. This indicates that a naive approach to visibility culling, where objects are constantly checked for being occluded, might actually hurt performance, and not achieve the full potential of the graphics board.

In fact, it is possible to slow down the fx6 considerably if one is unlucky enough to project the polygons in a back-to-front order, since none of the bounding boxes would be occluded. In their most recent offerings, HP has improved their occlusion culling features. The fx5 and fx10 accelerators can perform several occlusion culling queries in parallel\[6\]. Also, HP reports that their OpenGL implementation has been changed to use the occlusion culling features automatically whenever feasible. For example, prior to rendering a large display list, their software would actually perform an occlusion query before rendering all of the geometry.

Add material about Nvidia occlusion-culling extensions (that count the number of visible pixels; allow for batch executions).
2.7 Differences between DirectX and OpenGL

3 Summary

4 Further Reading

Add reference to material that discuss the evolution of 3d cards.
Part II

Point-based occlusion culling
CHAPTER 8

View volume and back face culling
CHAPTER 9

Image-Space From-point Occlusion Culling

1 Luebke and George
(here!)

2 HZbuffer

3 Extensions (antialiasing, coverage masks)

4 HOM
5 Wonka 99

6 Other implementations
   6.1 Hardware solutions HP, SGI, Bartz
   6.2 Renderman
CHAPTER 10

Object-Space From-point Occlusion Culling

1 Coorg and Teller
2 Hudson
3 BSPs Bittner
4 Hardly Visible sets
5 volumetric grid visibility
   (Or as a new chapter just after)
6 PLP
CHAPTER 11

Occlusion culling in entertainment industry

(or something like this)

1 Games

2 Flight simulators
Part III

From-region Culling
In this chapter we will visit visibility methods developed for the special case of architectural interiors. These methods exploit the prominent characteristics of architectural indoor scenes, namely the natural partition of the scene into cells, and that visibility occurs through openings, which are called portals, in this context. Typically, architectural interiors offer a lot of occlusion, and typically the visibility is quite limited to the immediate surrounding, while very rarely one can see remote geometry.

In the generic visibility methods we reviewed so far (from-point and from-region) all the objects of the scene were assumed to be a priory visible until an occlusion test defined them as occluded. Visibility methods for architectural interiors usually assume the objects to be a priory hidden, until they become visible through the portals. Of course, this does not have to be necessarily so, but maybe it is a natural choice as the size of the portals is relatively much smaller than the size of the occluders (the cell’s walls).

Historically, the first visibility methods were developed for these architectural environments. The work of Airey et al. \cite{airey1993} and Teller and Séquin \cite{teller1993} started the trend towards advanced occlusion culling techniques, and developed much of the foundation for recent work. Some of the key concepts introduced were the notion of “potentially visible sets” from a region of space, “conservative visibility”, and “densely occluded environments”.

Indoors methods can also be classified to from-point and from-region methods. More than in general scenes, any indoor walkthrough would gain from both. The from-region method can be used to generate a rough PVS quite efficiently. The from-point
method can then be applied on that PVS while still taking advantage of the characteristics of architectural indoor scenes [?].

1 Cell-to-cell visibility

The work of Airey et al [?, ?] proposes two different techniques for computing the PVS. A conservative technique which for each portal, computes whether it can see a given polygon in the model. His formulation of the algorithm leads to a \(O(n^3)\) algorithm. Also in his Ph.D. thesis, he describes several variations of a “sampling” approach, which roughly uses ‘ray shooting’ queries to determine visible geometry in some user-specified precision, and does not guarantee conservativeness in the computation of the visibility set (see Section ??).

The work of Teller is quite comprehensive, and covers different aspects of 2D and 3D visibility computations [?, ?, ?]. During pre-processing, the model is first subdivided into convex cells using a BSP tree. The main opaque surfaces, such as the walls, are used for defining the partitions and thus the boundaries of the cells. Smaller detailed scene elements are considered ‘non-occluding’ and are ignored at this step. Non-opaque portals, such as doors, are identified on cell boundaries, and used to form an adjacency graph connecting the cells of the subdivision. The adjacency graph is a powerful tool for traversing the cells in a front-to-back order through a sequence of portals. Each branch in a depth first search (DFS) traversal is such a sequence. See the example in Figure 12.1. The thick black lines are the walls that are used for partitioning into cells, and light gray are the portals. On the left, the adjacency graph shows which cells are directly connected through the portals.

The cell-to-cell visibility is determined by testing if sightlines exist that connect some point in one cell to some point in another. Actually, it is clear that if a line exists from one cell to another, it necessarily passes through a portal and thus we only need to determine if the portals are visible between them. For each cell, the adjacency graph is utilized to generate portal sequences, which are then 'stabbed' with the sightline. Finding the existence of a sightline that stabs a given 2D portal sequence can be solved efficiently using a 2D linear programming [?]. For example the tree on the right of Figure 12.1 shows the cells that are visible from cell A. The cells that are reached by the sightlines contain the potentially visible set (PVS) for a given cell.

2 from-point visibility

During an interactive walkthrough the cell-to-cell visibility can be further dynamically culled using the view volume of the observer, producing a superset of the visible scene data, the eye-to-cell visibility [?]. Testing each of the potentially visible cells with the following can refine the cell-to-cell visibility set:
Figure 12.2: Results from [?] showing the potentially visible set from a given cell. Courtesy of Seth Teller, UC, Berkeley.

- the cell is in the view volume
- all cells along stab tree are in the view volume
- all portals along stab tree are in the view volume
- a sightline within the view volume exists through portals

(* I hope to draw some figures to illustrate the above following Fig 9 from Teller91 paper*)

The above tests are in an increasing order of their computational cost, and in a decreasing order of their effectiveness. The first test is not too expensive, but it does not cull as much as the others. The last test is a sufficient condition, but it is the most expensive. One might decide to apply only some of these tests as a series of refinements. The geometry contained in each visible cell is then passed down the graphics pipeline for rendering.

(* above I suggest to two figures rather than one from teller91, figure 11 b and d, I think we need them in gray scale so they can be seen within the text*)

3 From-point cells and portals

Luebke and Georges [?] propose a from-point cells and portals technique, based on an earlier idea of Jones [?] and on the from-region methods discussed in the previous section [?, ?]. Instead of precomputing for each cell a set of potentially visible geometry, Luebke and Georges perform an on-the-fly recursive depth-first traversal of the cells using screen-space projections of the portals which overestimate the portal sequences, and performs conservative occlusion culling.

Their algorithm works as follows. First, the cell which contains the viewer is rendered, and its portals which are inside the view frustum are identified. Clearly, any remaining visible geometry has to lie inside the projection of those portals. The algorithm overestimates the portals by using the axial 2D bounding box of the projected vertices of each portal. Then, the same procedure is repeated for the cells adjacent to the portals. At each step, the new portals are clipped against the pre-existing portals therefore leading to smaller and smaller visible “windows”, until no visible portal remains.

This technique is simple and quite effective, and the source code (an SGI Performer library) is available for download from Luebke’s web page. ¹

(* here I will add a figure that illustrate this technique *)

¹Pfportals can be obtained at http://pfPortals.cs.virginia.edu.
4 extension to 3D ???

In [?, ?], Teller describes techniques for extending his original 2D framework to 3D environments, using a parameterization of line space. Doing so exactly, requires substantial mathematical sophistication beyond what is necessary for the 2D case, and it is beyond the scope of this survey to describe that material. We refer the interested reader to his seminal Ph.D. thesis [?].

However, he also proposes to compute a conservative approximation to the visible region [?, ?] through arbitrary portals in 3D. As each portal is added to the sequence, the separating planes bounding the visibility region are updated. These separating planes between the portals correspond to visual events. For each edge of the sequence of portals, only the extremal separating plane is considered. It is a conservative approximation because some complex non-planar visual events are not considered.

More recently, Jimenez et al. [?] proposed an alternative method to compute conservative visibility through a set of portals. Like Teller’s exact technique [?, ?], they perform computations in line space.
In the previous chapters various visibility culling techniques have been presented. In these techniques the occlusion is tested from a point. Thus, these algorithms are applied in each frame during the interactive walkthrough. An alternative is to find the potentially visible set (PVS) from a region or a viewcell, rather than from a point. The computation cost of the PVS from a viewcell would then be amortized over all the frames generated from the given viewcell. As mentioned before, the predictive capabilities of from-region methods are crucial for pre-fetching, especially in network applications.

1 Preliminaries

1.1 Definition of the problem

The from-region visibility problem can be formally defined as follows. Given a region of space $C$ and a set of objects $O$, we are required to compute a subset of objects $V \subseteq O$ that are visible from $C$. An object $o$ is said to be visible from a region $C$ if there exists a line segment $l$ whose end-points lie on the boundaries of $o$ and $C$, respectively, such that $l$ is disjoint in its interior from all objects of $O$. If this is not the case, $o$ is said to be occluded from $C$. The set $V \subseteq O$ is said to be conservative if the set $O - V$ contains only objects fully occluded from $C$. 

131
In almost all practical contexts it is assumed that the region $C$ is a parallelepiped (often referred to as the viewcell). It is easy to see that in order to find the objects visible from a parallelepiped in 3D, it is enough to find the objects that are visible from each of its faces. Thus, in some cases we will discuss the problem of “from-face” visibility (although we may continue to refer to it as from-region visibility), and use the term viewface to describe the view region.

It should be emphasized that the above definition means that it is enough that there is a single line segment (or let’s call it a ray-of-sight or ray, in short) connecting some point within the viewcell to some point on a given object $o$, to make this object visible. This implies that the problem is not trivial as it has a high dimensionality. For example, let’s take a look at Figure 13.1. It illustrates a simple 2D case, where the viewcell $C$ and the object $o$ are 2D rectangles. This cell-to-cell visibility problem requires testing the visibility of all the rays connecting the two cells. This can be reduced to testing the visibility of all the rays emanating from one of the faces of $C$ (the cell’s boundary edges) and ending in one of the faces of $o$. The space of all these rays is two-dimensional, one coordinate parameterized along each of the cell’s boundaries.

Extending to 3D, testing the visibility between two cells, is testing the visibility of all the rays emanating from the faces of $C$ and ending in some face of $o$. Two coordinates can parameterize the faces of a cell. That is, the rays from $C$ to $o$ can be parameterized by four coordinates. This means that detecting the objects that are visible from at least one point in a three-dimensional viewcell is a four-dimensional problem.

As we learned in Chapter ?? the problem is non-linear and the exact solutions to the from-region visibility determination problem are considered hard. In fact, no such solutions have explicitly appeared in the computer graphics literature, with the exception of [?]. Instead, as we shall see, researchers have concentrated on providing practical conservative algorithms that overestimate the set of visible objects.

### 1.2 Some basic observations

Consider a polygon $C$ and an object $o$. Imagine that all points on $C$ emit light, that is, $C$ is a spatially-extended light source. The umbra $U$ of $o$ relative to $C$ is a region in space where any segment connecting a point in $U$ to some point in $C$ intersects $o$ (Figure 13.3). The penumbra $P$ of $o$ relative to $C$ is a region in space where for each point in $P$ there is a segment that connects the point to some point on $C$ and intersects $o$ (Figure 13.3). In simple words the umbra and the penumbra consists of the parts that are totally and partially, respectively, in the shadow with respect to the light source $C$.

The umbra can be defined to be part of the penumbra, but it may also be defined as a separate region.

One of the reasons that from-region turns to be an interesting problem is that often the occlusion created by two (or more) objects is larger than the union of their individual umbrae. Figure 13.4 illustrates the case where the union of individual umbrae of the objects is insignificant, while the aggregate umbra of the object is large and creates a lot of occlusion. We can further distinguish between two different scenarios. Two occluders can have their umbrae intersect, as in Figure 13.5 (a). It may also be the case that they create a fully occluded area where only their penumbrae intersect, as in Figure
1. PRELIMINARIES

13.5 (b). Following Wonka et al. [?], we call the two scenarios *umbrae interaction* and *penumbral interaction*, respectively.

Apparently, penumbra interactions are harder to handle correctly than umbra interactions. As we shall see later on, there are from-region algorithms that detect only the umbra intersection case, ignoring the cases where occlusion is caused by the intersection of the penumbrae. The first from-region algorithms did not handle any kind of interaction, and only considered the occlusion of each occluder separately. Later in Section 2 we will cover these techniques.

The fact that from-region algorithms need to deal with the penumbrae of occluders and not only with their umbrae makes it a much harder problem than from-point algo-
134 CHAPTER 13. GENERIC FROM-REGION VISIBILITY

Figure 13.3: The umbra and a penumbra.

Figure 13.4: (a) The union of the individual umbrae does not occlude much. (b) The aggregate umbra.

...
Figure 13.5: The aggregate umbra of the two occluders is larger than the union of the two individual umbrae. The difference region, which is part of the aggregate umbra, but not part of the individual umbrae, is marked in gray pattern. In (a) the two occluders intersect, and yield the gray area in-between. In (b) the two individual umbrae do not intersect, but still together they yield the gray area.

Figure 13.6: The umbra of an occluder with respect to a viewcell. Testing whether an object is fully occluded within the umbra is not easy.

To make this last point clearer, let’s look at it from another view. Let’s consider a polygonal occluder and a polygonal view region. The umbra volume is defined by the set of half-spaces created by the EV pairs of supporting planes: one edge (E) at the view region and one vertex (V) at the occluder, or vise versa. This by definition is a simple volume bounded by planes. Similarly, the penumbra is created by the set of separating planes, and it creates a simple volume. Now, let’s consider the umbra and penumbra created by more than a single polygon. The aggregate penumbra is simply the union of the by the individual penumbra, and thus, by definition, the union of polygonal volumes is also a polygonal volume. The aggregate umbra, on the other hand, is not the union of the individual umbrae. Here two individual umbrae interacts
and three edges (one from the view region and two from two occluders) can define a
EEE event, which is not a simple polygonal plane, but a ruled surface.

To further emphasize the complexity of a from-region problem let us consider an
apparently very simple problem, the occlusion of a single occluder. In the general
case, this occluder is non-convex and thus, its umbra with respect even to a simplest
viewcell, is non-convex (see Figure 13.6). The answer to the query whether a given
object is occluded by a single non-convex occluder or not, is not trivial. As we shall
see later (Section 2), if the occluder is convex, it is then much easier to test for its
occlusion.

It is important to note the relation between the viewcell size and the occluder size.
This relation is better understood in 2D where we consider both the viewcell and the
occluder as line segments and the term “size” refers to their lengths. The supporting
lines connecting the endpoints of the two segments define the umbra cast by the oc-
ccluder. The separating lines (recall the definition of the supporting and separating lines
from Section ??) define the penumbra region.

Consider the case where the viewcell is larger than the occluder, as illustrated in
Figure 13.7(a). The umbra behind the occluder is finite. In contrast when the viewcell is
smaller than the occluder, as illustrated in Figure fig:finite-umbra(b), the umbra is semi-
infinite and extends away behind the occluder. Obviously infinite umbrae are larger
and occlude more. In other words, for a given occluder, umbra becomes larger and
more effective as the size of the viewcell decreases. Note also that as the viewcells get
smaller the difference between the umbra and the penumbra diminishes. Sure enough,
when the viewcell get infinitesimally small, it converges into a from-point occlusion. In
fact, the occlusion or the umbra from a segment is the intersection of all the from-point
umbrae along the segment.
1.3 Motivation

The above might raise the following question: If the occlusion from a region is always smaller than the occlusion from a point, why bother to compute a from-region visibility, which seems to be a much harder problem than a from-point visibility? Well, the answer is "coherence". There is coherence in the visibility of nearby points that can be exploited to increase the effectiveness of the visibility computation. Moreover, as we shall see, there are situations where the from-region visibility is a vital tool for pre-fetching due to its indispensable predictive capabilities.

From-point algorithms are applied in each frame during an interactive walkthrough. A promising alternative is to compute the potentially visibility set (PVS) from a region or viewcell, rather than from a point. The computation cost of the PVS from a viewcell would then be amortized over all the frames generated from the given viewcell. Typically, the viewcell is large enough so that during the walkthrough many dozens of frames are generated while the user is walking or traversing through the viewcell. This means that even if the computation time of the PVS of that viewcell is say, five times longer than a from-point PVS, still it is cost effective. The from-region PVS can be executed by a separate thread that precomputes and pre-fetches the PVS of adjacent viewcells, so that the rendering thread never has to wait for the current PVS to be available. However, this requires the from-region PVS to be computed, loosely-speaking, fast enough. While this seems far-fetched for true 3D scenarios, recent techniques devoted to 2.5D urban environments are fast enough to compute the PVS online. Finally, we should remember that a from-region PVS is an overestimation and it typically it is larger than a from-point PVS, and it takes somewhat longer time to render.

In network-based applications the from-region visibility technique is indispensable since there is another crucial factor involved - network latency. Let’s assume a remote walkthrough scenario, where a large three-dimensional virtual environment is stored on a server. The client performs an interactive walkthrough, via a remote network connection, with no a priori information regarding the environment. The client is assumed to possess the computational resources equivalent to those of a personal workstation, not being able to render a significant portion of the environment in real time, nor even fit it into its memory.

This scenario requires using a visibility streaming, where the server monitors the client’s (virtual) location during the walkthrough, and transmits the relevant portions of the scene to the client. To keep the client’s frame-rate high, the server has to ensure that the relevant PVS is always available at the client’s side. From-point visibility algorithms are not suitable for such networking applications, since they necessitate transmitting visibility changes in every frame, giving rise to unacceptable communication lags, no matter how fast the visibility is computed. To avoid the problem of lag due to the network latency, from-region visibility must be used. By transmitting the PVS of a region, the client is free to render the visible parts of the environment independently to the server for a large number of frames, giving enough time to the server to predict and pre-fetch the PVS of adjacent regions. By the time the client leaves a viewcell, the PVS of the next viewcell has already been transmitted, and the walkthrough proceeds smoothly with no latency.

A from-region visibility computation algorithm can considerably accelerate various
tasks that are notoriously time-consuming: radiosity computation [?], ray tracing and Monte-Carlo based global illumination algorithms [?].

anything else? what is the exact reference for that Haines 94?? or something better *)

1.4 Classification of methods

For a given polygonal scene and a region in space, an exact algorithm calculates the set of polygons visible from the scene with complete accuracy. Exact algorithms that treat general scenes have a very high computational complexity, and are impractical for three-dimensional realistic-sized scenes. Moreover, exact algorithms are analytical, and are prone to robustness problems.

The understanding of exact algorithms can be helpful as a means of giving insight into the nature of the problem, and as a reference solution that helps to evaluate the quality of non-exact algorithms. Exact algorithms can also be a basis for practical conservative or non-conservative algorithms. For example, Plantinga [?] executes an exact algorithm on a small set of specially selected occluders, and uses the results to infer something about the visibility of all the objects in the scene. Bittner [?] and Nirenstein et. al. [?] use Plucker coordinates to compute an exact visibility set. Due to a series of optimizations their methods are significantly accelerated.

So far, conservative visibility algorithms have received most attention from researchers in the graphics community. This is because they are much easier to compute than exact algorithms, without imposing a significant overhead on the rendering, which is usually accelerated with a z-buffer hardware. Note, that the rendering, and the z-buffering in particular, are always needed to resolve the visibility at polygon level.

It should be noted that although a PVS is computed from a region, each individual frame is rendered from a point. Even an exact visibility set of a given region is still a conservative approximation (a potentially visibility set) of the exact visibility set from any given point in that region. Thus, it seems that there is no good justification for computing an expensive from-region visibility exactly and not conservatively.

In the following we focus mainly on conservative from-region algorithms since they practically advantageous over exact algorithms. Non-conservative algorithms are also interesting. Removing the conservativeness requirement and allowing some visibility errors provides the potential for designing simpler algorithms. It can also allow trading-off of quality for speed, or ensure an error bound or a specified quality of approximation. Most non-conservative algorithms are probabilistic, and can generally be more robust and easy to implement.

When rendering a non-conservative visibility set, errors appear when this set does not include some visible object. The problem is more noticeable during animation or a walk through the scene. Using non-coherent, non-conservative visibility sets in adjacent regions can cause visible objects to pop on and off the screen. Non-conservative algorithms are inappropriate when rendering quality is important. Using non-conservative algorithms must be carefully considered since in special cases, for reasonable qualities, it is not clear whether they are always significantly faster than conservative algorithms. In Section 3 we will learn such non-conservative algorithms.
As we have seen above, the from-region visibility problem is much more difficult than the from-point visibility problem. This is true for conservative algorithms as well. To design a practical conservative algorithm, it seems necessary to either limit the type of scene it can deal with or the type of occlusion it considers. More specifically, some algorithms can only consider the occlusion caused by 2.5D occluders, while others consider the occlusion of full 3D objects. Some algorithms consider only the occlusion created by a single convex occluder, while others can fuse the occlusion of two or more objects. Among the latter, some algorithms can only fuse the occlusions resulting from the intersection of the umbræ, while others can also fuse the occlusion resulting from the intersection of their penumbrae. However, non-conservative algorithms can deal with 3D objects while fusing the occlusions caused by the intersection of the umbræ as well as penumbrae.

In the following we first discuss various conservative from-region methods classified by their domains:

- Conservative methods that consider the occlusion caused by individual convex occluders.
- Conservative methods that consider the occlusion of 3D objects fused by umbræ intersections.
- Conservative methods that consider the occlusion of 2.5D objects fused by umbræ and penumbrae intersections.
- Non-conservative methods.

A separate section is devoted to methods where the visibility is computed in a ray space. These techniques are conservative and usually treat only 2.5D scenes.

From-region visibility designed to the special case of architectural interiors are covered in Chapter ??.

Before moving on it is interesting to briefly survey the short history of from-region visibility techniques. The problem of computing the visibility from a region is not new. Previous work appeared as early as the early 90’s [?]. Effective methods have been developed for indoor scenes [?, ?], or narrow passageways [?], but the algorithms that compute the visibility set from a region for general scenes appeared only in late 90’s. The earlier work on out-doors visibility streaming [?] was motivated by the emergence of the Internet and the increasing need for networking applications in which from-region techniques are necessary in order to avoid communication latency. The first generic from-region [?, ?] techniques were conservative, where the occlusion was based only based on individual convex occluders. Later non-conservative techniques appeared [?, ?].

In the year 2000, a number of conservative techniques developed that performed occluder fusion [?, ?, ?, ?]. Since then more from-region techniques were developed aiming at online computation of the visibility from a region. These techniques gained efficiency by restricting themselves to considering only the occlusion cast by 2.5D objects [?, ?, ?]. All of these methods are covered in this and next chapter.
2 Strong Occlusion and Convex Occluders

2.1 Strong Occluders

Determining whether a given object $o$ is visible or not from a region $C$ is non-trivial since it requires expensive analytical calculations. It should be emphasized that it is enough that a single ray-of-sight from some point in $C$ hits $o$, and that $o$ must be determined as visible and be included in conservative PVS. That is, a conservative algorithm yields a PVS that includes at least all the objects that are at least partially visible from $C$. Note that the terms ”visibility” and ”occlusion” seem simply to complement one another. However, since we are interested in a conservative visibility set, we can tolerate false-positive errors in the computation of the ”occlusion”, and define (some) occluded objects as potentially visible. Instead of computing the visible objects, practically all the out-doors conservative algorithms compute the set of objects which is guaranteed of being occluded from any point within the region $C$.

One way of guaranteeing that no single ray emanating from some point in $C$ can hit a given object $o$, is by ensuring that $o$ fully resides within the umbra of some occluder $T$. In [?] the term strong occluder is used for an object $T$ that fully occludes $o$ from any point in a viewcell $C$. If no strong occluder is found for a given object $o$, then $o$ is said to be potentially visible and included in the PVS.

Following this terminology, occluded objects for which the algorithm does not find a strong occluder, are said to be weakly occluded. The PVS computed in [?, ?] is conservative, and includes only strongly occluded objects. Figure 13.8 illustrates in
2. STRONG OCCLUSION AND CONVEX OCCLUDERS

Figure 13.9: (a) There is no strong occluder. (b) There is a strong occluder between each part of the viewcell and occludee.

2D the cases where an object (a segment in 2D) has a strong occluder (in (a)) and where it is weakly occluded.

However, the 2D case can be misleading, since the 3D case is significantly more complex than what it appears in the 3D case. In 2D the occluders are simply-connected segments and thus their umbrae are always convex, while in 3D the umbra of a general non-convex object is not necessarily convex.

2.2 Convex occluders

To determine whether a given object is entirely inside an arbitrary umbra volume can be quite complicated and requires expensive computation. Even if the tested object is a bounding box or any other bounding volume, the computation may still be costly. However, if the occluder is convex then its umbra is convex, and it is significantly easier to determine whether a given object is inside the umbra of the convex occluder. Instead of volumetric operations, the convexity of the occluder and its umbra provide a means to realize the test with a series of simple ray-shootings. Moreover, when the occluder is not given, the ray-shooting also accelerates, finding a convex strong occluder of a given object in a large scene [?].

Given a point $v$ and a polyhedral object $o$ with vertices $p_i$, if all the rays connecting $v$ and $o_i$ intersect a convex occluder $O$, then $O$ is a strong occluder of $o$ with respect to $v$. Now, given a viewcell $C$ with vertices $v_j$, if $O$ is a strong occluder with respect to all $v_j$, then $O$ is a strong occluder of $o$ with respect to the viewcell $C$. This directly implies on a naive algorithm to find a strong occluder of a given object. Cast rays from each vertex...
of the viewcell to each vertex defining the object. Each ray reports a list of potential convex occluders that have been intersected by the ray. Then if the intersection of these lists is not empty, the object is reported as being strongly occluded.

Instead of casting all the rays connecting all the vertices of the viewcell and all the vertices of the candidate object, it is enough to use only a subset of these rays. Rays that do not lie on the convex hull defined by the vertices of the viewcell and the vertices of the candidate objects are redundant. In the 2D case, this simply means that instead of casting four rays connecting the endpoints of the segments representing the viewcell and the candidate object, it is enough to cast only the two supporting rays and not the separating rays. This is in fact an efficient implementation of the test whether an occluder that stabs the shaft connecting the viewcell and the candidate object completely blocks it [2]. Figure 13.10 shows all the rays connecting the vertices of the viewcell and the vertices of the occludee. In (a) a strong occluder exists since it blocks all the rays, while in (b) the occluder does not intersect all the rays.

Cohen-Or et al. [2] present a further optimization of the above algorithm. One of their aims is to reduce the number of objects each ray has to intersect. It is shown that in densely occluded scenes, the strong occluder is mostly found within the first five intersected objects or trials. Thus, it is better to test one occluder at a time. One ray, the leading ray, is cast from one of the viewcell vertices, and retrieves the first candidate encountered by the ray. The rest of the rays test their intersection with the candidate hoping to quickly classify the candidate as a strong occluder. If one of the rays fails to intersect the candidate, then the leading ray retrieves the next candidate, until either a strong occluder is found or the leading ray reaches the object.

An important question is what the probability is of a given object having a strong occluder and being just weakly occluded. It was shown that a convex occluder is effective only if it is larger than the viewcell [2]. This means that a technique based on the occlusion of an individual convex occluder can be effective only when the viewcell is smaller than the average potential occluder. This restriction is in many cases too
Figure 13.11: (a) The union of the individual umbrae is small and insignificant (the area that is in the aggregate umbra of these occluders is drawn in a diagonal pattern). (b) By placing an occluder in the umbra overlap of the left most occluders, a larger umbra is created (drawn in a diagonal pattern).

severe for real applications. For example, the objects in Figure ?? are smaller than the viewcell, and their umbrae (with respect to the viewcell) are rather small. Their union does not occlude a significant portion of the scene (see in (a)), while their aggregate umbra is large (see in (b)).

To alleviate this problem it is possible to subdivide both the viewcell and the candidate objects into smaller parts, $C_i$ and $o_j$, respectively. Now, if each possible pair $(C_i, o_j)$ has a strong occluder that occludes $o_j$ from $C_i$, then $o_j$ is guaranteed of being fully occluded with respect to $C_i$. The 2D case is illustrated in Figure 13.9. The segment $o$ is weakly occluded with respect to the segment $C$, but each sub-segment of $o$ has a strong occluder with respect to each sub-segment of $C$. By refining the subdivision more and more, the visibility test converges to an exhaustive point-to-point visibility test between $C$ and $o$. Thus, it is worth noting that there is always a sufficiently fine subdivision that results in the exact visibility set.

2.3 Narrow passageways

A somewhat similar approach is taken by Yagel and Ray for scenes that consists of narrow passageways [2]. For example, a cave with narrow tunnels, or similarly blood vessels (see Figure 13.13). Their technique is based on a regular space subdivision and applying a cell-to-cell visibility. Such scenes are densely occluded, where most of space is solid and visibility can occur only through the void space of the passageways. The cells of the subdivision through which the walls of the passages goes through are conservatively teated as void cells. Using a fine enough subdivision this does not add much overestimated visibility (see Figure 13.14).

The cell-to-cell conservative visibility is based on a raster approach. The algorithm uses a raster walk along the corridor between the two cells to determine the visibility.
Figure 13.12: (a) By placing another occluder in the umbra overlap of the right most occluders, a larger umbra is created (drawn in a diagonal pattern). (b) By placing another occluder in the umbra overlap of the larger umbrae the aggregated umbra is further extended (drawn in a diagonal pattern).

Figure 13.13: The void area (in white) is defined by the walls of passage within the solid area (in black).

between the two cells. If the raster walk is completed without encountering a non-void cell, then a visibility ray can pass through the corridor, and the end cell is defined as visible from the viewcell. Otherwise, solid cells were encountered. For each such solid cell, the algorithm checks whether this cell is part of a simply connected set solid cell that blocks the corridor. A simple test can just check the opacity of the two adjacent neighbors of the solid cells. If both are solid, it guarantees that the corridor is completely blocked (see, Figure 13.15). This is a conservative test, but fast. A more involved test, can check whether the solid cell is part of a simply-connected raster pass.
2. STRONG OCCLUSION AND CONVEX OCCLUDERS

Figure 13.14: The subdivision defines void cells (in white), solid cells (in black) and the cells (in gray) where the walls are defined are neither void nor solid, but are conservatively treated as void.

Figure 13.15: The raster walk is marked with circles. The state of the two vertically adjacent cells can be an evidence that the cell is part of a vertical blocker of the visibility corridor between the two end-cells.

that crosses the tunnel.

The above test is in fact a strong occlusion test, since it tests for the occlusion of the corridor by a single simply-connected blocker. There are of course cases where the occlusion is only guarantee by the combination of two or more blockers, which are not connected (see, Figure 13.16).

Note that here the view space subdivision is the same as the scene space subdivision. In one hand, the subdivision resolution needs to be fine to enable the definition of non-void cell as void cell without being overly conservative. On the other hand, this imply small viewcells, losing the effectiveness of a from-region computation. It also implies more cell-to-cell tests.
Figure 13.16: Although the corridor is blocked, each of the single blockers (c and d) is not a strong occluder.

3 Non-conservative visibility

So far we have seen only conservative visibility algorithms. As discussed in Section 1.4 approximate methods, or non-conservative allows some visibility errors. This relaxed requirement bears the promise of designing simpler or faster algorithms. A potential advantage of a non-conservative technique is the ability to handle scenes which cannot be handled by existing conservative algorithms. For example, a scene consisting of many very small occluders, or parts, it is likely that even the best conservative algorithms will greatly overestimate the visible geometry. Since this sounds promising, it is tempting to develop non-conservative visibility methods.

Maybe the simplest and most intuitive way to compute the visibility from a region is by applying a sampling method. Loosely speaking, all it requires is to sample the visibility of the scene from a number of points within the region, and define the from-region PVS as the union of the sampled from-point PVS. This can be realized by casting visibility rays from a random set of points within the viewcell toward the entity in question. For example, to test the visibility of a cell of some hierarchical representation of a scene, one can cast a series of rays from the viewcell toward the given cell. The first ray that hits the cell is a sufficient evidence that the cell is visible. If no rays hit the cell, then the cell is treated as being hidden. Since ray casting is a robust method and many acceleration techniques have already been developed for it, such a ray sampling method seems to be robust and potentially fast. The speed, of course, is a function of the number of rays that can be a reasonable evidence that the cell is indeed hidden from the viewcell.

The visibility sampling is inherently an under-sampling method (see Figure 13.17). It is enough that the cell is visible through a tiny passage for which only a small number of rays (maybe even a single one) leaving from a particular point in the viewcell pass through the occluders. These rays can always be missed by any sampling method. However, it is clear that by excluding special cases, a dense enough sampling will in general yield an exact sampling. Well, the crucial question is what is that density which is enough for an acceptable sampling. Another important question is whether that density won’t be over-redundant, making the sampling method extremely inefficient.

Sampling the visibility yields an aggressive visibility set. It never reports an occluded object as visible. Thus, the visibility set is usually a smaller set than the exact
visibility set, and never larger. One should bear in mind that a from-region visibility set is in sense always a conservative visibility set. Even an exact from-region visibility set is almost always a superset of the exact visibility set when it used from a point for rendering. So in a from-region context, the relative compact size of the aggressive visibility set is somewhat redundant. Recall that in any case, rendering a subset of exact visibility set does not avoid the need to remove hidden parts, since a visible polygon might be only partially visible.

Nevertheless, sampling methods are of interests and some attempts where made to exploits their simplicity, robustness and their capability to trade off speed for quality [?, ?]. As discussed above, a central problem in a sampling is defining the sampling density. A naïve solution is to apply a uniform sampling, where an under-sampling can cause too much error, and over-sampling can be overly expensive. An adaptive sampling seems to be an indispensable solution. Basically the idea is to sample at some initial rate, and then iteratively, based on some heuristics to decide whether the sampling rate should be increased to refine the visibility set.

To realize an adaptive sampling approach, the following question should be answered,

- What to sample?
- How to sample?
- How to refine the sampling?
- What is the error criteria?
- Can the error be bounded?

Basically the idea is to sample the visibility from a viewcell [?]. One can sample the visibility of the individual objects of scene [?], or the cells of some hierarchical
partition of the scene [?]. The sampling itself is applied by casting visibility rays from the viewcell. Gotsman et al. [?] cast rays that originate at random points distributed uniformly across the viewcell, towards random points over the target. Nirenstein et al. [?] sample the visibility by rendering the scene for a number of points within the viewcell and collecting from the frame buffer the indices of the visible polygons. This is equivalent to casting a large number of visibility rays from one origin. While Gotsman et al. cast a small number of rays towards many objects, Nirenstein et al. cast many rays towards each one of the triangles in the scene. The latter method seems attractive since the rendering is done by the hardware, still, it the collection of the indices of the visible polygons requires a slow access the pixels of the frame buffer. Also conceptually, the sampling of so many rays for a single origin raises the concern of an over-sampling.

Gotsman et al. refines the sampling by means of subdividing the viewcells. In their method, they partition the view space into a tree of viewcells and compute a list of visible objects for each such viewcell. They use a five-dimensional view space maintained by a kd-tree ($k = 5$), where three dimensions define the ray origin and two dimensions define the ray direction. The refinement of the sampling is applied by recursively increasing the depth of the tree by partitioning leaf whenever it is proved to be effective. By effective it means that the visibility of the two new leaves are significantly different from the visibility of the old node. This measures how close the partition separates the visible objects into two disjoint visibility sets. Gotsman et al. suggested weighting the effectiveness values with another term, which measures the balance of the partition to avoid unbalanced trees. It should be noted that finding a good partition is non-trivial optimization problem [?].

A different refinement approach is taken by Nirenstein et al. [?]. Their view space is a rectangular region (possibly a viewface) which is refined by a quadtree, where each node of the quadtree is rectangle whose vertices are sampling points. From each sampling point the scene is rendered where each polygon is assigned a unique index. The visible polygons for the sampled point consist of all the indices that contributed to the image. A quadtree node is adaptively subdivided if the visible polygons of the four sampled points are significantly different. Unlike the previous method, here the scene is not represented by some hierarchy; each and every triangle is tested.

Figure 13.18 show a case where two view samples admits since their visibility set are the same. However, they both miss an object that is visible from the viewcell. This example can be extended into a degenerate case where an array of view samples miss a large number of objects. If the objects are not distributed uniformly, sampling the individual objects (possibly after sampling the hierarchical cells) significantly increases the probability of not missing some visible objects. However, as demonstrated in Figure 13.18 it still cannot guarantee a good sampling. For example, assuming that one samples a wall by casting rays. Starting from a coarse sampling rate, one can easily miss a clock hanged on the wall, and no refinement will be applied. Sampling the visibility on a per object basis requires defining a detailed hierarchy (or not using any hierarchy at all) since at the clock level, one can still miss the clock arms.

A useful observation is the fact that visibility is commutative. That is, the visibility test can be applied in either directions. Testing whether a given object is visible from a viewcell is equivalent to testing whether the viewcell is visible from the object. Some-
times in practice, due to discretization and implementation issues the reverse order can be preferable [?].

3.1 Hardly-visible sets

A non-conservative visibility algorithm that is not based on a sampling method is presented by Andujar et al. [?]. Their main motivation was not intended to compute a visibility set, but to combine approximate occlusion culling with levels-of-detail (LOD) to accelerate rendering by controlling the tradeoff between accuracy and speed. The idea is to identify partially-occluded objects in addition to fully-occluded ones. The assumption is that partially-occluded objects take less space on the screen, and therefore can be rendered using a lower LOD. The authors use the term Hardly-Visible Set (HVS) to describe a set consisting of both fully and partially visible objects.

A set of occluders is selected and simplified to a collection of partially-overlapping boxes. Occlusion culling is performed from the viewpoint using these boxes as occluders to find the “fully-visible” part of the HVS. It is performed considering only occlusion by individual boxes [?]. There is no occlusion fusion, but a single box may represent several connected occluder objects.

To compute partially-visible objects, all the occluders (boxes) are enlarged by a certain small degree, and occlusion culling is performed again using these magnified occluders. The objects that are occluded by the enlarged occluders and not by the original ones are considered to be partially occluded from the viewpoint, and are thus candidates to be rendered at a lower LOD (see Figure 13.19).

Several parts of the HVS are computed by enlarging the occluders several times, each time by a different degree, thus, classifying objects with a different degree of visibility. During real-time rendering, the LOD is selected with respect to the degree of visibility of the objects.

It should be noted that this basic assumption of the degree of visibility is solely heuristic, since an object partially occluded from a region does not mean it is partially

Figure 13.18: View sampling is prone to under-sampling even in two view admit.
Figure 13.19: (a) The boxes C and D are hardly visible behind A and B. (b) One boxes A and B are enlarged enough C and D become hidden.

occluded from any point within the region. It could be fully visible at one point and partially visible or occluded at another.
CHAPTER 14

Advanced From-Region Visibility Techniques

In the previous chapter we learn the basic concepts of from-region visibility, and saw conservative techniques based on the occlusion cast by a single occluder as well as non-conservative techniques. In this chapter we cover advanced conservative techniques that fuse occluders to achieve more effective occlusion.

1 Occluder Fusion

In the previous section we learned about occlusion made by single occluders. Figure 13.11(a) shows a simple scene in flatland where four segments cast a large aggregate umbra (patterned in diagonal stripes). However, each of the four segments has a finite, relatively small umbra. The union of the four individual umbrae is not effective as they hide very little compared to the umbra of their combined occlusion. In this and the following sections we study techniques that can handle occluder fusion for from-region visibility.

In the previous chapter we examined some of the properties of the umbra and penumbra and compared the various cases of interaction between the umbra and penumbra of two occluders with respect to a viewcell. Occluder fusion methods are aimed at finding regions that are in the aggregate umbra of two (or more) occluders. The term
aggregate umbra just emphasizes the distinction between an umbra of an individual occluder and the umbra caused by a combination of two (or more) occluders. Note that the aggregate umbra of two occluders is not the union of their umbrae, nor the intersection of their penumbrae. However, the aggregate umbra of two occluders is necessarily included in the union of their two penumbrae.

Detecting all the regions of the aggregate umbra of a number of occluders is too difficult. Even if one aims at conservatively approximating the aggregate umbra, it still remains problematic. However, some methods, discussed in the following sections, find a very good approximation for 2.5D scenes. Some methods conservatively fuse the occlusion in 3D, but extend the occlusion into a subset of the aggregate umbra. Such methods are conservative as they only occlude objects that are guaranteed to be in the aggregate umbra. These methods are fusing occluders that their individual umbrae intersect.

The principle is illustrated in Figures 13.11-13.12. Abstractly, the idea is to somehow find the intersection between two (or more) umbrae and then somehow represent a larger occluder that is a fusion of these two (or more) occluders. As a result a larger umbra is created, that of the fused occluders. Then iteratively, another intersection between two (maybe aggregated) umbrae is found and the aggregate umbra is further extended. In Figure 13.11(b) the left most individual umbrae intersect and a fused larger occluder is placed through the intersection area and extends across the individual umbrae. The aggregate umbra region of these two occluders is then represented by a diagonal pattern. Then the right two occluders are similarly fused and the final step is to fuse the two fused occluders (see Figure 13.12(b)).

To realize the above abstract fusion method the following questions should be addressed:

- How does one detect the intersection of two or more umbrae?
- How does one represent the fused occluder?
- How does one test the occlusion of a given object?

These questions are interdependent and as we shall see, are not always directly addressed. In the following, we will look into a number of algorithms that fuse occluders based on the above principles. First, let us introduce two tools, which will simplify the understanding of the techniques.

**Ghost and Virtual Occluders**

An occluder polygon $O$ can be approximated and represented by a back-ghost polygon [?] as follows. Consider a plane that is parallel to the viewface and that is directly behind the polygon. Consider the projection of the polygon onto this plane, defined as the intersection of the plane with the polygon’s umbra with respect to the viewface. The umbra of the back-ghost polygon is included in the umbra of the original occluder $O$, and approximates it well (it only misses the region between the original occluder and its ghost occluder). Since in most cases the viewcell and/or the viewfaces are axis-aligned, in many cases it is simpler to use the back-ghost polygon that corresponds to polygon 0 as an occluder rather than $O$. 

Similarly, consider a plane that is parallel to the viewface and is directly in front of the polygon $O$, and consider the projection of the polygon onto this plane, defined as the intersection of the plane with the convex hull of the polygon and the viewface. The umbra of the front- ghost polygon includes the umbra of $O$, and approximates it well.

A conservative from-region culling algorithm can test whether a set of occluders $O_i$ occludes a given polygon $P$, by testing their simplified representatives. That is, if the front-ghost polygon $P$ is occluded by the aggregate occlusion of the back-ghost polygons of $O_i$, then $P$ is also occluded [?].

A generalization of the back-ghost polygons is the virtual occluder [?]. A virtual occluder is a polygon that is fully included in the aggregate umbra of a set of occluders. The virtual occluder is like a back-ghost polygon since it is a simpler representation of the occlusion. However, unlike the back-ghost that represents the occlusion of a single polygon, the virtual occluder represents the occlusion of a cluster of polygons. Koltun et al. [?] present a technique to incrementally construct such virtual occluders that capture the occlusion of a set of occluders. The virtual occluders are more general as they are not necessarily axis-aligned and are not associated with a single occluder, but with a number of them. A virtual occluder that represents the occlusion of more than one occluder is in fact a realization of occluder fusion, and will be discussed in Section 4.

2 Occluder fusion by space discretization

Operating in a discrete space can simplify many of the difficulties in occluder fusion [?]. The original object-space representation of occluders is voxelized conservatively.
Figure 14.2: The black voxels are those which are interior to the object. Note that the umbra of the interior voxels is significantly smaller than of the original object.

That is, each opaque voxel is fully included in the interior of the occluder (see Figure 14.2). This discrete representation is bounded by the original representation, and thus its umbra with respect to viewcell can never occlude the visible part of an occludee. Let’s further assume that the voxels are grouped into a hierarchical representation, for example, an octree. Now the discrete representation of the occluder consists of voxels of various sizes. An important feature of these voxels is that each of them is convex-shaped.

This choice of representation of the occluder facilitates extending the occluders into a larger occluder that has a larger umbra. Each opaque voxel can be grouped together with its adjacent opaque voxels into an extended occluder to form a large convex shape. By grouping adjacent voxels, larger box-shaped occluders are obtained (see Figure 14.3). Now, since these extended occluders are convex-shaped, it is relatively easy to build their umbrae. The umbrae of all (or maybe a subset of) the convex shapes (extended occluders and original voxels) are built and voxelized conservatively.

All the voxels included in the umbrae with respect to the viewcell, can also be considered and marked as opaque. In fact the interior of the original occluders and the interior of their umbrae, are the same with respect to the viewcell as both are occluded. We can also claim that the interior of the occluders is part of the umbrae. Now again either the opaque voxels or the extended occluder, which consists of groups of opaque voxels, can be further extended into even larger occluders (see Figure 14.4).

Extending occluders into an adjacent umbra is the act of occluder fusion, and is possible provided that the umbrae intersect. Note that the area of intersection must be wide enough to fully contain at least one voxel. Once a new larger occluder has been constructed, its larger umbra is again built and voxelized and the process of occluder fusion continue iteratively. Figure 14.4 shows a step in this discretized version of occluder fusion.
Figure 14.3: The initial box-shaped occluder is generated by extending a seed voxel to its non-empty neighboring voxels. Here a 3-voxel wide initial box-shaped occluder is defined.

Figure 14.4: (a) Two black occluders have two umbrae (in gray voxel) that intersect. (b) A wider occluder consisting of four voxels is generated by extending a box-shaped occluder (in white) from one umbra to the next.

It is still questionable how to select the candidate voxels to be extended. Also one needs to use a heuristic to decide on the direction and order of extending the occluders. Schaufler et al. [?] suggest several heuristics to maximize the effectiveness of the extended box-shaped occluders as well as an extension using L-shaped occluders.

(* much more can be said here on these heuristics and how to deal say with 2.5D and how to build the shafts/umbrae *)
Figure 14.5: The back plane is the projection plane on which the extended projection of the set of occluders is recorded as a bitmap. The occlusion test is taking place over that bitmap. [I MUST CORRECT THE SUPPORTING LINES IN FIG B]

A byproduct of the above discrete occluder fusion is a discrete representation of the occluded region. This occlusion information can also be effectively represented in a hierarchical data structure. Given a candidate occludee, commonly represented by its bounding box, the visibility test is applied by testing the inclusion of the bounding box in the discrete representation of the umbra, which can be implemented by simply query the inclusion of the vertices of the bounding box in the octree.

The volumetric discretization of the scene and the umbrae provides a simple means for detecting umbrae intersection and fusing occluders. However, the memory requirement is too great for large scenes. On one hand the finest voxel size must be small enough to be fully contained within the occluder. On the other, the space requirement of a high-resolution voxel space might be too large.

The initial discretization of the occluder poses another problem. The initial occluders have an extended box-shaped (or even L-shaped) form that is necessarily aligned with the main axis. As illustrated in Figure 14.2, a diagonally-oriented occluder, which has a large umbra, does not contain any effective axis-aligned box. Thus, the initial umbra might be ineffective and no discrete space fusion can be applied. This can be alleviated in 2D space by allowing L-shaped occluders to be extended by concatenating them into a larger shape that is convex with respect to the viewcell. However, the extension into 3D is not trivial.

(* Please check my last comment above. My senses tell me that working in discrete space might be simple, but it won’t really be effective in 3D, and in 2D we can do better *)
3. OCCLUDER FUSION BY PROJECTION PLANES

3 Occluder fusion by projection planes

The following occluder fusion technique represents the aggregate occlusion of two or more occluders with a discrete projection plane [?]. A projection plane is a finite rectangle that is parallel to one of the six sides of the viewcell (to a viewface). Each such rectangle is associated with a bitmap on which the projections of the occluders are discretized.

The occluders are projected onto the projection plane to form a bitmap, which records their combined occlusion. The projection of an occluder onto a projection plane is defined as the intersection of the plane and the occluder’s umbra, with respect to the viewcell. Durand et al. [?] term this projection, an extended projection to distinguish it from a perspective projection, which has a point as a center of projection. The black regions of the bitmap correspond to the portion of the projection plane that is fully occluded. The projection of two (or more) occluders whose umbrae intersect yields a discrete representation of their combined (fused) occlusion (see Figure 14.5).

Testing whether a given polygon $P$ is occluded by a set of occluders $O_i$ can be applied over the projection plane. If the bitmap representing the projection of the aggregate umbra of $O_i$ includes the projection of $P$, then $P$ is occluded (see Figure 14.5).

While this sounds simple, the combined projection of a set of occluders is not straightforward to achieve. This is because the projection with respect to a viewface is defined as the intersection of the projection plane with the individual occluder umbrae. In other words, the projection plane needs to be placed somewhere where all the individual umbrae intersect. As illustrated in Figure 14.6, there is no single plane that captures all of the umbrae intersection.

To overcome this problem it is possible to build the aggregate umbra in an accumulative fashion (see Figure 14.6). With respect to one of the viewcell faces - the viewface, the occluders are traversed in a front-to-back order. For each occluder, in turn, a projection plane is placed right behind it, and the occluders in front of it are projected onto it. A key point is to be able to project not only occluders, but also to reproject combined projections of previously projected occluders.

This reprojection mechanism is the actual act of the occluder fusion. In Figure 14.6, occluders $a$ and $b$ are first projected onto plane $R_1$ since only there their umbrae intersect. Then the bitmap on $R_1$ representing their fusion is reprojected onto plane $R_2$, where the umbrae of $c$ intersect with the aggregate umbrae of $a$ and $b$. The bitmap on plane $R_1$ accumulates the combined occlusion of $a$, $b$ and $c$, and is then reprojected onto plane $R_3$ to be fused with the occlusion of $d$ and $e$. Note that again the intermediate projection plane $R_2$ could not be skipped.

An important issue is where to place the projection planes. One possible option is exhaustive - behind each potential occluder, where “behind” is with respect to one of the viewfaces. Figure 14.7 shows that a projection plane placed behind the occluder might be too far where the umbrae do not intersect. This can be regarded as applying this occluder fusion to the back-ghosts of the occluders (note that the umbrae of the back-ghosts of the occluders in Figure 14.7 do not intersect). However, this only leads to a more conservative set.

Implementing occluder fusion with these projection planes in 3D requires dealing with non-convex occluders. Also it requires a mechanism to reproject the bitmap
Figure 14.6: The projection plane must be placed where the umbrae intersect. Thus the fusion of the five occluders requires using three projection planes and reprojecting the occlusion of in a front-to-back order.

Figure 14.7: The projection plane cannot be placed behind the (right) occluder since it does not capture and fuse the combination of their umbrae.

representations of the projected occluders. Non-convex occluders are projected onto a projection plane by means of the intersection of the non-convex occluder and the plane. This yields a discretized slice of the object represented by a bitmap just like a projection of the convex occluder. The bitmaps are thus a unified discrete representation of the occlusion of an occluder regardless of whether it is convex or concave. Thus, their fusion is based on a reprojectation of their bitmap from one projection plane to the next. Durand et al. employ a discrete reprojection technique similar to the technique of Soler and Sillion [? ,?] for soft shadow computation, which is based on discrete convolution
4. EXTENDED OCCLUDER FUSION

In the previous sections we have seen how two occluders are fused as a results of an intersection between their umbrae. We can restate that condition as follow: Given a viewcell C and two occluders A and B, an occluder fusion can be applied if A intersects the umbra of B, or if B intersects the umbra of A. So far the occluder fusion algorithms that were presented above were applied in discrete space, and thus they required a somewhat stronger condition for the fusion to take place.

Stepping back and looking at the problem in flatland, Koltun et al. [?] show other conditions by which an occluder fusion can be applied. Let’s take a look at Figure 14.8.

Figure 14.8: The blind areas.

4 Extended occluder fusion

In the previous sections we have seen how two occluders are fused as a results of an intersection between their umbrae. We can restate that condition as follow: Given a viewcell C and two occluders A and B, an occluder fusion can be applied if A intersects the umbra of B, or if B intersects the umbra of A. So far the occluder fusion algorithms that were presented above were applied in discrete space, and thus they required a somewhat stronger condition for the fusion to take place.

Stepping back and looking at the problem in flatland, Koltun et al. [?] show other conditions by which an occluder fusion can be applied. Let’s take a look at Figure 14.8.
The two supporting lines of an occluder $O_1$, together with the occluder form a region in front of the occluder in which the penumbra region of the $O_1$ is not visible. Let’s call this region, the *blind region* of $O_1$. It can also be defined as the convex hull of the endpoints of $O_1$ and the intersection of the separating lines of $O_1$ (with respect to the viewcell). An occluder $O_2$ that intersects the blind region of augments it. The two occluders, $O_1$ and $O_2$ *form a cluster*, and its blind region is defined by the intersection point of its supporting lines and the endpoints of the two occluders. More occluders that intersect the evolving blind region are included in the evolving cluster, expanding further the blind region and the cluster of occluder. That blind region has an invariant property: no ray leaving the viewcell can ever pass through the occluders of the cluster. In that sense, the convex blind region can be regarded as a larger convex occluder. Koltun et al. observed that an occluder that intersects both the blind region and the current supporting line is a sufficient condition to extend the aggregate umbra.

An example is illustrated in Figure 14.9, where a series of occluders are iteratively augmenting the blind region and forming a cluster of occluders that can be fused. The umbra of the cluster is larger than the initial individual occluders. Note that the individual umbrae do not intersect, and only due to the incremental generation of the large blind region the aggregate umbra is built. Note also that the order by which the algorithm insert the occluders into the cluster is important, which implies that it is advantageous to start with a large seed occluder.

![Figure 14.9: The order is important.](image-url)
5. \(\varepsilon\)-NEIGHBORHOOD VISIBILITY

Wonka et al. [?\] have introduced a generic technique by which from-region visibility can be calculated by an array of from-point visibility calculations. The idea is to shrink the occluders by \(\varepsilon\), and to use the umbrae of these shrunken occluders with respect to a set of points of view \(p_i\). It is based on the observation that the umbra cast by an object shrunk by \(\varepsilon\) is a "valid" umbra also for the \(\varepsilon\)-neighborhood of the point \(p\) with respect to the original occluder. In other words, a from-point culling by the umbrae of the shrunken occluders is a from-region conservative culling by the umbrae of the original occluders. As illustrated in Figure 14.10, any point which is the aggregate umbra with respect to point \(p\) is in the aggregate umbra with respect to the \(\varepsilon\)-neighborhood of \(p\).

The advantage is that a from-region conservative algorithm with occluder fusion is readily given based on any from-point visibility algorithm that considers occluder fusion. This of course yields an overestimate of the PVS since the visibility test considers smaller occluders. See the scenario illustrated in Figure 14.10. The aggregate umbra of the original occluders is larger than that of the shrunken occluders. It should be emphasized that the size of the umbra volume is not necessarily an important factor for a successful occlusion. Sometime its shape is more effective. For example, see Figure 14.11; where the apparently minor gap in the umbra prevents the classification of large bounding boxes as occluded.

Given a viewcell, or an arbitrary view region, its conservative visibility set can be computed by sampling the from-point visibility of an \(\varepsilon\)-dense set of points on the viewcell boundary. Each sample point guarantees the occlusion from its \(\varepsilon\)-neighborhood, and all sample points together guarantee the occlusion from the entire cell. There are two possible approaches to combine the results of the \(n\) individual from-point visibility tests [?]:

- Any point that is an umbra with respect to all the \(n\) points is an umbra with respect to the viewcell. Thus, one can explicitly compute the intersection of the
CHAPTER 14. ADVANCED FROM-REGION VISIBILITY TECHNIQUES

Figure 14.11: The aggregate umbra as the intersection of the three from-point umbra.

- Calculate the PVS from each sample point and take their union as the PVS from the viewcell.

The problem with the second approach is that the visibility of a given object (or a bounding box) would have to be calculated \( n \) times. The first approach is hard to implement since it requires maintaining a data structure of the umbrae and applying volumetric intersections. In these approaches the from-point visibility calculation is applied independently, where many objects are tested redundantly and classified as occluded. A third approach can apply them in a dependent fashion, where the scene is traversed top down, and each cell in the hierarchy is tested simultaneously with all the
n viewpoints. If a cell is classified as occluded, the traversal of its descendants stops
and no cell is redundantly classified as occluded. However, this would prevent using a
from-point algorithm as a black box for the implementation of a from-region visibility.

The $\varepsilon$-neighborhood visibility can also be used for a micro-region visibility test.
This can be as an ”instant visibility” test that is valid for a small number of frames [?].
The rendering consists of two processes. One computes the PVS and one renders the
PVS. Unlike the ”visibility streaming” [?], here the two processes are not remote and
there is no latency problem caused by the transmission of the PVS over communication
channels.

6 Ray Space Techniques

In this section I’ll review (again) the basics of ray space parameterization and discuss
their nice properties. Then I go into the details of

6.1 Koltun et al. 2001
6.2 Bittner et al. x 2

and

6.3 Leyvand et al.

6.4 Exact Visibility

I don’t know if and how much we should discuss these techniques in the book. I think
it is too much for a section in this chapter. To explain them we need a chapter.

Bittner [?] has proposed method that computes an exact from-region visibility in
polygonal scenes by performing set theoretical operations in line-space. The algorithm
maps a set of lines blocked by a convex polygon to a 5D polyhedron using Plücker
coordinates. The union of these polyhedra is maintained by a 5D BSP tree.

Pu [?] proposed a similar algorithm using a 5D BSP tree to represent from-region
visibility maps. This method captures a global visibility information and therefore the
memory complexity restricts its application to scenes with only a few polygons.

Nirenstein et al. [?] proposed an algorithm designed for the from-region visibility
culling. The method tests visibility between each scene polygon and the given region.
The exact visibility test is computed using 5D set operations similarly to [?]. Nirenstein
et al. proposed several optimization strategies that make the algorithm suitable for large
scenes.

7 The PVS storage space problem

Precomputing the PVS from a region requires solving a prominent space problem. The
scene is partitioned into view cells and for each cell a PVS is precomputed and stored
readily for the online rendering stage. Since the number of viewcells is inherently large, the total size of all the visibility sets is much larger than the original size of the scene. Aside for a few exceptions this problem has not received much attention.

In this section we will see two schemes that aims at compactly encoding and storing the visibility sets of all the viewcells.

Let’s first get a feel of the magnitude of the problem. Assume that the scene is a city model consisting of 50k primitives, and the scene is subdivided into 10k viewcells. Let’s further assume that in average from each viewcell about 20% of the scene is visible. This requires to store about 10k indices per viewcell, which total in 10M of indices. Since each index must be larger than 8 bits, and they must be linked together the storage requirement gets easily over 20M Bytes. Alternatively, it is possible to encode a large binary vector $A$, where each entry $A[i]$ indicates whether primitive $i$ is visible or not. Following our example, this would require about 6k Bytes for each viewcell. However, since this binary vector contains about 20% of 1’s only, it would tend to be well compressed with some LZ encoder. If it compresses down to 2k per viewcell, then the total again is around 20M Bytes.

Of course, the above representations of visibility sets are naive and they can be better encoded by taking advantage of the nature of the problem. That is, the visibility sets of nearby viewcell are in general coherent, and their visibility sets relatively similar. It should be noted that an efficient encoding scheme must also guarantee a fast decoding. In particular we would like to have a fast access to the visibility set of a given viewcell.

Van de Panne and Stewart [?] present a technique to compress precomputed visibility sets. They assume that the visibility set of a viewcell is encoded in binary visibility vector like we described above. The array of all the viewcells, thus form a visibility table $C[i, j]$, where the entry in row $i$, column $j$ is 1 if and only if polygon $j$ is visible from viewcell $i$. Their method compressed that visibility table by merging columns and rows that have similar binary codes. The essence of merging operation is clustering objects and viewcells of similar behavior.

Polygons that have similar visibility and clustered together are usually belonging to the same object or at least they are very close to each other. If the scene is well modeled by a hierarchy of bounding boxes and scene graph solve this clustering a-priory. Cell-to-cell visibility techniques [?, ?], cluster the polygons of a cell, and classify their visibility together. In a sense, the cell-to-cell visibility set is a compact representation of the cell-to-polygon visibility set. Similarly, one can use the cells of a hierarchical subdivision data structure (e.g., kd-tree) as a compact representation of the visibility. That is, the nodes of the hierarchy can be encoded as a binary tree, which requires only 2 bits per node. Note, that most visibility methods use a hierarchical data structure, so the indices of the polygons of a given cell are readily available.

Merging viewcells that see similar polygons is an effective operation that can be applied to any viewcell partition. Typically adjacent viewcells have similar visibility sets. Merging them save storage space, but traded for a stronger conservativeness of the merged visibility set. Gotsman et al. [?] present a hierarchical scheme to encode the visibility efficiently. Instead of simply merging similar adjacent viewcells, they encode the “dissimilarity” between them with respect to their merged viewcell (see Figure 14.12).

More specifically, assume the viewcells are arranged in some hierarchical data
7. THE PVS STORAGE SPACE PROBLEM

(a) plain bits per node. (b) encoding in B and C just the bits that are non zero in A, and (c) encoding in C just the bits which are not 1 in B.

Figure 14.12: The three possible encoding of a node A and its immediate sons B and C.

structure represented by a tree. The leaves of the tree are viewcells, each of which stores the visibility set by means of a binary code array. The visibility set of an internal node A is the union of visibility sets of its two children, B and C. Now, any entry in code array of A which is zero, is also zero in the corresponding entries in code arrays of B and C. Thus, encoding these known entries in B and C can be avoided. Similarly, there is a dependency between the codes of the siblings with respect to their parent node. A zero entry in the code array of B indicates that the corresponding entry in C must be non-zero since necessarily the parent node has a non-zero in the corresponding entry. Thus, again the encoding of trivial zero entries in C can be avoided. Now since the visibility sets are relatively small, the code arrays are sparse and they can be effectively compressed [?].

Intermediate storage

A completely different approach to solve the storage problem is to use an intermediate representation of the PVS rather than directly the set of visible objects. The intermediate representation is a means to generate the visibility at run-time. It must have a compact form to significantly alleviate the storage space problem, and it must be fairly simple and fast to compute the PVS at run-time on-the-fly during rendering. It should be emphasized that this intermediate representation is associated with each viewcell, and is thus a view-dependent representation.

Koltun et al. took such an approach [?]. In a preprocess stage they synthesize simple vertical rectangular polygons that encapsulate the occlusion caused by the original objects of the scene with respect to a given viewcell. These rectangular polygons are view-dependent convex virtual occluders (see Section 4). At run-time the virtual occluder can be used as a strong occluder (see Chapter ??) to compute the PVS on-the-fly during rendering.
The key point is to find a small set of effective virtual occluders. For a given view-cell, they first compute a large number of virtual occluders (as we have seen in Section 4 above), then they select a small effective subset. They report that 5-20 virtual occluders often represent almost all the occlusion from the viewcell. Since the virtual occluder is a vertical rectangle, it requires only 5 Bytes to encode it (one byte for the height and 4 for the two endpoints). Only 100 Bytes are needed to encode 20 virtual occluders. It should be noted that simplified occluders also means a somewhat larger PVS.

Ideally, the storage problem can be avoided all together by computing the PVS on-line during rendering. Some efforts has been put toward achieving this [?, ?, ?].
Part IV

Misc. occlusion culling
In the previous chapters we learned a large variety of occlusion culling methods. Their general concept is to traverse the objects of a scene, typically in a hierarchical fashion, and apply a visibility test for each candidate with respect to “some representation” of the occlusion. The meaning of “some representation” is exactly what distinguishes the various methods. It can either be represented in image-space or in object-space. It can represent the occlusion found so far during the traversal, or it can represent the occlusion of a preselected subset of occluders. In any case, it is clear that the complexity of the occlusion process is directly dependent on the complexity of the visibility test and on the complexity of constructing the occlusion representation.

To get a better understanding of this, let’s first take a look at a simple example. In the first chapter of the book, we have presented a simple occlusion algorithm where the visibility of the objects is tested against $n$ preselected large polygons. Clearly, as we have shown there, the complexity of the algorithm is directly dependent on $n$ - the number of visibility tests. The complexity is also dependent on the complexity of each individual test, which depends on the “shape” of each individual occluder. As we will see later, all the algorithms which we classified as “large occluders [?, ?, ?, ?] [DOUBLE CHECK HERE LATER] have a similar behavior.

In image-space techniques the complexity of the occluders is not exhibited in the test itself since the complexity of an image-space test is dependent on the image space
resolution. However, during the algorithm an image-space representation of the occlusion is constructed, and that’s where the complexity, or more precisely, the number of occluders counts. For example, in the hierarchical z-buffer algorithm \([1]\), a complete generation of image-space representation requires traversing the entire hierarchy all the way down to the leaf nodes. This implies drawing all the visible parts which is rather costly. To avoid this, Zhang \([2]\) selected only a number of “good” occluders in his image-space technique.

Of course, an intelligent selection of “good” occluders is required to make the algorithm efficient. In the following we will discuss this meaning of “goodness”, and also the selection process of “good” occluders.

However, there are many cases where there are no readily available “good” occluders. The input scene might consist of a large number of small polygons, possibly without the notion of objects. Or the scene may consist of many small objects, such that no object alone is a “good” occluder, and only a group or cluster of objects together, casts a significant occlusion. This has brought about the idea of synthesizing occluders that were not originally present in the scene. For instance, we might attempt to synthesize a large simply-shaped occluder from a group of small finely-tesselated objects, such that the new occluder partially encompasses, or represents, the occlusion caused by the original group of objects. Considering this occluder instead of the original group would clearly make it much easier for most algorithms to account for the occlusion.

The above suggests that we should distinguish between two types of processes. One is **occluder selection**, and the other is **occluder synthesis**. The first selects a subset of occluders from among the original objects of the scene, while the other creates, or synthesizes, new virtual occluders that correctly represent the occlusion cast by a subset of the original object of the scene. These two problems are quite related since in both cases one is interested in having an efficient set of “good” occluders in order to accelerate the visibility algorithm. In other words, one wishes to synthesize a virtual occluder which is very likely to be selected as a good occluder during run-time.

1 Classification of the problem

The occluder selection process is a much simpler process than the occluder synthesis. It typically requires just testing some properties of a given object and assigning it a rank. An occluder synthesis process necessarily involves complex geometric algorithms, and thus, is typically applied in a preprocess. The occluder selection process, on the other hand, can either be applied as a preprocess or at run-time, on-the-fly during the walkthrough itself. This classification of the algorithms to offline versus online algorithms is closely related to another two important classifications of the methods:

- **view-dependent** versus **view-independent**, and
- **from-point** versus **from-region**.

In a view-dependent method the selection or the synthesis is done upon the current location of the viewpoint. Typically this is applied on-the-fly during run-time \([3, 4, 5]\).
A CLASSIFICATION OF THE PROBLEM

?, but can also be precomputed offline for a region []. A view-independent algorithm, on the other hand, does not take into consideration the current location of the viewpoint and thus it is necessarily applied in a preprocess [?, ?, ?]. So, an occluder synthesis algorithm is typically view-independent and applied offline. But again, some occluder synthesis techniques are developed for a from-region scheme [?, ?]. Thus, we should distinguish between algorithms that are applied from a point and those that are applied from a region.

The exact meaning of an online algorithm that is computed for a region is somewhat different than that computed for a point. In a from-region scheme, the online process has much more time to spend on computation than in a from-point scheme. This may permit a moderately complex synthesis algorithm to take place in a from-region scenario [?, ?].

Let’s take a look at the illustrations in Figure 15.1. The objects of the scene are distributed around the viewpoint (marked with a circle), and the two black objects are those that were selected as good occluders. The object in the gray pattern below the viewpoint is a synthesized object that serves as a good occluder. It captures the view-dependent occlusion cast by the cluster of the three smaller objects around it.

![Figure 15.1](image-url)

**Figure 15.1**: A scene where two objects (in black) are selected as good occluders and one (in a gray pattern) is a synthesized occluder that captures the occlusion of the cluster of three objects. The viewpoint is marked with a circle.
2 Good occluders [needs to be improved]

Before we answer the question “What is a good occluder?”, let’s note the following two points. First, the set of all objects that contribute to the occlusion, is the exact visible set itself. In fact finding this set requires solving an exact visibility problem. Thus, selecting occluders is always selecting of a set that won’t necessarily hide all the occluded parts. In other words, the selected occluders set is a conservative set, and is effective mainly for conservative visibility algorithms. But that’s no problem since, as we saw, all the efficient algorithms are conservative ones.

The second point is that once we have agreed on a conservative set of occluders, the reader should be convinced that in a typical scene, a simplified version of a given occluder can occlude about as much as the original object. Moreover, when simplified occluders are fused, their aggregate occlusion is very much close to the aggregate occlusion of the original objects.

Figure 15.2 shows a typical example of the effect of simplifying an occluder. We can see that the simplified occluder retains most of the occlusion of the original occluder. In Figure 15.3 we show that the reduced occlusion of each simplified occluder is even less significant under occluder fusion. The simplified occluders might have less overlap among their umbrae, but this does not reduce the size of their aggregate umbra. The simplification only reduces the silhouette of the aggregate umbra, and that is typically a minor reduction.

Using the umbra metaphor, we can agree on the simple intuitive criterion that a good occluder is one that casts a large umbra. Assuming the objects are evenly dis-
Figure 15.3: The three black occluders in (a) are simplified to the three black boxes in (b). The gray objects are those occluded by the black occluders. Note that the occluded objects in (a) are also occluded by the fused occlusion of the three simplified occluders.

tributed, large umbrae occlude more. To cast a large umbra the occluder has to be relatively large. In a view-dependent scheme, this has to be with respect to the viewpoint location. As noted by Zhang [7], any object can be a good occluder for a given favorable viewpoint: when the viewpoint is really close to the given object, its umbra is really large. In a view-independent scheme, one has to consider how likely it is that the object will be viewed. For example, it is unlikely that the viewpoint will be too close to the arms of a clock hanging on a wall, or even to the clock itself. Thus, this clock is unlikely to be a good occluder in a sensible scenario.

An earlier proposition for a good occluder is due to Saona et al. [9] who select an object as an occluder based on its volume and the area of its largest face. This approach is global in nature, suitable as a view-independent criterion. It does not account for the fact that occluders that are suitable for one viewpoint in a scene may not be so for another. Moreover, the volume of a polyhedral object is quite expensive to compute. Zhang [7] essentially proposes selecting objects that are nearest to the viewpoint, disregarding all other criteria. The advantage of this simple criterion is that it is extremely easy to compute. Coorg and Teller [7] and Hudson et al. [7] propose selecting objects that have a large solid angle as the occluders for the visibility computation. Intuitively, this means that objects that appear “large” when seen from the viewpoint are selected as occluders. In the following we elaborate more on this criterion.
2.1 Solid angle

The solid angle measures the size of an object, as seen from a point in space. Its value is determined by the size \( d\omega \) of the object’s projection on the unit sphere centered around the viewpoint (Figure 15.4). Informally, we can say that the solid angle grows with the visible size of the object, and diminishes with the object’s distance.

\[
\text{d}A \propto \cos \theta \frac{r}{r^2}
\]

This measure is applicable to polygons, for which we know how to compute their area and their normal. This can easily be extended to convex polyhedra by summing the solid angles of the front-facing polygons. More complex objects can be approximated by a single convex polyhedron (i.e., a bounding box) and computing its extended solid angle.

To avoid this computation during the walkthrough, Hudson et al. [?] suggested precomputing for each potential occluder the region from which the given object is seen with a large solid angle. Then this information is stored as from-region information (we will elaborate more on this in Section ??).

2.2 Criteria for good occluders

In the above discussion we only considered large objects as good occluders. However, there are further criteria one would require from a “good” occluder. To be more concrete, let us consider a large set of neighboring polygons from which we would like to synthesize a good virtual occluder. Of course, the main requirement from the virtual
occluder is that it capture as much as possible of the occlusion of the given polygons or maybe of a subset of them. Now, let us examine what we expect from this virtual occluder so as to consider it a good occluder:

- **Size.** Clearly, large occluders cast larger umbrae and thus occlude more. One should also consider the occluder aspect ratio. If the occluder is large, but long and thin, it is not large in size in all directions. In a view-dependent content this is crucial. In a view-independent scheme, we would like the occluder to be large, and have a small aspect ratio.

- **Simple.** Dealing with complex objects that have a high polygon count, or vertex count, imposes a burden on any algorithm that uses them. It takes more time to render them, or to make any geometric or analytic computation with them. As we will see later, many algorithms try to either simplify a given occluder without losing much of its effective occlusion, or to synthesize an occluder with a very simple shape in the first place.

- **Convex.** One characteristic of being simple, is being convex. For some algorithms convexity can make a difference. The umbra of a convex occluder is convex as well. Thus, for example, testing whether a given vertex is within the umbra of a convex shape is a much easier test than a non-convex test.

- **Redundant.** Some large and simple occluders may not be as effective as would appear from their intrinsic properties. They might just be positioned in front of a larger wall that renders their occlusion redundant. The actual occlusion of an object needs to be measured with respect to the context of the other occluders.

These criteria do not only hold for selecting a good occluder, but also for the synthesis of such. In summary, we can say that basically a good occluder needs to be large and simple. In some cases it helps if the occluder is also convex. Regarding the last requirement above of effectiveness, as this is not an intrinsic property, it would usually be too hard to compute.

### 2.3 Sampling the “goodness” of an occluder

A better measure for a good occluder can be based on a sampling scheme. This is brute force and quite expensive, but correctly measures the effective occlusion a given object casts. This is essentially a precomputation and the results need to be stored for each region of some spatial partition. Let’s assume the view space is partitioned into a grid of viewcells. Then for each viewcell the visibility set of the entire scene is sampled from several viewpoints within the viewcell. For each candidate occluder, each sample is applied twice: once with and once without a given occluder. The difference between the two visibility sets measures the effectiveness of the given occluder from a particular sampled viewpoint. Averaging the samples yields the average effectiveness of the given occluder from within the viewcell.

Note that this sampling scheme is applied to each individual occluder separately. Thus, two adjacent occluders might have umbrae with a significant overlap (see Figure
15.5). Each might have a large umbra and not be classified as a good occluder, while in fact only one of them is redundant with (respect to) the existence of the other. One possible way to circumvent this, is by defining an order among the occluders. An occluder which is found not to be effective is removed from the scene for the rest of the sampling test (of the given viewpoint). Then the adjacent occluder will have a fair chance of showing its effectiveness.

Figure 15.5: None of the three black occluders is a good occluder with existence of the two others. Thus, a naive visibility sampling fails to detect that at least one of them is a good occluder.

2.4 Hidden occluders

Until now we have discussed either the selection of a good occluder from the existing objects, or the synthesis of a good occluder. However, these criteria do not only apply to the original objects or to the virtual objects, but also to any geometric entity that generated as a byproduct of the algorithm.

For example, sometimes during the application of the algorithm large regions are detected as hidden. These hidden regions can serve as effective occluders. In conservative algorithms, regions detected as hidden are necessarily such and any entity lying entirely inside this hidden region, is a conservative occluder. Schaufler et al. [?]
use this idea. They synthesize large boxes within the aggregate umbra detected so far. Koltun et al. [?] synthesize large vertical rectangles within the aggregate umbra of a cluster of objects. Similarly, Nirenstein [?] uses hidden bounding boxes as occluders. Note that the use of hidden parts as occluders is applicable only to view-dependent algorithms.

2.5 Good from-region occluders

It was shown by Koltun and Cohen-Or [?] that this approach may at times provide erroneous results. An example is given in Figure X, where object B that subtends a larger solid angle at the center of a viewcell is actually a worse occluder with respect to the viewcell than object A that subtends a smaller solid angle and is also farther from any point in the viewcell than B. Koltun and Cohen-Or further proposed a new criterion for guiding the selection of occluders for from-region visibility computation algorithms that treat such scenarios correctly, and suggested a hardware-assisted implementation.

2.6 Temporal coherence

Regrading the selection of occluders, Zhang [?] pointed out some interesting observations: there is a coherence between consecutive frames that can be used to improve the selection. Instead of selecting the occluders from among all the objects in the current view frustum, one can ignore all the objects that were detected as hidden in the previous frame. This is based on the observation that hidden objects from the last frame tend also to remain hidden in the new frame. However, in some cases, this is not enough as new object enter the current view frustum, and should also be considered as occluders. See Figure 15.6, where objects e and f are hidden in frame $n$ and are (likely to remain) hidden in frame $n+1$. Objects a-d are just entering the view frustum, and one of the objects is a good occluder.

Avoiding objects that were occluded in the previous frame as candidates for the current frame is merely a good heuristic. In rare cases, the previously occluded objects may become visible, and some time they can serve as a good occluder. An example is shown in Figure 15.6 where objects c and d become visible. However, as Zhang mentioned, since these cases are quite rare, their effect is only minor.

3 View-dependent techniques

3.1 Hudson 97

Although it was described above, we can repeat things here and elaborate a bit more. Especially on the combination of view-independent and dependent of the selection.

3.2 Koltun00

Koltun et al. [?] propose to synthesize virtual occluders locally (that is, a different collection of virtual occluders is created for each viewcell). Going even further than Law
and Tan [?], their virtual occluders are rectangles, making them extremely effective for visibility computation. Unfortunately, the synthesis algorithm is designed to work only with 2.5D scenes. However, it is able to fuse occlusion due to disparate objects, such that each virtual occluder encapsulates occlusion caused by a multitude of original objects. In fact, the experimental results suggest that less than 20 virtual occluders are often enough to represent all the occlusion in the scene with very little error [?].
3. VIEW-DEPENDENT TECHNIQUES

3.3 Bernardini00

Bernardini et al. [?] also deal with synthesis of occluders.

Figure 15.7: The silhouette of a objects can be simple from one view and non-simple from another.

Figure 15.8: The silhouette of a objects can be convex from one view and non-convex from another.

3.4 Hoops

Brunet et al. [?] describe a method of locally synthesizing view-dependent occluders, so-called “hoops”. Hoops are non-planar, non-convex polygonal polylines with the property that they look convex and simple from within a given viewcell, but probably invalid for another viewcell. Given a space partition into viewcells, their algorithm synthesize a number of hoops for each cell. These hoops are used as occluders and fused into larger occluders to compute a conservative visible set for each cell.

As we have seen in Chapter ??, many from-region algorithms, the convexity of the occluder is a necessary condition. The fundamental assumption is that if a convex occluder hides an object from all vertices of the viewcell, then it hides it from any viewpoint within the viewcell. However, a key observation is that for a from-region visibility, the occluder can be non-convex as long as it umbra is convex with respect
The silhouette of an object can be simple from one view and non-simple from another.

to the view region. The convexity of the umbra from the cell is sufficient, while the
occluder itself can be concave. This is a weaker condition. The main idea of using
hoops is that these non-convex occluders can be more effective provided that their
view-dependent umbra is convex.

A hoop is defined as a non-planar, closed polyline that appear simple and convex
from all the vertices of a viewcell. A polyline is a sequence of edges and it has no
interior. It is convex or concave according to the enclosed region within it. Non-planar
means that its vertices are arbitrary in 3D and not necessarily embedded in a plane.
Simple means that the polyline does not intersect itself (See Figure 15.7).

The shape of the umbra of an occluder is determined by the silhouette of the oc-
ccluder as viewed from a viewpoint. For certain viewpoints the silhouette of the object
appears as convex, while it may appear as non-convex for others. A hoop can approx-
imate the silhouette of any object, not necessarily a polyhedron. Note that a hoop can
also approximate the silhouette of a curved solid as illustrated in Figure 15.9.

Given a viewcell C we say that a hoop is a C-hoop if it appears simple and convex
from all vertices of the cell C. Hence the hoop definition depends on the viewcell. A
hoop P can be C1-hoop but not C2-hoop, where C1 and C2 are viewcells (see Figure
15.8). Recall that convex occluders allow us to infer the visibility from aa viewcell just
from the visibility of its vertices. This also holds when an occluder generates extended
umbra that is convex from all vertices of the cell. An extended umbra from a point q
contains, in addition to the umbra, the occluder itself and the region between q and the
occluder. A C-hoop generates convex extended shadow from all vertices of the cell C,
and thus it can be used to infer the visibility from the vertices of C alone.

The hoop synthesis algorithm is a iterative algorithm. It starts with a seed which is
an initial hoop and then enlarges it repeatedly to get a better solution while preserving
its C-hoop property. The algorithm uses as input an octree representation of a given
object O. Each node includes a color black, white or grey and eight pointers to its sons.
A black node is completely inside the object, white is completely outside, and grey
contains part of the boundary. Namely the object interior is represented by the black
nodes in the tree such that the smaller nodes are located closer to the boundary.

The silhouette of one of the biggest black node as viewed from a cell C is taken as a
seed. A seed black node is valid only if it has the same silhouette from all vertices of C. Then the algorithm try to enlarge this seed occluder by adding, or “popping”, vertices from the list of candidate vertices. The list of candidate vertices contains the vertices of the black nodes, sorted by their size.

Each candidate vertex is examined in turn and if the vertex pops the hoop P, then P is enlarged. A candidate vertex v augments P if v is outside all extended umbrae of P from all the vertices of C. Otherwise it is rejected since it cannot enlarge the hoop. The next test checks if P can be enlarged to contain v while still being a C-hoop, thus convex. The new vertex introduced two edges. This test checks the convexity between them and between each of them and its adjacent edge. The last test ensures the conservativeness and guarantees that P does not hide something that O does not. It checks if the resulting hoop’s umbra is contained in that of the object O. This is done by checking for the existence of a surface S that is contained in O and whose silhouette matches that of P.

To improve its performance, the algorithm is starts with few seeds, rather than one, and generates few hoops. At the end, the best hoop is chosen as the one whose projected area (shadow area) on the separating plane is the largest.

4. View-independent techniques

Law and Tan were among the first to consider occluder synthesis [?.], and have coined the term virtual occluders. They proposed a method of synthesizing a view-independent convex occluder with low polygonal complexity from a finely tesselated non-convex object. The idea is essentially to perform a restricted simplification by which the original object is simplified such that the result is within a certain error bound from the original. Then they “fix up” the result by bringing the simpler occluder completely inside the original object. This is done by detecting vertices that are outside the original object and moving them so that they are bounded by it. Finally, they perturb the location of the edges of the simplified object ensuring that the occluder is convex. It should be noted that this does not necessarily creates a valid object with significant volume. Further more, it works on a single object and has no means to create a simple occluder that encompasses the occlusion of multiple objects, as the following techniques do.

4.1 Virtual Overlapping Boxes

Andujar et al. [?] introduce a coherent view-independent occluder synthesis methods. The algorithm generates a set of, possibly overlapping, boxes from non-convex polyhedra with arbitrary number of shells. The synthesized boxes are aimed to be as large as possible so to maximize their umbrae from arbitrary direction. The algorithm guarantee that each one of the boxes is completely inside the original object. This method can take as an input a polygon soup and create valid convex occluders that well represent the occlusion cast by input polygons. However, the polygon soup must be a valid representation of a closed polyhedra with a significant volume.

The algorithm consists of two stages. First the volume of the polyhedra is discretized, and then large (overlapping) boxes are extracted. Andujar et. al. call these
Figure 15.10: A 3D model of St. Paul’s cathedral (on the left) is converted to a set of voxels (in the middle), out of which, a set of overlapping axis-aligned large boxes are extracted.

Figure 15.11: The 2D polygon in light gray is converted into a discrete representation consisting of an array of cell. Only the cells which fully inside the polygon are drawn in dark gray. Starting from the two seed cells (marked with a black circle), two axis-aligned boxes (in white) are synthesized.

two stages, aggregation and convex extraction, respectively. The idea is illustrated in Figure 15.10, and a simplified 2D version in Figure 15.11.

First, the input objects is discretized into an octree. Nodes of the octree that are completely inside the objects correspond to parallelepiped or axis-aligned boxes that can serve as conservative occluders. Then these boxes are extended into neighboring ones, while retaining their box-like shape, provided that they are all inside the input objects. The algorithms aims to find a set of such, possibly overlapping, extended boxes that well represent the occlusion of the original object. Finding the optimal solution would take exponential time. Thus, a heuristic approximation is used: A number of large seed boxes are selected and in a greedy fashion, the algorithm extend them by elongating their. Each such extended box is stretched along one dimension at a time. The stretch is greedy, since it elongate the dimension of the box as much as possible, as long as it remains within the inside the original objects. A 2D example is illustrated in Figure 15.12. Note, that the order of dimensions along which the box is stretch affect the results. See, Figure 15.12 where unlike the previous example, here, the box is first stretched horizontally to yield a box that can no longer be stretched. However, this can be alleviated by running the box extension procedure several time with randomly
Figure 15.12: The greedy algorithm for extracting axis-aligned boxes. In (a) the inner voxels (cells) of the object are in dark gray. The selected seed voxel is marked with a black circle. In (b) the seed voxel is expanded down to create an stretched box. Then in (c) that box is stretched to the left to yield a large box bounded by the original object. That box is not the largest. In (d) a different voxel is picked as a seed, and extended first horizontally to the left. Then in (e) it is further extended vertically to yield a larger box than in (c). In (f) a third seed voxel can yield yet another large box. Note, that if in (a) the voxel would have first stretched to the left, it wouldn’t have yield a large box.

shuffled ordering of directions. Practically, the authors report that this is not needed if several seeds are used. Moreover, one can let the algorithm generate a large number of such extended boxes generated from a various seed boxes, and then select a subset of effective boxes.

The above algorithm is designed to synthesize occluders that represent the occlu-
of typical buildings. The algorithm takes advantage of the nature of architectural building whose walls are typically axis aligned. Thus, the extended boxes successfully “slide” along the internal walls of the building and easily expand. A necessary preprocess step is to align the model with the axes of the octree. This is done by applying a linear transformation to the model so that its main direction become axis-aligned (see the algorithm for generating an oriented bounding box, Section ??). If the bounding box of the model does not well represent the orientations of the walls, a more sophisticated algorithm is necessary. However, the extraction of large convex shapes from arbitrary n-vertex object, is not

Andujar et. al. also suggest a variant of the algorithm that can produce larger occluders if some degree of error is allowed. They treat small cavities, holes and transparent parts as being opaque and therefore allowing the generation of larger boxes. By increasing the error tolerance a level-of-detail of occluder can be defined. This can improve the occlusion of the individual boxes. However, when the synthesized occluders are fused by the visibility algorithm, the aggregate occlusion of the error-free (conservative) overlapping boxes is effective enough for a typical scene.

Figure 15.13: The original building in (a) has seems rather simple geometry as the details of the building are not displayed. By identifying three critical heights, three simple blocks are extracted (b-d).
4.2 Synthesizing Facade

The above technique of synthesizing boxes was signed to deal with architectural models. However, their method is generic and can be applied to a general 3D model. The following technique exploits the fact that in most cases urban models consist mainly of 2.5D parts. Assuming the model has no overhanging parts, like balconies, the occlusion of such a 2.5D object can be well represented by a set vertical walls or facades. Germs and Jansen introduce an object-space method that synthesize such facades.

The given 2.5D model can be approximated by a set of blocks. A block is a 2D contour with an associated height. In other words a block is a closed sequence of quadrilateral facades with the same height defines. Since overlapping among the blocks is allowed, the facades and thus the blocks can be always emerge from the ground plane to yield larger blocks.

The method of Germs and Jansen first computes the footprints of contiguous parts of the model that have about the same size. The footprint is a 2D projection of these parts on some reference plane. The footprint is simplified and approximated by a closed sequence of segments which form the basis of the block. Their method uses heuristics to define the critical heights of the given model, for which it is effective to compute a block. Figure 15.13 illustrates the idea. The original building in (a) has seems rather simple geometry as the details of the building are not displayed. However, one should assume that the building consists of a large number of polygons which describe the the fine details (e.g., the windows). By identifying three critical heights, three simple blocks are extracted (b-d). Each of those blocks has a simple polygonal base (footprint) and facades emerging from the segments of the footprint up to the critical height. Note that the shorter block in front of the building does not meet one of the critical heights, and thus its volume contributes to none of the blocks.

Unlike the technique of Andujar et al., this technique avoids the need to discretize the volume of the given model, and it synthesizes the occluders significantly faster. However, it is limited to deal with 2.5D only. If the model contains some overhanging parts, the algorithm will fail to be conservative.

4.3 Lev-Yehudy 2003
CHAPTER 16

Integration with other acceleration techniques

1 Level of detail
2 Image-based acceleration
3 Real-time rendering
   3.1 Performer
   3.2 Funkhouser
4 Strategies of integration
   what to do when from-region + run-time
5 Hardly visible sets
CHAPTER 16. INTEGRATION WITH OTHER ACCELERATION TECHNIQUES
Part V

Shadows
17 Preliminaries of Shadows

AW: If what we have written is too long, here are the candidate sections we can consider removing: 2.3.2, 2.3.3, 2.6, 2.7, 2.8, 3.6, 3.7; or reduce 3.3.

PP: we need to start with a definition of what we consider to be a shadow, and general concepts to be more thorough, and actually better introduce shadows.

A shadow is a region in 3D space where light emitted or reflected is completely or partially occluded. As such, computing shadows corresponds to computing visibility of the light emitter for a region.

1 Why Shadows

Using almost any measure of the image quality, the computation of shadows is essential. They cause some of the highest intensity contrasts in images; they provide strong clues about the shapes, relative positions, and surface characteristics of the objects (both occluders and receivers of shadows); they can indicate the approximate location, intensity, shape, size, and distribution of the light sources; and they represent an integral part of the total effect in architecture with many objects in the environment.

In fact, in some circumstances the shadows constitute the only components of the scene, as in shadow-puppet theatre and in Pinscreen animation, as developed by Alexander Alexeieff and Claire Parker [?]. Another wonderful treatment of shadows has an artist named Paul Pacotto [?] (at an art gallery in St. Paul de Vence, France)
using shadows as part of his sculptures, in which the sculpture forms ordinary shapes such as a rose, but the shadows from the rose looks like a woman. See Figure 17.1.

![Figure 17.1: Paul Pacotto sculpts both surfaces and their shadows.](image)

*PP: I will check a couple of books in computer vision about shadows, such as Horn...*

Wanger *et al.* [?] evaluate various depth cues that are useful for displaying inter-object spatial relationships on a 2D screen. Shadows form an important visual cue among the depth cues. However, comparing the usage of hard shadows versus soft shadows as visual cues, they determined that hard shadows are actually more beneficial as a visual cue. Studies have been done in which the availability of shadows improve the interaction of object positioning and improve accuracy of spatial relationships between objects [?, ?], and the perception of realism [?]. In fact, without the presence of shadows, surfaces often appear as if they are floating over a floor, when they are actually lying on the floor – this is why shadows are one of the most crucial elements in augmented reality applications.

One conclusion to draw from these studies is that shadows form important visual cues for spatial relationships between objects and light sources in a 3D scene. However their *exact* determination might not be as important as long as they are "consistent" with our expectations. In fact, exact shadows when dealing with extended light sources can be very surprising and unnatural for common observers. The two images in Figure 17.2 show the shadow from a linear light aligned with the object (left) and with the object rotated by 20 degrees around the vertical direction. Notice the discontinuity within the shadows.

This can lead to potential shadow approximations that exploit visual expectations, and therefore to simplifications over the visibility algorithms reviewed in the first part of this book. Direct image visibility is always more demanding than shadows, which
2. HARD SHADOWS VS. SOFT SHADOWS

**Figure 17.2**: An object formed of three axis-aligned ellipsoids is rotated by $20^\circ$ around the vertical axis. The soft shadow cast on the plane below from an axis-aligned linear light source becomes discontinuous on the right image.

is an indirect phenomena. The next two chapters in this part on shadows will show a large number of such approximations that are commonly used for shadows.

Note that not all shadows rendered need to be physically accurate in terms of the shadows reflecting the shape of the occluder. There can be very “approximate” shadows on planar receivers, which can be very effective in certain real-time environments such as games. For example, in the 3D version of the Super-Mario game on the PS-2, Super-Mario casts a circular shadow on the ground. When Super-Mario jumps, the shadow circle gets smaller. Though the shadow does not reflect the silhouette of (the occluder) Super-Mario, it is a very effective, real-time, visual cue as to where Super-Mario is with respect to the ground.

2 Hard Shadows vs. Soft Shadows

Shadow determination, in the context of occlusion from other surfaces, can be considered as some variation of the visibility determination problem. However, instead of computing visibility from the eye, shadow determination computes visibility from the light source. One main difference is that for shadow determination, it is not necessary to calculate the closest visible surface, it is only necessary to determine if there is occlusion between the surface and the light.

There are basically two shadow types: hard and soft shadows. We will discuss them in detail in the next two sections. We will finish off this section with a brief discussion of colored shadows, which is not really discussed much in general. For existing shadow algorithm surveys, refer to [?] for an older survey, [?] for an up-to-date real-time shadow algorithm survey, and [?] for an up-to-date real-time, soft shadow algorithm survey.
2.1 Hard Shadows

A hard shadow is the simplest type of shadow, displaying only the umbra section. By umbra, we mean that a region of space is either completely occluded or completely lit, thus forming “hard” boundaries between the shadowed and lit regions. Calculation of hard shadows involves only the determination of whether or not a point in the scene lies in shadow of occluding surfaces. This is a binary decision problem on top of the illumination model. In other words, multiply a value of either 0 or 1 by the light intensity, indicating in shadow or not in shadow, respectively. The domain of light sources truly generating hard shadows include a point light, spotlight, and directional light – see Figure 17.3. Chapter 18 will cover algorithms that generate hard shadows.

![Figure 17.3: Light types casting hard shadows. PP: to add figure of spotlight](image)

2.2 Soft Shadows

The other type of shadow is a soft shadow, meaning the inclusion of a penumbra region along with the umbra for a higher level of quality. Full occlusion from the light causes the umbra region, and partial occlusion from the light causes the penumbra region. The degree of partial occlusion from the light results in different intensities of the penumbra region. The penumbra region causes a “softer” boundary between shadowed and fully lit regions. The resultant shadow region is a function of the shape from the light source and the occluder. Instead of a binary decision on top of the illumination model as for hard shadows, “a fraction in the range of [0,1] is multiplied with the light intensity, where 0 indicates umbra, 1 indicates fully lit, and all values in between indicate penumbra”. Needless to say, soft shadows are slower to compute than hard shadows, and the soft shadow algorithms are more complex.

The domain of light sources generating soft shadows includes linear, polygonal/area, spherical, and sky (hemispherical) lights - actually any extended light – see Figure 17.4 for an example of an area (triangular) light. As well, algorithms considering global illumination, motion blur, and transparency can also produce soft shadows, which will
be briefly discussed in Sections 6 and 7. We will, however, focus on soft shadow algorithms from extended lights, which will be covered in Chapter 19.

![Figure 17.4: aFig10.eps about here.](image)

Actually, the statement made above can be ambiguous, and thus it is important to understand the nature of the integral that must be evaluated in order to compute the soft shadow resulting from the direct illumination of an extended light source. The integral of the irradiance, $I$, over a diffuse surface element $dA_r$ can be expressed as follows:

$$I = \int A_e \left( \frac{L \cos \theta \cos \phi}{\pi r^2} \right) dA_e$$

(17.1)

where $A_e$ is the extent of the light source as seen from the receiving surface element $dA_r$, $L$ is the radiance received from the light, $V$ is the visibility of a surface element $dA_e$ on the light source, $\theta$ and $\phi$ are angles measured with respect to the normal of the receiving surface element and of the emitting surface element on the light source respectively, and $r$ is the distance between the two surface elements. Figure 17.5 illustrates the configuration for these variables.

![Figure 17.5: Configuration of the variables for the integral of the irradiance.](image)

In order to obtain the exact irradiance, one must evaluate the integral with care. Ideally, if the domain of integration can be reduced to the fragments of the extended light that are visible from the point to be shaded (i.e., the points on $A$ for which $V$ is 1), then the integral reduces to a direct illumination integral over precisely these fragments. Unfortunately, determining these fragments is a difficult problem in itself.

A common simplification the computation of the integral is to assume that it is separable, and to integrate the visibility and the irradiance separately (as originally stated in †). Many of the soft shadow algorithms presented in Chapter 19 will assume this separation of the visibility term. This gives rise to the following approximation of the irradiance:

$$I = \left[ \int_{A_v} V dA_e \right] \left[ \int_{A_e} \left( \frac{L \cos \theta \cos \phi}{\pi r^2} \right) dA_e \right].$$

(17.2)
Note that whereas this approximation will generate soft shadows, the accuracy of
the results will not necessarily be reliable, especially in cases where the solid angle
subtended by the light is large (e.g., the light source is large and/or close to the surface),
or is more or less perpendicular to the surface. The inaccuracies from the decoupling
of the irradiance integrand and of the visibility factor will be most evident when the
parameters \( \theta, \phi, \) and \( r \) vary greatly over the domain of the integrand, and the parameter
\( V \) is non-constant.

The appeal of such a simplification is great, given that an analytic solution can be
computed for the non-shadowed illumination integral \([?, ?, ?]\) and that distribution ray
tracing can be used to approximate the visibility of the light \((i.e.,\) the integral of \( V \) over
\( A_e )\). The approximation is also simpler for radiosity solutions, for some direct illu-
mination algorithms \([?, ?, ?]\), for sampling criteria in final gathering of penumbras \([?]\),
and may also be applied for specular integrands. However, care must be taken that the
artifacts resulting from such an approach are reasonable, though such approximations
are more than acceptable and not visually discernable in many situations and applica-
tions. See Figure 17.6, with the same linear light relative to the receiver, where the two
different cases would result in the same shading evaluation if the shadowing function
is outside of the irradiance integral, but using the correct integral evaluation, the results
should be quite different.

![Figure 17.6: aFig3a.eps about here.](image)

**PP:** Figure 9 from \([?]\) is a simple good illustration of how shadows from combined
objects is more complex than the combined shadows of each object.

### 2.3 Colored Shadows

Implicit from the above descriptions is that the occluding surfaces are opaque. One
assumption is that because we are only dealing with opaque surfaces, a shadow appears
like a darker region. This is not always the case, especially when there are multiple
colored lights, and “colored” shadows can be the end-result. Take the example of a
blue light and a red light. A grey region occluded only from the blue light will appear
red, and a region occluded only from the red light will appear blue, and only a region
occluded from both lights will appear black.

Another form of “colored” shadows can come from occlusion of semi-transparent
3. Classes of Shadow Algorithms: Basic Ideas and Properties

In the upcoming sections, we will review the major classes of algorithms available for shadow generation, at a high level. This is to ensure that the reader has some basic understanding, before delving into the details in the upcoming chapters. In fact, based on the information provided here, the reader may choose to ignore certain classes of algorithms due to the high level descriptions because certain algorithms clearly do not fit his needs. The major classes of shadow algorithms include: shadow depth map, shadow volumes, area subdivision and preprocessing, ray tracing, and axis-aligned shadow receiver algorithms.

3.1 Shadow Depth Maps

In the literature, the terms “shadow map” and “shadow buffer” are often used in an inconsistent fashion. They can mean the same thing, which is a “shadow depth map”, or they can mean something else totally. For example, a shadow buffer used in the context of voxels may refer to a 2D or 3D shadow buffer. In the context of this book, “shadow depth map” will be used explicitly to indicate a 2D shadow map that stores a single depth value per pixel. Any shadow map or shadow buffer references will indicate explicitly what information is stored per element, and in how many dimensions.

Williams [?] uses a Z-buffer depth map algorithm to generate shadows, known as the shadow depth map. The Z-buffer approach to determine visibility and intensity with respect to the eye is repeated for the light source. Thus, as a preprocess as seen in Figure 17.7, it creates a buffer with respect to the viewpoint of the light source $L$, except that the buffer contains only $z$ depth values, not shading values or object information. During rendering of the camera view, each point $x$ to be shaded is projected towards the light and intersects the shadow depth map pixel. If $\|x - L\|$ is larger than the $z$ depth value $Z_n$ (as shown in Figure 17.7) from the projected shadow depth map pixel, then $x$ lies in shadow, otherwise it is fully lit. The papers by Williams [?] and Reeves et al. [?] are the most often cited and implemented versions of the shadow depth map approach.

From a GPU-based implementation, several papers [?, ?, ?] use a GPU texture (typically 16 or 24 bit texture) to represent a shadow depth map, projectively texture it onto the scene, then compare the depth values in the texture during fragment shading (Everitt et al. [?] use combiner-registers on nVidia GPU) to achieve per-pixel shadows. However, due to an overload of usage of GPU textures in some instances (for real texturing), the use of the GPU texture to achieve depth map shadows may be a bottleneck.
An example is shown in Figure 17.8.

Figure 17.8: GPU-based shadow depth map image. Courtesy of Everitt et al.

Shadow depth map has been very successful in many graphics environments because:

1. It is simple to implement.
2. The performance can be close to real-time without GPU assistance. Conservative occlusion culling algorithms as described in this book (which chapters?) can also be applied for the light source rendering to achieve good performance while handling large datasets.
3. It can handle surfaces other than polygons.
4. It can be used as a sort of “protocol” of shadow information from different data representations, or different renderers.

5. It can be simply implemented in the GPU as a hardware texture for real-time applications.

6. Its quality has been improved so much that it can be used in film Major software renderers use some variation of the shadow depth map, such as Pixar’s Renderman, Alias|Wavefront’s Maya renderer, Mental Images’ Mental Ray, NewTek’s Lightwave, etc.

7. Soft shadows from extended lights have been developed based on the depth map shadows approach.

8. Though the basic shadow depth map approach can only deal with shadows from opaque objects, there are variations to handle non-opaque object shadowing as well.

The disadvantages include:

1. Rendering quality issues relating to filtering and self-shadowing issues have not been completely resolved.

2. Only z-depth is stored in a shadow depth map pixel, thus handling semi-transparency and self-shadowing is more difficult. "There is no association information between the occluder and receiver."

3. Shadow determination is more complex for point lights, requiring more than one shadow depth map per light. This is also true for spotlights with a large angle of view, because an angle of view larger than 90 degrees will likely result in poor quality renderings if only one shadow depth map is generated.

4. The rendering quality is particularly poor when the view focuses on a specific region that only covers a small part of the shadow depth map. Some algorithms can deal with this issue, but most other variations of the shadow depth map do not deal with this problem except at a manual level.

5. Changes in the shadow coverage region can result in changes in rendering quality. By shadow coverage, we mean the world space region represented by the shadow depth map. Changes in the shadow coverage may be needed to get the sharpest image quality by encompassing only particular objects, and the particular objects’ occupied world space changes during an animation.

3.2 Shadow Volumes

Crow proposes an approach to generate polygonal shadow umbrae from the surfaces in the scene, then place them into the rendering data structure as invisible surfaces. The
original set of polygons is also included in this rendering data structure for shadow determination, which is also sometimes known as the “light cap”. To compute shadow determination, a shadow count is used. An initial shadow count is calculated by counting the number of shadow volumes that contain the viewing position. Then the shadow count is incremented by 1 whenever there exists a front-facing polygon (that is, entering the shadow umbra) crossing in front of the nearest visible surface. The shadow count is decremented by 1 whenever there exists a shadow back-facing polygon (that is, exiting the shadow umbrae). If the final shadow count is 0, then the visible surface does not lie in shadow; if positive, it is fully lit. See Figure 3.2, where the initial shadow count is 1, then gets decremented/incremented to 0, 1, 2 until it hits the surface to be shaded – since the final shadow count is greater than 0 (2), then that point is in shadow.

To implement shadow volumes in the GPU, Heidmann [?] draws the scene polygons shaded only by an ambient light with a hardware Z-buffer. Front-facing shadow polygons are then drawn (using front-facing culling test), incrementing shadow counts in an 8-bit GPU stencil buffer, if visible (called z-pass test) for each affected pixel. Similarly, visible back-facing shadow polygons decrement their respective shadow counts. Finally the scene polygons are drawn with diffuse and specular shading only where their stencil shadow count is 0.

The original shadow volume approach has been equal in popularity to shadow depth maps in many graphics environments because:

1. It is computed at object precision, is omni-directional, and can produce volumetric effects [?].


3. Near real-time variations without the need for the GPU have been developed [?, ?, ?, ?].


5. Though the general shadow volume is only capable of dealing with shadows from polygonal objects, there are more advanced variations that can deal with other geometric representations [?, ?, ?].
6. Conservative occlusion culling algorithms as described in this book (which chapters?) can be applied to significantly reduce the shadow volumes required, to achieve good performance while handling large datasets.

The disadvantages include:

1. It is primarily effective for polygonal representations.
2. It needs well-formed objects, with adjacency information to optimize silhouette detection.
3. Many (up to one quadrilateral per edge per light source) long shadow polygons need to be scan-converted (high fill rate).
4. Limited representation (8-16 bits) for hardware-based shadow counts.
5. Aliasing errors in the shadow counts due to scan-conversion of very narrow shadow polygons.
6. Transparent objects cannot easily receive shadows when this algorithm is implemented in the GPU. The problem is that a pixel on the screen has the shadow state of only one surface, normally the closest opaque surface, stored for it. There is no additional storage for transparent objects that cover the pixel.

3.3 Area Subdivision and Preprocessing

Nishita and Nakamae [?], and Atherton et al. [?] use clipping transformations for polygon shadow generation. In this two-pass hidden surface algorithm, the first pass transforms the image to the view of the light source and separates shadowed and lit portions of the polygons via a hidden surface polygon clipper. It then creates a new set of polygons, each marked as either completely in shadow or completely lit. The second pass encompasses visible determination from the eye and shading of the polygons, taking into account their shadow flag.

This class of shadow algorithms has received very little research attention since the original papers. Reasons for reduced focused with this class of shadow algorithms are likely due to the significant increased complexity with medium-to-large datasets, as well as potential numerical instability issues [?], particularly difficult is a GPU based implementation due to the reduced numerical precision (until recently) in the GPU. Ghali et al. [?, ?] take the subdivision approach and store the shadow edge and adjacency information in a visibility map, which does avoid polygon clipping instability issues, but do not resolve the other issues. Such edges can also store penumbra information for extended lights. However, a practical algorithm to compute the visibility map is sought, which makes this extended subdivision approach more of theoretical interest at the moment.

Similar to area subdivision techniques, Appel [?], and Bouknight and Kelly [?] generate shadows during the display using an extended scanline approach. During preprocessing of each polygon, all polygons that lie between the light source and the
polygon itself are identified and stored in a list. During the display phase, polygonal boundaries from the scanned polygon’s list are projected down onto the scanned polygon from shadow boundaries, clipped within the boundaries of the scanned polygon, then projected onto the viewing screen. The intensity of a scanned segment changes as it crosses the shadow boundaries.

Due to the lack of research papers and limited usages of the above algorithms, this class of shadow algorithms will not be further discussed in this book.

### 3.4 Ray Tracing

Shadow determination using ray tracing is trivial: a shadow ray is shot from the point to be shaded towards the light source \( L \). If the shadow ray intersects any object between \( x \) and \( L \), then it lies in shadow; otherwise, it is fully lit – see Figure 17.10. There have been quite a number of papers written on the topic of accelerating shadow computations in a ray tracing environment because ray tracing is so flexible (in just about any environment) and simple to implement. The flexibility of ray tracing can be seen particularly in the generation of soft shadows, such as work done by the following papers: [...]. However, ray tracing remains very CPU bound and expensive, and not generally capable of being used for real-time applications yet, unless some GPU-based version of a ray tracer can be implemented. The closest proposal on a generally accessible GPU-based ray tracer can been seen in the work by Purcell et al. However, many of the acceleration concepts used in ray tracing can partially be applied to much faster shadow algorithms and of theoretical interest.

![Figure 17.10: Basics of the ray tracing shadow algorithm. PP: need to change P by L in figure.](image)

#### 3.5 Axis-aligned Receivers of Shadows

In simpler applications of real-time shadows, certain assumptions may be made about the environment. One such example is that hard shadows are projected only on a (planar) floor. The floor is generally perpendicular to one of the \( X, Y, Z \) axes, thus a single transformation matrix resembling oblique or perspective screen projection matrices is all that is necessary to project the polygons’ vertices onto the floor. In fact,
the projected shadows can even be modeled as dark polygons, so that little additional rendering code is necessary. Note Figure 17.11, in which point $P$ on the box projects a shadow on the floor as point $S$, as a result of light $L$.

![Figure 17.11:Projected shadow polygons on a planar floor.](image)

This simple algorithm has been used in many real-time implementations due to its simplicity and speed. There has been little followup on this algorithm (because it is pretty much complete for its use), except that Hallingstad [?] provides details and implements the fake shadows on a plane in the GPU, using OpenGL along with a stencil buffer.

Additional reading material include [?].

## 4 Self-shadowing

From a computer graphics standpoint, there are typically two main components of determining the shadowing: shadows due to occlusion from other surfaces, and self-shadowing. Most of the discussions in this book focus on occlusion from other surfaces. However, we do want to cover a few essential topics on self-shadowing, to allow a more complete understanding of shadowing. Most self-shadowing issues assume the simulation of non-smooth surfaces and relate to (or are considered as part of the) illumination computations or the reflection models employed, such as the $N \cdot L$ check, bump mapping, and anisotropic reflection models. An additional topic of the "surface acne" is also discussed, though this is more of a limited precision problem than related to illumination computations.
4.1 $N \cdot L$ Check

The simplest example of self-shadowing can be done using a dot product check $N \cdot L$, where $N$ is the surface normal and $L$ is the light direction with respect to the point to shade. This check is done in almost all rendering systems. This means that no light reaches the portion of the surface that is facing away from the light without further computations; shadowing from other occluding surfaces are only checked when $N \cdot L > 0$. This is a concept similar to back-face culling from the view direction, except for the lighting direction in this case. While this check is physically correct, natural, and optimal, there are consequences to this check that should be understood, such as specular highlight cutoff, the terminator problem, and translucency, which are discussed below.

Specular Cutoff

The first consequence comes from bad specular highlight cutoff [?]. Because the $N \cdot L$ evaluation also happens to be the diffuse reflection amount, and the specular component is calculated independently of the diffuse evaluation, there can be cases where $N \cdot L < 0$, but the specular component is positive, indicating a specular contribution when the diffuse has no contribution. Thus the self-shadowing check appears to have prematurely cut off the specular component – see Figure 17.12. This problem is not usually visible, due to the exact circumstances required to hit this situation.

![Specular Cutoff](image)

Figure 17.12: Specular cutoff: The specular reflection component spreads where direct light should not reach because the $N \cdot L$ check was not performed.

Terminator Problem

The second consequence comes from the terminator problem. This problem results from improper self-shadowing due to polygonal mesh approximation for a smooth sur-
face. In Figure 17.13 (left), polygons $A$ and $B$ represent polygonal approximations to the smooth surface. At point $x$ on $A$, the vertex-interpolated normal $N$ is used to compute the illumination, as opposed to the plane’s normal $N'$. Since $N \cdot L > 0$, light contribution is present, and the shadow occlusion from other surfaces must be computed to determine whether $x$ is shadowed. The shadow ray for point $x$ intersects $B$ and incorrectly concludes that $x$ is in self-shadow. This artifact is usually visible as shadow stair-casing – see Figure 17.13 (right) where the stair-casing occurs between the dark and lit regions. The simple solution is to offset the shadow ray origin by a small amount along $N$, to avoid the self-shadowing (see point $x'$ in Figure 17.13). Unfortunately, it is difficult to figure out what is the correct offset value, and this offset typically assumes convex region behavior, because a concave region should ideally have a negative offset. Furthermore, though this problem has been described in the context of ray tracing, it is actually a problem in most shadow algorithms (except less so in shadow depth map algorithms due to the usage of bias factor), and unfortunately, an “offset” workaround is not possible in some non-ray-tracing based shadow algorithms.

![Figure 17.13](image)

**Figure 17.13**: Shadow terminator problem: (left) interpolated normal on the polygonized sphere; (right) resulting shadow stair-casing.

**Translucency**

A third consequence comes from translucency. The interesting illumination aspects for translucency include the region that faces away from the light. So the $N \cdot L$ check cannot be used for self-shadowing. In fact, it is very difficult to account for shadows from other surfaces when computing translucency, because the point to be shaded needs to know whether other parts of the surface has been occluded from light. Thus most algorithms handling translucency assume the translucency component is always well-lit, except for the brief discussions in Section 7.
4.2 Bump Mapping

Another example of self-shadowing problems is bump mapping [?], where surface normals are perturbed to give the impression of a displaced, non-smooth surface. However, bump mapping does not actually displace the geometry (as in displacement mapping [?]). As a result, shadowing for bump mapped surfaces appear as if the surface is perfectly smooth, because shadow determination does not use surface normal information at all.

Horizon mapping [?] approximates the shadows cast by the bumps on the same surface. Interpreting the bump function as a 2D table of height values, horizon mapping computes and stores, for each height value, the angle between the horizon and the surface plane at 8 or more azimuthal directions on the surface plane. During rendering, the horizon angle at the intersection point is interpolated from the light direction and the horizon map. If the horizon angle from the surface normal exceeds the light angle, then this point lies in shadow.

Sloan and Cohen [?] extend the horizon mapping algorithm for the GPU using multi-pass rendering of texture lookups and stencil buffering. The 8 horizon mapping directions are encoded into two four-channel texture maps, and multiplied with the basis functions (expressed as a texture map) to determine the amount of shadowing. Kautz et al. [?] also permit self-shadows from bump maps in the GPU. As a preprocess, rays are shot into the upper hemisphere for each bump map pixel in many directions, and a tight ellipse that contains as many lit rays as possible (and as little shadowed rays) is computed. To determine shadowing per pixel, the lighting direction must have a hit inside this ellipse to be lit, otherwise it is in shadow. Testing if inside the ellipse is cleverly converted into dot products for fast computations using the GPU.

The above approaches compute self-shadowing of bump mapped surfaces onto themselves. Noma and Sumi [?] use ray tracing to cast other surfaces onto a bump mapped surface. This is done simply by offsetting the bump mapped surface as the starting shadow ray origin. Shooting the shadow ray with this offset can result in convincing shadows on the bump mapped surface. Note that this technique can be applied to shadow depth maps as well. While all the above approaches solve some part of bump mapped shadows (self-shadowing, other surfaces onto a bump mapped receiver), one of the main issues that remains unsolved is the shadowing of other bump mapped surfaces onto other (possibly bump mapped) surfaces. Figure 17.14 illustrates these situations.

Figure 17.14: Problems with shadows from/on bump mapped surfaces.

4.3 Anisotropic Reflection Models

Say somethings about anisotropic reflection models – Pierre – can you take of this section?

PP: measured BRDFs include the masking/shadowing factors, some reflection models include the shadowing factor within the model itself Blinn77, Cook84, Poulin90, He91.
4.4 Surface Acne Problem

The surface acne problem is the improper self-shadowing due to either lack of numerical precision (e.g., ray tracing) or lack of required resolution (e.g., shadow depth map – see Section 1.2 for more in depth coverage of this issue). The intersection point just happens to be off by a bit to make the computations believe that it is in shadow – see Figure 17.15. For example, in ray tracing, the intersection point is just below the surface, so that the shadow hits the surface itself again. Offset solutions to escape bad self-shadowing are employed in both ray tracing and shadow depth maps for this. For example, in ray tracing, a shadow hit is confirmed only as such if the hit distance is larger than some epsilon (instead of 0); other solutions are proposed by Woo et al.

Figure 17.15: Surface shadow acne due to numerical precision.

5 Considerations for Choosing an Algorithm

The choices of a shadow algorithm will get more and more complex, as the following issues need to be considered while the reader is going over the algorithms described in the next two chapters:

1. Is real-time or interactive speeds a requirement?
2. Is real-time feedback from dynamic scenes a requirement?
3. Is an appropriately advanced GPU available for your environment?
4. What platform dependencies or constraints exist?
5. What is considered “good enough” quality or accuracy for your needs? In most circumstances, this usually applies to soft shadows only.
6. What “features” from the shadow algorithms are needed?
7. Are there any constraints on the type of geometry used for rendering purposes?
8. Are there dependencies on the visibility determination algorithms used?
9. How much user-intervention and user-input are acceptable?
10. Are consistent performance behaviors important (i.e., around the same speed of rendering no matter where the camera or objects might be)?
11. Is the algorithm you wish to implement within a product already patented?

There are some aspects of shadow algorithms that are not discussed in this book. They may affect your choice of algorithms quite differently. Additional readings can be found in the following papers:

- Non-photorealistic (NPR) environments: [?, ?, ?, ?, ?, ?].
- Integrating 2D images with 3D graphics: [?, ?, ?, ?, ?, ?].
- Shadows as interaction tools: [?, ?, ?, ?, ?, ?].
This chapter will cover algorithms that generate hard shadows. The main classes of algorithms include the shadow depth map, shadow volumes, and ray tracing. Image-based rendering techniques are also listed as they may be easily employed in just about any environment supporting texture mapping. To finish off, atmospheric shadows, semi-transparent shadows, and shadows from other geometry types are also briefly discussed. Although those three topics apply to both hard and soft shadows in general, most papers and research have focused their solution within the hard shadow domain.

1 Shadow Depth Maps

While the shadow depth map appears simple in its description in Chapter 17, there are many things to consider for achieving an efficient and robust rendering of datasets. Among them include how to deal with different light types, and the many performance and quality issues (including bad self-shadowing).

1.1 Dealing with Different Light Types

For the shadow preprocess rendering, it is important to realize that a spotlight maps to a perspective rendering from the light. However, the circular region often represented by the spotlight must be totally inside the perspective rectangular region, so there is some wasted rendering region. Similarly, a directional light maps to an orthographic...
rendering from the light. See the hard shadows in Figure 17.3 of Section 2.1. Extended light types produce soft shadows and therefore will be discussed in Chapter 19.

Things get a bit more complicated when dealing with point lights, because a single perspective rendering cannot cover a viewing frustum of 360 degrees. For a point light source, multiple shadow depth maps need to be generated using a perspective rendering. The most common approach produces six 90 degree-view shadow depth maps, such that the six views form a full cube around the point light. In this way, full coverage in both the azimuth and altitude of the point light is considered for shadow computations. During the actual shading phase, the point to be shaded is projected on one of the six shadow depth maps to determine the shadow occlusion. However, this method tends to result in discontinuous artifacts when the shading point projects close to the border of one of the six shadow depth maps. This is due to the perspectives changing drastically at the borders when moving from one shadow depth map to another. A simple workaround is to slightly overlap the six shadow depth maps, computing six 95 degree-view shadow depth maps instead of 90 degrees. When a point projects to any of the overlapping regions, then the shadow lookup is done for multiple shadow depth maps (up to a maximum of three) and their results are averaged. This alleviates most of the discontinuous artifacts.

Note that six such shadow depth maps are not always required for point lights. In many cases in which the point light is above all surfaces, the number of shadow depth maps can be reduced – or put another way, if this optimization is not done, some shadow depth maps may be empty. Gerasimov [?] goes over the details of code using DirectX-9 for point lights using the six 90 degree-view shadow depth maps, but not at the level of discussing the above issues.

Brabec et al. [?] deal with the point light shadow depth map problem by employing a dual-paraboloid mapping. Instead of six planar shadow depth maps forming a full cube, only two hemispherical shadow depth maps are needed. Each paraboloid map can be seen as an image obtained by an orthographic eye viewing a perfectly reflecting paraboloid. However, with large polygons, linear interpolation performed during rasterization could cause non-linear behaviors. As well, it is likely that some discontinuity artifacts might appear near the hemisphere borders.

1.2 Avoiding Bad Self-shadowing Problem

One rendering quality problem with the basic shadow depth map approach is the bad self-shadowing artifacts that can result from a shadow depth map pixel projecting to many screen pixels. The resulting artifacts usually appear as some Moiré-like, narrow, darker strips – see Figure 18.1 where dark strips on the planar floor are due to the floor incorrectly self-shadowing itself. The situation is particularly bad when the light direction is close to parallel to the surface being shaded – see Figure 18.2, where $Z_n$ is the shadow depth map pixel value but the pixel projects to a wide region in space. As a result, for a point to be shaded $x$, $\|x - L\| > Z_n$, that indicates that $x$ is in shadow, but clearly it is not the case. It is also clear that if the shadow depth map resolution is increased, this problem is reduced, but never fully eliminated; it also significantly slows down the shadow rendering process. Partial solutions to this problem, without needing to increase the shadow depth map resolution, will be discussed below.
Object or Surface ID

Hourcade and Nicolas [?] address this problem by using a surface ID. Each surface is assigned a unique ID, and the shadow map stores for each pixel the surface ID that represents the closest surface at that pixel. Then during rendering, instead of comparing z-depths, the surface IDs are compared— if the shadow map pixel has a different surface ID than the current surface, then it is in shadow; otherwise it is not. However, this general technique does not allow for self-shadowing of surfaces, and thus the surfaces
must be convex. This technique also does not eliminate incorrect shadowing cases, however, since a surface can be composed of many small surfaces, and the shadow map pixel only has information to a single surface ID – unless all the pixel’s visible surface ID values are stored, incorrect shadowing can result, but then the storage required and the during-rendering comparison can be expensive.

Dietrich [?] and Vlachos et al. [?] apply the self-shadowing surface ID approach [?] on the GPU. Surfaces are split up into convex parts and given a unique surface ID, put in a sorted order from front-to-back with respect to the light and the alpha channel is used for comparing objects to the corresponding value in the shadow depth map. Alternative hybrid approaches have the surface ID approach used for occlusion between different surfaces, and localized shadow depth map or localized shadow volumes are used to determine proper self-shadowing.

Bias

Reeves et al. [?] address this self-shadowing artifact by using a user-specified value called a “bias”. This bias value is used to escape self-shadowing by displacing the surface closer to the light by a small distance. Thus, there is shadowing only if $||x - L|| > Z_n + \text{bias}$. However, choosing a good bias value is very difficult in many cases, because a single bias value must pertain to the entire scene – i.e., if the bias value is too large, there can be shadowed cases where the bias value addition causes the algorithm to indicate fully-lit; if the bias value is too small, then bad self-shadowing can continue to occur. To improve this situation, Reeves et al. [?] introduce a bias value which has a lower and upper bound on biasing, and apply a stochastic factor to have different per pixel bias value in between, i.e.,

$$bias = \text{minBias} + \text{random} \times (\text{maxBias} - \text{minBias}).$$

In this way, it is hoped that a good amount of cases would properly detect self-shadowing, and for those cases that are inaccurate, it is hoped that the stochastic component will result in noise, which is more acceptable than Moiré-like patterns. However, good guesses at bias values are still needed.

Grant [?] and Wang and Molnar [?] also try to deal with the bad self-shadowing case by examining the surface normal at the point to be shaded. They determine a good bias based on the relationship between the surface normal and the light vector direction. However, because only the current point’s surface normal is being evaluated, it cannot possibly determine self-shadowing in many concave surfaces.

To achieve a shadow bias in OpenGL, \texttt{glPolygonOffset} can be used that uniformly offsets the depth values. However, there is no such equivalent in DirectX. Everitt et al. [?] apply a small texture transform offset (i.e., a small scale factor) so that the surfaces sent to the GPU for shadow preprocessing is a bit smaller than the surfaces rendered for the view. However, both bias approaches on the GPU are fixed (i.e., non-stochastic). PP: is smaller correct, or better further? PP2: what I actually meant is that the surface is scaled down by a bit, so becoming smaller should also make it a bit more distant to the light source. I don’t understand what is a “fixed” approach, is it a special term?
Alternative Z-depth Value

Woo [?] indicates that keeping the closest \( Z_n \) value in the shadow depth map may not be the best choice. A z-depth value between the closest \( (Z_n) \) and second closest surface \( (Z_f) \) would be a better choice: in practice, the mid-distance \( Z_m \) between the two z-depth values is chosen for implementation \( (i.e., Z_m = (Z_n + Z_f)/2) \) – see Figure 18.2 – thus we refer to this approach as the “mid-distance” approach in this book. If there is no second closest surface, then \( Z_m \) equals some large value. Thus the check \( \|x - L\| > Z_m \) indicates shadowing. In this case, the closest surface would always be in light, and the second closest surface would always be in shadow. This reduces the need to have a good bias value. However, when \( Z_n \) and \( Z_f \) happen to be very close together, then the original bad self-shadowing problem may still persist. Thus, it may still be useful to have the shadowing check as \( \|x - L\| > Z_m + \text{bias} \), where bias is some small value (but less dependent upon in most cases). Even with these modifications, there can be some shadow leaks at the intersections of surfaces, at the boundary between surfaces. For examples, refer to Figure 18.3, where we see two cases where the region labeled “not in shadow” is incorrect, because they are clearly in shadow, due to the usage of the mid-distance approach.

![Figure 18.3: Two cases where the mid-distance approach [?] is incorrectly labeled.](image)

PP2: inserted definition, should we (all authors) think to write a glossary for the book? A shadow leak appears as smaller band of shadow that spreads in an illuminated region, or respectively for an illumination or light leak in a shadowed region. Figure 18.4 illustrates both cases.

![Figure 18.4: (left) shadow leak; (right) light leak.](image)

To resolve problems found with the mid-distance extension, Snyder et al. [?] and Weiskopf and Ertl [?] independently arrive at a solution by comparing a depth value using a dynamically generated bias value that is the smaller of a (user-input) bias value and the \( Z_m \). In other words, the shadowing check is \( \|x - L\| > \min(Z_m, Z_n + \text{bias}) \). However, an appropriate bias value is still required from the user, though this bias value can again be more relaxed. As well, it is unclear whether this approach would really solve the self-shadowing problem any better. This approach also requires more
storage to store both $Z_m$ and $Z_n$ per shadow depth map pixel.

Wang and Molnar [?] also take a very similar approach to the mid-distance extension, but choose the second closest surface $Z_f$ as the z-depth value. The authors assume that all their surfaces are closed, which is often not the case in many applications, so that the second closest z-depth $Z_f$ would not be concerned about being in light, because the $N \cdot L$ check would already put it in shadow. This technique would also be unable to handle well the cases at silhouette edges or with thin objects.

Everitt et al. [?] implement the mid-distance variation [?] in the GPU by applying depth peeling [?]. Depth peeling works by first doing the shadow depth map pass, then the nearest surface is peeled away in favor of some combination with what is immediately behind it. The mid-distance variation is enticing for the GPU because it relaxes the need for high precision of the depth values, as the GPU textures used for the shadow depth map are typically only 16 or 24 bits per pixel.

1.3 Filtering Considerations

Because the basic shadow depth map uses the typical Z-buffer to compute z-depth values, the results tend to be aliased. Reeves et al. [?] use percentage-closer filtering with a basic Z-buffer to achieve anti-aliasing (Hourcade and Nicolas [?] employ a similar filtering scheme). This results in a blurred result rather than correct anti-aliased shadows, though the rendered results are quite pleasing in most cases. A filter region size is user selectable, where a large filter region fakes soft shadows. A typical region size is $3 \times 3$, in which z-depth comparison is done for some number of samples. The location of those samples can be stochastically chosen within this region so as to not needing to sample all $n \times n$ filter region per point to be shaded. The shadowing amount is the average of the shadowing hits based on each z-depth comparison – see Figure 18.5. To get high quality shadow depth maps in the GPU, Brabec and Seidel [?] implement the percentage-pixel filtering. Bunnell and Pellacini [?] also implement the percentage-pixel filtering in the GPU, using a fixed $4 \times 4$ texel region, by taking 4 samples within this region, to get some anti-aliasing.

![Figure 18.5: Percentage-closer filtering to reduce the shadow depth map aliasing.](image-url)
In the original paper by Reeves et al. [?], care must be taken around the boundary cases, especially at the spotlight’s cone of illumination. When sampling occurs outside this cone (due to the extra neighborhood pixels to achieve anti-aliasing), the result returned must indicate the sample is in full shadow. Otherwise, spurious brightly lit lines may appear at the junction between lit and shadowed regions. Reeves et al. provide source code that has this error. However, to confuse matters, in an augmented reality environment, where 3D graphics components are added to the live-action picture, such borders must indicate that those samples are fully lit (the reverse of what is recommended in normal situations). This is because the shadowing at those samples has already been taken into consideration in the digital picture.

Instead of using percentage-closer filtering, Keating [?] and Snyder et al. [?] store fragments per pixel, a depth value per fragment, and employ an A-buffer to anti-alias the shadow edges. Because there is fragment information per shadow depth map pixel, the shadow depth map resolution can be decreased to get a similar quality. Similarly, other buffer variations to achieve anti-aliased shadows include Tanaka and Takahashi [?] that use a cross-scan buffer, van Ee and van Overveld [?] that use an Ee buffer, and Isshiki et al. [?] that use an ED buffer.

1.4 Performance and Quality Considerations

High Quality Results with Performance Considerations

There are many considerations to accelerate the shadow depth map approach. For starters, in static scenes of flyby animations, a common shortcut renders the shadow depth map just once and reuses it for the entire flyby animation. This is because the shadow depth map is generated in world space (typically, except for the perspective shadow depth map [?]) and if the surfaces and light do not change, the shadow depth map should remain exactly the same for all frames. This particular shadow depth map resolution can even be higher (in case it represents the shadowing for an entire city) since only a single shadow rendering is needed. Similarly with moving object animations, a shortcut involving two shadow depth maps can be generated then merged – one for the static objects (rendered once), and one for the moving objects (much reduced complexity but rendered per frame). So during the shadow testing phase (for a point \( x \) to be shaded), it is necessary to test against both shadow depth maps. This is useful because in many applications, only a few surfaces in a large environment might be moving. As a result, the second shadow depth map tends to be smaller in resolution and faster to preprocess render as well as check during shading.

It is also prudent to consider rendering the smallest region possible for the preprocess shadow depth map generation. This gives higher quality results for a specific resolution. For example, for a car moving about a static road, lit by a directional light, it is not necessary to include the road into the shadow depth map, only the car would do. In this case, all the details of the car are included into the shadow depth map. The same can be said of spotlights, in which a region smaller than the spotlight angle of view can be used for the shadow preprocess render. However, keep in mind that quality can change for the same shadow depth map resolution if the region to be rendered changes – thus some of the above performance considerations should be considered for
entire units at a time, not surfaces which would fly apart and back during an animation (dramatically changing the size of the region to be rendered). See Figure 18.6, where the same shadow depth map can result in different quality due to the different coverage of 3D space, where the shadow depth map pixel between the two cases is of different sizes.

![Figure 18.6](image)

**Figure 18.6**: In an animated sequence, the size of a pixel in a shadow depth map with the same resolution can change.

Regardless of the above different flavors for the shadow depth map approach, it is crucial to realize that the shadow depth map generation preprocess can employ many, if not all, of the occlusion culling algorithms described in this book, in order to deal with large datasets very efficiently. These culling algorithms can be used because the shadow depth map generation preprocess is the same as a regular rendering, except that it is from the light source instead of the eye. For example, Govindaraju et al. use a potentially visible set (PVS) algorithm to cull out polygons that are not visible from the eye and are not illuminated. Those culled polygons do not need to be processed from a shadow standpoint any further. A shadow depth map is created to determine if the remaining polygons are fully lit, fully occluded, or partially occluded. Any polygon that is fully lit or occluded is easily handled. Any partially occluded polygon goes through a shadow-polygon clipping process, whereby the polygon is clipped between the lit and shadowed regions. To reduce the number of such partially occluded polygons which matter in the final rendering, a pixel-based criteria is chosen to avoid clipping against polygons whose clipped results would not be that visible anyways. Due to the above optimizations, the list of such partially occluded polygons should be small, and therefore the shadow-polygon clipping process should not be slow. The list of hybrid approaches can also be split up among three GPUs to render the shadows in real time. In the paper, the shadow-polygon clipping indicates having better quality than perspective shadow depth maps, but the complexity is far better than shadow volumes. However, a numerically robust shadow-polygon clipper remains difficult to implement.

**Alternatives to the Linear and Fixed Shadow Depth Maps to Achieve High Quality Results**

Instead of using multiple shadow depth maps along with some high resolution shadow depth maps (as described in Section 1.4), alternatives are available to achieve good quality while maintaining reasonable shadow depth map resolutions – they include applying some sort of hierarchical or adaptive resolution approach, a non-uniform shadow
SHADOW DEPTH MAPS

depth map approach, and a perspective shadow depth map approach. And unfortunately, none of the approaches can use most of the “tricks” as described in Section 1.4.

Hierarchical or Adaptive Shadow Depth Maps Fernando et al. [?] apply an adaptive resolution to the shadow depth map, focusing more resolution to the shadow boundaries. This way, they do not need to store large shadow depth maps to get a high quality. The criteria for higher resolution sections include z-depth discontinuity, and whose shadow depth map resolution is lower than the eye view resolution. Tadamura et al. [?] have a similar approach by using multiple shadow depth maps of varying resolutions. Such techniques look to run slower than the usual shadow depth map for typical cases, may suffer from quality discontinuities at the different resolutions’ boundaries, and cannot easily be implemented in the GPU to run fast.

Arvo [?] uses tiled shadow maps to achieve better quality results. The light view is initially divided up into equally sized tiles. The observation is that each tile should not be considered equal, i.e. each tile should be computed with a different resolution to maximize the eventual shadow quality. Each tile computes a weight which determines the resolution of the shadow depth map, where the file weight is the summation of the pixels’ depth (within that tile). The pixel weight is a function of the depth discontinuity (whether it contains shadow boundary), depth difference (between the occluder and receiver), surface distance (between the surface and eye view – higher resolution when the surface is close to the eye view, which is the basis of the perspective shadow depth maps, as discussed in Section 1.4).

The main concern with hierarchical or adaptive shadow depth maps is the discontinuity between the different resolutions, which could be visible in the shadow boundaries in which the quality of the shadows could shift in a vertical or horizontal fashion.

Non-uniform shadow depth maps Sen et al. [?] try to generate a piecewise linear approximation to the occluder’s silhouette region, so that the advantages of shadow depth map is retained, while the silhouette approximation would result in high quality shadow edges. A standard shadow depth map is created as usual, along with a silhouette map, which is computed using a dual contouring algorithm to store shadow edge information. During shading, if the four closest samples all agree on the shadowing occlusion (i.e., all in shadow, or all in light), then this pixel is considered to be in shadow or in light without any further computations. If, however, there are shadow occlusion differences, then the silhouette is found and silhouette map is used to determine the exact amount of shadowing for this current pixel. The main issue with this approach is that when edges are too close to each other, the boundary artifacts could remain because the shadow edges may not be easily identifiable. Similarly, Bujnak [?] also focuses on just the silhouette to achieve better shadow edge quality. Instead of shadow edges as in Sen et al. [?], silhouette information is encoded inside the shadow depth map using one or two quads per silhouette edge.

Aila and Laine [?], and Johnson et al. [?] independently come up with an approach where the actual shadow depth map (the map itself) is actually not required any more. Instead of a fixed, linear map that contains depth values, some non-uniform pattern is stored that provides depth information that is important to the visual quality. This
pattern is determined by eye-view visible points \((x,y,z)\) which require shadow information. These \((x,y,z)\) can be mapped to \((x',y',z')\) in light view space, and a \(z''\) can be computed in light space which represents the first visible hit for \((x',y')\). Thus shadow determination compares \(z'\) with \(z''\) – if \(z' > z''\), then \((x,y,z)\) is in shadow. To achieve fast computation of all the necessary \((x',y',z'')\) values as a preprocess (instead of one by one brute forced ray tracing approach), Johnson et al. \([?]\) use a non-uniform Z-buffer to compute the non-uniform set of values, and Aila and Laine \([?]\) use a 2D BSP tree. Then during actual rendering to determine shadowing, the \(z'\) and \(z''\) values are compared based on the \((x,y) \rightarrow (x',y')\) transformation to determine which light view pixel is being compared. While this approach is promising, the different preprocess step is slower than the standard shadow depth mapping, and the inability to apply some simple filtering \([?]\) can be inhibiting.

**Perspective Shadow Depth Maps** Stamminger and Drettakis \([?]\) introduce the “perspective shadow depth maps” to concentrate the depth map shadow details according to the current view by computing and comparing depths in post-perspective space. Thus we can get very high precision and high quality with respect to shadows seen close to the eye view. The shadows far away from the eye view are coarse, but then their coverage in screen space is small, so it should not matter (although Chong and Gortler \([?]\) found otherwise). See Figure 18.7, in which the two leftmost images represent the regular shadow depth maps, whereas the perspective and current view nature – as seen in the two rightmost images – allow the shadow depth map to concentrate on a certain section and covers more details in that section.

**Figure 18.7:** Perspective shadow depth map.

Perspective shadow depth maps have since attracted much research, since the basic approach is so promising, but has a number of issues that prevent it from being production ready, among them:

1. Martin \([?]\) indicates that, for the perspective shadow depth maps, the changes
1. **SHADOW DEPTH MAPS**

    in the near and far clipping planes over an animation may also cause shifts in
    quality.

2. The virtual camera issue, in which shadow occluders behind the camera is not
    numerically stable in the perspective space, and the suggestion by Stamminger
    and Drettakis [?] was to shift the camera back to handle such occluders. However,
    the numerical behaviors will worsen and produce poor quality results.

3. The shadow quality depends heavily on the light direction relative to the camera
    view [?]. If the light direction is pointed towards the camera, the shadow image
    quality can be poor. Chong and Gortler [?] also indicate poor quality in common
    situations when the surface is far from the camera.

4. Martin [?] and Kozlov [?] indicate that due to the non-linear behavior of the z-
    depth values, it is very difficult to come up with a good bias value (to avoid bad
    self-shadowing – see Section 1.2).

5. It is unknown how a point light, under 6 shadow depth maps forming a cube, will
    perform in terms of discontinuity artifacts between the 6 shadow depth maps.

To improve on issue #1 above, Martin and Tan [?] generalize the perspective shadow
depth map, and instead of post-perspective space, approximate the light frustum as a
trapezoid and represent shadow depth values in trapezoidal space. The trapezoidal
space also attempts to concentrate the shadow depth map information for close by sur-
faces. *PP2: for close by? AW is supposed to check.* As explained in Section 1.4,
the danger of concentrating shadow depth map information (present in both perspec-
tive and trapezoidal shadow depth maps) is that there can be jumps in quality over a
sequence of frames, because the region of the concentration can change significantly
over a frame. A scheme is devised to lower the frequency of the jumps by having the
trapezoidal transformation map the focus region to within 80% in the shadow map –
see Figure 18.8. This heuristic lowers but does not eliminate jumps.

![Figure 18.8: Trapezoidal shadow depth map.](image)

Kozlov [?] also studies the perspective shadow depth map in detail to resolve is-
sues 2-4, and focuses on the special cases that make perspective shadow depth maps
difficult to robustly implement. For the virtual camera issue (issue #2), the post-
perspective camera transform is reformulated such that a slightly larger area of cov-
erage is achieved, without having to extend back the lighting frustum to cause as much
image quality sacrifice.
For issue #3, Wimmer et al. [?] attempt to improve this problem by employing a logarithmic perspective transformation. As well, Chong [?, ?] interprets the perspective shadow depth map as an orientation selection of the shadow depth map projection plane, according to the visible points in the current view. He presents metrics to automatically choose an optimal orientation in 2D, indicating these could be extended in the normal 3D environments.

As for the bias issue (issue #4), it may be partially solvable by employing the mid-distance variation instead (see Section 1.2) though it has not been attempted. Kozlov [?] suggests instead an offset where

$$P_{\text{biased}} = (P_{\text{orig}} + L(a + bL_{\text{texel}}))M,$$

where $P_{\text{orig}}$ is the original point position, $L$ is the light vector direction in world space, $L_{\text{texel}}$ is the shadow map pixel size in world space, $M$ is the final shadow depth map matrix, and $a$ and $b$ are bias coefficients.

In general, while perspective shadow depth maps remain a very promising approach, the above problems are not solved robustly enough at this point to be used for production purposes.

**Simplifications to Achieve Performance**

Some GPU-based performance considerations and shortcuts can be considered. For example, Oh et al. [?] use shadow depth map results to project shadow textures onto receivers. The shadows are generated as a result of multi-pass rendering of the shadow textures with the receivers. The shadow textures do not take self-shadowing into account, and it appears that the approach is only suitable for only one light source. To further improve performance, Nagy [?] discusses some optimizations, and Isard et al. [?] distribute shadow depth map generation, as hardware textures, over texture units on each graphics card.

In the case of games, cheating variations of the shadow depth map are often used. For example, in the game “Outcast” [?], only moving objects need to have their shadows computed dynamically using the shadow depth map. For those objects, instead of checking all the pixels of the screen for shadow determination, only the outline of the shadow is processed. First, one point in shadow is found. Search through the scanline to find the first point of the outline of the shadow. Then start a shadow boundary detection algorithm that will retro-project and check only the outline of the shadow. The algorithm generates span lists that are rendered later. *PP: the algo here should written in proper sentences...* Because typically only one object is moving, the boundary detection should be simpler and non-conflicting since there are no other objects that come into play.

In another context, to achieve shadows in between view-frames, Chen and Williams [?] interpolate the results between a fixed set of shadow depth maps so that not all frames require shadow depth map generation. Such a technique can be extended for stereo viewing as well.
Memory Issues

From a memory standpoint, the storage expense of shadow depth maps for multiple lights can become prohibitive when one shadow depth map is required per shadow casting light. One simple way to reduce memory usage is to tile each shadow depth map and store the tiled shadow depth maps on disk. Each tile can employ a simple RLE (runlength encoding) compression, where z-depth increments can be used as runlengths. A least-recently-used (LRU) array of tiles is stored in RAM, and when a shadow sample is needed, its corresponding tile will be loaded from disk, decompressed, and added to this LRU array. Because we are usually sampling in the same area, the decompressed tile would be useful for quite a number of consecutive samples. When the LRU array is full, least-used tiles are dumped in favor of incoming tiles. This scheme is effective because the LRU array keeps the memory usage to a constant, and the decompression of RLE-tiled shadow regions is very fast. While this is not talked about from a software perspective, other than some mention of a LRU scheme by Fernando et al. [?], variations of this have been implemented in most GPU.

Different level-of-detail geometry is used quite often to reduce the memory used in a renderer. However, when using different LOD representations between the eye view and the shadow depth map generation process, there can be bad self-shadowing problems because the depth values of the different representations may cause discrepancies during comparison of depth values. An example of LOD can come from optimized tessellation of higher order surfaces (such as NURBS or subdivision surfaces), or on-the-fly tessellation of displacement mapped surfaces [?], based on the eye view, and the pre-generated shadow depth map needs to make sure that its optimized tessellation for the light view cannot be too different from the eye view. Though this problem is an issue for most shadow algorithms, care must be taken to avoid this problem in particular for the shadow depth map because the shadow depth map generation pass (from the light’s point of view) is entirely separate of the eye view rendering.

Another example of the different LOD comes from the hybrid rendering of points and polygons. The premise is that for all those polygons that are much less than the area of a pixel, there is no need to spend all the time rasterizing, interpolating the polygonal information. In those cases, Chen and Nguyen [?] replace the tiny polygons with points, and got much faster rendering times. Shadows were not considered in the paper. Since the choice of point or polygon is determined on the fly, based on the view, care must be taken to make sure that there is no improper self-shadowing due to the tiny polygon shadowing the replaced point. However, fixing a “depth” for each tiny polygon, encoded in the replaced point, may help resolve the self-shadowing issue, but may also miss proper self-shadowing within the region represented by a single point. Guthe et al. [?] employ perspective shadow depth maps for the above hybrid rendering, but have not addressed the above self-shadowing issue.

1.5 Miscellaneous Considerations

Zhang [?] introduces the concept of “forward shadow mapping” to exploit spatial coherence in the shadow depth map when rendering. The shadow depth map is computed as usual. Then rendering from the eye view is done, without accounting for shadow-
ing. Then each shadow depth value per pixel is warped to the eye view. Each eye view pixel’s visibility depth value is compared against the warped shadow depth value. If they are approximately the same, then the eye view pixel is lit. The lit/non-lit status is put into a modulation image. The modulation image is blurred to achieve anti-aliasing of the shadow edges. This has the advantage that the shadow depth map pixels, as well as eye view pixel values are accessed in order, so that spatial coherence is achieved. This also has the advantage in the GPU because a hardware texture map is not needed to represent the shadow depth map, and texture maps are generally used extensively. The main problems with this approach are that the extra preprocessing may not warrant the spatial coherence advantage, and the anti-aliasing may not generate high quality results over an animation.

Brabec et al. [?] identify some quality problems with the shadow depth map, with the source of the problem coming from unevenly spaced depth values and not-so-tight near and far clipping planes. Mostly due to low precision, such as in the GPU, it is important to have evenly spaced depth values – the most common technique actually stores $1/z$, which would be non-linear. While this artifact can be seen in a pure software implementation in terms of eye visibility, such as with a very large scene that are sparsely distributed but densely distributed in the occupied regions, visual artifacts from this problem are generally very rare. This may be due to either the software implementation having high enough precision, or the mid-distance variation [?] allowing such problems to go unnoticed. As for the tight near/far clipping planes, it can also be a problem with respect to eye visibility, in which tight near/far clipping planes for rendering would result in much more stable results. Brabec et al. [?] suggest tight near/far clipping planes that are not axis-aligned, and do not need to encompass all shadowing objects. While this is a good idea in general, it should also be a concern that an animation with too-large variations in near/far clipping planes may show artifacts over the animation due to noticeable shifts in numerical precision.

2 Shadow Volumes

Although the earlier description of shadow volumes appear simple, details as to how to compute the initial shadow count, how to intersect the shadow polygons to advance the shadow count, and performance issues need to be considered. As well, GPU implementations for shadow volumes are possible and much research has been involved in this area.

2.1 Visibility Algorithms to Intersect the Shadow Polygons

The description in Section 3.2 is actually vague because it does not address how the shadow count incrementing and decrementing is achieved. The easiest solution is to employ a Z-buffer for shadow count incrementing and decrementing [?]. The Z-buffer is first used to render the non-shadowed pass. Then the shadow polygons are scan-converted into the Z-buffer, and any shadow polygon that resides before the z-depth will be used to increment or decrement the shadow count. In fact, the Z-buffer approach
is almost identical to the stencil buffer based method directly extended for implementation in the GPU [?]. A variation of the Z-buffer approach to employ a scanline-based solution has also been achieved [?].

Slater [?] suggests that it might also be interesting to apply a light buffer [?] approach (see Section 3.1), in which an hemi-cube surrounds the light source. Each cell (pixel) in this light buffer indicates the surfaces that projects in it. Thus to determine the shadow count, the eye ray only visits the relevant cells to get at a small candidate set of shadow polygons that the eye ray might hit. Slater [?] compares this light buffer approach with the SVBSP approach [?] (explained in Section 2.5) and found no conclusive evidence which approach provides faster rendering speeds.

Eo and Kyung [?] suggest a ray tracing solution, in which the shadow polygons are scan-converted into a voxel data structure, and the viewing ray traverses the voxels to find the candidate shadow polygons it should intersect to increment or decrement the shadow count. Of course, this is quite slow, as Eo and Kyung are trying to use it in conjunction with regular voxel traversal – see Section 3.1.

The Z-buffer (and in some cases, scanline-based) variations tend to be the more commonly accepted implementations. And to be shown in Section 2.4, the Z-buffer for shadow volumes in the GPU is employed almost universally in the form of a stencil buffer. Due to the Z-buffer and stencil buffer approaches, the only ways to anti-alias the shadow results is to increase the resolutions of the required buffers. The exception is the slower ray tracing approach [?], in which adaptive and/or stochastic samples can be used to anti-alias shadows.

2.2 The Initial Shadow Count

When the eye is outside of all shadow volumes, then the initial shadow count is 0. Otherwise, the initial shadow count needs to be correctly set as the number of shadow volumes that contain the viewing position. In addition to generalizing shadow volumes for non-planar polygons, Bergeron [?] indicates the need to close (cap) shadow volumes so that a correct initial shadow count of how many shadow volumes the eye is within can be computed. Then by rendering of a single (arbitrary) pixel, and counting all shadow polygons that pierces this ray – call this \( E_{svc} \) – then the initial shadow count is simply \(-E_{svc}\).

Another simpler way is to treat it as a ray tracing problem, in which a ray is shot towards the light from the view point, to count the number of hits \( E_{svc} \) that the ray encounters – then \( E_{svc} \) is the initial shadow count. Since this only needs to be done once per rendering, this expensive ray tracing solution is quite reasonable.

For software-based solutions, the initial shadow count can be a single value. In GPU variations, where the shadow polygons are clipped to the near clipping plane, the initial shadow count may be different for particular pixels – see Section 2.4 for detailed coverage of how to deal with initial shadow counts when using the GPU.
2.3 Reducing the Complexity of Dealing with Large Number of Shadow Volumes

There have been a number of ways to reduce the complexity of the general shadow volume approach. As Slater [?] points out, a shadow volume completely enclosed within another shadow volume can be eliminated – to improve the chances of such cases occurring, conservative occlusion culling techniques as discussed in this book can be used where facades’ PP: need to refer to section in book shadow volumes can replace other objects’ shadow volumes which reside in the facades’ shadow frusta. Also by processing shadow volumes in a front-to-back order from the light source, simple configurations of shadow volume clipping can reduce the extent of shadow volumes closer to the light source. For example, in Figure 18.9, occluder A’s shadow volume can eliminate the need for the shadow volume for occluder B, as a result from this sorting.

![Figure 18.9: Eliminating the need for enclosed shadow volumes.](image)

Crow [?], Batagelo and Junior [?], and Bestimt and Freitag [?] eliminate many redundant shadow polygons within a single polygonal mesh by exploiting some silhouette identification. These silhouettes mainly correspond to the polygon edges that have different $N \cdot L$ positive/negative values (the view-dependent boundary case), and in turn also edges that have no other shared edges (the view-independent boundary case). Such polygon edges are the only edges that require the generation of shadow polygons – i.e., any internal polygon edges do not need shadow polygon generation. In fact, an interesting observation is that the degree of a silhouette vertex is even [?], where the degree $n$ means the vertex on such silhouette can be connected by $n$ silhouette edges.

As well, the original mesh (i.e., the light cap) forming the occluder needs to be part of what the shadow count needs to consider. However, those polygons can be culled because of their irrelevance to the shadow count computations [?, ?]. For example, if the occluder is between the light and the viewpoint, and the view direction is pointing away from the occluder, these polygons are not necessary for consideration for the shadow count – see Figure 18.10 (left). Similarly, if the viewpoint is between the occluder and the light, and the view direction is pointing away from the occluder, then these polygons do not need to be considered for the shadow count computation – see Figure 18.10 (right). In fact, in the latter case, all the shadow (silhouette) polygons for this mesh can be culled from any shadow count computation. This work can reduce the fill rate for GPU based solutions even more – see Section 2.4.

A very interesting contribution in reducing the complexity of the general shadow volume approach comes from McCool [?]. A standard shadow depth map is first cre-
2. Shadow Volumes

Z-pass Algorithms

As indicated in Section 2, Heidmann [?] is one of the first to present a GPU implementation of the original Crow [?] approach – this is referred to the Z-pass approach. The exact steps to achieve Z-pass shadow volumes include:

1. enable the depth buffer in order to render the scene (without shadows) onto the color buffer;
2. clear the stencil buffer and disable depth buffering;
3. draw all front-facing shadow polygons and increment the shadow count for the relevant pixels in the stencil buffer;
4. draw all back-facing shadow polygons and decrement the shadow count for the relevant pixels in the stencil buffer;

The interesting interpretation of the McCool approach is that it represents a global silhouette approach, while the approaches by Crow [?], Batagelo and Junior [?], and Bestimt and Freitag [?] represent localized silhouette approaches (localized to just the mesh itself).

2.4 Shadow Volumes on the GPU

Figure 18.10: Reducing the number of silhouette edge shadow polygons in the shadow volume algorithm.
5. darken the pixels in the color buffer where the stencil values are not equal to 0.

Unfortunately when using the GPU, shadow polygons must be clipped against the viewing frustum, thus introducing erroneous initial shadow counts for some of the image pixels if the eye view is inside the scene. One simple way to resolve this is to compute the value $E_{svc}$ for the eye point in terms of its initial shadow count, then initialize the stencil buffer with $E_{svc}$ instead of 0 [?]. This method is fairly accurate if the near clipping plane is very close to the eye point. However, if the near clipping plane is not close to the eye point or the view frustum is large, the single eye point’s $E_{svc}$ stencil initialization value may be different for the region occupied by the near clipping plane. Figure 18.11 (left) shows that a single initial shadow count is valid, but it is not the case for the Figure 18.11 (right), where the shadow volume cuts across the near clipping plane. Some authors [?, ?, ?] propose to cap the shadow volumes with new shadow polygons, where the shadow volumes are clipped against the viewing frustum’s near clipping plane. However, Everitt and Kilgard [?] explain why it is difficult to robustly implement this capping technique that needs to account for all the different combinations.

**Figure 18.11:** Potential errors in the initialization of the eye shadow count $E_{svc}$ when the near clipping plane intersects some shadow polygons.

**Z-fail Algorithms**

Another set of solutions [?, ?, ?] to deal correctly with a correct initial shadow count, starts from the observation that this point-in-shadow-volume test could be evaluated along the line of sight, but from the visible surface to infinity instead (called Z-fail test). Thus the shadow count for the viewpoint is simply initialized to 0, occluded back-facing shadow polygons increment their shadow counts, front-facing ones decrement them, and shadow polygons need not be clipped by the near plane. A shadow count greater than 0 indicates shadowing. However capping shadow volumes against the rest of the viewing frustum is still necessary to produce correct shadow counts. See Figure 18.12 for visual examples of the Z-pass versus Z-fail situations. Note that Z-fail can be considered the computation of the Z-pass in the reverse direction.

**Figure 18.12:** Figure 1?

Some hardware extensions [?, ?] avoid the need to compute correctly the capping with respect to the viewing frustum. Another solution, in software, simply eliminates
2. SHADOW VOLUMES

the far clipping process with a small penalty to depth precision; as a bonus, capping at infinity for infinite light sources is not necessary as all shadow polygons converge to the same vanishing point. Everitt and Kilgard [?] and Lengyel [?] realize that the Z-pass approach is generally much faster than the Z-fail, they detect whether the camera is within a shadow volume, and employ Z-pass if not, and Z-fail if so. However, the sudden shifts in speed difference may be annoying to the user.

Reducing the Fill Rate

Notice that many of the optimizations applied in the CPU, as discussed in Section 2.3, can be applied to the GPU to reduce the fill rate. More GPU specific fill rate reduction approaches are discussed below.

For both the Z-pass and Z-fail approaches, additional performance implications need to be considered (mostly scan-conversion) in terms of reducing the fill rate of the shadow polygons onto the stencil buffer, i.e., the smaller amount of coverage on the stencil buffer, the better the performance. An example of reducing the fill rate comes from Lengyel [?] and McGuire et al. [?], in which they reduce the shadow polygon sizes for point lights. The intensity of the light decays as the square to its distance. The shadow polygons are therefore clipped at a distance where the intensity becomes so small that it does not change the shading. Lengyel uses a sphere (around the point light) to bound the shadow polygon sizes, and McGuire et al. use a tighter bounded cube.

Lengyel and McGuire et al. also improve the fill rate by reducing the number of polygons that need to be included in the scan-conversion and computation of the shadow count for the stencil buffer – see Section 2.3. Lloyd et al. [?] reduce the fill rate for shadow volumes which do not contribute to the final image, by introducing three culling and clamping approaches:

- Shadow volume culling: culling shadow volumes in a manner very similar to Slater’s work [?] (see Section 2.3).
- Continuous shadow clamping: shadow volumes are clamped by using axis aligned bounding boxes around the shadow receivers to compute clamped volumes.
- Discrete shadow clamping: shadow volumes are clamped to intervals defined by slicing planes.

Finally, if there is an easy way to detect whether the shadow polygons of a mesh actually do not intersect any view-visible geometry, then this mesh’s shadow polygons do not need to be considered at all because they will only contribute to shadows that are not visible to the current view and can thus reduce the fill rate. This consideration will likely optimize the Z-fail approach more than the Z-pass approach. This approach was hinted at from McGuire [?], and attempted in a hierarchical shadow volume algorithm [?] as well as in an hybrid shadow rendering algorithm [?]. In the hierarchical shadow volume algorithm [?], the screen is divided up into many $8 \times 8$-pixel tiles. If the shadow polygons do not intersect the bounding box formed by the objects within this $8 \times 8$ tile, then the points within this $8 \times 8$ tile is either entirely in shadow, or entirely in light, and
the fill rate can be reduced because the stencil value for this $8 \times 8$ tile will be the same, and can be computed with any arbitrary ray through the $8 \times 8$ tile. This approach has shown data that improves the performance greatly, because only the shadow boundaries will require the scan conversion, thus greatly reducing the fill rate. In the hybrid shadow rendering algorithm [?], the shadow depth map is initially generated. For the non-silhouette regions, we know exactly whether the region is entirely in shadow or entirely not in shadow. For the silhouette regions, shadow volumes are used to determine exact shadowing, thus significantly reducing the fill rates. How these two similar approaches compare in terms of performance is unknown.

**Other Considerations**

Brabec and Seidel [?] use the GPU to compute the actual shadow polygons, to further speed up shadow computations, as well as avoid differences in numerical properties (between the CPU and GPU) that can result in some shadow leaks. Supposedly there is a McGuire and Hughes 2004 on this topic as well “Contour rendering on programmable hardware”. Proc. NPAR, June 04. PP: must check this reference as NPAR is done. In fact, numerical instability issues can potentially arise due to shadow polygons created from elongated polygons, or when the normal of the polygon is close to perpendicular to the light source direction. Care must be taken in dealing with such cases.

If the silhouette optimization techniques discussed in Section 2.3 are not used, then there are likely other numerical problems. In particular, because all shadow polygons (not just the silhouette) are created for shadow volumes, there are likely visibility problems due to the limited precision of the $z$ values – this problem is sometimes referred to as Z-fighting.

Roettger *et al.* [?] replace the stencil buffer by the alpha or screen buffers (because the stencil buffer can be overloaded in its use to cause bottleneck), and the computations of shadow counts by blending operations – *i.e.*, replace the increment and decrement shadow counts with multiplying and dividing by 2 operations in the alpha buffer, respectively. As shadow polygon fill rate is the main bottleneck in shadow volume approaches, these two buffers can be computed at lower resolutions (at the additional cost of a re-rendering at this lower resolution), copied into textures, and shadows treated exploiting texture bilinear interpolation. However, the basic approach may have incorrect results with specular highlights, and may suffer with complex objects in terms of the performance that can be achieved. PP: *I am not sure to understand. PP2: we seem to both feel this is fishy, probably we should leave as is.*

### 2.5 Other Shadow Volume Variations

Another variation of shadow volumes employs the combination of shadow volumes and binary space partitioning (BSP) trees [?]. A BSP tree is constructed that represents the shadow volume of the polygons facing the light. The shadow determination is computed by filtering down this shadow volume BSP (known as SVBSP) tree. This variation is quite different from the standard shadow volume approach because there is really no shadow incrementing nor decrementing, and also has the advantage that there
is no need to compute an initial shadow count. This variation tends to be inappropriate for high complexity scenes however.

For example, Figure 18.13 illustrates what the SVBSP tree looks like with the insertion of polygons (edges) \(ab, cd,\) and \(ef,\) respectively. Note that clipping is needed to ensure non-overlapping cases, which means numerical stability issues must be handled with care. For a particular point \(x\) (to be shaded), its shadow status can be determined by traversing this tree – left and right traversal is determined where \(x\) is with respect to the slices (at each node).

In terms of dynamic scenes, Chrysanthou and Slater \([?]\) employ a BSP tree and a shadow tiling approach (in the form of a hemi-cube) to get quick updates in dynamically changing scenes. The same authors \([?]\) then extend the SVBSP tree \([?],\) to get dynamic updates in dynamic scenes by being able to quickly add and delete changing elements of the scene, as well as merge the modified SVBSP tree.

### 3. Ray Tracing

The basic ray tracing approach to compute shadows is very simple. Thus just about all of the research has been focused on how to speed up the ray tracing approach, in this context, for shadow generation. The methods of acceleration include: shadow culling algorithms, dealing with many lights, and speeding up anti-aliasing.

### 3.1 Shadow Culling Algorithms

There are many intersection culling algorithms to accelerate ray tracing, and many have been discussed in detail in other references \([?].\) Of particular interest for shadow algorithms include light buffer \([?],\) hybrid shadow testing \([?],\) and voxel occlusion testing \([?].\) These algorithms are discussed because parts of their algorithms have shown to be useful for real-time purposes.
Light Buffer

The light buffer \([?]\) consists of 6 grid planes forming a box surrounding the point light source \(L\) – see Figure 18.14. Each cell of the buffer contains information on the closest full occlusion from \(L\) (closest surface that occludes the entire cell) and sorted approximate depth values of candidate occlusion surfaces that project in the cell. For each point to be shaded, if the depth value of its corresponding light buffer cell is greater than the closest full occlusion distance, then this point is in shadow. Otherwise, shadow determination requires intersection tests with candidate occlusion surfaces of the cell. They are performed inorder with respect to the depth values (starting with the surface closest to \(L\)) until either an intersection hit is found (in shadow) or the depth value of the candidate occluding surface is greater than the intersection point (not in shadow). This data structure has been employed in some variation of shadow volume algorithms \([?]\).

Figure 18.14: The light buffer of a point light source. Each pixel of the light buffer contains a sorted list of encountered objects up to an object producing full occlusion.

Hybrid Shadow Testing

Hybrid shadow testing \([?]\) uses a voxel data structure to store the shadow polygons as invisible surfaces (as in shadow volumes of Section 2). The shadow count is updated as the ray traverses the voxels. No shadow rays need to be generated for this scheme, but intersection calculations with the shadow polygons in traversed voxels are necessary. When the closest intersected surface is found, the shadow count is checked. If the count is 0, then the surface is not in shadow; otherwise the surface is in shadow. However, since it might need to deal with a large number of shadow polygons, the implementation resorts to the traditional shadow ray approach under such circumstances.

Voxel Occlusion Testing

For voxel occlusion testing \([?]\), each voxel uses up an extra 2 bits per light source. Its value indicates the level of opaque occlusion of the voxel with respect to each light source: full occlusion, null occlusion, and unknown occlusion – see Figure 18.15. The shadow umbra of all surfaces is scan-converted into the voxel occlusion bits, such that if the voxel entirely resides inside or outside the shadow umbra, then the voxel occlusion
value is full or null occlusion, respectively. If the voxel contains the boundary of the shadow umbra, then it is marked with unknown occlusion. When the view ray intersects a surface in a voxel that has either full or null occlusion, then any intersected surface in the voxel must be in shadow, and thus no shadow rays need to be generated. If unknown occlusion is found instead, then the fastest method of shadow determination is to resort back to voxel traversal of the shadow ray. But as each voxel is traversed, its occlusion value is checked – a quick exit condition for the shadow ray traversal is available if any of those traversed voxels show a full or null occlusion value, which then will again conclude that the point to be shaded is fully shadowed or fully lit, respectively.

For interaction applications, a high enough voxel resolution may be used to reduce the occurrence of unknown occluded voxels. For example, Schaufler et al. [?] implement a variation of voxel occlusion testing [?] using an octree. Because the lights and models are not changing during the visualization, the fixed shadow occlusion values can be embedded in voxels to indicate whether a certain voxel region is in shadow or not. They also improve the detection of full occlusion due to combined occlusions (occluder fusion) of disjoint occluders. When the voxel contains regions that are in shadow and some region not in shadow, and would need additional shadow rays to be shot to determine the shadow occlusion. PP: I feel this last sentence does not fit with Schaufler...

Other Considerations

Other papers [?, ?, ?, ?, ?] try to accelerate the computation of shadows from simple lights in the context of using voxels as the culling approach; they have not enjoyed as much success as hoped. This is likely due to the performance bottleneck in any renderer no longer being the reduction of floating point computations (as was the goal of the above papers), but rather, locality of memory references within RAM that has become the bottleneck with modern platforms.

3.2 Dealing with Many Lights

In scenes that contain many shadow casting lights, it becomes quite too costly to compute shadows for each point to each of those lights. To achieve a similar looking result, a number of algorithms have been proposed. Ward [?] determines, among all the lights,
an ordering of which lights contribute most to the current point to be shaded. He only calculates lighting and shadowing based on the lights that have significant contributions. As a result, shadow rays only need to be shot to some small number of lights per shaded point.

Shirley et al. [?, ?] attempt to find good probability density functions, using Monte Carlo integration, for direct lighting of extended light sources (for soft shadows) as well as for a large number of lights. The algorithm subdivides the scene into voxels, and for each voxel, separates the set of lights into either an important and unimportant set. While each light in the important set is sampled by shooting shadow rays per light, only one arbitrary light from the unimportant set is chosen for the shadow ray shooting to represent the entire unimportant set of lights. The use of exactly one shadow ray provided good results, however, a few shadow rays may generate even better results over one shadow ray. The optimal number of shadow rays to achieve accurate enough quality has not been researched in this work.

Fernandez et al. [?] also subdivide the scene into voxels, more specifically octrees. The basic preprocessing results in a voxel indicating whether the entire space occupied by the voxel is fully occluded, partially occluded, or totally unoccluded for each light. The storage is quite large: for each light, fully occluded voxels store nothing, totally unoccluded voxels store a pointer to the light, and partially occluded voxels store a pointer to the light and links to potential occluders for that voxel. During the shading phase, for fully occluded voxels, nothing needs to be done; for totally unoccluded voxels, the lighting equation is computed; for partially occluded voxels, shadow rays are shot to determine the shadowing. To accelerate the case of many lights, Weber’s law (Equation 18.1) is applied to totally unoccluded and partially occluded voxels to identify lighting that could be eliminated from computation, based on quick evaluations of whether each lighting case per voxel contributes that much to the final shading.

\[
\text{(18.1)}
\]

Note that though the above work has been done with respect to ray tracing, it can be extended to some other shadow algorithms as well, as the point of the above work is to avoid shadow computations for each point for all lights. In addition, though the above research work is very exciting, it is still quite immature and not used in production work. In production work, in order to save rendering time, there may be a lot of lights, but only a few lights are tagged for shadow calculations. As well, options such as associating certain objects with certain lights (so that the object is lit by only those lights to reduce rendering computations) and its shadowing are available in most software renderers. Finally, there are also options to select certain surfaces as shadow casters (or not) or as shadow receivers (or not) — i.e., not all surfaces need to be considered for shadowing. PP: maybe we should make an entire section on production habits with virtual lights if we can find info?

### 3.3 Anti-aliased Shadows

To get better anti-aliased (hard) shadows without shooting many shadow rays all the time, some cone tracing [?] (see Section 3.2) or beam tracing [?] extensions have been
4. IMAGE-BASED RENDERING

implemented. For example, Genetti and Gordon [?] use cone tracing [?] not to determine visibility, but as an indication as to how many shadow rays need to be shot and in which section of the cone to get good anti-aliasing. The criteria of the presence of different polygons is used to determine how complex that cone region might be, and thus drives how many rays need to be shot. This may be ideal for parametric surfaces, implicit surfaces, or large polygons, but not ideal for polygonal meshes, as polygonal meshes will always require many shadow rays to be shot based on the above criteria. Similarly, Ghazanfarpour and Masenfratz [?] use beam tracing [?] to get anti-aliased shadows. It encapsulates the different cases in which the objects will intersect the beam. This approach is again mainly ideal for large polygons, but not polygonal meshes, because the meshes will slow down the approach.

Other faster anti-aliased shadow algorithms involve acceleration with information from the current pixel’s ray results. For example, Woo [?] assumes that anti-aliasing is achieved by some form of adaptive super-sampling [?]. However, instead of always shooting a shadow ray when subpixel samples are needed, the ray trees of the surrounding subpixel samples are checked. Additional shadow rays are shot only if the invocation of the subpixel samples is due to shadow aliasing. Otherwise, the current subpixel sample’s shadow value is the same as the surrounding subpixel samples.

4 Image-based Rendering

PP: Andrew has had little experience with IBR, so this section looks weak...

Image-based rendering techniques [?] make use of “impostors” in a scene. These impostors replace many detailed geometries that can be rendered very quickly, usually using dynamically changing GPU textures. The image-based rendering techniques allow such impostors to be used in certain views and can interpolate the impostors based on the changing views. Impostors are particularly useful in urban environments because of the many buildings and features (e.g., trees) that can be faked almost seamlessly.

Zhang and Nakajima [?] experiment with shadowing of impostors from trees. Features of the trees are captured as irregular lines and shadow generation is produced based on this assumption – the results are not accurate, but for background features, that is not nearly as crucial. They [?] then apply computer vision techniques to extract shadow silhouettes from image-based objects to achieve shadows on a planar ground. They [?] also combine that work by assuming very simple geometric form from buildings in an urban environment, and extract the shadow silhouettes from simple buildings to achieve shadows. Qin et al. [?] use 2D buffers and smaller resolution voxels to approximate different aspects of shadowing.

Meyer et al. [?] render many trees on a landscape by also using image-based rendering techniques. A hierarchy of bidirectional textures is used to store different LODs to represent the tree, and the appropriate LOD texture is sent to the GPU. Shadowing is accounted for in the following combinations: a hierarchy of visibility cube maps is built where each tree’s cube map accounts for visibility of other trees and other geometries (e.g., landscape); self-shadowing of the tree onto itself is taken care of in the
bidirectional texture functions (BTFs) and horizon mapping techniques [?]; a shadow depth map [?] is used to incorporate all of the above.

Dinh et al. [?] generate shadows from impostors assuming only planar receivers such as a ground plane. The silhouette, from the perspective of the light source, is identified for one of the pre-rendered views (of the impostor). This silhouette, represented by shadow vertex locations, is projected onto the planar receiver to match the object as seen from any interpolated view. Loscos et al. [?] represent humans walking in virtual cities, in which each human is an impostor made of hardware textures. Shadowing of buildings onto the humans is achieved by representing the impostor humans with height fields. The height fields are combined with shadow depth maps to calculate shadows.

5 Additional Reading on Other Hard Shadow Algorithms

PP2: I feel this is a bit wimpy as getting out of explaining these minor papers...

A number of algorithms have investigated for the specific cases of convex polygonal meshes [?] and the use of GPU textures [?, ?]. Rather than providing insufficient details, we simply refer the interested reader to their original papers.

6 Atmospheric Shadows

Sunlight scattering in the air causes the atmosphere to glow. This glow is particularly visible in presence of shadows. Thus a ray shot to determine the closest visible surface, the critical question is not just whether the intersection point is in shadow. The segments along the ray (not shadow ray) that are visible from the light are just as crucial – see Figure 18.16. The information is necessary to acquire atmospheric shadows assuming only a single scattering model for light diffusion. If the ray is not illuminated, then the shading calculations are the same as in the original illumination model, including direct attenuation due to the media. However, with partial illumiance, an additional component (atmospheric shadows) is included.

6.1 Using Shadow Volumes

Max [?] uses shadow volumes to calculate the ray segments that are visible from the light to achieve the above, employing a single scattering model. This approach can be extended to voxel occlusion testing [?], in which the voxel occlusion values that the viewing ray traverses are used to determine which regions illuminate or shadow the ray segment. The span of the ray segment within each voxel can be trivially computed, since it is readily available in voxel traversal algorithms [?]. Similarly, Ebert and Parent [?] also use subdivision grids to accelerate atmospheric shadows.
6.2 Using Shadow Depth Maps

It is possible to achieve atmospheric shadows from shadow depth maps as well, assuming a conical atmospheric effect from a spotlight. Instead of comparing depth values of the point to shade, the viewing ray is projected to the shadow depth map pixels, and the depth comparisons are done per shadow depth map pixel. An accumulation of the shadow tests determines the amount of shadowing that occurs in the atmospheric effect. However, doing the comparison per shadow depth map pixel can be quite expensive because the viewing ray can potentially extend through the entire shadow depth map. To improve speed, only every (stochastically chosen) \( n \)th shadow depth map pixel comparison is done – this can result in noise in the atmospheric shadows however.

6.3 Integrating It into Radiosity

Rushmeier and Torrance [?] apply zonal method used in heat transfer to the computation of radiosity (see Section 6). Their method discretizes the medium into small volumes for which the form factors volume/volume and volume/surface are calculated. The shadows are generated with the hemi-cube extended to a full cube.

7 Semi-transparent and Translucent Surfaces

Most shadows from semi-transparent surfaces are accounted for by ray tracing approaches [?, ?]. PP2: I will try to get a few sentences in for these two papers.

For faster computations, shadow depth map variations have been developed to account for such shadows, though the semi-transparency is assumed to be linear and
causes no refractions of light.

Lokovic and Veach [?] introduce “deep shadow maps”, in which a shadow map pixel contains a beam (extension of the pixel in 3D space) of transmittance information. Each point to be shaded accesses its shadow depth map pixel, determines where it resides with respect to the transmittance function, and returns a shadow (attenuation) result. Because the transmittance function carries a lot more information per shadow depth map pixel, the authors indicate that a lower resolution deep shadow map can match the quality (i.e., capture the details as well as) of a much higher resolution regular shadow depth map.

Kim and Neumann [?] use alpha blending in the GPU to achieve transparency. Slices of opacity map planes, each perpendicular to the light source, are created. For each opacity map plane, the opacity of each pixel is determined for the region that crosses the opacity map plane and encloses the opacity pixel. Then during shading, the point to shade is projected to the corresponding opacity plane and pixel, and the shadow occlusion from the opacity is interpolated between the two opacity map planes that contain the point to shade.

Dachsbacher and Stamminger [?] use a structure based on the shadow depth map to store depth, surface normal, and irradiance of the directly illuminated points of a translucent object. To efficiently determine the sub-surface light contribution at a 3D point, the point is projected into the translucent shadow map and the values within a determined shadow map radius are hierarchically filtered with a pattern of 21 samples according to a formulation for the translucency computation. The technique is suitable for current hardware implementation, and they achieved interactive to real-time display rates for polygonal models between 10K to 100K triangles. The current technique is however limited to translucent objects for which all visible points can be linked by a line segment within the translucent material to the contributing points directly illuminated by the light source.

8 Other Geometry Types

8.1 Higher Order Surfaces

Rendering of higher order surfaces, such as parameteric (e.g., NURBS), subdivision surfaces, and implicit surfaces (including blobbies), can be achieved either through direct rendering of the surfaces applying numerical iteration techniques, or polygonization. Many numerical iteration techniques have been proposed but numerical robustness to guarantee correct intersection hits can be difficult to achieve results in potential holes in the renderings. Let us simply point to the following work [?, ?, ?, ?] for implicit surfaces and [?, ?, ?, ?, ?] for parametric surfaces.

Polygonization - tessellation of the higher order surface into a polygonal mesh – is the more often used technique in industry since it can then apply any of the rendering and shadowing algorithms described in this book, the rendering can be fast because the polygonal mesh can be accelerated in the GPU, and most importantly, it allows to handle the different
higher-order surfaces can be self-contained. However, polygonization has its costs, including the need for larger amount of memory to deal with the polygonal mesh, as well as the need to approximate a smooth surface with sufficient number of polygons, and the need to deal with the shadow terminator problem (as described in Section 4.1). See Figure 18.17 of an automobile tessellated from NURBS surfaces, with shadows rendered.

![Figure 18.17: Shadows from a tessellated car. PP: the left and right images provide any additional information, or just two views?](image)

Many other geometry types also employ polygonization for the final rendering and shadowing step due to some subset of the above reasons, such as Marching Cubes \[?\] to convert voxel data to polygons, and conversion of point cloud data to polygons \[?, ?\]. Both numerical iterations and polygonization require detailed discussions and are out of the scope of this book, except that the following references might be useful. PP: we need to find references. Direct shadow rendering algorithms for those other geometry types are briefly discussed in Section 8.3.

### 8.2 Shadow Volume Extensions

Shadow volumes can be used to achieve shadowing for other surface types. For example, Jansen and van der Zalm \[?\] use shadow volumes to calculate shadows for constructive solid geometry (CSG) objects. Shadow volumes are first calculated for the shadow generating parts of the silhouette (from the viewpoint of the light source) of each CSG object. A very large shadow CSG tree per object is formed. Then they consider only the front facing parts of the CSG object to simplify the complexity of the shadow CSG tree. The shadow trees tend to be quite large, and are needed on a per object and per light basis.

Heflin and Elber \[?\] extend the shadow volume approach to free-form surfaces. To handle free-form surfaces, the silhouette from the viewpoint of the light source needs to be identified first. Then the actual shadow volume is a set of trimmed surfaces of which intersection would occur against to increment and decrement the shadow count. PP: must check Fuchs Pixel-Plane papers if they did not handle shadows from quadric surfaces...
8.3 Voxels

Lichtenberger [?] provides a survey of voxel representation shadow algorithms. In our very brief coverage, we discuss some voxel representation approaches that relate to the techniques already discussed in this book. First, Discrete Ray Tracing [?] uses ray tracing of the voxels to determine shadow occlusion. Similar to voxel occlusion testing [?], the shadow occlusion is encoded as a bit in each voxel, whether a voxel is in shadow or not. So in a static environment, querying this bit would indicate whether the region inside this voxel is in shadow or not. Secondly, a variation of the shadow depth map approach can be used [?]. A pre-generated shadow depth map is rendered, and the depth to the closest voxel is stored in the shadow depth map. When rendering the voxel, it projects onto the shadow depth map, and if the distance of the voxel is farther away from the shadow depth map value, it is in shadow; otherwise it is not – see Figure 18.18. To avoid self-shadowing, the bias approach is chosen, as the surfaceID or mid-distance approaches are not easily extended for voxels. An interesting note, however, is that there is no obvious shadow volume extension that can be applied to a voxel shadow algorithm.

Additional reading on voxel shadow algorithms include papers based on splatting [?, ?, ?, ?], papers based on shear-warp [?, ?], papers based on hardware textures [?], and papers based on ray tracing [?, ?].

Figure 18.18: Voxel-based model of a head with its shadows.
8.4 Height Fields

There are many references on the web and in the industry where the term “voxels” is misused to indicate height fields \[?\]. Though voxels can be enumerated to represent height fields, voxels are 3D elements, and can handle concavities that height fields cannot handle. A height field is a 2D uniform grid, and each grid element is associated with a single height value to generate common geometries such as terrain or mountains. Shadowing for a height field is simple. It involves traversing the 2D uniform grid from the point to be shaded to the light source. If the visited elements indicate that the element height is higher than the traversal height, then that point is in shadow. Simple discussions and implementations can be seen in a few papers \[?, ?, ?\]. Furthermore, Stewart \[?\] and Heidrich et al. \[?\] apply horizon mapping techniques \[?\] (Section 4.2) to achieve hard shadowing for height fields.

*PP: shouldn’t we present horizon mapping somewhere? like here?*

8.5 Point Clouds

With respect to point clouds, we again do a very brief survey relating to familiar techniques already discussed in this book. Grossman and Dally \[?\] use a hierarchy of Z-buffers to detect tears. Since tears are located, a shadow depth map approach can be trivially used to produce (gap-less) shadows. In contrast, Schaufler and Jensen \[?\] use ray tracing to render and calculate shadows for point clouds. A ray used in this context is a closed cylinder with radius \(r\), where \(r\) is a value slightly larger than the radius of the largest spherical neighborhood that does not contain any point. Intersection hits occur when there are points in this cylinder. The intersection hit point and normal is a weighted average of the nearest points inside this cylinder. The same idea is extended for shadowing, using shadow rays. Automatic determination of the \(r\) value sounds like it would require an \(O(n^2)\) preprocess step, where \(n\) is the number of points in the cloud. Structures such as octrees and kd-trees can significantly reduce this cost.

Additional reading on point cloud shadow algorithms includes \[?, ?\].

9 Trends and Analysis

In reviewing the contents of Chapter 18, it is clear that there is no clear winner in terms of one particular algorithm for all situations. Ray tracing is, by far, the most flexible, but unfortunately inappropriate if good frame rates are desired. Shadow depth map and shadow volume algorithms tend to be much faster, if not interactive or real-time. However, the trend seems to indicate that shadow depth maps are much more used in high quality renderings (such as film and video, as shown in major software such as Renderman, Maya, Mental Ray, Lightwave, etc.) whose users are willing to experiment with user parameters, and shadow volumes are slightly more popular for real-time applications (such as games). *PP: Carmack seemed to indicate the scan-conversion of elongated shadow volumes kills the pipeline.*

It is not entirely clear why the above is the case, but the authors suspect that shadow depth maps allow higher quality renderings due to user-input tweaking (and the users’
willingness to tweak parameters) and the percentage-filtering capabilities to achieve fake soft shadows, which can also mask the straight edges of polygons that shadow volumes cannot. As well, shadow depth maps can also provide shadows from semi-transparent surfaces. Finally, shadow depth maps can easily integrate shadows from any surface type, with the flexibility of using different visibility determination algorithms for each surface type (e.g., Z-buffer for polygons, ray-tracing for algebraic surfaces, simplified splatting for voxels, etc.). Shadow volumes, on the other hand, are quite at home in real-time, GPU environments because polygons are the dominant primitive for real-time applications, and shadow volumes can deal with polygons very efficiently, with little to no user-input. It also does not require GPU textures for its approach that can be the bottleneck of a GPU based render. And the straight edges are not as discerning in real-time applications.

Note that when referring to shadow depth maps above, we are referring to the linear and fixed shadow depth maps. More recent and promising techniques such as the adaptive, non-uniform, or perspective shadow depth maps are still pretty much in the research phase. Finally, if only shadows casted on a floor in real-time is required, the fake shadows as suggested by Blinn [?] is clearly the winner and most often used implementation under such circumstances.
In addition to the visually pleasing aspects of soft shadows, studies by Rademacher et al. indicate that the presence of soft shadows contribute greatly to the realism of an image, which makes the generation of soft shadows that much more important. In this chapter, we mostly discuss soft shadows from extended light sources, such as linear lights, polygonal, spherical, and general area lights, and volumetric lights, skylight, etc., where soft shadows would be generated from direct illumination. The only exceptions are the sections on global illumination and motion blur algorithms.

The most straightforward approach to consider soft shadow generation from extended lights is to apply any existing hard shadow algorithm for a point light source, and simulate the extended light shadow with multiple point lights, where each point light represents a small region of the extended light. This is actually the basis for the approach by Brotman and Badler, for distribution ray tracing (see Section 3.1), and Herf and Heckbert’s approach (see Section 5), among others. However, just a straightforward implementation of multiple point source simulation tend to be very slow, often require a large amount of memory for the shadowing structures, and usually result in separation or “banded” results – i.e., there are noticeable shadow boundaries from the point sources – see Figure 19.1 of this separation/banding vs. smooth soft shadows. This produces a very low quality result, because our eyes are very sensitive to and unforgiving of banding artifacts. Banding from multiple point sources is reduced if:
1. Many such point sources are used – e.g., Herf and Heckbert use as many as 256-1024 point sources to simulate an extended light, which results in very slow rendering times.

2. Some non-uniform or stochastic sampling patterns are employed – as in distribution ray tracing techniques, see Section 3.1. Such patterns usually result in noise (to mask the banding artifacts), and our eye is a lot more accepting of noise than banding.

3. Some filtering, interpolation, or blending is done between the multiple samples.

![Figure 19.1](image.png)

**Figure 19.1:** Treating soft shadows as an averaging of hard shadows from multiple point sources can result in shadow bands (left), but ultimately converges to proper soft shadows (right).

Due to the above performance and banding quality issues, multiple point sources have not been the only approach explored in the literature. Many of the algorithms to be discussed in the following sections tend to start off with a good approximation to the soft shadows (without multiple point sources), and sometimes use a small number of point sources to improve the approximation even more. But the difficult question, in all cases (whether it be small number of points, or rely on many point sources), is how many point sources are sufficient, and this question if barely addressed at all in the literature. Thus soft shadow algorithms can come in many forms, from very accurate to very approximate, and evaluating soft shadow algorithms is more complex than hard shadow algorithms due to this extra accuracy variable.

The penumbra region, when compared to the umbra region, can actually be partitioned into an “inner penumbra” and an “outer penumbra”. See Figure 19.2. The inner penumbra is the soft shadow region inside the umbra, produced from a single representative point source. Algorithms computing just the inner penumbra usually result in under-sized shadows. The outer penumbra is the soft shadow region outside this umbra, and algorithms computing just the outer penumbra usually results in over-stated shadows. This distinction is useful in some approximations, because certain algorithms compute only the outer penumbra [? ] or inner penumbra [? ] for performance reasons – all other shadow algorithms discussed below assume both inner and outer penumbrae. Physically accurate results should produce both inner and outer penumbrae.
A starting point for generating soft inner or outer penumbra often relies on a single representative point on the light source. It is used as the central point for extensions from the shadow depth map algorithms, as the source of shadow volumes and wedges, etc. While the results appear often of very good visual quality, one must however pay attention to the fact that silhouettes thus identified only once, from this representative light point, will be static. Therefore a rounded occluder will not cast soft shadow on itself above these silhouettes. Another issue is that not all light source shapes are well represented by a single point.

Note that many of the discussions are also categorized (as in the hard shadow chapter) into shadow depth maps, shadow volumes, ray tracing, plus new categories of ray tracing of depth images, motion blur algorithms, and global illumination approaches. Most of the approaches are possibilities for interactive or real-time performance, with the exception of ray tracing algorithms. Finally, note that there is no clear set of obvious or complete solutions in terms of the soft shadow algorithms, as this area remains a topic of significant recent research.

1 Shadow Depth Maps

There are basically three types of shadow depth map algorithms to produce soft shadows, and they include single shadow depth map and silhouette based shadow depth maps. These have the light source being represented by one or very few point sources. The third type of shadow depth map approach is the multiple point source approach, which tends to be slower though more physically accurate.

1.1 Single Shadow Depth Map

There are a number of approximating and very fast shadow depth map algorithms [?, ?, ?] (based on a ray tracing approach suggested by Parker et al. [?]) that simulate the look of visually pleasing but inaccurate soft shadows from a point source. They fade the penumbra intensity as a function of the distance from the occluder, but otherwise assuming only a single point source. This makes sense because the farther away the occluder is from the shaded point, the more likely that the shaded point is partially lit, or lit by other non-local sources such as global illumination effects. These algorithms are fast because exactly one shadow depth map is computed and rendered against.

Ikedo [?] uses an intensity function that is based solely on the $z$ value difference between occluder and receiver, and the shadow depth map is used as the first occlusion hit distance. Should we say this is valid for light larger than occluder? Slightly more sophisticated schemes are available [?], in which for each point to be shaded, if it is in shadow, then the shortest distance to the closest shadow depth map pixel that indicates it is in light is identified. Inversely, for each point to be shaded, if it is in light, then the shortest distance to the closest shadow depth map pixel that indicates it is in shadow is
identified. The penumbra intensity is a function of this shortest distance and the $z$ value difference between occluder and receiver. Note, however, this shortest distance can be capped to some maximum value so that the search does not become the bottleneck of the algorithm. As well, note that if only the inner penumbra were computed, the computation of the shortest distance is only needed for points in shadow. Another advantage of computing the inner penumbra is that a shadow-width map can be pre-computed \cite{Heidrich2006} to store the “shortest distance” information per shadow depth map pixel.

### 1.2 Silhouette Detection based Solutions

Heidrich et al. \cite{Heidrich2006} simulate soft shadows for linear lights. Two shadow depth maps $S_i$ are constructed at the end points of the linear light – see Figure 19.3 (left). If the depth comparisons for both $S_i$ indicate shadow, then the final illumination is 0 (fully shadowed). Equivalently, if the depth comparisons for both $S_i$ indicate no shadow, then the final illumination is 1 (fully lit). If depth comparisons indicate both shadowing and lighting, then $L_i = \text{visibility} \times \text{local illumination}$, and the final illumination value for the linear light is $\sum L_i$, for $i$ sample points on the linear light. The visibility interpolation (see the visibility function in Figure 19.3 (left)) is achieved through an associated visibility map, by warping all the triangles from one view to another, and the shadow depth discontinuities are identified – this is known as the “skin” which produces the penumbra region – see Figure 19.3 (right), where the skin for the shadows from the boxes are illustrated in the two shadow depth maps. When the light source is large, accuracy of the above approach may be poor, and the number of shadow depth maps generated can be increased above two, so that the accuracy can be improved. Ying et al. \cite{Ying2009} extend that approach for (polygonal) area lights, in which a visibility map is computed per light boundary edge.

![Figure 19.3](image)

**Figure 19.3**: (left) visibility function of a linear light (left); (center) resulting shadows; (right) “skins” generated for each shadow depth map at the end of a linear light.

Wyman and Hansen \cite{Wyman1991} generate cones at silhouette vertices and intermediate sheets along silhouette edges similarly to Haines \cite{Haines1988}. These volumes are then scan-converted
in a penumbra map – see Figure 19.4. For each cone or sheet pixel, the corresponding illuminated 3D point in the shadow depth map is identified. The distances from the light to the occluding vertex (or point on the silhouette edge), the distance in the penumbra map, and the distance to the corresponding 3D point in the shadow depth map are used to produce a smooth attenuation approximating the soft shadow from a spherical light. This method is suitable to hardware implementation, and interactive to real-time results are reported. Arvo and Westerholm extend this approach to handle both the inner and outer penumbra regions, where Wyman and Hansen only deal with outer penumbra.

Similarly, Chan and Durand extend each silhouette edge of polyhedral models along the two normals of the edge. PP: are they bisecting normals from the polygons, or oriented along the shadow depth map plane? PP2: AW’s answer is yes, perpendicular, ?? I will check. This quadrilateral, called a “smoothie”, is textured with an alpha ramp, simulating the penumbra produced by a spherical light source. These smoothies are then rendered (alpha and depth) from the light source in an alternate buffer – see Figure 19.5. If two smoothies fall on the same shadow depth map pixel, only its minimal alpha value is kept. In the final rendering, if a 3D point projects with a larger depth value than in the shadow depth map, it is considered in full occlusion. Otherwise if it projects with a larger depth value than in the alternate buffer, its alpha value is used as an attenuation of the illumination. Otherwise, the 3D point is considered fully illuminated. Even though the shadows are only approximate and the algorithm only computes the outer penumbra, their technique still produces extended penumbrae of good visual quality at real-time to interactive rates on current graphics hardware. It also inherits many fundamental limitations of shadow depth maps, but extend its possibilities to produce anti-aliased soft shadows.

Arvo et al. also uses a single point source to simulate a spherical light, and applies a flood-fill algorithm to achieve soft shadows. The scene with umbra shadows are first rendered using the standard shadow mapping techniques. A shadow silhouette algorithm is used to determine the silhouette of the umbra region in the image. The penumbra rendering pass spreads out this umbra, where each pass spreads out by 2
pixels – the rendering passes terminate once the “spreading out” is completed with the silhouette hitting the edge of penumbra. With these potentially many penumbra rendering passes, this algorithm appears to be quite slow, especially for larger sized lights. In addition, overlapping objects may cause artifacts, as seen in Figure 19.6.

Figure 19.6: The figure with the green background.

1.3 Multi-point Source Shadow Depth Map Algorithms

In the previous two sections of the shadow depth map algorithms, those algorithms employ one or a couple of point (light) samples, which make those algorithms very attractive in terms of the real-time performance. In this section, multiple point (light) samples are relied upon to achieve more accurate results, but tend to be slower.

Sung [?] identifies the triangles that reside in the volume defined by the point to be shaded and the rectangle encompassing the extended light. Those triangles are inserted into a hardware Z-buffer to automatically do the scan-conversion and determine the amount of occlusion, with the point to be shaded as the view point, and the view direction being the center of the light. The scan-conversion is meant to simulate a set of faster distribution ray tracing computations. Note that this approach can be seen as an inverse shadow depth map approach, where the Z-buffer visibility is done from the perspective of the points to be shaded, not the light source origin.

Herf and Heckbert [?, ?] take advantage of fast hardware Z-buffer and texturing to stochastically sample an area light source at many points. For each such sample point (on the light), for each polygon that acts as an occluder, its shadow is projected onto all other planar receivers to form umbra regions stored as attenuation shadow maps. Essentially, each sample point pass produces hard shadows. In the end, all the results are averaged via an accumulation buffer to get penumbra regions. If enough sample points are used (e.g., 256), the results can look quite good and very accurate. However, for real-time purposes, if a small number of sample points is chosen, then banding (between the discrete sample points) would result. Isard et al. [?] distribute the above attenuation shadow maps, as hardware textures, over texture units on each graphics card, and claim very fast performance numbers.
Agrawala et al. [?] render shadow depth maps from different sample points on an area light source. These shadow depth maps are warped to the center of the light source and combined into one layered depth image, where each layer comprises of information for the same object. Each pixel in the layered depth image consists of depth information and a layer-based attenuation map (i.e., amount of occlusion for this layer), which stores the light attenuation at each depth. During display, each visible pixel is then projected to the layered depth image and the proper attenuation is computed by comparing the depth values with each layer. Note that there are similarities between the layered shadow depth map described above and multiple depth image ray tracing as described in Section 4, but layered shadow depth maps are more capable of interactive rates.

St-Amour et al. [?] also warp each sampled point light to the center of the extended light, as in the above approach. However, the information is stored within a deep shadow map [?], instead of a layered depth image. This is important because the layered depth image can contain shadow leaks, whereas the deep shadow map would not. This approach also allows the combination of semi-transparency, soft shadows, and motion blurred shadows all dealt with in a single algorithm. However the approach results in a large structure where the extended light and the occluders are assumed static. Soft shadows can be cast over moving objects, but the objects themselves cannot cast soft shadows computed from this structure.

Jeng and Xiang [?] can visualize soft shadows interactively with changes to parameters of an extended light source (including size, location, intensity, color, etc.). A large set of data structures are created to store the lighting and scene information, such as a slab of shadow depth maps (each shadow depth map represents a point on the extended light) and differential maps being created for the sample points in a light template. The shadow depth maps are sampled adaptively and stored in quadtrees to reduce an already large memory cost. As changes to the extended light occur, the light template controls which bits to turn on and off for the shadow evaluation. The same authors [?] propose a similar approach to Agrawala et al. [?], but in reverse order, projecting information from the light to the area light map to the screen buffer. Due to the sampling done in light space, dynamic splatting [?] is used to cover the holes in the screen buffer.

### 2. Shadow Volumes

Unlike shadow depth maps, there are really no “categories” of shadow volume based soft shadows – each of the approaches is quite different and do not really have much in common. Such approaches include the shadow volume BSP algorithm, penumbra wedges, cell counter shadow volumes, and plateau shadows. Note that the cell counter shadow volumes is the only multi-point source approach of all the shadow volume approaches, and thus the slowest in terms of performance. Of the shadow volume approaches described below, the penumbra wedge algorithms appear to be the most promising, in terms of performance, quality, and flexibility.

Additional reading on shadow volume approaches producing soft shadows that are not discussed in this book include [?, ?, ?].
2.1 Shadow Volume BSP (SVBSP) Algorithms

Chin and Feiner [?] extend their own work on SVBSP data structure [?] (see Section 2.5) to achieve soft shadows. Instead of one BSP shadow tree as in their original paper [?], two shadow trees are created: an umbra tree and a penumbra tree. The polygon to be shaded first goes through the penumbra tree. If it reaches an “out” cell in the penumbra tree, then the polygon is fully lit. If it reaches an “in” cell, then it may be in umbra or penumbra. This ambiguity is resolved by going through the umbra tree. If it reaches an “out” cell, then the polygon is in penumbra; otherwise it is in umbra. See Figure 19.7 for an example how to add polygons to the umbra and penumbra trees. Note, however, that this approach is not well suited for complex scenes, as both the penumbra and umbra trees can grow very quickly in size and complexity.

The intensity of the penumbra for the SVBSP tree is based on contour integration for diffuse surface elements. The light source may also need to be partitioned so that a unique BSP-tree traversal order will be generated, i.e., so that the split up area light will be entirely on one side of partitioned planes. Wong and Tsang [?] indicate that this can be very wasteful and identify cases where this extra partitioning of the area light can be avoided.


2.2 Penumbra Wedges

Akenine-Möller and Assarsson [?] modify the definition of a shadow volume to produce soft shadows. Instead of a quadrilateral forming one face of a shadow volume, a wedge is formed that represents the penumbra region. Each wedge has a shape similar to a prism, and all the wedges surround the umbra – see Figure 19.8. To determine visibility, instead of incrementing and decrementing a shadow count, a “lighting intensity” counter is used, incrementing and decrementing the lighting intensities between the entry point on the wedge and the closest to the eye of the exit point on the wedge and of the visible surface point (acquired from a Z-buffer render pass). A higher-precision stencil buffer is needed to deal with this lighting intensity counter. The intensity of the penumbra region has a linear decay between the wedge boundaries. Thus the penumbra intensities are approximate, and when penumbra regions cross, the results may not be correct, though the rendered images indicate pleasant-looking results.

Assarsson and Akenine-Möller [?] improve the original algorithm’s performance by separating the umbra and penumbra rasterizations. The umbra computations are virtually identical to the typical shadow volume approach, using the midpoint of the light as the umbra light source. The penumbra is only of interest if the visible surface point resides in a wedge (i.e., in penumbra). If within a wedge but not in shadow from the umbra pass, then the intensity value is increased; if within a wedge but already in shadow from the umbra pass, the intensity value is decreased to reduce the “hardness” of the umbra. Separating the two rasterizations allows less intersection computations than in the previous work [?], and since only visible surface points that reside in a wedge are of interest, the bounding box minimum and maximum z-depths can be used to cull out a lot of penumbra rasterizations. The same authors [?] use a very similar two-stage rasterization process to the previous paper, but provide a more robust computation of the actual wedges. Using penumbra wedges, the penumbra produced is fairly accurate unless the size of the light source is large in comparison to its shadow casters, to which the authors suggest subdividing the light source into smaller sized light sources. In addition, a 4D GPU texture lookup is used to compute the coverage of silhouette edges onto a light source using a V-buffer. Finally, the same authors [?] include optimizations such as tighter shadow wedges and an optimized GPU pixel shader for rectangular and spherical lights.

Figure 19.8: (left) shadow wedges from a spherical and a polygonal lights; (right) shadow intensity modification within a shadow wedge.
2.3 Plateau Shadows

Figure 19.9: In the plateau approach, a vertex generates a cone and an edge generates a sheet. Hardware texturing interpolates the soft shadowing.

Haines [?] uses the basic approach from Parker et al. [?] (see Section 3.3) to achieve soft shadows in a real-time environment, but assumes that the receiver is always a planar surface. The umbra region is always the same as if it came from a point source. As in Figure 19.9, a conical region and sheet-like region are created outside of the umbra silhouette vertices to define the penumbra region. The penumbra region is a function of the distance from the occluder and the distance from the umbra vertices. The algorithm can be implemented in the GPU, by using the Z-buffer to create the soft shadow hardware texture, which is then mapped onto the receiving surface. Both approaches assume the computation of only the outer penumbra, and can produce over-stated shadows. However, the results tend to look quite good – see Figure 19.10 for a comparison between the Haines [?], and Herf and Heckbert [?, ?] (see Section 1.3) approaches.

Figure 19.10: Visual comparison between the plateau algorithm and the multi-point shadow depth map. Courtesy of Haines.
2.4 Cell Counter Shadow Volumes

Brotman and Badler [?] stochastically choose points to model extended light sources. Their algorithm generates shadow polygons (in the form of shadow volumes) for each point source. A 2D depth buffer for visible surface determination is extended to store cell counters. The cell in which the point to be shaded resides is found, and the associated cell counter is incremented by 1 if the shadow polygons for that point source enclose the whole cell. If the corresponding cell count equals the total number of point sources, then the point to be shaded is in full shadow. If the cell count is less than the total number of point sources but higher than 0, then the point lies in penumbra region. AW: Steal picture from thesis... Diefenbach and Badler [?] and Udeshi [?] implement the above shadow volume extension [?] in the GPU using multi-pass OpenGL rendering.

3 Ray Tracing

Ray tracing remains the most flexible algorithm category to achieve soft shadows. Most of the work resides in distribution ray tracing, though some work has been done in analytic solutions. Distribution ray tracing approaches tend to be easier to implement, but to get high quality results with minimal noise and without resorting to too many point source samples tend to be challenging. Analytic solutions tend to be more difficult to implement, but they often provide smoother looking results. The main decision criteria to choose between the basic approaches is fundamentally how many point samples can match the analytic solution in terms of performance and quality – this is a difficult question to answer in a general context.

3.1 Distribution Ray Tracing

In terms of soft shadows within ray tracing, variations of the distribution ray tracing approach [?] have become the most used technique. An example is the work using distribution ray tracing for linear and area light sources [?] and curve light sources [?]. By distribution ray tracing, we mean shooting a number of rays towards the extended light to come up with an average for the shadow occlusion fraction or to estimate the irradiance. This does not imply that it always requires stochastic sampling as implemented in the original work of Cook et al.’s [?]. Deterministic sampling approaches (e.g., adaptive super-sampling [?], stratified super-sampling [?], Hammersley and Halton point sequences [?], N-rooks with a static multi-jittered table [?], blue noise tiling patterns [?], etc.) for distribution ray tracing can work just as effectively. This may be a factor to consider because stochastic sampling is patented. Other variations of the distribution ray tracing techniques are listed below.

Shirley et al. [?, ?, ?] compute good estimators of probabilistic locations on the light source for lighting and shadow computation, in a distribution ray tracing environment. Those ideal probabilistic locations are discussed for various types of extended light sources. See more detailed description of this work in Section 3.2.
Jensen and Christensen [?] apply a preprocessing step of computing a photon map [?], by sending photons from the light source. For each intersection the photon hits, “shadow” photons are continued along the same direction as if the original photon had not hit any surface. To compute the illumination at a particular point \( x \), the nearest photons around \( x \) are selected. If the nearest photons are all shadow photons, then \( x \) is considered in complete shadow. If the nearest photons are all regular photons, then \( x \) is in full illumination. If the nearest photons contain both regular and shadow photons, then \( x \) is either on the boundary of the shadow or it is in the penumbra of an extended light. One can take the number of regular and shadow photons to determine the shadowing fraction, but the authors [?] indicate that the result is not very good unless a very large number of photons are generated from the light. Instead, shadow rays are shot to the light to determine the shadowing in such cases. In essence, the photon map is applied as a “shadow feeler”. Using this technique, the authors claim that as many as 90% of shadow rays do not need to be cast. However, the preprocessing cost may be large for only directly illuminated scenes, and because the photons are shot based on probabilities, small objects can be missed.

Genetti et al. [?] use pyramidal tracing to quickly compute soft shadows from extended lights. A pyramid from the point to be shaded to the extended light is formed. If there are different objects in this pyramid, then the pyramid is subdivided into smaller pyramids, just as in typical adaptive super-sampling approaches [?]. The subdivision criteria remain the same geometric identification as used in their earlier work [?], which means that the same limitations still hold — i.e., the criteria of the presence of different polygons is used to determine how complex that cone region might be, and thus drives how many rays need to be shot. This may be ideal for parametric surfaces, implicit surfaces, or large polygons, but not ideal for polygonal meshes, as polygonal meshes will always require many shadow rays to be shot based on the above criteria.

Hart et al. [?] precompute an occluder list per image pixel per light source by tracing a very small number of shadow rays. When an occluder is found, a check is done to determine if adjacent image pixels also see the same occluder, so that the adjacent pixels can also register this occluder. During the shading of the pixel, the pixel’s occluders are projected and clipped against the extended light to analytically determine the visible portions of the light. This algorithm ensures consistency of occluders between adjacent pixels, but can result in missed occluders (i.e., light leaks), especially for small geometry occluders. The storage of occluders per pixel can also be large.

Additional reading on other distribution ray tracing variations includes [?, ?].

### 3.2 Structures for Exact Back-projection

Amanatides [?] extends the ray tracing concept of a ray to a cone. Instead of point sampling, cone tracing does area sampling. Achieving anti-aliasing requires shooting exactly one conic ray per pixel. Broadening the cone to the size of a circular light source for shadow cones permits generation of soft shadows — a partial intersection with a surface not covering the entire cone indicates penumbra; an intersection with a surface covering an entire cone indicates umbra. Due to significantly changing surfaces, it may be necessary to divide up a single cone into a set of smaller cones to get a better approximation for soft shadowing, which can be complicated. Note that it is ideal to
use cone tracing only for spherical lights, and not linear or area light sources, because it is difficult to approximate well a single cone with the linear or area light. Figure 19.11 illustrates how cone tracing generates soft shadows.

Figure 19.11: Approximating the shadowing of the pixel-cone intersection by a spherical light shadow cone.

Poulin and Amanatides [?] introduce two structures to soft shadows for a linear light. The first involves a light triangle, where a light triangle is defined by the end points of the linear light and the point to be shaded. The regular grid voxels that encompass this triangle are identified through 3D scan-conversion, and the objects in these voxels are intersected against the light triangle. All objects that intersect the light triangle are projected to the light to determine the occlusion. Though this algorithm provides an analytic solution to the shadow value, the authors indicate that the results are slow due to the expensive scan-conversion. And in fact, this analytic solution is about one of the few that generate the correct soft shadow integral evaluation. See Figure 19.12 of the light triangle and how the shaded regions are the final segments of the linear light that are fully lit.

Figure 19.12: The light triangle formed by the point to be shaded and the linear light is intersected against the objects of the scene. The intersected objects are back-projected onto the linear light to determine the light illuminating segments.

Poulin and Amanatides [?] also propose a more efficient linear light buffer represented by an infinite cylinder oriented along the light. The cylinder is subdivided into arcs along its radius. Objects residing in an arc are added to the arc’s list. Any point to be shaded identifies the arc it resides in, and projects the arc list’s objects on the
linear light. To make the regions smaller, each arc is divided into three sections: left, center, and right side of the linear light. They [?] found this algorithm to be much faster than the scan-conversion approach, though it does require additional preprocessing and more memory to store pointers to entire object set.

Tanaka and Takahashi [?] extend the idea of the linear light buffer. The candidate object lists are partitioned as in the previous scheme [?], but are also partitioned into layers along the radius of the infinite cylinder, as well as overlapping, partitioned bands which are $\phi$ and $\psi$ angles from the linear light axis – see Figure 19.13. The bands represent the regions for potential shadow rays. A loop through all the partitioned regions is executed, projecting all hit segments with the light triangle to get the final shadow occlusion value. The memory requirements for this scheme appear to be quite large.

Figure 19.13: Angles $\phi$ and $\psi$ from the linear light axis [?].

The same authors [?] extend their approach to deal with area lights. Partitioned bands are created in each direction of the area light, and a 2D array of bands is produced. The light pyramid (instead of light triangles for linear lights) is intersected against the objects in the appropriate band. Instead of employing a typical and expensive polygon clipper to identify the lit regions on the area light to compute the visible regions of the area light analytically, a “cross-scanline clipping algorithm” [?] is used instead. Both algorithms provide analytic solutions to the shadowing of extended lights. Because of rendering systems which super-sample to anti-alias, it is unclear if distribution ray tracing [?] with a few shadow rays per super-sample would converge to the analytic solution just as easily, without incurring the cost of the exact analytic solution.

3.3 Single Point Analytic Solution

Parker et al. [?] apply ray tracing to generate inaccurate but pleasant-looking soft shadows, by shooting only one shadow ray per spherical light. The basic idea is to identify the silhouette of objects and make semi-transparent a region about the silhouette. Thus the penumbra would come from these semi-transparent sections. Any penumbra region outside the umbra results in some interpolation from the umbra to achieve soft shadows – i.e., computation of the outer penumbra. Note that the umbra section tends to be over-stated, and a spherical light is assumed. This approach was as well the basic motivation for non-ray tracing approaches for soft shadow approximation [?, ?, ?, ?]. See Figure 19.14.

4 Ray Tracing Depth Images

Lischinski and Rappoport [?] achieve computationally faster soft shadows by ray tracing multiple depth images instead of the slower ray tracing of scene objects. A
Figure 19.14: A region around the object is marked as semi-transparent. Any ray traversing this region is considered partly occluded, depending on its distance to the real object.

shadow multiple depth image is generated that represents the view from the center of the extended light. By itself, multiple depth images can result in missed shadow hits. To alleviate this, each depth is superimposed onto fixed boundaries forming discrete depth buckets [?], resulting in larger, flat surfaces (e.g., panels). To improve performance, a 32-bit word can be used to describe 32 depth buckets at a pixel. Shadow rays traverse the depth buckets (to the extended light) and the bit-masks are OR-ed to determine shadow visibility for each shadow ray – see Figure 19.15. For shadow depths that may cause self-shadowing, the original $z$-depth values are kept around for comparison. Though this scheme is fast and reduces the risks of missed shadow hits, light leaking can still occur and the discrete depth buckets may show shifting artifacts over an animation. As well, the soft shadows are very approximate because the multiple depth image represents a single view from the center of the extended light, and shadow rays are shot to the extended light. As a result, the size of the extended light tends to be small for the results to be visually acceptable.

Agrawala et al. [?] deal with the light-leaking problem in a different way, using an approach that is similar to ray tracing height fields. Multiple depth images are generated, from points emanating from the extended light – thus this approach is more accurate than Keating and Max [?], which assumes only a single point from the extended light. To compute shadowing for each depth image, an epipolar ray traverses the depth image. As the traversal occurs, the epipolar depth interval $[Z_{\text{enter}}, Z_{\text{exit}}]$ is maintained. If the depth image depth $Z_{\text{ref}}$ is inside the epipolar depth interval, then it is in shadow. To speed up the time needed for traversal, a quadtree is used. Because a fixed set of depth images are preprocessed, if the number of such depth images is too small, banding artifacts in the soft shadows may occur.

PP2: I will rewrite this description, but only if there is a high probability to get
Figure 19.15: The scene is converted in a multiple depth image. The soft shadows result from tracing shadow rays through this structure and testing for intersection with the flat panels in occupied voxels.

Figure 19.16: Epipolar traversal of depth image is tough to understand without a figure.

St-Amour et al. [?] have the most complete incorporation of both motion blur shadows and extended light soft shadows, into a single deep shadow depth map, though that approach does not take into account moving lights, just moving occluders.

5 Other Soft Shadow Algorithms

Soler and Sillion [?] realize that the shape of the penumbra is a function of both the occluder and of the light source shapes. In an ideal situation where the light source, occluder and receiver are parallel to each other, this “function” would be a convolution of the occluder and the light source. This is achieved by rendering the occluder and light source are rendered onto shadow maps without Z-buffering being turned on. In other words, convolution, in its simplest form, means that the shadow of the occluder is blurred, with the distance from the occluded object controlling the amount of blur. The authors choose some heuristics to have representative and virtual light source, occluders, and receivers, projected onto parallel planes. However, choices of such virtual representations can result in error, and can easily miss self-shadowing cases. In particular cases such as radiosity meshes, these limitations may be less severe.

Stark et al. [?] compute the exact shadow irradiance using polygonal splines. As it currently stands, the derived mathematics is not usable in a complex environment.

PP: I don’t feel there is such a link between Stark and Ouellette... The exactness
of the solution here is almost a direct opposite approach taken by Ouellette and Fi-
ume [?, ?]. They determine the approximate locations of integrand discontinuities for
points in the penumbra. These points correspond to the approximate boundaries of the
visible portions of the linear light. The authors take advantage of the fact that even
with complex occlusions, there are usually only one or two such discontinuities. They
demonstrate that knowing the exact locations of these points is not necessary, provided
that the average location of these discontinuities varies continuously throughout the
penumbra. They introduce the Random Seed Bisection algorithm, a numerical iterative
technique for finding these discontinuities in a continuous fashion. The same authors
[?] follow up with research to locate integrand discontinuities for area lights. The
visible domains of the integrand is approximated by a polygonal boundary, then subdi-
vided into triangles so each of the triangulated integrands can be computed analytically
using a low degree numerical cubature (3\textsuperscript{rd} degree analogy to a numerical quadrature
technique).

Additional reading on other soft shadow algorithms that are not covered in this
book include [?].

6 Global Illumination

Global illumination algorithms take into account both the direct illumination from a
light source as well as the indirect illumination that reflects/refracts/transmits through-
out an environment. As a result, such algorithms tend to provide more realistic solu-
tions, but are much slower than direct illumination algorithms (which are the algo-
rithms mostly discussed in this book so far). Frequently, production software simulates
the look achieved by global illumination algorithms such as soft shadows by a judicious
placement of extra virtual lights in order to compensate for the lack of indirect illumi-
nation. If such scenes were to be used with a global illumination algorithm, they would
require manual adjustment to the lighting levels to avoid having an overlit scene. PP:
I am not sure to follow this logic... As well, some of the direct lighting features may
not translate well into global illumination algorithms, particularly radiosity algorithms
[?, ?]. In general, both direct and global illumination algorithms can generate realistic
looking images, but lighting a scene appropriately requires skills and, very often, a
trial-and-error approach.

Most, if not all, already take shadows into account and are inherent in the global il-
lumination approaches. Because we are considering global illumination here, the shadows
produced tend to be soft shadows. This makes sense in that the inter-reflections
of light come from surfaces, just like that of extended lights. In fact, it is sometimes
difficult to distinguish a direct illuminated rendering of soft shadows from extended
lights versus a scene rendered with global illumination. This can in fact provide some
heuristics to allow good placement of a few point or extended lights to generate the
global illumination look [?].

PP: I will work on this section, going over the various global illumination books,
and checking of how they treat shadows specifically.

There are basically three types of global illumination algorithms: radiosity [?, ?]
(and its ray tracing variations to determine visibility between mesh elements), Monte
Carlo ray tracing [?], and Photon mapping [?, ?, ?] (of which was partially covered in Section 1.1). Radiosity is well suited for purely diffuse environments in general. Monte Carlo ray tracing and photon mapping are well suited for diffuse and specular environments. Among those variations, some special techniques are worth listing here in terms of global illumination techniques that address shadows specifically.

PP: Arno (sp? Ref?) ray tracing assumes hemispherical direct illumination and obtain very soft shadows with only a few dozens of rays, among the best softer shadows.

In radiosity algorithms [?, ?] for diffuse environments, discontinuity meshing has become an important research area. By identifying mesh boundaries along the shadow discontinuities in the zero-th (zero-th derivative only exist for light sources that can produce a hard shadow discontinuity, such as a point light, or a linear source that is parallel to the edge of an occluding object), first and second derivatives and intersecting them with the scene, radiosity solutions can converge faster at a higher visual quality. This is because the mesh elements will not exhibit blockiness artifacts due to Gouraud shading (from hardware shading, for example), as the shadow discontinuities are set at the mesh boundaries. This also avoids “shadow leakage” where a shadow extends on mesh elements straddling illuminated and shadowed regions. Important work on this area comes from the following papers: [?, ?, ?, ?].

To compute even more accurate irradiance on top of discontinuity meshing techniques, the back-projection technique is used to build a “complete discontinuity mesh” [?, ?, ?, ?, ?, ?, ?]. The back-projection in a region contains the set of ordered lists of emitter vertices and edge pairs such that for every point to shade, the projection of those elements through this point onto the plane of the light source form the back-projection instance at this point. The partition of the scene into regions having the same back-projection is what forms the complete discontinuity mesh. Given the complete discontinuity mesh, the penumbra irradiance can be efficiently computed. The different papers (listed above) utilize back-projection and propose optimizations on top of the back-projection technique.

PP: This back-projection corresponds then to an exact visibility, referred to for linear light as an analytical solution, but for regions of space rather than for each point to shade...

There are alternative algorithms that produce high quality shadows within radiosity algorithms without resorting to some form of discontinuity meshing. Zatz [?] uses 40x40 shadow maps with sampled and interpolated visibility values as a shadow mask into a high order Galerkin radiosity solution. However, the determination of a good shadow map resolution and placement of the shadow maps are not discussed. Soler and Sillion [?] use texture convolution [?] (see description in Section 5) to produce a visibility texture applied to the mesh as a shadow mask. Duguet et al. [?] use a robust â method to compute shadow boundaries analytically without the need to generate a subdivided mesh, and small shadow features can be merged while preserving connectivity.

Sloan et al. [?, ?] use a lot of preprocessing, spherical harmonics (an excellent tutorial is available from Green [?]) and irradiance environment maps [?] to compactly express and compute radiance transfer in a real-time environment, capturing soft shadows (including self-shadowing), inter-reflections, and caustics. Because spherical harmonics are used as a linear approximation, the results are often blurred and are
only useful in a low-frequency lighting environment. The blurring means that sharp shadow boundaries are not captured. Ng et al. extend the work above by applying non-linear, spherical wavelets instead of spherical harmonics, so that high frequency effects such as sharp shadows can be captured. Note that though pre-computed radiance transfer are very exciting, limitations such as static scenes and assumptions about view changes (e.g., some specular components are not accurately accounted for) are required.

7 Motion Blur

Motion blur is now used extensively in rendering images for film. Without motion blur, the renderings will appear noticeably jittery, in the temporal domain over the animated frames (e.g., in the stop-motion filming techniques) and as a result, not as convincing as a special effect. With respect to its shadows, the blurred – thus soft – shadows come from the occluder's or light's motion captured in a single frame – see Figure 19.17.

![Figure 19.17](image)

**Figure 19.17**: Soft shadows resulting from motion blur when the light and/or the occluder move.

Other than distribution ray tracing, which shots rays that intersects against surfaces in stochastically chosen time domains, there has been little published research on achieving motion blurred shadows. It may sound like a simple thing, but it is not, though it is simpler in a ray tracing environment. The problem is difficult enough when just considering motion blurring of hard shadows (resulting in soft shadows) from a moving occluder but stationary receiver and light. The problem gets even more complex when motion blurred soft shadows (from extended lights) need to be considered, or motion blurred shadows from a moving light.

One can render motion blurred shadows by pre-generating several shadow depth maps, each shadow depth map indicating a different snapshot in time. Then integrate the shadowing based on sampling each of the shadow depth maps. However, the merging of such results would likely produce banding of the blurred shadows if not enough time snapshots are done, which is very visually distracting in an animation. Filtering the banding result may cause shifts over an animation. Stochastically choosing a
subset of snapshots (by shifting the light source) during shading would result in very bad blocky noise if not enough time snapshots are generated.

One could also do a time-jitter per shadow depth map pixel. However, because each shadow depth map pixel may project to multiple screen pixels, the motion blurred shadows would look very blocky. Lokovic and Veach [?] apply this trick to the transmittance functions of the beam of the shadow map pixel; this is then essentially jittering in 3D and thus can achieve decent motion blurred shadows. Because storage in terms of beams in a shadow map pixel is possible, it would also be interesting to consider an analytic motion blur coverage solution [?] in the shadow domain to be embedded in those beams.

As mentioned before, the complexity of the problem is greatly increased if such variables as moving occluders, moving lights, extended lights are considered. Sung et al. [?] formalize the motion blur equation without explicitly considering shadows:

\[
i(\omega, t) = \sum \int \int R(\omega, t) g(\omega, t) L(\omega, t) dtd\omega
\]  

(19.1)

where \(\omega\) is the solid angle intersected with a pixel at time \(t\), \(g(\omega, t)\) is the visibility of the geometry, \(R(\omega, t)\) is the reconstruction filter, and \(L(\omega, t)\) is the shading equation. Imagine the worst case scenario in which both \(g(\omega, t)\) and \(L(\omega, t)\) must be folded into the penumbra integrand in order to compute a motion blurred soft shadow; the above integral becomes very difficult to solve... Actually, that is not exactly correct either, because the \(g(\omega, t)\) term needs to be part of the soft shadow integral.

8 Trends and Analysis

To date, other than some form of ray tracing or global illumination approaches in high quality still scenes (or willingness to wait for a sequence of high quality images), soft shadows have remained mostly an academic research topic. This is because in high quality renderings for output such as film and video, faking the penumbra regions (such as percentage closer filtering) is usually good enough. As well, in real-time applications such as games or web applications, it is already a bit of a minor miracle that real-time hard shadows can be achieved with moderately complex scenes, features such as accurate soft shadows have not really been exercised.

In general, if the main criteria were performance and accuracy of the soft shadows is secondary, then the single shadow depth map cheating algorithms (see Section 1.1) are likely the best choice. If the performance and accuracy of soft shadows have equal weighting, it is better to resort to algorithms from penumbra wedges (see Section 2.2), and perhaps one of the silhouette based shadow depth map solutions or the multi-point shadow depth map solutions (see Sections 1.2 and 1.3). Finally, if the main criteria were accuracy of the soft shadows, then ray tracing based solutions would be most ideal (see Sections 3 and 4).
8. TRENDS AND ANALYSIS

<table>
<thead>
<tr>
<th>Paper</th>
<th>Perf</th>
<th>Quality</th>
<th>Light Type</th>
<th>Assumptions</th>
<th>Required Hardware</th>
<th>Manual Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>Herf-Heckbert</td>
<td>Int</td>
<td>*</td>
<td>polygonal</td>
<td>1 planar receiver</td>
<td># sources</td>
<td></td>
</tr>
<tr>
<td>Plateau</td>
<td>Int</td>
<td>**</td>
<td>sphere</td>
<td>1 planar receiver</td>
<td>size of plateau</td>
<td></td>
</tr>
<tr>
<td>Single sample cheat</td>
<td>RT</td>
<td>*</td>
<td>sphere</td>
<td>shadow map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Silhouette-based</td>
<td>Int</td>
<td>**</td>
<td>linear, polygonal</td>
<td>2D convolution</td>
<td># point sources</td>
<td></td>
</tr>
<tr>
<td>Convolution</td>
<td>Int</td>
<td>**</td>
<td>polygonal</td>
<td>no self-shadowing</td>
<td>2D convolution</td>
<td></td>
</tr>
<tr>
<td>Penumbra wedges</td>
<td>RT</td>
<td>***</td>
<td>sphere, rectangle</td>
<td>fragment shading</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SVBSP</td>
<td>Int</td>
<td>**</td>
<td>polygonal</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brotman-Badler</td>
<td>Int</td>
<td>*</td>
<td>polygonal</td>
<td></td>
<td># point sources</td>
<td></td>
</tr>
<tr>
<td>Layered depth images</td>
<td>Int</td>
<td>*</td>
<td>polygonal</td>
<td></td>
<td># point sources</td>
<td></td>
</tr>
</tbody>
</table>

Table 19.1: Shadow algorithms performances and requirements.
Part VI

Misc.
CHAPTER 20

Other

1 Global illumination
2 Art gallery
3 viewpoint planning
4 light from shadows
5 sound visibility
6 pursuit-evasion
Conclusions

Philosophical discussion on solved and unsolved issues!

1. approximate visi
2. temporal coherence
## CONTENTS

### Introduction

1. A historical perspective ........................................ 6
2. Audience .................................................................. 6
3. Overview of the book ............................................. 6
4. Acknowledgments .................................................... 6

### I Basics

#### 1 A Visibility Starter

1. Urban walkthrough ................................................ 10
2. Basic culling .......................................................... 10
   2.1 Backface culling ................................................... 10
   2.2 View-frustum culling ............................................ 10
3. Occlusion culling using large polygons ................... 11
   3.1 Basic method ...................................................... 12
   3.2 Some important notions ....................................... 13
4. Accelerating things a little bit ................................ 14
   4.1 Bounding boxes .................................................. 14
   4.2 Occluder selection .............................................. 15
   4.3 Other display acceleration techniques .................. 17
5. Shadow computation .............................................. 17
6. Summary .............................................................. 19
2 Important Concepts and Issues 21
  1 Occlusion culling ........................................ 21
    1.1 Image and object precision .......................... 21
    1.2 From-region visibility ................................ 23
    1.3 Architectural scenes and cell/portal structure ...... 24
    1.4 2D vs. 3D Visibility ................................ 24
    1.5 Exact, conservative, approximate and aggressive culling ... 25
    1.6 Occluder fusion ...................................... 27
    1.7 Occluder limitations and synthesis ..................... 28
    1.8 Spatial acceleration .................................. 28
    1.9 Dynamic scenes ..................................... 29
  2 Shadow computation ..................................... 29
    2.1 Hard and soft shadows ................................ 29
    2.2 Sampling vs. object precision ......................... 29
    2.3 Precision and aliasing ................................ 30
    2.4 Dependence on particular rendering algorithms ...... 30
    2.5 Can all objects be blockers? Receivers? .............. 30
    2.6 Accuracy ............................................ 31
  3 Discussion .............................................. 31

3 Analytical visibility 33
  1 Visual events ........................................... 34
    1.1 2D visual events .................................... 34
    1.2 3D visual events .................................... 35
    1.3 Triple-edge events .................................. 37
    1.4 Visual events of smooth objects ....................... 38
  2 The aspect graph ........................................ 39
    2.1 2D aspect graph ..................................... 39
    2.2 3D aspect graph ..................................... 41
    2.3 Implications and applications ......................... 42
    2.4 Complexity .......................................... 45
    2.5 Implementation ...................................... 45
  3 Line-space visibility ................................... 46
    3.1 2D dual space and the Visibility Complex ............ 47
    3.2 Lines parameterization in 3D .......................... 52
    3.3 Visual events, critical sets and extremal stabbing lines ... 55
    3.4 The 3D Visibility Complex and Skeleton ............ 56
  4 2D vs. 3D visibility ................................... 58
    4.1 A line is not a hyperplane in 3D ...................... 58
    4.2 Complexity of the visual events ...................... 59
    4.3 combinatorial explosion .............................. 59
  5 Summary .................................................. 60
## CONTENTS

### 4 Basic Tools

1 Visibility Queries
   1.1 Point-to-Point
   1.2 Point-to-Region
   1.3 Region-to-Region

2 Scene Data Structures
   2.1 Bounding Volumes
   2.2 Bounding Volume Hierarchies
   2.3 Space Partitioning
   2.4 Building a better hierarchy

3 Intersecting the Query and Scene Data Structures
   3.1 Ray intersections
   3.2 Shadow frustum tests
   3.3 Shaft culling

4 Acceleration Methods
   4.1 Ordering
   4.2 Coherence and Lazy Evaluation
   4.3 Silhouettes
   4.4 Shadow Volume BSP tree

5 Conclusion

### 5 Robustness

1 Issues
   1.1 Numerical precision
   1.2 Degeneracy

2 Robust numerical computations
   2.1 Perturbation
   2.2 Interval arithmetic

### 6 Visible surface determination

1 Hidden-Line Removal
   1.1 Robert
   1.2 Appel
   1.3 Curved objects

2 Image-precision methods
   2.1 Scan-line
   2.2 Adaptive subdivision
   2.3 The z-buffer
   2.4 Ray-casting

3 Object-precision hidden-surface removal
   3.1 Depth order and the painter’s algorithm
   3.2 Exact area-subdivision

4 Antialiasing
7 Graphics Hardware
1 Graphics Pipeline Overview ...................................... 103
2 The Graphics Pipeline ........................................... 105
  2.1 Command Stage .............................................. 105
  2.2 Geometry Processing ........................................ 106
  2.3 Rasterization .................................................. 106
  2.4 Texturing ....................................................... 108
  2.5 Fragment Operations .......................................... 109
  2.6 Advanced Topics and Examples ............................. 111
  2.7 Differences between DirectX and OpenGL ................. 113
3 Summary .................................................................. 113
4 Further Reading ..................................................... 113

II Point-based occlusion culling

8 View volume
  and back face culling ............................................. 117

9 Image-Space From-point Occlusion Culling ................. 119
  1 Luebke and George .............................................. 119
  2 HZbuffer .......................................................... 119
  3 Extensions (antialiasing, coverage masks) ................... 119
  4 HOM ................................................................ 119
  5 Wonka 99 .......................................................... 120
  6 Other implementations .......................................... 120
    6.1 Hardware solutions HP, SGI, Bartz ......................... 120
    6.2 Renderman ...................................................... 120

10 Object-Space From-point Occlusion Culling ............... 121
  1 Coorg and Teller ............................................... 121
  2 Hudson ........................................................... 121
  3 BSPs Bittner ....................................................... 121
  4 Hardly Visible sets .............................................. 121
  5 volumetric grid visibility ...................................... 121
  6 PLP ................................................................. 121

11 Occlusion culling in entertainment industry ............... 123
  1 Games ............................................................. 123
  2 Flight simulators ................................................ 123

III From-region Culling

12 Architectural interiors visibility ............................. 127
  1 Cell-to-cell visibility .......................................... 128
  2 from-point visibility .......................................... 128
## View-independent techniques

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Virtual Overlapping Boxes</td>
<td>181</td>
</tr>
<tr>
<td>4.2 Synthesizing Facade</td>
<td>185</td>
</tr>
<tr>
<td>4.3 Lev-Yehudy 2003</td>
<td>185</td>
</tr>
</tbody>
</table>

## Integration with other acceleration techniques

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Level of detail</td>
<td>187</td>
</tr>
<tr>
<td>2 Image-based acceleration</td>
<td>187</td>
</tr>
<tr>
<td>3 Real-time rendering</td>
<td>187</td>
</tr>
<tr>
<td>3.1 Performer</td>
<td>187</td>
</tr>
<tr>
<td>3.2 Funkhouser</td>
<td>187</td>
</tr>
<tr>
<td>4 Strategies of integration</td>
<td>187</td>
</tr>
<tr>
<td>5 Hardly visible sets</td>
<td>187</td>
</tr>
</tbody>
</table>

### V Shadows

#### 17 Preliminaries of Shadows

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Why Shadows</td>
<td>191</td>
</tr>
<tr>
<td>2 Hard Shadows vs. Soft Shadows</td>
<td>193</td>
</tr>
<tr>
<td>2.1 Hard Shadows</td>
<td>194</td>
</tr>
<tr>
<td>2.2 Soft Shadows</td>
<td>194</td>
</tr>
<tr>
<td>2.3 Colored Shadows</td>
<td>196</td>
</tr>
<tr>
<td>3 Classes of Shadow Algorithms: Basic Ideas and Properties</td>
<td>197</td>
</tr>
<tr>
<td>3.1 Shadow Depth Maps</td>
<td>197</td>
</tr>
<tr>
<td>3.2 Shadow Volumes</td>
<td>199</td>
</tr>
<tr>
<td>3.3 Area Subdivision and Preprocessing</td>
<td>201</td>
</tr>
<tr>
<td>3.4 Ray Tracing</td>
<td>202</td>
</tr>
<tr>
<td>3.5 Axis-aligned Receivers of Shadows</td>
<td>202</td>
</tr>
<tr>
<td>4 Self-shadowing</td>
<td>203</td>
</tr>
<tr>
<td>4.1 $N \cdot L$ Check</td>
<td>204</td>
</tr>
<tr>
<td>4.2 Bump Mapping</td>
<td>206</td>
</tr>
<tr>
<td>4.3 Anisotropic Reflection Models</td>
<td>206</td>
</tr>
<tr>
<td>4.4 Surface Acne Problem</td>
<td>207</td>
</tr>
<tr>
<td>5 Considerations for Choosing an Algorithm</td>
<td>207</td>
</tr>
</tbody>
</table>

#### 18 Hard Shadows

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Shadow Depth Maps</td>
<td>209</td>
</tr>
<tr>
<td>1.1 Dealing with Different Light Types</td>
<td>209</td>
</tr>
<tr>
<td>1.2 Avoiding Bad Self-shadowing Problem</td>
<td>210</td>
</tr>
<tr>
<td>1.3 Filtering Considerations</td>
<td>214</td>
</tr>
<tr>
<td>1.4 Performance and Quality Considerations</td>
<td>215</td>
</tr>
<tr>
<td>1.5 Miscellaneous Considerations</td>
<td>221</td>
</tr>
<tr>
<td>2 Shadow Volumes</td>
<td>222</td>
</tr>
<tr>
<td>2.1 Visibility Algorithms to Intersect the Shadow Polygons</td>
<td>222</td>
</tr>
<tr>
<td>2.2 The Initial Shadow Count</td>
<td>223</td>
</tr>
</tbody>
</table>
## CONTENTS

2.3 Reducing the Complexity of Dealing with Large Number of Shadow Volumes ........................................... 224
2.4 Shadow Volumes on the GPU ........................................... 225
2.5 Other Shadow Volume Variations ........................................... 228

3 Ray Tracing ........................................... 229
3.1 Shadow Culling Algorithms ........................................... 229
3.2 Dealing with Many Lights ........................................... 231
3.3 Anti-aliased Shadows ........................................... 232

4 Image-based Rendering ........................................... 233
5 Additional Reading on Other Hard Shadow Algorithms ........................................... 234

6 Atmospheric Shadows ........................................... 234
6.1 Using Shadow Volumes ........................................... 234
6.2 Using Shadow Depth Maps ........................................... 235
6.3 Integrating It into Radiosity ........................................... 235

7 Semi-transparent and Translucent Surfaces ........................................... 235
8 Other Geometry Types ........................................... 236
8.1 Higher Order Surfaces ........................................... 236
8.2 Shadow Volume Extensions ........................................... 237
8.3 Voxels ........................................... 238
8.4 Height Fields ........................................... 239
8.5 Point Clouds ........................................... 239

9 Trends and Analysis ........................................... 239

### 19 Soft Shadows

1 Shadow Depth Maps ........................................... 243
1.1 Single Shadow Depth Map ........................................... 243
1.2 Silhouette Detection based Solutions ........................................... 244
1.3 Multi-point Source Shadow Depth Map Algorithms ........................................... 246

2 Shadow Volumes ........................................... 247
2.1 Shadow Volume BSP (SVBSP) Algorithms ........................................... 248
2.2 Penumbra Wedges ........................................... 249
2.3 Plateau Shadows ........................................... 250
2.4 Cell Counter Shadow Volumes ........................................... 251

3 Ray Tracing ........................................... 251
3.1 Distribution Ray Tracing ........................................... 251
3.2 Structures for Exact Back-projection ........................................... 252
3.3 Single Point Analytic Solution ........................................... 254

4 Ray Tracing Depth Images ........................................... 254
5 Other Soft Shadow Algorithms ........................................... 256
6 Global Illumination ........................................... 257
7 Motion Blur ........................................... 259
8 Trends and Analysis ........................................... 260
VI Misc. 263

20 Other 265
1 Global illumination ................................................. 265
2 Art gallery ......................................................... 265
3 viewpoint planning .................................................. 265
4 light from shadows .................................................. 265
5 sound visibility ..................................................... 265
6 pursuit-evasion ...................................................... 265

21 Conclusions 267
1 approximate visi .................................................... 267
2 temporal coherence ................................................... 267

Contents 269

List of Figures 277
# LIST OF FIGURES

1.1 Backface culling. The faces facing away from the viewer are eliminated. The dot product between their normal and a vector reaching the viewpoint is negative. (a) 3D example: Face $B$ can be culled. (b) 2D example: Face $B$ can be culled.

1.2 Three types of visibility culling techniques: (i) view frustum culling, (ii) back-face culling and (iii) occlusion culling.

1.3 The tree $O$ is occluded by the façade $R$, since it lies inside the pyramid defined by $V$ and $R$.

1.4 Shadow frustum of $R$ with respect to $V$.

1.5 Testing if an object lies inside a frustum.

1.6 Pseudo code of the basic main display routine.

1.7 Axis-aligned bounding box.

1.8 The bounding box of $O_1$ is not completely inside the frustum of $R$, but most of the leaves of the tree are.

1.9 Pseudo code of the main display routine with occluder selection and bounding volumes.

1.10 Occluder selection. Occluder $C$ is not effective because it is occluded by $A$ and $B$. Occluder $D$ is small and does not cause a lot of occlusion. $E$ is distant but the configuration of the scene makes it effective.

1.11 Shadow.

1.12 Pseudo code of the main display routine performing shadow computation.

1.13 Shadow volume of a directional light source $\vec{L}$.

2.1 Image-precision occlusion culling. (a) Object space. (b) Z-buffer from $V$. The bounding box of occludee $O_1$ is tested against the current z-buffer.
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>From-region visibility.</td>
</tr>
<tr>
<td>2.3</td>
<td>Cell and portal structure of an architectural interior. The PVS of room A includes rooms {C, D, E, F}. The sequence of portals through which room F is visible is highlighted.</td>
</tr>
<tr>
<td>2.4</td>
<td>A non-conservative occlusion algorithm may miss-classify visible objects as hidden.</td>
</tr>
<tr>
<td>2.5</td>
<td>Approximate from-region visibility using sampling.</td>
</tr>
<tr>
<td>2.6</td>
<td>Aggressive occlusion culling. An approximate strategy might rightly determine (O_1) as hidden, but miss-classify (O_2) as hidden too.</td>
</tr>
<tr>
<td>2.7</td>
<td>The tree (O) is occluded by the conjunction of (R_1) and (R_2).</td>
</tr>
<tr>
<td>2.8</td>
<td>Occluder fusion. Neither (A) nor (b) completely occludes (C), but their combined effect occludes it.</td>
</tr>
<tr>
<td>2.9</td>
<td>(a) An extended light source (L) casts a soft shadow. (b) The view of the light source from point (P_2).</td>
</tr>
<tr>
<td>2.10</td>
<td>Soft shadow boundaries.</td>
</tr>
<tr>
<td>3.1</td>
<td>Visual event in flatland.</td>
</tr>
<tr>
<td>3.2</td>
<td>Visual event and occlusion.</td>
</tr>
<tr>
<td>3.3</td>
<td>Visual event. As the viewpoint goes down, the church disappears behind the house. This occurs when the apex of the church tower and the edge of the roof are superimposed in the view.</td>
</tr>
<tr>
<td>3.4</td>
<td>The visual event (ev) correspond to the limit of umbra.</td>
</tr>
<tr>
<td>3.5</td>
<td>Visual event. As the viewpoint goes down, the structure of the visible part of house (B) changes. The view from (V_0) is not represented.</td>
</tr>
<tr>
<td>3.6</td>
<td>Image structure graph of the four views from Figure 3.5.</td>
</tr>
<tr>
<td>3.7</td>
<td>EEE event. (a) The roof of the church (C) is visible. (b) When the viewpoint goes down, the roof disappears behind the roofs of (A) and (B). (c) The EEE visual event corresponds to a ruled quadrics.</td>
</tr>
<tr>
<td>3.8</td>
<td>EEE event in an axis-aligned scene. Box (C) will disappear when (e_1, e_2) and (e_3) meet.</td>
</tr>
<tr>
<td>3.9</td>
<td>Tangent crossing singularity.</td>
</tr>
<tr>
<td>3.10</td>
<td>Swallowtail visual event for a rotating torus. At viewpoint (b), the end junctions sand the t-vertices disappear.</td>
</tr>
<tr>
<td>3.11</td>
<td>(a) Viewing space partition of a 2D scene composed of two segments. (b) Corresponding aspect graph.</td>
</tr>
<tr>
<td>3.12</td>
<td>(a) viewing space partition of a 2D scene composed of two triangles. (b) Corresponding aspect graph. (c) Viewing space partition for three triangles.</td>
</tr>
<tr>
<td>3.13</td>
<td>Aspect stable under perspective projection but not under orthographics projection. (a) set of perspective viewpoint that see face 2 and 3. (b) Representative view. (c) An orthographic view shows 2 and 3 only when the view direction is parallel to face 1 and 6. (d) Corresponding visual visual event under orthographic projection.</td>
</tr>
<tr>
<td>3.14</td>
<td>Aspect graph of a convex cube.</td>
</tr>
<tr>
<td>3.15</td>
<td>Aspect graph of a L-shaped polyhedron under orthographic projection.</td>
</tr>
</tbody>
</table>
3.16 Viewing space partition and soft shadow. We show an EV visual event corresponding to an umba boundary. The aspect of the light source for a sample cell is represented in the inset. ........................................ 44
3.17 Partition of orthographic viewpoint space for a dimple object .......... 46
3.18 Raytracing and view in line space ........................................ 47
3.19 \((a, -b)\) duality. Lines are mapped to points, and points are mapped to lines. ................................................................. 48
3.20 \((a, -b)\) duality. A segment is mapped to a double wedge. .......... 48
3.21 The set of lines going through two segments is a quadrilateral. The four dual vertices correspond to the four lines joining pairs of segment endpoints in the primal. ............................................. 49
3.22 The set of lines going through two segments can be the union of two regions in the dual space (which can be seen as a quadrilateral that spans infinity). ..................................................... 49
3.23 Occlusion in the dual space. .................................................... 50
3.24 The dual arrangement classifies all lines depending on the objects they intersect. ................................................................. 50
3.25 Topology of maximal free segments ........................................ 51
3.26 A face of the visibility complex ............................................. 52
3.27 Line parameterizations ......................................................... 52
3.28 Plücker space and line orientation ........................................ 53
3.29 Lines stabbing a triangle in Plücker space ............................... 54
3.30 \(EEE\) in Plücker spaces ...................................................... 54
3.31 Visual event \(ev\) seen as a critical line set. ................................. 55
3.32 Four lines in general position are stabbed by two lines (adapted from [?]). ................................................................. 56
3.33 Viewing space partition and visibility skeleton. (a) Vertex \(v_1\) of the viewing space partition corresponds to node \(v_0\)' of the visibility skeleton, while \(v_2\) does not correspond to any node. (a) Viewing space partition. (b) Abstract embedding of the visibility skeleton. .......... 57
3.34 Images computed using erarchical radiosity and the Visibility Skeleton. (a) A scene with multiple sources. The skeleton construction took 2min 23s and the lighting simulation 8min. (b) A scene mainly lit by indirect light. The skeleton construction took 4min 12s and the lighting simulation 6min 58s. Note the shadows caused by indirect illumination, cast by the books on the back wall. ............................................. 58

4.1 Visibility between two points. .............................................. 62
4.2 Visibility from a point along a certain direction. ....................... 63
4.3 pyramid ................................................................. 64
4.4 Stolen from Fredo :) I will probably draw a different one. ............ 65
4.5 stolen from Haines :) will do a better one soon! ....................... 65
4.6 Various common bounding volume alternatives ....................... 67
4.7 Show in more detail the AABB and the OBB ......................... 68
4.8 Construction of BV hierarchy ............................................ 70
4.9 A regular grid is more suitable when the objects are fairly evenly distributed ........................................ 71
4.10 The ray traverses through 7 cells and intersect 5 objects until it hits the first object .................................................. 72
4.11 The ray traverses through 16 cells and intersect 2 objects until it hits the first object .................................................. 72
4.12 A partition by a quadtree and its corresponding graph. A quadtree is the 2D counterpart of the 3D octree. ................. 73
4.13 A partition by a quadtree and its corresponding graph. A quadtree is the 2D counterpart of the 3D octree. ................. 74
4.14 ?? ........................................................................ 74
4.15 With a quick comparison with Figure 4.15 we can see that a kdtree can produce a much more compact subdivision. .................. 76
4.16 Outline of the kdtree construction .................................. 76
4.17 The ray intersects the box, if the intervals $(t_{\text{min}}, t_{\text{max}})$ and $(t_{\text{ymin}}, t_{\text{ymax}})$ overlap. ................. 78
4.18 On the left, if the determinant of the quadratic equation, resulting from the substitution of the ray into the sphere equation is positive then we have two points of intersection $t_1$ and $t_2$, while at if it zero we have one point. On the right, a faster approach, since we are not interested in the actual $t$ values, is to check if $d < R$. .................. 79
4.19 The red points are the intersection points of the ray with the vertical grid line, and the blue points with the horizontal lines. Note that the distances along the ray between consecutive points of the same color is the same. That sequence define the entry and leaving points between adjacent cells. ........................................ 81
4.20 A parametric Ray Traversal Algorithm .......................... 82
4.21 The initial ray $R$ is tested against the tree by recursively inserting it, starting from the root node. This is done in a top-down, near-to-for manner. Here we show how the ray is partitioned during the recursion with the "near" parts shown in red, and the "far" in blue. Note that in the bottom right figure, the blue part will be discarded since an intersection with the scene is found by the red piece. .................. 83
4.22 Ray traversal of a binary partitioning tree ...................... 84
4.23 Frustum testing a BV hierarchy. ................................. 85
4.24 Frustum testing a BSP tree. ....................................... 85
4.25 Traversing the tree to get a front-to-back order ............... 87
4.26 Ordering the kd-tree cells with respect to a region ............. 88
4.27 Shadow volume BSP tree ........................................ 88

6.1 Z-buffer: darker pixels are closer to the viewpoint. (a) View of Figure ??, (b) View of Figure ?? ..................... 96
6.2 A buffer .......................................................... 97
6.3 Classic example of a cycle in depth order ......................... 98
6.4 A priori depth order ............................................... 99
LIST OF FIGURES

7.1 Graphics Pipeline (after Akeley and Hanrahan). Depicted are different views of the graphics pipeline. In (a), different functional units of the rendering pipeline are shown. In (b), we show the difference between operations performed in object versus image space. (c) highlights the fact that the major data type used during object-space processing are primitives composed of vertices, while for image-space, vertices are replaced by fragments. ...................................................... 104

7.2 Block diagram of modern graphics architecture (based on the model used by NVIDIA Cg. In current architectures, users can write their own functions which are executed for each vertex and fragment as they are generated (see programmable boxes). ......................... 105

7.3 Geometry processing (after Akeley and Hanrahan). Under a traditional rendering pipeline, it is here that geometry is transformed, clipped, and lighting computations are performed. ......................... 107

7.4 Rasterization (after Akeley and Hanrahan). Continuous primitives are broken up into discrete fragments, which are used throughout the rest of the rendering pipeline. ................................. 107

7.5 Texturing (after Akeley and Hanrahan). Data in the fragments are used for computing, often through simple interpolation and table lookup, a final color and opacity for the fragment. Recent advancements allow complex computations where multiple texturing operations are performed. ........................................ 108

7.6 Per-fragment operations (after Segal and Akeley). The way fragments contribute to the framebuffer is a complex set of steps. First, a fragment undergoes multiple culling steps, which can reject or accept a fragment based on a set of individual test. Then, logical and blending operations can possibly be applied to combine the fragments color contribution with existing framebuffer data. ................................. 109

12.1 Cells and portals: the adjacency graph and stab tree ..................... 128

12.2 Results from [?] showing the potentially visible set from a given cell. Courtesy of Seth Teller, UC, Berkeley. ......................... 129

13.1 The occlusion test in flatland is a 2D problem of all the rays connecting the occluder boundary and the occludee boundary. ................. 133

13.2 The occlusion test in 3D is a 4D problem of all the rays connecting the occluder boundary and the occludee boundary. .................... 133

13.3 The umbra and a penumbra. ............................................ 134

13.4 (a) The union of the individual umbrae does not occlude much. (b) The aggregate umbra. ................................................ 134

13.5 The aggregate umbra of the two occluders is larger than the union of the two individual umbrae. The difference region, which is part of the aggregate umbra, but not part of the individual umbrae, is marked in gray pattern. In (a) the two occluders intersect, and yield the gray area in-between. In (b) the two individual umbrae do not intersect, but still together they yield the gray area. .......................... 135
13.6 The umbra of an occluder with respect to a viewcell. Testing whether an object is fully occluded within the umbra is not easy. .......... 135
13.7 (a) A finite umbra is cast when the viewcell is larger than the occluder. (b) An infinite umbra is cast when the viewcell is smaller than the occluder. .................................................. 136
13.8 (a) The gray object has a strong occluder. Note that it fully resides within the strong occluder umbra. (b) The gray object does not have a strong occluder. However, it is occluded by the combination of three occluders. .................................................. 140
13.9 (a) There is no strong occluder. (b) There is a strong occluder between each part of the viewcell and occludee. .................. 141
13.10 In (a) a strong occluder exists since it blocks all the rays, while in (b) the occluder does not intersect all the rays. ............... 142
13.11 (a) The union of the individual umbrae is small and insignificant (the area that is in the aggregate umbra of these occluders is drawn in a diagonal pattern). (b) By placing an occluder in the umbra overlap of the left most occluders, a larger umbra is created (drawn in a diagonal pattern). .................. 143
13.12 (a) By placing another occluder in the umbra overlap of the right most occluders, a larger umbra is created (drawn in a diagonal pattern). (b) By placing another occluder in the umbra overlap of the larger umbrae the aggregated umbra is further extended (drawn in a diagonal pattern). 144
13.13 The void area (in white) is defined by the walls of passage within the solid area (in black) .................................. 144
13.14 The subdivision defines void cells (in white), solid cells (in black) and the cells (in gray) where the walls are defined are neither void nor solid, but are conservatively treated as void. .............. 145
13.15 The raster walk is marked with circles. The state of the two vertically adjacent cells can be an evidence that the cell is part of a vertical blocker of the visibility corridor between the two end-cells. .... 145
13.16 Although the corridor is blocked, each of the single blockers (c and d) is not a strong occluder. ................................. 146
13.17 Sampling is always prone to undersampling. ................. 147
13.18 View sampling is prone to under-sampling even the two view admit. ............................................... 149
13.19 (a) The boxes C and D are hardly visible behind A and B. (b) One boxes A and B are enlarged enough C and D become hidden. .... 150
14.1 Front and Back ghost polygons. .................................. 153
14.2 The black voxels are those which are interior to the object. Note that the umbra of the interior voxels is significantly smaller than of the original object. ................................. 154
14.3 The initial box-shaped occluder is generated by extending a seed voxel to its non-empty neighboring voxels. Here a 3-voxel wide initial box-shaped occluder is defined. .......................... 155
14.4 (a) Two black occluders have two umbræ (in gray voxel) that intersect. (b) A wider occluder consisting of four voxels is generated by extending a box-shaped occluder (in white) from one umbra to the next.

14.5 The back plane is the projection plane on which the extended projection of the set of occluders is recorded as a bitmap. The occlusion test is taking place over that bitmap. [I MUST CORRECT THE SUPPORTING LINES IN FIG B] .......................... 156

14.6 The projection plane must be placed where the umbræ intersect. Thus the fusion of the five occluders requires using three projection planes and reprojecting the occlusion of in a front-to-back order. .............. 158

14.7 The projection plane cannot be place behind the (right) occluder since it does not capture and fuse the combination of their umbræ. ............. 158

14.8 The blind areas. .................................. 159

14.9 The order is important. .................................. 160

14.10 The rectangular (patterned) occluder is shrunk by epsilon into the black rectangular occluder. The region that is in the umbra of the shrunk occluder with respect to \( p \), is guaranteed to be occluded from any point close to \( p \). ........................................................ 161

14.11 The aggregate umbra as the intersection of the three from-point umbra. 162

14.12 The three possible encoding of a node \( A \) and its immediate sons \( B \) and \( C \). (a) plain bits per node. (b) encoding in \( B \) and \( C \) just the bits that are non zero in \( A \), and (c) encoding in \( C \) just the bits which are not 1 in \( B \). 165

15.1 A scene where two objects (in black) are selected as good occluders and one (in a gray pattern) is a synthesized occluder that captures the occlusion of the cluster of three objects. The viewpoint is marked with a circle. ........................................................... 171

15.2 The black occluder in (a) is simplified to the black box in (b). The gray objects are those occluded by the black occluders. Note that the most of the occluded objects in (a) are also occluded by the simplified occluder. 172

15.3 The three black occluders in (a) are simplified to the three black boxes in (b). The gray objects are those occluded by the black occluders. Note that the occluded objects in (a) are also occluded by the fused occlusion of the three simplified occluders. .......................... 173

15.4 The solid angle of the surface is proportional to the size \( d\omega \) of its projection on the unit sphere. ........................................................... 174

15.5 None of the three black occluders is a good occluder with existence of the two others. Thus, a naive visibility sampling fails to detect that at least one of them is a good occluder. ................................. 176

15.6 Temporal ................................................ 178

15.7 The silhouette of a objects can be simple from one view and non-simple from another. ........................................................... 179

15.8 The silhouette of a objects can be convex from one view and non-convex from another. ........................................................... 179

15.9 The silhouette of a objects can be simple from one view and non-simple from another. ........................................................... 180
15.10A 3D model of St. Pauls cathedral (on the left) is converted to a set of voxels (in the middle), out of which, a set of overlapping axis-aligned large boxes are extracted. 182

15.11 The 2D polygon in light gray is converted into a discrete representation consisting of an array of cell. Only the cells which fully inside the polygon are drawn in dark gray. Starting from the two seed cells (marked with a black circle), two axis-aligned boxes (in white) are synthesized. 182

15.12 The greedy algorithm for extracting axis-aligned boxes. In (a) the inner voxels (cells) of the object are in dark gray. The selected seed voxel is marked with a black circle. In (b) the seed voxel is expanded down to create an stretched box. Then in (c) that box is stretched to the left to yield a large box bounded by the original object. That box is not the largest. In (d) a different voxel is picked as a seed, and extended first horizontally to the left. Then in (e) it is further extended vertically to yield a larger box than in (c). In (f) a third seed voxel can yield yet another large box. Note, that if in (a) the voxel would have first stretched to the left, it wouldn’t have yield a large box. 183

15.13 The original building in (a) has seems rather simple geometry as the details of the building are not displayed. By identifying three critical heights, three simple blocks are extracted (b-d). 184

17.1 Paul Pacotto sculpts both surfaces and their shadows. 192
17.2 An object formed of three axis-aligned ellipsoids is rotated by 20° around the vertical axis. The soft shadow cast on the plane below from an axis-aligned linear light source becomes discontinuous on the right image. 193
17.3 Light types casting hard shadows. PP: to add figure of spotlight 194
17.4 aFig10.eps about here. 195
17.5 Configuration of the variables for the integral of the irradiance. 195
17.6 aFig3a.eps about here. 196
17.7 Basics of the shadow depth map algorithm. PP: needs to replace $P$ by $L$. 198
17.8 GPU-based shadow depth map image. Courtesy of Everitt et al. 198
17.9 Basics of shadow volumes. 200
17.10 Basics of the ray tracing shadow algorithm. PP: need to change $P$ by $L$ in figure. 202
17.11 Projected shadow polygons on a planar floor. 203
17.12 Specular cutoff: The specular reflection component spreads where direct light should not reach because the $N \cdot L$ check was not performed. 204
17.13 Shadow terminator problem: (left) interpolated normal on the polygonized sphere; (right) resulting shadow stair-casing. 205
17.14 Problems with shadows from/on bump mapped surfaces. 206
17.15 Surface shadow acne due to numerical precision. 207

18.1 Self-shadowing of the planar floor on itself, appearing as darker strips of shadows. 211
18.2 Self-shadowing in the shadow depth map algorithm. The distance of point \( x \) to the point light \( L \) is larger than the depth sampled in the shadow depth map pixel, \( i.e., \| x - L \| > Z_n \). PP: change \( P \) by \( L \). 211
18.3 Two cases where the mid-distance approach \([?
]\) is incorrectly labeled. 213
18.4 (left) shadow leak; (right) light leak. 213
18.5 Percentage-closer filtering to reduce the shadow depth map aliasing. 214
18.6 In an animated sequence, the size of a pixel in a shadow depth map with the same resolution can change. 216
18.7 Perspective shadow depth map. 218
18.8 Trapezoidal shadow depth map. 219
18.9 Eliminating the need for enclosed shadow volumes. 224
18.10 Reducing the number of silhouette edge shadow polygons in the shadow volume algorithm. 225
18.11 Potential errors in the initialization of the eye shadow count \( E_{svc} \) when the near clipping plane intersects some shadow polygons. 226
18.12 Figure 1? 226
18.13 SVBSP traversal to determine if a point \( x \) is in shadow. 229
18.14 The light buffer of a point light source. Each pixel of the light buffer contains a sorted list of encountered objects up to an object producing full occlusion. 230
18.15 In voxel occlusion testing, each voxel of the regular grid is marked as full, null, or unknown occlusion from one light source. PP: need to remove figure on right. 231
18.16 Shadowing in presence of light scattering media. (left) schematized ray segments; (right) resulting light beams. 235
18.17 Shadows from a tessellated car. PP: the left and right images provide any additional information, or just two views? 237
18.18 Voxel-based model of a head with its shadows. 238
19.1 Treating soft shadows as an averaging of hard shadows from multiple point sources can result in shadow bands (left), but ultimately converges to proper soft shadows (right). 242
19.2 (left) Umbra; (center) outer penumbra; (right) inner penumbra. 243
19.3 (left) visibility function of a linear light (left); (center) resulting shadows; (right) “skins” generated for each shadow depth map at the end of a linear light. 244
19.4 (left) shadow depth map from the representative point source; (center) penumbra map; (right) penumbra cones and sheets generated from an occluder edge. 245
19.5 (a) shadow depth map from the representative point source; (b) smoothie depth; (c) smoothie alpha; (d) outer penumbra soft shadows. 246
19.6 The figure with the green background. 246
19.7 Adding polygons to the umbra and penumbra trees. 248
19.8 (left) shadow wedges from a spherical and a polygonal lights; (right) shadow intensity modification within a shadow wedge. 249
19.9 In the plateau approach, a vertex generates a cone and an edge generates a sheet. Hardware texturing interpolates the soft shadowing.

19.10 Visual comparison between the plateau algorithm and the multi-point shadow depth map. Courtesy of Haines.

19.11 Approximating the shadowing of the pixel-cone intersection by a spherical light shadow cone.

19.12 The light triangle formed by the point to be shaded and the linear light is intersected against the objects of the scene. The intersected objects are back-projected onto the linear light to determine the light illuminating segments.

19.13 Angles $\phi$ and $\psi$ from the linear light axis [?].

19.14 A region around the object is marked as semi-transparent. Any ray traversing this region is considered partly occluded, depending on its distance to the real object.

19.15 The scene is converted in a multiple depth image. The soft shadows result from tracing shadow rays through this structure and testing for intersection with the flat panels in occupied voxels.

19.16 Epipolar traversal of depth image is tough to understand without a figure.

19.17 Soft shadows resulting from motion blur when the light and/or the occluder move.
REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


[NGR] NGRAIN. Software from ngrain technology at ngrain corporation.


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES


REFERENCES

[WREE67] C. Wylie, G.W. Romney, D.C. Evans, and A.C. Erdahl. Halftone per-


[WS99b] Peter Wonka and Dieter Schmalsteig. Occluder shadows for fast walk-
60, September 1999.

algorithm for real-time application. In *Workshop on Object-Oriented

[WSP04] M. Wimmer, D. Scherzer, and W. Purgathofer. Light space perspective

102, October 1998.

[WW92] Alan Watt and Mark Watt. *Advanced Animation and Rendering Tech-

[WWS00] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility pre-
processing with occluder fusion for urban walkthroughs. *Rendering
Techniques 2000: 11th Eurographics Workshop on Rendering*, pages
71–82, June 2000.

[WWS01] Peter Wonka, Michael Wimmer, and François X. Sillion. Instant visi-

[Yan85] Johnson K. Yan. Advances in computer-generated imagery for flight
August 1985.


[YKCS95] K. Yoo, D. Kim, K. Chwa, and S. Shin. Efficient algorithms for com-
puting shadow volumes from an area light source. In *Proceedings of

[YKSC98] K. Yoo, D. Kim, S. Shin, and K. Chwa. Linear-time algorithms for find-
ing the shadow volumes from a convex area light source. *Algorithmica


