Master of Science Thesis
Department of Computer Science
Lund Institute of Technology

Optimizations for Shadows in Interactive Environments

Jon Hasselgren [d99jh@efd.lth.se]

December 2003

Supervisor: Lennart Ohlsson, Lund Institute of Technology

Abstract

The purpose of this master thesis is to research possible optimizations for shadows in interactive graphics. In interactive graphics performance is always an important factor since it limits the complexity of the images that can be drawn in a pace high enough to generate a sequence of images that seem animated to the user. There already exist several optimizations for shadows but in this thesis an alternate approach is taken. The optimizations are based on a pre-computational phase that generates important data that can be saved to the hard drive and used in the interactive part of the application to achieve higher run-time performance.

The thesis report presents an algorithm for high performance light and shadow rendering under the assumption that many of the light and objects in the world are static during the whole execution time. The algorithm has been evaluated on mainstream graphics hardware and a demo application has been constructed that implements both the algorithm described in this paper as well as an optimized version of a common shadowing algorithm.

Contents

1 Introduction

Graphics drawn on a screen is one of the most important means of communication between computer software and the user. Traditional computer graphics is based on rendering (drawing) two-dimensional images on the screen. Images are represented as a grid of pixels where every pixel has a color assigned to it.

In three-dimensional graphics the goal is to compute a two-dimensional image on the screen based on a representation of the three-dimensional world and a view position. The representation may vary from implementation to implementation but the most common representation is geometrical and based on building objects from planar polygons. Common linear algebra and vector geometry can be used to perform varying computations for the objects.

Computer graphics, in particular three-dimensional graphics, is divided into the two main fields: Cinematic graphics and Interactive graphics. Cinematic graphics is generated one single time and then stored as a two-dimensional image or movie which makes the actual rendering time a less important factor. In Cinematic graphics the greatest focus is on realism, very accurate simulations of physics is used which gives images that are (almost) photo-realistic. The field of Interactive graphics is a lot more dynamic. The user should be able to interact with the virtual world and the delay from a user action to an update on the rendered image should be small enough to feel instant to the user. In reality, this means that the computer should be able to render at least around 20 frames (images) per second. Thanks to the development of hardware designed solely for the purpose of rendering three-dimensional graphics, the image quality gap between Interactive graphics and Cinematic graphics is constantly decreasing. The programmability has increased as well as the performance, which allows for much more complex and application specific effects to be performed.

An important part of the three-dimensional graphics is lighting, this is the part where most of the extra computational power used in Cinematic graphics is spent. Programs designed for Cinematic graphics often use accurate simulations of light, where light reflected by the other objects in a scene play an important role. In Interactive graphics this light is typically ignored and up until recently the same has been true for shadows caused by objects obstructing the light from reaching another object. A common strategy for Interactive graphics, called light-mapping, has been to divide the world into static and dynamic objects and to perform advanced lighting on the static objects which is stored as an image of the incoming light for every polygon. This gives high quality lighting for the static objects, but since dynamic lights and objects must be computed in real-time the quality of this lighting will be much lower.

Another important aspect of three-dimensional graphics is shadowing, which of course is closely related to lighting. Although techniques used for computing shadows in Interactive graphics have existed for more than a decade they have been rarely used. This is almost solely because of the very large pressure put on the graphics hardware compared to the other, less complex, light models. The techniques adds a fair amount of extra polygons to a scene, but the most important limitation is the fill-rate (pixel processing power) required by the hardware which is still often the limiting factor. In this thesis project, a technique intended for optimization of the fill-rate used for rendering scenes with shadows, has been developed. Just like light mapping, this technique is based on differing between static and dynamic objects. The difference however is that the technique used for static and dynamic

objects are identical except for the optimizations. It's also possible to perform correct shadow interactions between dynamic and static objects which has always been a problem light-mapping suffered from.

The presented algorithm contains several new contributions as well as further usage for existing methods. Many of the new contributions are related to allowing dynamic objects to receive shadowing and lighting of the same quality as the static geometry. Also, a method for pre-computing relevant data is presented that although it is related to work done by ATI was developed independently. The contributions include

- Pre-computational algorithm to generate data that is crucial to optimizations
- Using dynamic object bounding volumes to further improve common shadow volume optimization techniques and how these properties will affect different parts of the engine pipeline and performance.
- Special shadow volume construction from the optimized pre-computed meshes.
- Robust implementation of the faster "depth pass" shadow volume algorithm in special cases.

The presented approach has been compared with an optimized implementation of the common shadow volume algorithm. The evaluation was performed on mainstream (low cost) hardware and showed an encouraging result of 50% better performance in high resolutions with the presented algorithm.

2 Background

To understand the mechanics of the optimizations presented it's important to have a basic understanding of the basic lighting and shadowing algorithms used in modern interactive graphic engines. This section describes the common algorithms used to perform the lighting and shadowing along with the strengths and weaknesses of every such algorithm.

2.1 The Phong reflection model

The Phong reflection model approximates light by dividing it into two separable parts illustrated in *Figure 1*. The first part is the diffuse light which is light reflected equally in all directions and therefore is independent on the viewer's position. The second part is the specular light which is directly reflected of the surface just as if it where a mirror, the specular lighting for a point therefore gets dependant on the viewer's position. These light types are mixed together to produce the output result.

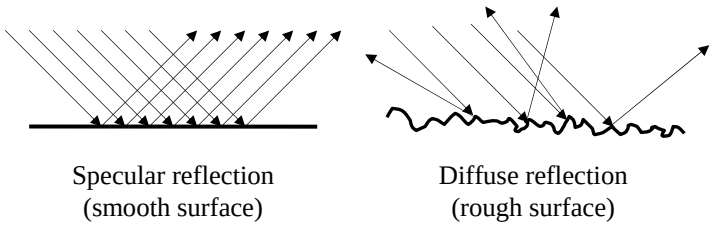

Figure 1 *Conceptual differences between specular and diffuse reflections*

For a given light source the pong reflection model can be written as

$$I(p_c, p_s, p_l) = \left(k_{dps}\left(L \cdot N_{ps}\right) + k_{sps}\left(R \cdot C\right)^{\alpha_{ps}}\right) \cdot light\left(p_l, p_s\right)$$

Equation 1 *The Phong reflection model*

Where the output of $I(p_c,p_s,p_l)$ is a 3 component (r,g,b) color value describing the intensity of the light reflected from the point p_s (which is a point somewhere on a surface in the scene) towards the viewer. The other inputs, p_c and p_l are the positions of the viewer and the light source for which to compute the illumination. The vectors L, R and C may be computed directly from the input parameters, they are: L – normalized vector from p_s to p_l, R – normalized reflection of L in the plane formed by N_{ps} and p_s and C – normalized vector from p_s to p_c. The N_{ps}, k_{dps}, k_{sps} and α_{ps} values defines the material properties of the surface for the point p_s, texture maps are commonly used to describe those properties for each point on a given surface.

The function $light(p_l,p_s)$ computes a scalar value for the properties of a certain light, here functions such as light distance attenuation is included. In the original Phong reflection model this attenuation is computed using *Equation 2*

$$light(p_l, p_s) = \frac{1}{a|p_s - p_l|^2 + b|p_s - p_l| + c}$$

Equation 2 *Attenuation equation*

2.2 Light maps

Light mapping is based on assigning a unique (low resolution) texture map to every polygon in the world. This texture map, called a light map, is used to store pre-computed and pre-mixed lighting for every sample point used in the world. Thanks to light maps the whole lighting computation is moved out of the real time part of a program. This allows us to use complex algorithms such as radiosity since time is less of an issue, also computing shadows can be done by simple ray tracing between the two sample-points currently used to compute light distribution.

The real magic with light maps are seen on rendering time, to render a scene with full lighting one must simply go through the polygons that needs to be rendered and use texture combining to multiply the base diffuse texture of a polygon with the light map for that polygon.

A problem with the light maps is that they are too static. The fact that intensity of a sample point in the light map is a mix of all lights affecting that point makes it problematic to introduce physically correct shadows for dynamic objects. The most common solution is by removing light from the area shadowed by a dynamic object but this will also subtract light from areas that are already shadowed by the static environment itself. Since the shadows in that case already exist in the light maps a shadow within a shadow will occur, the same goes for if two dynamic objects are shadowing the same position.

Last but not least light maps consume a great amount of video memory. Especially if trying to achieve the same picture quality with light maps as with per pixel lighting, in which case the lighting must be sampled with very high resolution.

2.3 Per pixel lighting

Lighting computations have traditionally been made on a per-vertex basis using simple interpolation schemes to compute the lighting values for each pixel. However, new hardware is flexible and powerful enough to compute the lighting of every pixel individually. This does not only give better image quality but also allows using texture maps to describe the different parameters in the Phong equation giving artists possibilities to create much more realistic models without adding more polygons and different materials to it.

Per pixel lighting is often implemented using pixel and vertex shaders. A shader capable card can execute programs defined by the coder or artist for every pixel rendered on screen and every vertex of the geometry processed. The first generation of graphics hardware called programmable could only execute rather simple and restricted programs but the newer cards constantly remove more and more of the restrictions on what programs can be written.

The attenuation factor of the Phong model (*Equation 2*) can also be implemented in a somewhat different fashion because of two problems related to it. The first of these problems is the division, only the newest cards such as the ATI R300 and NV30 series

currently supports division on a per pixel basis. Also, the attenuation factor in *Equation 2* never reach zero, this introduces some problems related to efficient culling for light sources although they are quite easily solved using a threshold value. In my implementation I have chosen to implement the attenuation function based on the mathematical expression in *Equation 3* (Dietrich 2001) which may not be physically correct but produce a visually trustworthy result.

$$light(p_l, p_s) = \max\left(1 - k \cdot |p_s - p_l|^2, 0\right) = \max\left(1 - \left|\sqrt{k}(p_s - p_l)\right|^2, 0\right)$$

Equation 3 *Per pixel distance attenuation*

As can be seen in *Equation 3,* the internal expression $\sqrt{k}(p_s - p_l)$ can be linearly interpolated since k is constant for every light. This means that it can be performed in a vertex program rather than on a per pixel basis. This leaves only a subtraction and a dot product which can even, on some hardware, be collapsed into a single instruction. Alternatively, texture lookup tricks with two or three dimensional textures can be used to compute this function.

A frequent function needed in a Phong pixel shader is the normalization of vectors on a per pixel basis. Normalization is rather slow to compute on new hardware and impossible on old hardware unless using polynomial approximations of the reciprocal square root. The best alternative is to use a cube map (Kilgard 2000). Cube maps are widely supported and they only use the direction of a vector to address a pixel. A pre-computed cube map with a normalized vector representation for every fragment in the map efficiently normalizes the interpolated vector used for addressing a fragment in the texture.

Looking back at the Phong equation (*Equation 1*) there's a lot of new material components that may be moved to a texture map evaluated on a per pixel basis instead of the previous per vertex basis. Components that can be moved to a texture map are N_{ps}, k_{dps}, k_{sps} and $\alpha_{ps.}$

Of special interest is the N_{ps} texture. As can be seen in the Phong equation this texture represents the normal of a surface in a given point. It allows us to simulate roughness in an otherwise plane surface as far as lighting goes, this technique is called bump mapping (Blinn 1978, Kilgard 2000).

The values for pixels in the normal map texture are interpreted as vectors rather than colors, thus a question arise of in which coordinate system these normals should be represented. There are mainly two coordinate systems that are efficient to use, object space and tangent space. The object space normal representation usually gives better performance, but it suffers from memory problems on very large meshes such as the world mesh due to the fact that every surface needs to have a unique normal map. The tangent space coordinate system is built from the coordinate space of the texture map, thus the vectors building this coordinate system for a specific face are the vectors for the main texture axes and the normal for that face. One usually assumes an orthogonal coordinate system for the texture map axes, which makes the transformation to tangent space a simple matter of computing the dot products of a vector in object space and the vectors building the tangent space for a single face. The usage of tangent space allows you to assign a certain normal map to a material rather than to a surface as was the case in the object space based representation.

Many implementations use the Blinn shading model in *Equation 4* to perform the per pixel lighting. The purpose is to save the rather expensive per-pixel reflection computations needed for the Phong model. This model uses the half angle vector instead of the reflection vector furthermore the half angle vector is often approximated on a per pixel basis with

linear interpolation. Unfortunately this approximation doesn't work particularly well with the geometry generated by the shadow algorithm, described in this paper, due to non-uniform and rather low tessellation. In this case the Phong illumination model may be preferable since the reflection based on two normalized vectors is also a normalized vector while the normalization of a half angle vector that is computed on a per-pixel may be troublesome on some hardware.

$$I(p_c, p_s, p_l) = \left(k_{dps}\left(L \cdot N_{ps}\right) + k_{sps}\left(H \cdot N_{ps}\right)^{\alpha_{ps}}\right) \cdot light(p_l, p_s)\ ,\ H = \frac{C+L}{|C+L|}$$

Equation 4 *Blinn illumination model (Assumes the C and L vectors are normalized)*

The α_{ps} component raises some problems on older hardware. The new cards have support for power instructions which makes the specular computation easy alternatively a dependant texture lookup can be used to emulate the power function. A problem is the GeForce3/4 cards which are too limited to allow for the dependant texture lookup and a normalization of the half angle vector at the same time. This can be worked around using a lower degree polynomial to approximate the power function (Beaudoin 2002). The polynomial coefficients for a given power can be looked up from a texture map.

All in all, implementation of these shaders is simple straightforward coding for new hardware with good pixel shader capabilities. Implementation on old hardware or using old capabilities for performance reasons make the implementation a more challenging task because of both instruction limitations and the amount of textures simultaneously accessible.

Up to this point we have forgotten all about shadowing for the per pixel lighting. Shadows do their fair share to the realism of a scene, especially for dense indoor scenes. Shadowing is a quite simple problem with time consuming solutions. The goal of every shadow algorithm is to determine if a point is visible from the light source. If it's not visible, it's in shadow and no light should be rendered there. There are two commonly used and well supported methods for computing in real time if a pixel is in shadow or not given a specific light. They are described in the following sections

2.3.1 Shadow maps

The Shadow maps algorithm (Williams 1978) is based on using a common depth buffer and depth comparisons to compute shadows. In a first step all polygons visible to the light source is rendered into an off-screen depth buffer often called the shadow map. This is done in a light-space coordinate system, and furthermore the polygons must be projected in some way to be able to render them.

When the shadow map has been created we have enough information to determine if a pixel is in shadow or not. When rendering lighting to the scene the coordinates for every pixel is computed not only in image space, but also in light space. The light space coordinate representation is used to compare the depth component with the value in the shadow map. If a pixel has a higher depth value than the one in the shadow map it's rejected. The shadow map algorithm can be described by the following pseudo-code

```
Pass 1.  For every polygon visible to the light source
             Transform to light-space coordinate system
             Render to off-screen depth buffer

Pass 2.  For every polygon visible to the viewer
             Transform to light-space coordinate system
             Transform to view-space coordinate system
             For every pixel in polygon
         if (light-space depth <= depth in shadow map)
             render shaded pixel and add to the current color for this pixel
         else
             discard pixel, in practice add black to the current color
```

The shadow map algorithm is very simple and elegant. It is also rather well supported on hardware and can be emulated through pixel shaders or the alpha testing unit possibly at some precision cost, if no support exists.

Unfortunately there are also many problems related to the shadow map technique. Since the scene must be projected and most hardware only supports pixel accurate projection of the scene onto a plane, omni-directional lights are both problematic and slow to implement. Other problems with this algorithm are often related to precision issues, since the shadow map is represented in light space there's no guarantee that one pixel of the shadow map maps to a single pixel when rendering the scene. There are a workarounds for this (Fernando 2001, Stamminger 2002, Sen 2003), but shadow maps are seemingly better suited for cinematic graphics due to the possibility of choosing appropriate shadow map resolutions based on the light, geometry and the camera path. This is not as easy for interactive environments due to the unpredictability of these parameters.

2.3.2 Shadow Volumes

"Shadow volumes" is also an old algorithm. It was originally designed by Frank Crow in the late seventies (Crow 1977, Bergeron 1985, 1986).

The first step in this algorithm is to compute the "shadow volume", which is a polygon mesh that encloses the subspace that lies in shadow for a specific light. The shadow volume mesh can be constructed given the original geometry of the scene and the position of the light source. This is done by finding the silhouette edges of the world mesh. A silhouette edge is defined as an edge between two faces where one face faces the point (light source) and the other face faces away from the point. When the silhouette edges has been found the shadow volume mesh can be constructed by inserting a quad for every single silhouette edge. The quad is constructed using the vertices for the silhouette edge and the same vertices pushed infinitely far away from the light. The shadow volume construction of a cube is shown in .

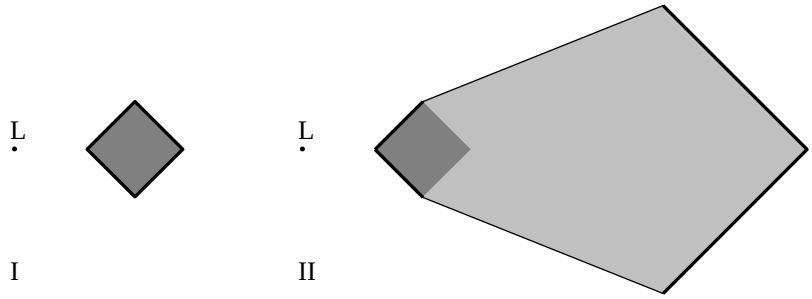

Figure 2 *I) A cube and light source. II) Shadow volume mesh constructed from the cube based on the position of the light. The original cube is shown with a darker shader of gray. The faces added in silhouette edges are shown as the thinner lines. The thick lines in the shadow volume meshes show the faces of the original geometry used to close the shadow volume mesh, the thinner lines are the quads inserted in silhouette edges.*

If you want to achieve a robust shadow volume implementation, it is often essential that the shadow volume mesh is a closed mesh (Bergeron 1986). This can complicate the procedure somewhat but if the object mesh is a simple closed mesh enclosing a finite subspace the process is rather straightforward. In this case a closed shadow volume mesh can be constructed by combining the quads inserted in silhouette edges with the original geometry where the faces facing the light are kept at their original position while the faces facing away from the light are pushed back enough to make sure the resulting shadow volume mesh is connected.

It's quite likely that al least some of the geometry will not enclose a finite subspace. This is for instance true for a simple room modeled through a cube where all polygons face toward the center of the cube. In this case a slightly different technique is better for the shadow volume construction. Find all polygons front facing or back facing the light source and compute the edges for these faces. Every edge only belonging to one polygon will be a silhouette edge and treated just as in the previous algorithm. Closing the volume can be done using the original front or back facing polygons and copies of those pushed back from the light far enough to close the mesh. The choice of using front facing or back facing polygons for shadow volume extraction will have influence on the robustness and performance of the algorithm. Using back facing polygons will work in all but very special cases and produce a quite small shadow volume mesh while using front facing polygons works in every case but results in a larger more complex shadow volume mesh. In this thesis the back facing polygons was used since it covers all desired cases and gives better performance. All in all, this algorithm isn't much harder to implement than the previously described algorithm for shadow volume extraction. However, the previous algorithm may be adapted to execute completely on the vertex program unit of graphics hardware. This algorithm on the other hand must currently be executed almost entirely on the CPU.

It's worth to note that shadow volume meshes quickly become more complex when the shadow caster geometry is more complex. *Figure 3* shows a slightly more complex mesh than the nicely convex cube. The generated shadow volume is a closed but self intersecting mesh and the number of levels of self intersections in a shadow volume mesh is equal to the number of overlapping concavities in the shadow caster from the lights point of view.

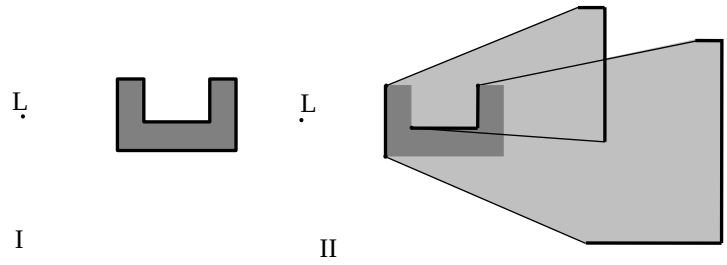

Figure 3 *I) A more complex shadow caster object. This object is concave from the lights point of view. II) The shadow volume mesh shown just as in . Note that this mesh gets more complicated and even intersects itself due to the concavity in the shadow caster.*

Once the shadow volume mesh is constructed we can use it to determine whether a point is in shadow or not. It is equivalent to testing whether the point lies within the shadow volume mesh which can be tested using a variation of the crossings test (Haines 1994). Imagine yourself standing on a field where an area is enclosed by fences. Start walking straight in an arbitrary direction, if the number of fences you had to cross is an odd number you started inside the enclosed area and otherwise you started outside of it. This algorithm works in the three-dimensional case as well.

The crossings test does not deal with some issues related to shadow volume meshes that are constructed using the silhouette edges such as self intersection. However, the algorithm can be extended to computing the difference between number of "back facing" fences and the number of "front facing" fences (assuming there is such a thing as a front/back facing fence). One can realize that you started inside the field if the number is greater than zero and that you started outside the field in the zero case. *Figure 4* gives a graphical example of the crossings test in action.

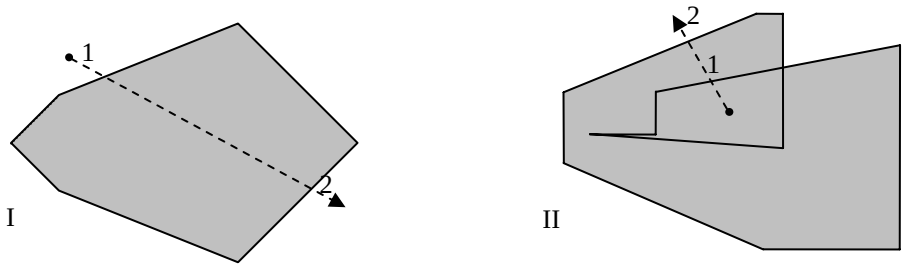

Figure 4 *The crossings test performed along the arrow line on the shadow volume meshes constructed in . I) Note that on this simple shadow volume constructed by the cube, the odd/even crossings test is sufficient. II) For this more complex shadow volume the odd/even test returns 2 (even) which would mean that the point isn't in shadow. Computing #back facing fences crossed - #front facing fences crossed instead will give the number 2 (2 back facing fences) which correctly states that the point is in shadow.*

In practice this test is implemented with the viewpoint as the starting point and walking from the viewpoint towards the point that we want to do the shadow test for. In this form the algorithm can be accelerated by most common hardware using the stencil buffer (Heidmann 1991). The stencil buffer handles integer values and can perform simple operations such as incrementing and decrementing the values stored in the buffer, it also provides testing mechanisms to discard output on a per pixel basis when rendering. The

idea is to use the depth buffer in combination with the stencil buffer to do the counting. This can be done using the following algorithm, which is also illustrated in *Figure 5*.

1. Render the whole world geometry from the cameras point of view to set up the correct depth buffer values. This is also a good opportunity to render effects independent of the lighting such as ambient light, reflection maps and similar.

2. For every light L
   - Clear the stencil buffer to value zero, Turn off depth buffer updates
   - Compute the shadow volume mesh for the geometry and L
   - Render the **front facing** polygons (from view position) of the shadow volume mesh and **increment** the stencil buffer value if the **depth test passes**
   - Render the **back facing** polygons (from view position) of the shadow volume mesh and **decrement** the stencil buffer value if the **depth test passes**
   - Render the lighting for L with the stencil buffer hardware setup to reject pixels with a stencil buffer value not zero

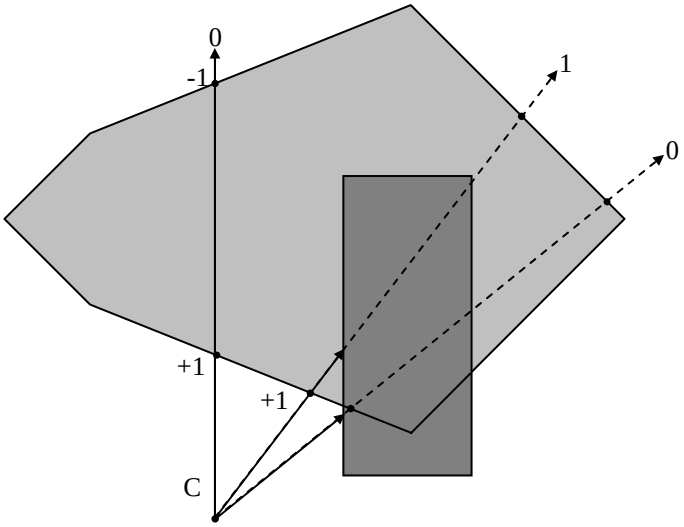

Figure 5 *Using the stencil and depth buffer to determine if a pixel lies in shadow. This figure shows the shadow volume of the cube object in and a rectangular object. To simplify the figure the cube object and shadow volume of the rectangular object is left out but in practice every object and the shadow volume of every object is processed. Every pixel rendered to the screen can be thought of as following one ray as shown in the figure and performing the crossings test (Figure 4). The depth buffer assures that the ray followed "ends" at the position of the original geometry since no counting is performed for pixels failing the depth test*

This algorithm, often called the depth pass shadow volume algorithm, suffers from many problems related to the view frustum. Counting will not be made for those pixels clipped by the camera near clipping plane which results in artifacts when the camera moves in and out of the shadow volume mesh. If the camera is completely in the shadow volume mesh the artifacts can be fixed by clearing the stencil buffer to a value corresponding to the crossings test result for the cameras point of view but when the near clipping plane of the camera intersects a polygon of the shadow volume it will result in some crossing tests starting in shadow and some starting outside. This is illustrated in *Figure 6*

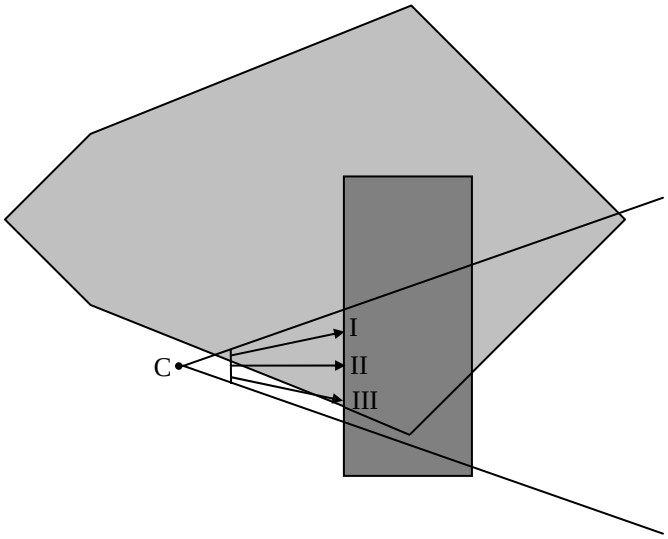

Figure 6 *The rays used for the crossings test starts at the near clipping plane rather than the camera point when the depth pass implementation is used. Ray II and III will cross a shadow volume polygon and therefore indicate that the pixels lie in shadow. Since ray I starts inside the shadow volume mesh it doesn't cross any shadow volume polygons, this results in that the pixel will be considered not to be in shadow although it is.*

To work around this unwanted behavior the algorithm can be changed to the following (changes are marked in bold), which is also illustrated in *Figure 7*.

1. Render the whole world geometry from the cameras point of view to set up the correct depth buffer values. This is also a good opportunity to render effects independent of the lighting such as ambient light, reflection maps and similar.

2. For every light L
    - Clear the stencil buffer to value zero, Turn off depth buffer updates
    - Compute the shadow volume mesh for the geometry and L
    - Render the **back facing** polygons (from view position) of the shadow volume mesh and **increment** the stencil buffer value if the **depth test fails**
    - Render the **front facing** polygons (from view position) of the shadow volume mesh and **decrement** the stencil buffer value if the **depth test fails**
    - Render the lighting for L with the stencil buffer hardware setup to reject pixels with a stencil buffer value not zero

This simple reverse (Bilodeau 1999, Carmack 2000) moves the artifacts from appearing when a polygon is clipped by the near clipping plane to appearing when a polygon is clipped by the far clipping plane. It can also be thought of as letting the rays used for the crossings test come from points infinitely far away and travel towards the camera instead of the other way around. The reverse introduces need for a fully closed shadow volume mesh, at least when the camera is positioned within the mesh. This variation of the algorithm is often called the depth fail shadow volume algorithm since the counting operations are performed when the depth test fails rather than passes. It is often combined with setting an infinite far clipping plane or using the depth clamping (instead of clipping) extension supported by some graphics hardware (Everitt 2002, Lengyel)

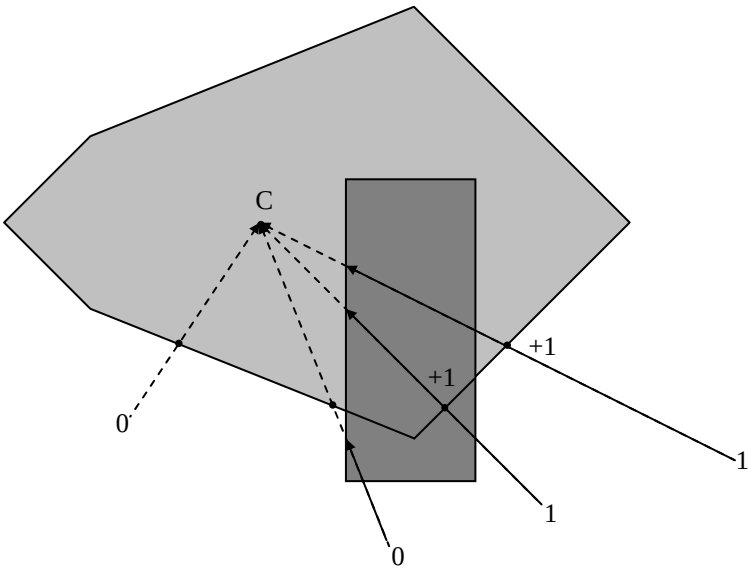

Figure 7 *Changing the stencil counting algorithm to depth fail instead of depth pass. Closing the shadow volume mesh is important at least when the camera is positioned inside the shadow volume. Note that the algorithm works just as in Figure 5 but performs operations when the depth test fails which is equivalent to performing the crossings test in the opposite direction, from infinity towards the camera.*

The shadow volume algorithm has through several years of development reached a point where it can be considered robust. Its strengths lie in the geometrical representation of shadows which gives a pixel accurate representation of the shadows as well as possibility to include omni-directional lights with no modifications to the algorithm.

As for weaknesses the shadow volume algorithm also has its fair share. The fist one concerns hard shadows, due to the fact that shadows are pixel perfect there will be a razor sharp edge separating lit from shadowed. This makes an image look synthetic since real shadows usually have a fuzzy edge due to the fact that lights in reality are volumetric or have an area instead of being a single point. Since there are implementations of the shadow volume algorithm to render soft shadows (Akenine-Möller 2002) this gets less and less of an issue.

Another weakness of the shadow volume algorithm is its massive usage of rendering resources. Since the shadow volume mesh is pushed away from the light it often tend to occupy more pixels on screen than the original model. This is illustrated in wire frame mode in *Figure 8*.

Finally the shadow volume algorithms operates on mesh geometry while the shadow map algorithm operates on a per pixel level. Thus, the shadow map algorithm can be altered to work on semi transparent surfaces and surfaces where texture maps control the level of transparency. This is not easily done using the shadow volume algorithm.

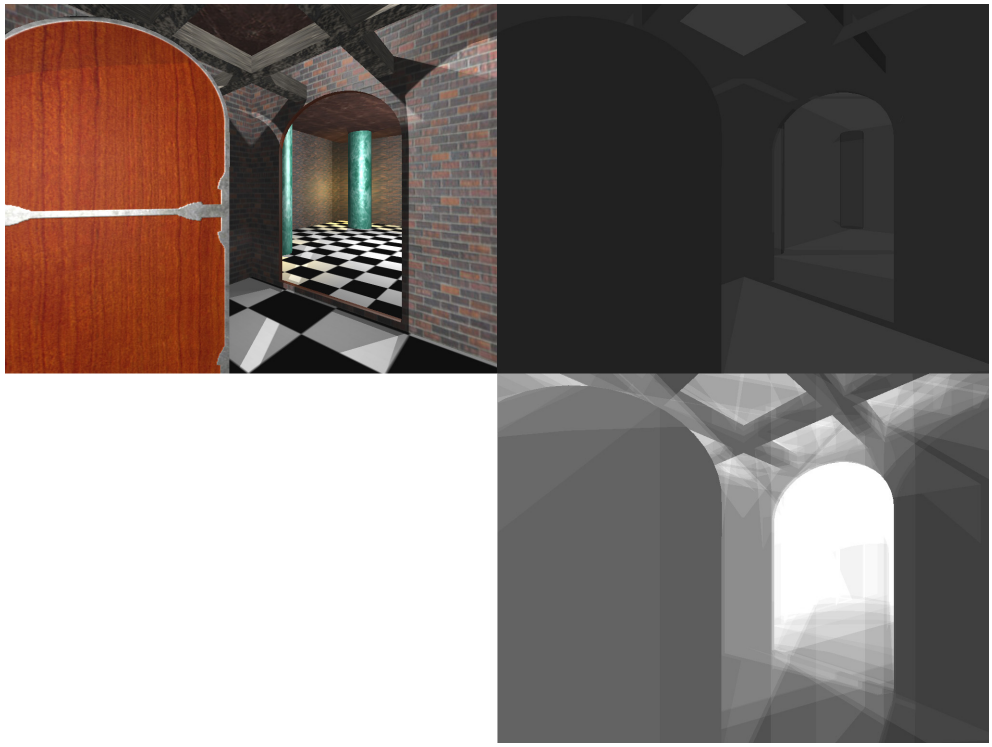

Figure 8 *Top left: Rendered image. Top right: Overdraw factor for lighting only. Bottom right: Overdraw factor for lighting and shadow volumes. Pure white means the pixel is processed at least 20 times. Pixels failing the depth test were ignored although they contribute to further slowdown.*

2.4 Rendering

As can be seen in the previous algorithms the rendering pipeline of an engine designed for per pixel lighting is a bit different from the traditional vertex lighting pipeline. When using OpenGL or some other API for rendering, one usually renders every object in the scene together with information about the relevant lights. However, per pixel lighting is at least a two pass rendering process for every single light. The rendering of one frame is performed in the following way.

- Normal visibility calculations, note that good view frustum and occlusion culling is even more important in an engine with complex lighting. The ambient pass gives great opportunities for "free" hardware accelerated occlusion culling.
- Render visible objects with ambient lighting, mainly to initialize the depth buffer.
- For every visible light
    - Visibility calculations based on both light position and position of viewer
    - Render shadow maps or shadow volumes for objects visible to light
    - Render lighting on objects visible to viewer with shadow testing for every pixel

2.5 Culling

Culling (Policarpo, 2001) is the process of removing what is not visible to the viewer. This is done using gross representations of the objects in hope that you gain more performance by quickly removing the invisible objects than you loose when performing the culling tests.

In practice this will almost always be true since the tests performed are very simple and may remove objects consisting of several thousands of elements. The tests are generally performed using so called bounding volumes, a kind of simple geometric primitive enclosing the entire object. If the bounding volume is invisible the object must be too but the bounding volume can be visible although the object is not. The first part of *Figure 9* shows a typical culling operation between the view frustum and the bounding sphere of an object

The exact implementation of culling is dependant on what data types are used in the engine and the quality and performance of the different types of culling often have a major impact on the complexness of the scenes that can be rendered at interactive frame rates.

When it comes to per pixel lighting and shadowing rendering is a two pass process for every light source visible to the viewer which means that there's a great deal to win by performing culling not only for the view frustum but also based on the properties of the light when lighting is rendered. There are also pitfalls since an invisible object may still cast a visible shadow so we need to isolate the culling into two cases: light culling and visibility culling. Visibility culling can be performed just as before and is mainly used during the ambient rendering pass. The light culling process on the other hand is dependant on the technique used for shadows. With shadow maps the culling can be done using the frustum for the light as well as the position and radius of influence, after all rendering the shadow map is no different from a common rendering operation so the same culling methods can be used. When using shadow volumes the light culling starts like when using shadow maps with the only difference that a frustum for the light may not exist if the light is omni directional. The result of this culling will be a list of objects visible to the light and inside it's radius of influence which can then be used to determine objects with a visible shadow. The objects with a visible shadow can be found by extracting the shadow volume mesh for the bounding volume and cull it by the view frustum. The complete light culling process is shown in the second part of *Figure 9*. Everitt et al (2003) presents an alternative culling approach for this last step which alters the view frustum and therefore doesn't have to compute the shadow volume meshes for the bounding volumes.

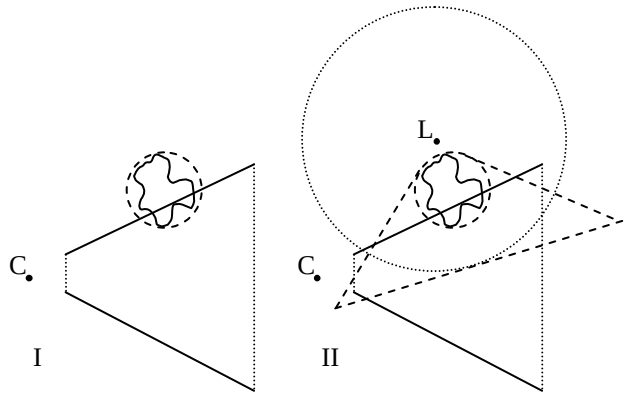

Figure 9 *The culling process. I) A complex object is represented through the bounding sphere, since the sphere intersects the view frustum the object is assumed to be visible which is also the case in this example but must not be true at all times. II) Light culling, first a complete object culling is done based on light position and radius. This is followed up by culling the shadow volumes of bounding volumes to the view frustum.*

We now have lists of objects for the ambient rendering pass and the shadow volume rendering pass the only thing left is to compute a list of objects for light rendering, this is

simply the intersection of both culling operations which is equivalent to the objects existing in both lists.

3 Optimizations for shadow volumes

3.1 Concept

Shadow computations are done on a per-pixel level when using volume shadows. The advantage of this is that the shadow computations get relatively simple, increasing/decreasing a value and a simple comparison is all that is needed to do the computations in the shadow volume case. The downside is of course that a lot of computations must be done all in all due to the large amount of pixels that must be rendered to get the desired result.

Another possibility of generating shadows is to operate on the polygons of a scene rather than a single pixel. These algorithms are almost totally disregarded today because they are much too complicated to perform even on modern graphics hardware. They also scale worse than volume shadows and shadow maps when the number of polygons in a scene grows. My intent is to use a combination of these algorithms where I mix the advantages of both worlds to achieve better overall performance.

If we take a closer look at the shadow volume algorithm it could just as well be performed on a polygonal level as well as the pixel level. Cut all polygons by the shadow volume mesh for a single light and remove all resulting polygons that are inside the shadow volume mesh. This would save us not only the fill-rate needed to render the shadow volume polygons, but we would also skip drawing what's in shadow rather than actually drawing it and check which pixels should be skipped. The only problem associated by performing the shadow volume algorithm on a per-polygon level is the vast amount of computations required to cut all polygons by the shadow volume mesh and determine which are inside it.

Let's assume for a while that all meshes in the world and all lights in the world are static. In that case there wouldn't be a problem with cutting polygons by the shadow volume mesh for every light since this could be offloaded to a pre-computation phase. In this phase we can spend quite some time and resources on performing the shadow volume algorithm on a per-polygon level rather than a per-pixel level and store it in a file for later use. If this is done for every light in the world we would end up with the original world mesh and a mesh for every light containing polygons only completely visible from the light. Upon rendering time these pre-calculated light meshes could be used to render lighting with correct shadows without any testing what so ever at runtime.

In reality we can't assume a completely static world with completely static lights but what we can assume is that a majority of the polygons and lights will be completely or at least partially static. Since we want to support dynamic objects as well, shadow casting must work both ways between dynamic and static geometry. This results in eight different cases of shadowing if considering both dynamic and static lights and geometry. As can be seen in *Table 1* there is in practice only three different methods as only two of the combinations can be optimized and the rest of the cases are performed using a standard stencil buffer based shadowing algorithm. The cases on which to use the standard algorithm are all cases where light or shadow caster is dynamic, since all pre-computations are based on static data there's no additional information that can be used to optimize performance.

Light type	Shadow caster type	Shadow receiver type	Shadowing algorithm
Static	Static	Static	Pre-computed lit polygons
Static	Static	Dynamic	Optimized stencil buffer shadow volumes
Static	Dynamic	Static	Stencil buffer shadow volumes
Static	Dynamic	Dynamic	Stencil buffer shadow volumes
Dynamic	Static	Static	Stencil buffer shadow volumes
Dynamic	Static	Dynamic	Stencil buffer shadow volumes
Dynamic	Dynamic	Static	Stencil buffer shadow volumes
Dynamic	Dynamic	Dynamic	Stencil buffer shadow volumes

Table 1 *Different cases of lighting and geometry (static or dynamic) and the algorithm used in every case.*

The case where static geometry casts shadows on dynamic geometry should be handled with care because this case conflicts with the optimizations done through cutting the world polygons since it means that we still need to render the shadow volumes of the static geometry if we want to use the stencil buffer based method to compute those shadows. However, we still have a good advantage over the other cases with dynamic parameters where shadow volumes are used. The advantage is that the shadow volume mesh is known and static and that shadows on static geometry are already handled. The only varying parameter is the objects that receive the shadow. From these dynamic objects important information can be rapidly computed using bounding volumes, this information can be used with traditional shadow volume optimization strategies. For instance graphics hardware is able to limit rendering to a given rectangle of the screen using a so called scissor box. Usually the size of these boxes are determined by the projection on the screen of a light but since shadows are already correct on the static geometry the scissor boxes can be limited to the intersection of the projection of dynamic objects and the light which is often considerably smaller than the projection of the light and thus saves fill-rate. Another optimization that can be done is to construct minimal view frustums that contain dynamic objects and the camera and use these to cull many unnecessary shadow volumes that the scissor boxes don’t handle.

The rendering approach found in the previous section has to be changed when using this algorithm. For the dynamic lights everything is more or less the same but static lights are treated differently to be able to get as much optimization as possible out of the case where static geometry shadows dynamic objects.

- Normal visibility calculations, note that good view frustum and occlusion culling is even more important in an engine with complex lighting. The ambient pass gives great opportunities for "free" hardware accelerated occlusion culling.
- Render visible objects with ambient lighting, mainly to initialize the depth buffer.
- For every visible static light
    - Visibility calculations based on both light position and position of viewer
    - Render shadow volumes from dynamic objects
    - Render lighting on the static geometry with stencil testing
    - Compute sets of dynamic objects that will be treated as one entity during shadow volume rendering
    - For every such set of dynamic objects
        - Render static shadow volumes optimized based on the properties of objects in this set
        - Render lighting on dynamic objects in the set, with shadow testing
- For every visible dynamic light
    - Visibility calculations based on both light position and position of viewer
    - Render shadow volumes
    - Render lighting on objects visible to viewer with shadow testing for every pixel

3.2 Static light computations

In the pre-computation phase where we wish to compute the meshes containing lit polygons only, a number of basic tools are needed. These are shortly presented before the actual algorithm for computing the "light meshes". The most important component of this algorithm is a robust mesh by mesh clipper. That in combination with basic ray tracing and silhouette extraction is everything needed.

3.2.1 Edges in a mesh

An edge is defined by two vertices $\{V_a,V_b\}$ and a number of polygons connected to the edge. For convenience, the number of polygons connected to a single edge may not exceed two. This introduces some limitations on supported meshes but it's rarely needed or even practical to have more than two polygons per edge in a mesh. However, sometimes CSG operations in modeling packages may result in these types of anomalies which should then preferably be corrected before the operations are performed. In *Figure 10* a very simple mesh and its edges are illustrated.

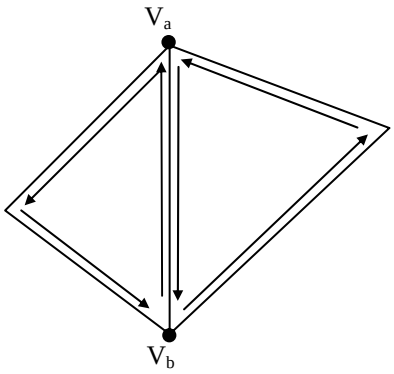

Figure 10 *Mesh with 5 edges where the edge between V_a V_b is shared by both triangles in the mesh.*

A silhouette edge from the viewpoint $V=[V_x,V_y,V_z]$ is defined as:

- An edge between two planar convex polygons with one of the polygons front facing the viewpoint and the other polygon back facing the viewpoint
- Any edge that only belongs to a single polygon

The second point is not needed if the mesh is a closed mesh, but is practical if we want to compute shadow volumes based on the computed light mesh which will be an open mesh due to the removed parts that lie in shadow

3.2.2 Mesh clipping

Clipping a mesh A by mesh B is equivalent to ensuring that no polygons in mesh A cross any polygon in mesh B. The two dimensional case of mesh clipping, which is to clip a polygon by another polygon in the plane, is shown in *Figure 11*.

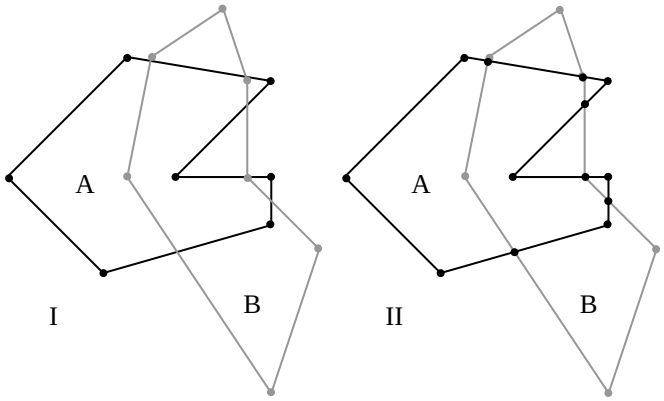

Figure 11 *Polygon by polygon clipping in the plane (two dimensional case of mesh by mesh clipping). I) Two polygons, before the clipping operation is performed. II) Polygons after clipping A by B. All line segments in A that cross any boundary of B is split into many new segments and new vertices are inserted in the intersection points.*

The first attempt in creating a mesh clipper may be to implement a polygon by polygon clipping method. However, this is not as easy or straightforward as a line clipper in the two dimensional polygon by polygon clipping case. *Figure 12* shows a possible case for

polygon clipping, the main problem here is that the result from clipping polygon A by B is a polygon just like A but with an infinitely thin hole in it. If both meshes used for clipping are closed meshes all the infinitely thin holes caused by clipping will form a line curve or line loop splitting the polygon. It is probably possible to implement a mesh by mesh clipper that operates using exact polygon clipping but there will be many problems to solve related to finding the whole intersection line curves or loops. Also the resulting polygons may be concave which means that they need to be tessellated to convex polygons before rendering.

A simpler and much more useful operation to base a mesh by mesh clipper on is to cut a polygon by an infinite plane. This is done by traversing the edges of a polygon and inserting a new vertex if the two vertices of the edge cross the cutting plane. If the input polygon is a convex polygon that crosses the plane the resulting polygons will be two convex and planar polygons, one behind and one in front of the cutting plane. *Code 1* shows a simple pseudo-code function for cutting a polygon by a plane and *Figure 12* illustrates the polygon by plane clipping for the same example as the polygon clipping.

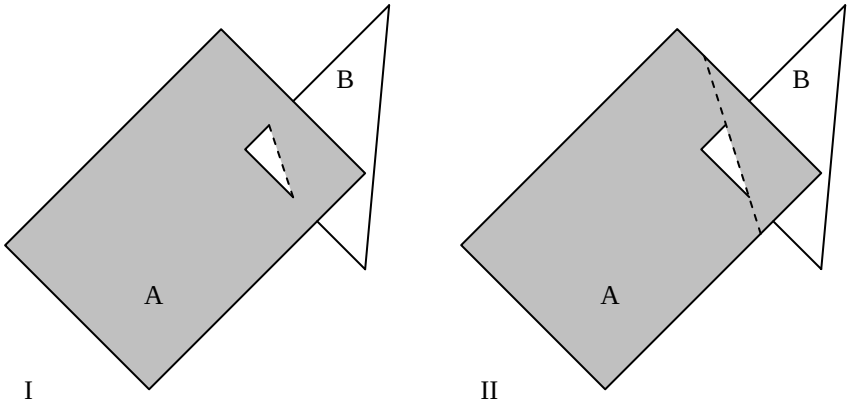

Figure 12 *I) The problem of polygon clipping, how is polygon A clipped by B? Is a resulting polygon with an infinitely thin hole in practical to work with? The answer is no. II) Another approach, clip polygon A by the plane of polygon B. The result is two convex and planar polygons which is easier to work with than the result in I*

Clipping all polygons in one mesh by the planes of all polygons is simple enough to code but will produce an output mesh with a lot of polygons. Since performance is not only dependant on fill rate but also on the geometrical complexity it is of interest to minimize the number of polygons and vertices inserted in the clipping operation. One thing that can be done is to only perform the polygon by plane clipping operation if the polygons of the different meshes actually intersect. To find out if two polygons intersect (Möller 1997) is pretty close to cutting a polygon by a plane. Find the intersection points between the first polygon and the plane of the other polygon and vice versa. If both polygons intersect the plane of the other polygon, the result will be two three dimensional line segments, one for each polygon. The line segments are segments of the same infinite line, the line where the planes of both polygons intersect. This means that the segments can be represented with one-dimensional intervals. If these intervals overlap the polygons intersect. The overlapping test can be done using the largest coordinate component of the interval vectors which saves the trouble of parameterizing the line segments.

```
[back,front] = cutPolyByPlane(Poly,Plane)
    List FrontVerts,BackVerts
    for evert vertex V in Poly
        N = cyclicNextVertex(V)       // chooses vertex 0 if V is last vertex

        if sideOfPlane(V,Plane) == FRONT
            add(FrontVerts,V)
        else
            add(BackVerts,V)

        // Test if edge cross the plane, if it does insert a new vertex
        if sideOfPlane(N) != sideOfPlane(V)
            alpha = distance(V,Plane)/(distance(V,Plane)-distance(N,Plane)
            newVertex = linearInterpolation(V,N,alpha)
            add(FrontVerts,newVertex)
            add(BackVerts,newVertex)

    back = face with vertices BackVerts
    front = face with vertices FrontVerts

    return [back,front]
```

Code 1 *Pseudo-code that clips a polygon by a plane. The resulting is two new polygons, one in front of the plane and one behind the plane.*

Writing a robust mesh by mesh clipper can be a more problematic task than it seems. As a rule of thumb, floating point comparisons should be avoided whenever possible due to precision related issues. A problem that is introduced when only performing polygon by plane clipping for polygons that actually intersect is that T-junctions may appear. An example clipping case when a T-junction appears is illustrated *Figure 13*.

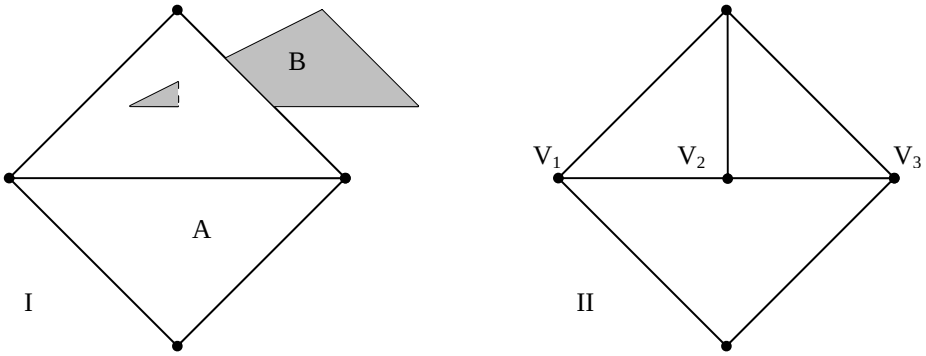

Figure 13 *Image of a T-junction. I) The original mesh A and another mesh B used for clipping. II) The resulting mesh after the mesh clipping operation. Note that the resulting mesh is "opened" along the edge where only one polygon is clipped and the other is not.*

T-junctions introduce precision related pixel sized gaps when rendering geometry and they will also open the resulting mesh in an unnecessary place which will possibly create more silhouette edges and thus unnecessary shadow volumes. Removing the T-junctions can be a quite troublesome functionality to code. Especially since floating point comparisons should be avoided whenever possible. I use an edge "cache" in my clipping utility which stores clipped edges when clipping a mesh by a specific polygon (mainly to ensure only one extra vertex was inserted for an edge shared by clipped polygons) this cache is used to insert the extra vertex in all none clipped polygons sharing a clipped edge. However, this might not be efficient or possible depending on which types of optimizations that are used in the clipping code. The structure of a mesh clipper is outlined on a high level in *Code 2*.

```
clipMeshByMesh(meshA,meshB)
    for every polygon Pb of meshB
        for every polygon Pa of meshA
            if intersects(Pa,Pb)
                [back,front] = cutPolyByPlane(Pa,Pb.plane)
                meshA.removePoly(Pa)
                meshA.addPolys([back,front])

        RemoveTJunctions(meshA) //T-junctions can be removed per polygon

    // Or here, per mesh which is harder but possibly faster
```

Code 2 *Very high level pseudo code for mesh clipping*

3.2.3 Ray tracing

Ray tracing will also be used in the algorithm for visibility testing purposes. Tracing a ray for intersections can be used to efficiently check visibility between two points. Even though we work with polygons in the end point visibility can be used. Tracing a ray is as simple as getting all intersections between a three-dimensional line segment and all the polygons in the scene. Given a position $P = [P_x,P_y,P_z]$ and a direction $D = [D_x,D_y,D_z]$ the ray is given by *Equation 5*

$$R(t) = P + tD$$

Equation 5 *Ray parameterization from a point and vector*

For the intersection between a ray and a plane of a polygon (ax+by+cz+d=0) we can solve the t parameter as the minimal distance between P and the plane divided by the length of the projection of the direction vector D on the normal of the plane. Call the plane normal N = [a,b,c] and we can write the following equation (*Equation 6*).

$$t = -\frac{\frac{N \cdot P + d}{|N|}}{\frac{N \cdot D}{|N|^2}} = -\frac{(N \cdot P + d)|N|}{N \cdot D}$$

Equation 6 *Intersection between a ray and a plane (can be optimized if normal is normalized).*

Once the plane intersection parameter value is established the intersection point can be computed directly from the ray equation (*Equation 5*) once the intersection point is found it's a matter of an inside/outside test for the intersection point, this could be done using the 2-dimensional crossings test that was described in the shadow volume section.

Ray tracing can be sped up by a number of different tweaks where trees are probably the most important. Using a hierarchical structure, many triangles may be culled with very little effort. One may also traverse the tree in an ordered fashion which gives an opportunity for early exits if only the first intersection point is desired. The implementation made for this paper used an oc-tree based ray-tracer with early exit. A brute force ray-tracer was also implemented for debugging purposes.

3.2.4 Algorithm

In this pre-computational phase we wish to compute a mesh for every light containing only polygons lit by the light. Polygons partially lit should be cut to construct polygons only completely lit by the light. The following tree steps sum up the algorithm quite well:

- Compute shadow volume mesh
- Create a new mesh by cutting a copy of the static geometry mesh by the shadow volume mesh
- Test every polygon in the new mesh for visibility by tracing a single ray to a point in the polygon

Given the previously described tools this is a rather simple operation but it can get quite slow. Common data structures such as trees or portal/zone partitioning can be used to optimize the performance of the algorithm.

The first step, computing a shadow volume mesh, is done using the definition of a silhouette edge. Gather a list of silhouette edges by testing all edges in the mesh to the silhouette edge conditions. Once this is done the shadow volume mesh can be constructed by inserting a quad for every silhouette edge. The quad is defined by the vertices of the silhouette edge and the same vertices pushed directly away from the light source in a straight line. The vertices must be pushed back far enough to ensure that the pushed back edge of the quad does not intersect the bounding sphere of the light-source.

The second step is to cut the world geometry mesh by the constructed shadow volume mesh. If a robust mesh clipper (as outlined in previous sections) is already implemented this is just a matter of a single function call.

Once the clipping is performed, we will have the same world mesh with a couple of new polygons caused by the clipping operations. A very convenient property with these polygons are that they are either completely in shadow or completely visible to the light-source. Visibility can now be tested from any point in the polygons to the light-source this is easily done by tracing a ray from the mid-point of every polygon to the light-source position if all polygons where an intersection of the ray occur are removed from the mesh we will end up with a mesh of lit polygons only. Alternatively one can construct a closed shadow volume mesh using the original back and front facing geometry plus the silhouette edges and then use this mesh to do an inside/outside test for the mid-point of every polygon on this mesh.

Since many 3D engines already have implemented simple polygon clipping and ray tracing tools it should not be too hard to implement this algorithm with reasonable performance (Performance is not that important in pre-computational phases unless the time required gets very high). The pseudo code representation of the outer (conceptual) loop of the algorithm is listed in *Code 3*

```
LitMesh = computeLitPolys(Mesh M,Light L)

// Extract shadow volume polygons from the silhouette edges
Mesh ShadowVolumes
    for every edge E in M
        if L.attenuation(E.minDistance(L)) > TRESHOLD && E.silhouetteEdge(L)
            ShadowVolumes.addPolygon(MakeShadowVolume(E,L))

    Mesh MCopy
    for every polygon P in M
        if L.attenuation(P.minDistance(L)) > TRESHOLD && P.FrontFacing(L)
            MCopy.addPolygon(P)

    Mesh LitMesh = MCopy.CutByMesh(ShadowVolumes)

    For every polygon P in LitMesh
        if M.rayIntersects(Ray(P.MidPoint(),L))
            LitMesh.removePolygon(P)
    Return LitMesh
```

Code 3 *Pseudo code for the main loop for cutting polygons to fit the completely lit/shadowed criteria*

The different methods used by this algorithm have already been outlined in previous sections, the main optimizations that can be done to this algorithm lie within the mesh by mesh cutting utility and the ray-tracer as these parts of the algorithm is $O(n^2)$ whereas silhouette extraction is only O(n). As previously mentioned the most rewarding optimization to these functions is to use a tree structure.

All of the different steps of the algorithm are illustrated in *Figure 14*

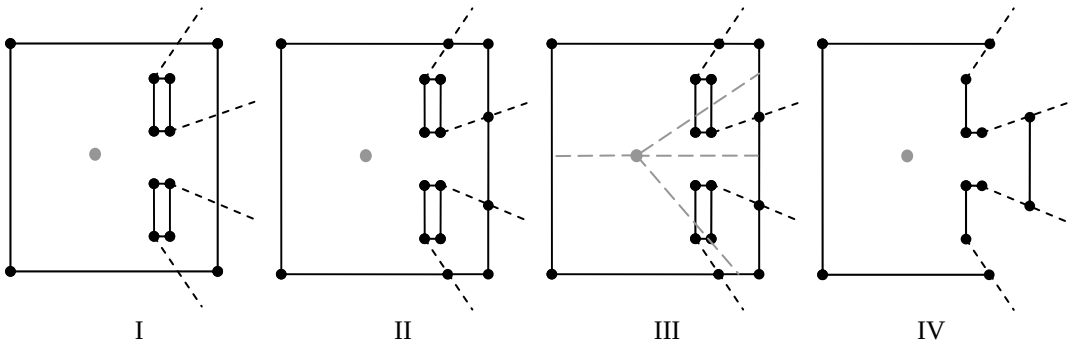

Figure 14 *The three major operations when cutting the world geometry. I) Shadow volume extraction. II) Cutting static geometry by the shadow volume mesh. III) Ray tracing from light to polygon to determine visibility (only some of the rays are shown). IV) Lit polygon mesh.*

The whole polygon cutting algorithm should be performed for every static light in the scene. This gives one mesh containing the original static geometry and one mesh for every light containing the lit parts of the static geometry for the light. Since the computations involved are not fit for real-time applications it's a good idea to store this information in the file used for describing the world geometry.

In this pre-computational phase, it's also a good idea to produce a potentially visible set for the light, this goes hand in hand with optimizations of the algorithm. Since my implementation of this algorithm is portal-cell based it's only natural to use this property to speed up the algorithm. This is done by testing the portals for visibility with exactly the

same type of cutting algorithm used for the shadows. Simply clip the portal polygon with all shadow volumes, the resulting polygons are tested for visibility with using ray-tracing and if any test pass the portal is visible. After this optimization of the algorithm, the potentially visible set of cells for a light is simply all cells that were processed. The pseudo-code listed as *Code 4* outlines the implementation of the portal-cell based version of this algorithm.

```
computeLitPolysRec(Cell C,Light L,ShadowVolumes Sv,OutData Data)

// The cell may have been entered through another portal
if !Data.PVS.cellExists(C)
    Data.PVS.addCell(C)

Mesh CellMesh = C.getMesh()

// Compute new shadow volumes and merge with shadow volumes computed for
previous
// cells. Also computes a list of lit polygons for this cell, only
difference in
// the computeLitPolys function is that it doesn't compute shadow volumes
and
// ray-tracing must be performed recursively through portals or be performed
on all
// meshes in "Data"
Sv.addShadowVolumes( ComputeShadowVolumes(CellMesh,L) )
Mesh LitMesh = computeLitPolys( CellMesh, L, Sv )

// Adds the light mesh if cell isn't inserted before, otherwise merges it
with
// previously inserted for this cell.
Data.addLitMesh(C,LitMesh)

for every portal P in C
    if !P.alreadyTraversed()
        ShadowVolumes isectsPortal = findIntersectingFaces(P,Sv)
        computeLitPolysRec(P.otherCell(C),L,isectsPortal,Data);
```

Code 4 *Outline of a recursive portal-cell based implementation of the pre-calculation algorithm. It also computes the potentially visible set for every light.*

3.3 Dynamic objects

Since we're dealing with real-time graphics, dynamic objects will of course play an important role. The dynamic objects should be able to move around in a static world and the lighting quality for the dynamic objects should be close to or on par with the lighting for static geometry. It's also important that the shadow casting between static and dynamic geometry works as it's supposed to or the rendered image will not be credible.

There are two types of shadows from static lights which can be treated in different ways. They are the shadows caused by dynamic objects on dynamic/static geometry and the shadows caused by static geometry on dynamic objects. The dynamic objects casting shadow on dynamic/static geometry is a simple case because it cannot be optimized at all. In this case the shadow volumes are dynamic which means that no complex data structures or pre-computations can be used. Another issue is that every object in the world including the static geometry is a potential shadow receiver. This gives fewer opportunities for optimizations of the shadow volume algorithm.

The case where static geometry casts shadow on dynamic geometry only includes static shadow volumes which mean that tree-structures may be used to accelerate algorithms. Also, the shadow receivers are only the dynamic objects since the pre-computational step already ensures shadows are correct on the static geometry. Since the dynamic geometry is

assumed to be relatively small in relation to the static geometry we can use the bounding volumes of dynamic objects to further improve common shadow volume optimizations such as scissor boxes and culling. *Figure 15* shows a typical case with a small dynamic object positioned in a static world with several lights. The figure also displays overdraw factor for all shadow volumes and after all optimizations implemented in this thesis project, described in detail later in this section. In the top row of the figure it's apparent that the projection of the dynamic object on the screen is very small, this results in a small scissor box used to clip the static shadow volumes and therefore the overdraw factor is very low except for in the small box around the sphere. The second row of figures shows another case where the projection of the sphere on the screen is large. Since the resulting scissor box will be large most of the optimizations will have to be done by culling shadow volume polygons. Thankfully, under the right circumstances this is a good scenario for shadow volume culling and as can be seen in the figure, many of the shadow volume polygons in the background have successfully been culled.

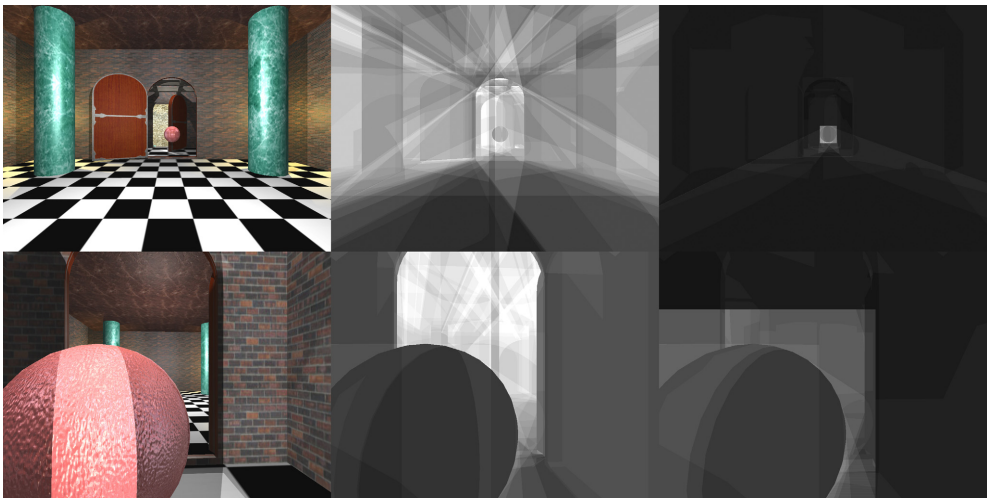

Figure 15 *Left: Rendered image with a dynamic object (the sphere). Mid: Shadow volume complexity visualized through the overdraw factor. Right: Near optimal overdraw under the assumption that all shadows on static geometry is pre-computed (except for the sphere shadows)*

3.3.1 Shadows cast by static geometry on dynamic geometry

As stated before this is the most important shadowing case, optimizations here will have a direct outcome on the performance of the engine at least fill-rate wise. The first and most obvious approach is to render the complete shadow volume mesh for the static geometry to set up the stencil buffer. However, this is also the worst approach from a fill-rate perspective although it is the best if we consider CPU utilization. Since all shadow volume polygons will be rendered it efficiently cancels out all optimizations that was done during the pre-computational phase.

What can be done is to use simple information about the dynamic objects, such as bounding volumes, to try to prevent as many shadow volume polygons and pixels as possible from actually being rendered. The more rendering that can be prevented the more performance gained from the pre-computational phase can be kept during run time.

To achieve optimal performance there are several factors that must be taken into account. There are three conflicting factors, fill-rate utilization, geometry utilization and CPU utilization. During the rendering process it's possible to make the decision to "merge"

several dynamic objects and treat them as one single entity as far as shadows is concerned. Doing this will result in lower geometry and CPU utilization but also in a larger entity which means less spatial information and also less opportunities to cull shadow volume pixels that does not contribute to shadows on the dynamic objects.

Common shadow volume optimizations for hardware (Everitt, 2003 and Kilgard 2003) include using scissor boxes and view frustum culling of the shadow volumes to render. Since the shadows only need to be correct on the dynamic objects we can use the information about all objects in every such merged group to further narrow the scissor box and the frustum used for culling. It's important to understand how merging of different objects will affect the resulting scissor box and how the culling operations. This is described in the following sections

3.3.1.1 Scissor boxes

All modern hardware is capable of clipping polygons against a view frustum to make sure only pixels that are actually visible are processed. The clipping can be setup to force the card to consider only a part of the screen as a visible surface, this operation is called scissoring and is often used to try to minimize the number of pixels that needs to be processed when rendering shadow volumes.

The most common optimization is to compute a screen-space bounding rectangle for every visible light in a scene and use that to discard all the pixels that cannot be affected by the light both when it comes to shadow volume rendering and light rendering. Since the static shadows are already computed for the static geometry it's only necessary that the light and shadow rendering is correct on the pixels occupied by the dynamic objects. This means that the bounding box of a dynamic object can be projected to a bounding rectangle on the screen and then used as well as the bounding rectangle of the light to compute a new bounding rectangle of the pixels that needs correct rendering for that particular object. The resulting rectangle will be the intersection of the light and object rectangles.

When more than one dynamic object is visible on screen the bounding rectangle problem gets considerably more complicated. If two bounding rectangles exist for two different objects we can either choose to merge them to a single bounding rectangle which is the union of the two original rectangles. This will increase the area of the boxes and therefore increase fill-rate usage. On the other hand, if the two bounding boxes are not merged everything will be rendered twice which means a higher amount of polygons sent to the graphics card. The bounding rectangles for the object may also overlap which means that if they are not merged the area shared by both rectangles will be rendered twice. This may not sound so bad but the double stencil buffer rendering in the shared area will cancel each other out or otherwise produce artifacts, the stencil buffer values in the shared area must be cleared which unfortunately also clears out possible values from a previous pass where dynamic object shadowing dynamic or static geometry is computed, also using the stencil buffer. An alternative way to clear the effect of shadow volumes rendered if using the invert stencil buffer operations is to render the shadow volumes once again which means that the geometry must be rendered three times. The differences fill-rate wise and geometry wise between two overlapping boxes is illustrated in *Figure 16*.

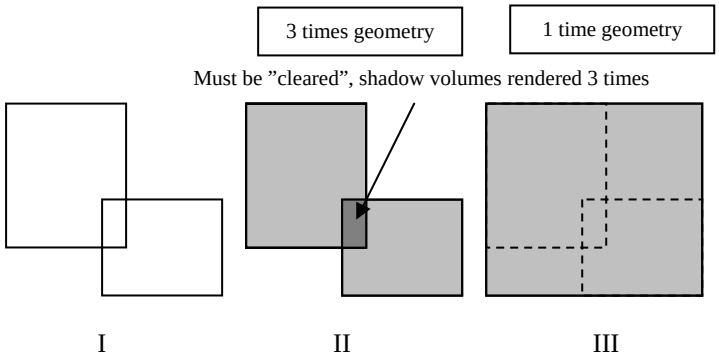

Figure 16 *Two rectangles approximating the area of two dynamic objects in screen space. Shows shadow volume rendering based on the rectangles individually and a combined box for both objects. I) Original bounding rectangles for the two objects. II) Shadow volume rendering process if objects are treated individually. III) Objects treated as one entity for the shadowing process.*

3.3.1.2 Frustum culling

The choice of shadow volume implementation has great impact on the frustum culling optimizations that can be made. The shadow volumes algorithm can be implemented in two ways, depth pass and depth fail, which are both described in the background section. As the name suggests one of the implementations performs stencil buffer operations when the depth test passes and the other implementation performs stencil buffer operations when the depth test fails, this is illustrated in *Figure 5* and *Figure 7*.

Take the case of a single dynamic object, in the previous section the bounding rectangle of an object was used to compute a scissor box. Another common shadow volume optimization is to properly cull shadow volume meshes by the view frustum to ensure they are visible before rendering them. Culling is often done on a per object basis but since shadow volumes are static in this case a tree data structure can be used and therefore it is reasonable to assume that polygon accurate culling may be performed. Since our only interest is correct shadowing on the pixels occupied by dynamic objects we could compute a frustum based on the bounding rectangle used for scissoring and an approximate nearest or furthest point on the dynamic object. The frustum creation is dependant on the shadow volume algorithm chosen. If depth pass is chosen the frustum will be defined by the bounding rectangle and have the furthest point on the object as a far clipping plane because all shadow volume polygons completely behind the object will only fail the depth test and therefore not contribute to the final stencil buffer value. The depth fail implementation on the other hand will use the nearest point on the object as a near clipping plane and use no far clipping plane at all. In *Figure 17* a simple fictive scene is shown as well as the culling possibilities using the depth pass and the depth fail shadow volume algorithms. There’s a consistent relation between the size of an object onscreen (intersection with the near camera plane in the figure) and the number of shadow volumes that can be culled away.

The depth fail technique has the most extreme cases. It’s possible to cull many shadow volumes when the projected image of an object is small but when the projected image grows larger the number of shadow volumes that can be culled shrink drastically. A very good best case but also a very bad worst case where we’ll end up with a large scissor rectangle and a large overdraw.

The depth pass algorithm on the other hand has the complete opposite behavior. When the projected image of an object shrinks, the number of shadow volumes between the camera and object increase and when the projected image grows, the number of shadow volumes decrease. This gives us a constant performance rather than the two extreme cases.

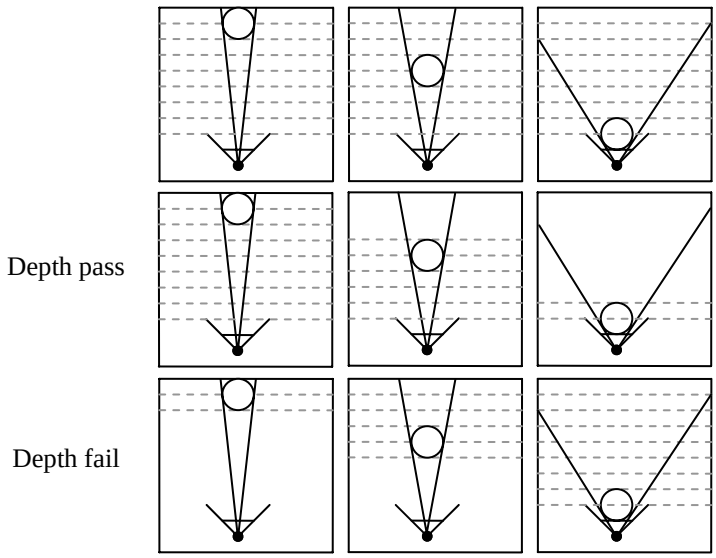

Figure 17 *Room with many shadow volumes (Gray dashed lines) where the sphere represents a dynamic object. On second and third row shadow volumes for depth pass respectively depth fail stencil shadow algorithms has been culled.*

Unfortunately this adds another dimension to the previous problem of determining whether to merge the shadow and light rendering of two objects or not. Now we also have the number of shadow volumes that can be culled to take into consideration, if we perform a merge we must of course base the frustum construction on the worst case which would be the nearest point for a depth fail shadow volume implementation and the furthest point object for a depth pass implementation. In an extreme case this could mean that it would be better not to merge the light and shadow rendering of two objects even if one is completely covered by the other due to the number of shadow volumes that cannot be culled after the merge. Take for instance *Figure 17*, if we assume that depth pass is used and the objects in the leftmost and rightmost images exists in the same world it might be favorable not to merge the light and shadow rendering since the object in the leftmost image ensures no shadow volumes can be culled and the object in the rightmost image has a large bounding rectangle.

It can also be noted that the computed largest and smallest distances for the merged objects can be used with the GeForce FX depth bounds testing capabilities to accelerate the shadow volume rendering further. The depth bounds test rapidly rejects pixels if the current value in the depth buffer is not within the defined depth bounds range.

3.3.1.3 Putting it together – Merging heuristics

Computing the optimal merge of objects for the shadowing pass from a fill-rate point of view is a rather complex operation which is unlikely to yield high performance since it must be carried out completely on the CPU and will require lots of frustum culling operations. It’s as important not to underestimate the ability of the graphics card just as well

as not overestimate it. Therefore I used a simple heuristic to perform the merging, namely to merge all dynamic objects affected by the current light. This heuristic doesn't result in minimal overdraw in all cases, but it's fast and easy to implement and also assure no "clearing" needs to be done since there will be no overlapping scissor boxes. This very simple heuristic still works quite well in the case it was meant for, namely when visibility from light sources is limited due to dense static geometry and relatively small light radiuses. The probability that many objects onscreen are affected by the same light source is relatively low and the dimensions of the light will limit the maximum dimensions of the frustum used for shadow volume culling.

3.3.2 A robust depth pass stencil buffer implementation

The reason a depth pass implementation of shadow volumes is desirable is that it is usually less fill-rate intense than the depth fail implementation. It also removes the need for closing the shadow volume mesh and as seen in the previous section it allows much more aggressive frustum culling in our case. The problem lies in making a robust depth pass implementation that doesn't suffer from artifacts when the camera moves in and out of shadow. To do this we must look into how shadow volumes can be extracted and how the depth pass algorithm can be implemented. An unfortunate problem is that the depth pass implementation presented here will not work on ATI hardware due to lack of support for removing the near clipping plane. The alternatives lie in falling back on the depth fail implementation instead or setting the near clipping plane close enough to the view point to make the camera approximately point sized, this have a large negative effect on the depth buffer precision.

3.3.2.1 Shadow volume meshes constructed from the static geometry

There are several different approaches that can be taken to compute shadow volumes for the static geometry to use for run-time purposes. One can for instance use the original geometry to compute a shadow volume mesh for every light in a common fashion, a problem with this is that we won't use the visibility information extracted for every light in the pre-computational phase and may therefore end up with an unnecessary amount of shadow volume polygons which may result in higher fill-rate usage.

Another simple approach is to use the list of visible polygons for a light as a means to compute the shadow volumes. This can easily be done by inserting shadow volume quads in edges shared only by one polygon in the visible polygons list, if a closed shadow volume mesh is desired this can be achieved using the original geometry visible to the light and the same geometry pushed away from the light. If correct winding order is desired the polygons pushed away from the light must have the reversed vertex order of the original geometry. A nice property of the shadow volume meshes generated this way is that it will not self-intersect which makes it possible to use a simpler version of the stencil buffer algorithm where the stencil buffer values are inverted for both front and back facing polygons. The shadow rendering can be simplified a bit since the stencil buffer operations are identical. It doesn't buy anything in fill-rate performance and can be achieved for modern hardware which supports different stencil buffer operations on front and back facing polygons but this shadow volume computation method ensure that no double shadowing occurs.
The differences between the two approaches for shadow volume computation are illustrated in *Figure 18*.

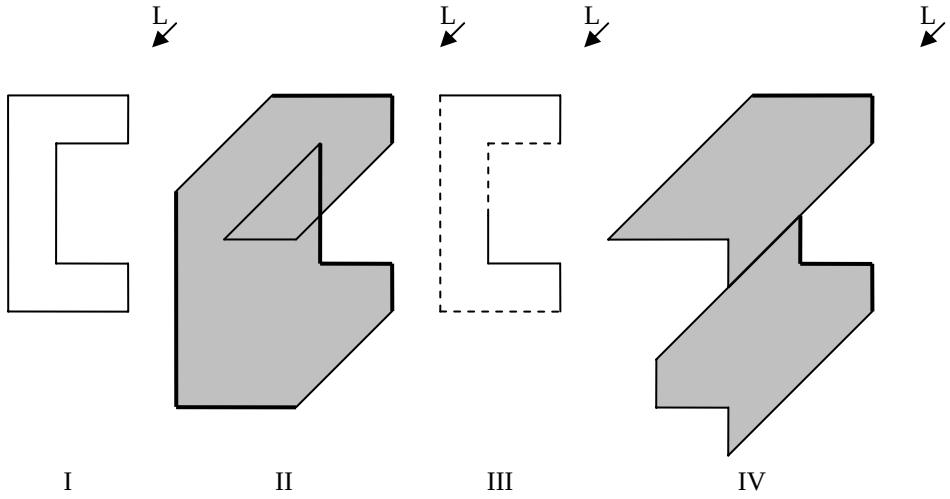

Figure 18 *Two different kinds of shadow volume extraction based on a directional light source. I) Shows the original geometry mesh II) The common shadow volume extraction algorithm based on the geometry in I, the resulting shadow volume mesh is self intersecting. III) Geometry visible to the light source (equivalent to the "light mesh"). IV) Shadow volumes extracted from the mesh in III, this shadow volume mesh consists of several none intersecting segments*

3.3.2.2 Implementation

A robust implementation of the depth pass shadow volume algorithm (Heidmann, 1991) can prove to be a quite challenging task. But with the depth clamping capabilities of some cards it is possible to remove the near clipping plane at the expense of assigning the same depth value for pixels that lie between the near clipping plane and the viewer. Since the scene is still rendered using normal clipping the depth value anomalies doesn't have any effect on the shadow volume rendering.

Assume that we use the method in the previous section which uses the list of lit polygons for a light to compute the shadow volume mesh. In this case the stencil buffer shadow volume algorithm can be performed using bitwise invert operations rather than increasing/decreasing a value. *Figure 5* shows that if the near clipping plane is removed the depth pass algorithm will work unless the viewer is inside a shadow volume mesh, if the viewer is inside a shadow volume mesh the stencil buffer must be initialized to the correct value before the shadow volumes are rendered. With a common self intersecting shadow volume mesh or a shadow volume mesh with double shadowing the correct value can be quite hard to find and conflicts with possible optimizations through removing shadow volume polygons outside the radius of impact for a light. Since the method used for shadow volume extraction assured that no double shadowing or intersection of shadow volume meshes occurred, the problem is much simpler. If the viewer is positioned in a shadow volume mesh the stencil buffer can simply be inverted to get the correct value, the viewpoint is in shadow if a ray from the viewpoint to the light intersects any of the polygons in the mesh of lit polygons.

3.3.3 Summary and alternatives

In the previous sections we've examined a possible approach to optimize the important case where static geometry casts shadow on dynamic geometry. The approach was based on

common strategies for shadow volume optimizations that were refined using the bounding volumes of the dynamic objects to further lower the load on the graphics hardware. The problem whether to merge dynamic objects and treat those as one entity as far as shadow optimizations are concerned plays a major part and is unfortunately also a hardware dependant problem. The best blend lies in a simple heuristic that grossly minimize the fill-rate utilization without putting too much work on the CPU.

It's important to note that the usage of the optimizations described herein does not override the use of common optimizations in the graphics engine. An example of this is to test if an object is inside the radius of a light before performing any lighting or shadowing operations and of course to test if the object is visible or not. Another possibility is to use the occlusion culling functionality that is more or less standard even on older graphic hardware. This occlusion culling doesn't buy you anything in terms of fill rate performance in most cases since the graphics hardware needs to render all pixels of an object to determine if it's visible or not. However, in our case the test can be performed when rendering the ambient or depth buffer setup pass for almost no extra cost at all. If an object is occluded it can be completely discarded for light rendering and the merging process described in this section which gives a fair speedup. Note that the shadow volume of an object can be visible even if the object is not.

There are also other several alternative approaches that can be done but haven't been tested so far. These include

- **Shadow volume construction:** In the pre-computational pass when shadow volume polygons are used to cut the static geometry to produce lit polygons the reverse process can be done as well. Cut the shadow volume polygons with the world geometry and remove all polygons that are not visible from the light source using ray tracing. The result is a shadow volume mesh with higher polygon and vertex count but it ensures that the shadow volumes polygons have minimal area. Closing the shadow volume mesh would be a definite problem but it would not be necessary if the depth pass implementation is used.
- **Other heuristics:** More complicated heuristics could be used. One could for instance assume that the shadow volume polygons are approximately uniformly distributed. In a first step objects with overlapping bounding rectangles would be merged to make sure no "clearing" issues needs to be dealt with. Secondly further merging could be done based on some heuristic like: frustum volume after merge / frustum volumes before merge < constant. A problem is that the constant would be machine dependant since more frustums means more culling and therefore a higher CPU utilization while the volume of the frustums affect the utilization of the graphics hardware.
- **Shadow maps:** It's possible to make a different approach altogether and use shadow maps instead of shadow volumes for the case where static geometry casts shadow on dynamic geometry. If the target hardware is good enough to handle things such as cube shadow maps this is definitely something that should be considered. Shadow maps suffers from more aliasing problems than stencil buffer based shadows and the static shadow algorithm described in here, but may still be enough for the dynamic objects. Alternatively the algorithm presented in (Sen 2003) could be used to give more satisfactory results. The strong point with using shadow maps is that they can be pre-computed and stored which means that on run-time they only need to be projected on the dynamic objects and used to render the lighting. This only gives a very slight overhead to the shading and doesn't introduce any of the problems in the other approach based on shadow volumes, the price to pay is higher memory overhead and possible aliasing problems. Using shadow maps to solve this case could be combined with either using shadow maps for everything but the pre-computed static shadows or using shadow maps

exclusively for the static geometry casting shadow on dynamic geometry case while using stencil buffer shadows for the other cases.

3.4 Fake soft shadows

The algorithm so far can work in combination with stencil shadow volumes and produce the same type of hard shadows as the shadow volume algorithm. In real life light-sources are never a single point entity but rather a volume of glowing matter, since it's a volume it's possible that a point is partially visible to the light source which gives shadows a smooth penumbra where the light fades out. *Figure 19* illustrates that even with a rough approximation of the penumbra soft shadows are more realistic than its counterpart.

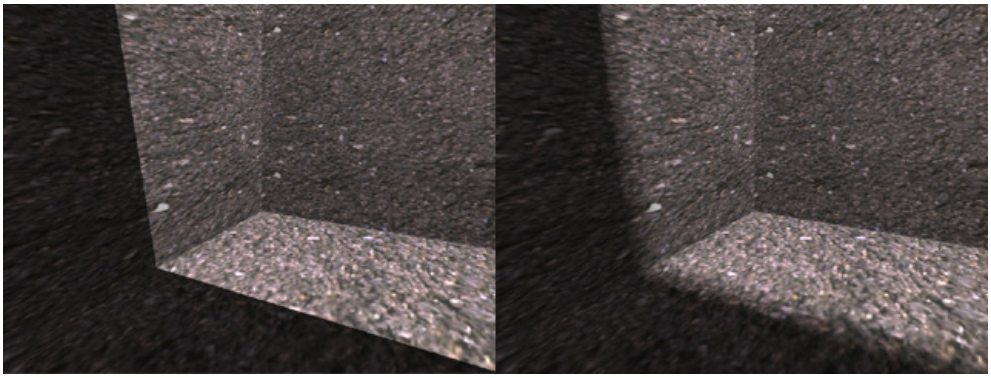

Figure 19 *Hard vs. fake soft shadows*

The title of this section quite nicely sums up what we're trying to do. Rather than a physically correct representation of soft shadows the goal is to come up with something that's visually credible yet simple enough to run with good performance. For the performance we need to consider the extra amount of polygons that the cutting phase will add and the accuracy with which we will compute the visibility approximate for every pixel in the penumbra regions on run time. The goal is to perform the same kind of cutting of the world geometry as the previous algorithm but this time resulting polygons should be either lit, shadowed or in a penumbra region. This can be done using a more complex mesh than the shadow volume mesh for cutting. This mesh would consist of penumbra wedges (T. Akenine-Möller, 2002), a geometrical primitive which completely encloses the volume of a penumbra caused by a silhouette edge. The wedge is essentially built from two quads which are constructed from the silhouette edge and the light's max/min visibility positions and two triangles that tie both quads together, closing the wedge. All points contained in the penumbra wedge is considered to lie in the shadow region, since this algorithm operates on polygons rather than a per pixel level it's desirable that the penumbra wedges fit together to remove unnecessary polygon cutting. *Figure 20* shows a penumbra wedge constructed using a volumetric light source and a silhouette edge.

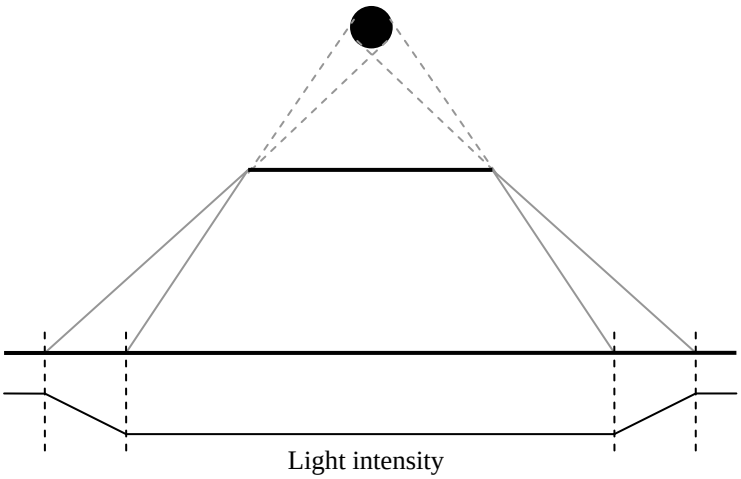

Figure 20 *Penumbra wedges. Inside the volume of a penumbra wedge all points are considered to lie in the shadow penumbra. The lighting intensity of a pixel in the wedge is proportional to how much of the light is visible from that point. In this project this was approximated using linear interpolation, the intensity function is shown below the figure.*

The algorithm I use for penumbra wedge construction is very simple and also probably a rough approximation but on my test cases it looks relatively good and overall cause a rather low amount of polygon splits. First of all, silhouette edges are still computed from a light's position which is a single point rather than a volume. Given the light radius (the radius of actual glowing matter) and a vertex on a silhouette edge we wish to compute points in the light's volume that represent minimal/maximal visibility and use these positions for pushing back that vertex when creating the quads for wedges. The points of minimal/maximal visibility can be approximated using the pseudo normal of a vertex. The max position is roughly in the direction of the pseudo normal of a vertex while the min position is in the opposite direction. Note that using this method, a quad constructed by a silhouette edge and the vertices pushed back for min or max visibility position, may not be planar. In the test scenes used this was not a large issue and assuming planar quads worked very well, but if a problem would arise this could be worked around simply using two triangles rather than a single quad although it would result in more polygon splits.

Once the wedges have been constructed the world geometry should be cut using them. From the resulting mesh, polygons completely in shadow are removed. Since the light is now a sphere rather than a point, ray-tracing is now less efficient. It is a good idea to maintain a mesh based on the polygons in the wedge mesh representing maximum visibility and use it to do an inside outside test with the crossings test. The mesh must be closed in order to make the test work.

For later use, light-source visibility estimates are computed for every vertex. I computed these estimates using the penumbra wedges by finding the wedge containing a vertex and using linear interpolation based on the distances to the polygons representing max/min visibility from the light. If a point lies within more than one wedge the smallest visibility estimate is chosen. All vertices not contained in a wedge are set to full visibility.

Quadratic interpolation would intuitively seem more accurate than linear since the intensity in a point is proportional to the area of the light visible from the point. However my tests did not show any significant improvement in visual quality when using quadratic interpolation, linear interpolation was chosen since it's cheaper than quadratic.

In practice it may be convenient to combine pre-computed static soft shadows with hard shadows on dynamic objects. Although the difference in shading may clash there are currently plenty of performance related reasons for doing so. If this is the case it might be good to compute wedges based on the minimum visibility and the point light visibility rather than the min and max visibilities. Even though this can cause recognizable anomalies in some cases it will ensure that the dynamic shadows “fits” in nicely in the scene. For instance imagine a common doorway. If the door is considered a dynamic object while the wall and doorframe is static, a mix of soft and hard shadows can make a closed door “leak” light. However if the wedges are constructed as previously mentioned, the maximum visibility will be equal to the shadow volume mesh of the dynamic object which will because the shadows to fit together. This is how my implementation of fake soft shadows worked. The shadows generated are far from physically correct but in most cases when textures is added the result looks quite realistic, especially when compared to hard shadows. In *Figure 21* fake soft shadows are compared soft shadows, using the mean of ray tracing to random points on the light, rendered using popular model and render software. Due to the implementation the penumbras on the fake shadows will be half the size of the other penumbras. Note that this was to allow hard and soft shadows to blend well together, if an engine is targeted for soft shadows only it should be possible to use full penumbras. The shading is also slightly different in the two images, it’s probably related to the attenuation and attenuation approximations used in the pixel shaders.

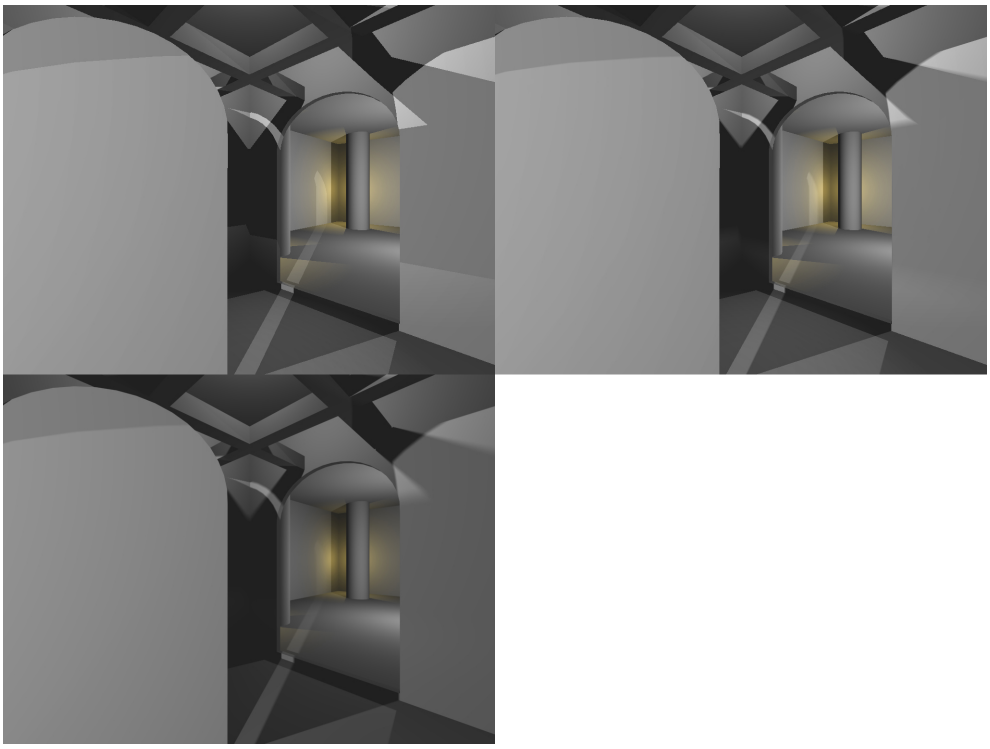

Figure 21 *Comparison between the fake soft shadows and real soft shadows rendered with a popular modeling and rendering program. Top left: hard shadows. Top right: Fake soft shadows. Bottom left: Accurate soft shadows*

4 Evaluation

It's hard to perform benchmarks comparing the method described in this paper, but for testing purposes I also implemented an optimized implementation of dynamic shadows. Furthermore I used a high resolution and low polygon meshes for benchmarking to try to make up for the performance loss introduced by real time shadow volume generation in the dynamic algorithm that could have been eliminated through using a shadow cache.

For benchmarking, three different setups were tested: No shadows, dynamic volume shadows and finally an implementation of the algorithm described in this paper. This implementation features fake soft shadows for the static shadows and (only) hard, per pixel shadows for dynamic objects. The benchmarking was performed on an AMD XP 2000+ (1666 MHz) with 512 Mb SDRAM and a GeForce FX 5200 (Mainstream/Low end card). Screenshots of the different test images can be found in *Figure 22*.

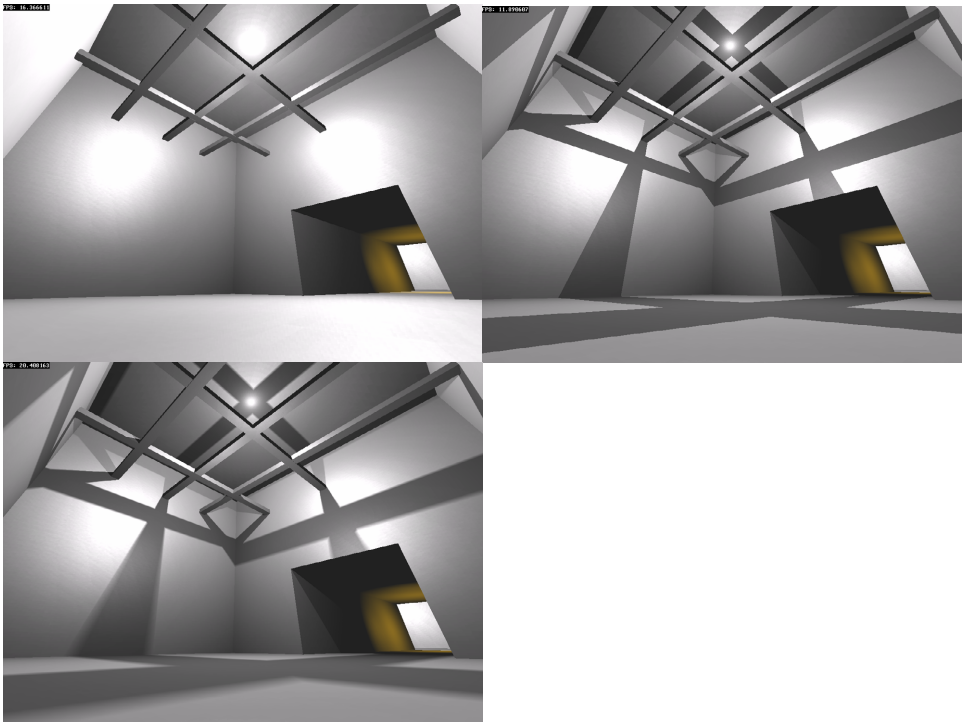

Figure 22 *Screen shot of the test scene with 4 lights (textures/bump maps are not displayed). The pictures show the test scene with no shadows, hard dynamic shadows and soft static shadows.*

I also decided to perform some tests using a pixel shader which utilizes the flexibility of programming languages for modern hardware (often known as ps2.0). This allows the shading to be collapsed into a single pass and therefore eliminates repetitive stencil buffer testing for the same pixel favoring the dynamic implementation somewhat. Noticeable is that these shaders performed very poorly next to the old shaders, they were outperformed by over 100% but I believe this difference is lower with high end hardware.

The tests were performed on a simple scene with a varying number of dynamic objects and static lights. This was to try to measure the performance drop of the static method as the

number of dynamic objects in a scene increase. Note that no lights were dynamic in the tests since these cannot be optimized and must be implemented using the common algorithm for dynamic lighting and shadows at any rate.

The results, shown in *Table 2*, indicate that the algorithm presented outperforms not only a common implementation of dynamic shadows but also an implementation with per pixel lighting only. The reason for this is that all shadowed areas that can be swiftly skipped save more hardware resources than the rendering of the necessary shadow volumes. The results also show lower performance gains with DirectX ps2.0 or equivalent shaders although the difference is only noticeable in the test with one light and two dynamic objects

Benchmark settings	Only lighting (FPS)	Dynamic shadows (FPS)	Optimized Static shadows (FPS)
1 Light, 2 dynamic objects	30.2	21.6	32.2
4 Lights, 2 dynamic objects	14.0	10.1	16.1
1 Light, 0 dynamic objects	34.4	25.4	39.2
4 Lights, 0 dynamic objects	16.1	11.9	20.2
1 Light, 2 dynamic objects ps2	13.3	11.4	14.8
4 Light, 2 dynamic objects ps2	5.5	5.0	7.3
Avg. relative performance	1.29	1.0	1.51

Table 2 *Performance measurements for the different algorithms implemented. The performance is measured in FPS (Frames per second)*

All in all, the performance gain is almost constant around 50% although it may differ based on the scene geometry. The strengths of static shadows lie specifically in scenes with shadow details that will rarely affect dynamic objects. The shadow pre calculation phase is also an excellent opportunity to produce a potentially visible set for every light which can further help the culling of shadow volumes, this can further help to cull objects that is within the falloff radius of a light but completely in shadow.
It may seem odd that the static shadows often outperform the test run with lighting only. This is because the lighting equations are very heavy on the per-pixel resources of a card. The test using static shadows doesn't need to perform the lighting computations in shadowed areas for a specific light. This is most apparent in the lower right corner of the pictures in *Figure 22*, in the lighting only picture notice the bright specular spot caused by the light in the distant room. The gain in performance due to static shadowing was larger than the loss from rendering dynamic shadows for the dynamic objects in the scene.

Another aspect that may lower performance gain is the high end cards that have currently appeared on the market, specifically GeForce FX 5900 and Radeon 9700/9800 models. These cards are more or less optimized to maximize shadow volume performance being able to push vast amounts of colored or single textured fragments per second. I still believe that the optimized algorithm will outperform the common algorithm with a significant factor. Especially when used in combination with more complex soft shadow methods (U. Andersson, 2003), where even more opportunities for optimization may arise. The technique used for these soft shadows are more complex than the simple method used in this paper, it breaks down to the following steps:

1. Render a pass of the geometry where every fragments position in world-space is written to a floating point texture
2. Render a pass with the common shadow volume algorithm and initialize shadowed fragments to 0 and lit to 1. This pass serves as a crude approximation and initializes shadow fragments to "black" but it also causes a problem since some fragments in the penumbra wedge will be initialized to 0 and others to 1. One can say that the shadow volume polygon divides the penumbra wedge in two halves

3. Create shadow penumbra wedges that completely enclose the penumbra of the shadow. A good property with this algorithm is that they may overlap/intersect freely.
4. Render penumbra wedges with a rather complex fragment shader that computes the area of the light source that is "covered" by the silhouette edge used to generate the wedge. This fragment shader is based on geometrical properties of the light, for instance for a spherical light a cone based on the fragment's world-space position and the light sphere is used in combination with the edge to compute a coverage factor. What are computed in practice are the coverage factor for one of the halves of the penumbra wedge and the visibility factor for the other half. These are stored in two textures and then subtracted respectively added to the texture that was generated during the common shadow volume pass.
5. Render lighting with the visibility estimate texture generated in the last step.

I believe that using the technique described herein to cut the scene with penumbra wedges generated and then through some technique remove all polygons completely in shadow would give great possibilities for optimization of the algorithm described above.

If only looking at static geometry and shadows one could remove the need for subtractive/additive contribution since the shadowed polygons are already removed. The world-space position texture map would not be needed for static shadows since polygons in the penumbra could be sent rather than the polygons enclosing the penumbra. This also has the effect that the world-space position of a fragment can be linearly interpolated over the polygon rather than looked up from a texture map which may very well make it possible to move out a large part of the complex fragment shader to a vertex shader and interpolate it linearly. This part of the fragment shader is devoted to creating a translation matrix to translate the penumbra causing edge's coordinates into a local light-position coordinate system. If the world-space position can be interpolated linearly it sounds reasonable that the edge could be translated in a vertex program and then using linear interpolation to get the edge's coordinates to the fragment program.

Even though the soft shadow method described in this paper is a rather simple approximation by comparison there are still plenty of reasons to use it at least for the static geometry. First of all it's inferiority in quality is not as easy to notice as one might think. When diffuse-, bump- and gloss-maps are mixed in a lighting pass there's enough noise that it's not easy to notice the lower quality. Also it requires no advanced programming features what so ever to work since the only thing added is a linear interpolation of colors. Still, it should be used with care because when the radius of lights increase, the errors introduced by interpolation and approximation are bound to be more and more obvious.

5 Related work

Vlachos et al (2003) presents a pre-computational algorithm for shadow optimizations called composite shadows. This presentation is based on the same original concept as the pre computational algorithm in this thesis report where static shadows are optimized through a pre-computational cutting phase, although they have been developed independently. The article is based on one of the technology demos used to promote ATI's Radeon 9700 PRO series graphic cards.

The composite shadows article also suggests a pre-computational pass to cut static shadows out of the geometry thereby creating lists of polygons either completely lit or completely in shadow. However, this approach is much more aggressively optimized and contains more approximations. First the static lighting is rendered completely, after this the shadow caused by dynamic geometry is removed from the image to remove lighting in areas shadowed by dynamic geometry. To avoid removing too much light from areas where the angle of impact or distance from the light affects the lighting intensity, a darkening value is computed for every pixel when the lighting is rendered. If the pixel is multiplied by the darkening value only the ambient lighting term should be left. The shadow volumes of dynamic objects are then rendered to set up the stencil buffer and for every pixel that lie in shadow, the current lighting intensity is multiplied by the previously computed darkening value. This will result in incorrect shadowing where lights overlap since the lighting will always be forced to the ambient level in shadowed areas even though the point may be shadowed from one lights point of view but lit by all the other lights. The presentation also totally ignores shadows on dynamic geometry cast by static geometry. In all the test cases for this algorithm this would have a severe impact on the realism of the rendered images. It is also the single most important performance limiting factor of our approach.

All in all, the approximations are fair for demo style scenes with pre-determined animation paths, small dynamic objects that don't need to receive shadows from static geometry and light positions that are strategically positioned to not show the possible artifacts. For a game engine more freedom is generally desired and although scenes as the one shown in *Figure 23* can be designed without revealing any of the weaknesses of the algorithm there's bound to be cases it shows in an interactive program.

Figure 23 *This figure is a screenshot from a tech demo by ATI showing a similar optimization for static shadows. All dynamic objects are kept at a small size, traveling in paths where they won't enter shadow cast by static geometry.*

Chin et al (1989) presents optimizations for shadow volumes using BSP Trees and cutting all geometry by shadow volumes to gradually remove resulting polygons that lie in shadow. Unlike traditional implementations, using shadow volumes on a per pixel basis in combination with a depth buffer, this algorithm operates on the original polygonal geometry using a BSP tree to speed up polygon clipping and shadow determinations.

Even though the article itself is outdated due to the high performance of graphics hardware and the possibility to implement shadow volumes on a per pixel basis using the stencil buffer, the concept is the same as in the pre-computation step that creates light meshes. In fact, this algorithm could be used in the pre-computational phase to compute the lists of completely lit polygons. It's probably very fast but has the drawback of producing several extra polygons since it's working on planes rather than polygons. This will cause slower rendering overall at runtime and of course a larger memory footprint.

Everitt et al (2003) and Kilgard et al (2003) discuss general optimization strategies for shadow volumes on modern graphics hardware. This includes scissor boxes to limit fill-rate usage, the depth range extension of GeForce GX 5900, view frustum clipping, shadow volume generation and similar. These optimizations play a central role of the algorithm presented in this report.

In the static geometry casting shadows on dynamic geometry case of the algorithm, much more spatial information than usual is available. The shadows only need to be correct on the pixels occupied by dynamic objects which mean that the objects screen-space bounding rectangles and placement can be taken into consideration both for the scissor box computations and the usage of the depth range extensions. Furthermore the view frustums to use for shadow culling can be optimized to only contain the dynamic objects rather than the whole visible volume.

Akenine-Möller et al (2002) and Assarsson et al (2003) presents two algorithms for rendering soft shadows. The approach used resembles most to the first of the two reports as a simple model based on linear interpolation is used rather than the more computationally intense but better approximation in the second paper.

A base concept taken from these reports is the penumbra wedge primitive that is used to enclose the volume that lies in the shadow penumbra. The penumbra construction itself differs quite much from the method presented in the papers. These methods are originally meant for per pixel computations whereas my method operates on polygon geometry. The method presented in the second paper only ensures that the penumbra region lies somewhere inside the penumbra wedge, it produce lots of overlapping wedges which will result in many splits and higher polygon counts. Since the needs are different for a geometry based algorithm and a per pixel algorithm I chose to use a simplified version of penumbra wedge computation which is aimed more at getting a penumbra region than making sure it's physically correct.

Kilgard et al (2001) presents some information about stencil buffer based shadow volumes using the invert operation rather than the increment and decrement operations. The algorithm based on bitwise inversion of the stencil buffer values only work for none overlapping and intersecting shadow volume meshes, however we can create these types of shadow volume meshes using the pre-computed lists of lit polygons. This simplifies the case of static geometry casting shadow on dynamic geometry and the implementation of a depth pass shadow volume algorithm.

6 Conclusion

To summarize, let's look at the advantages and disadvantages of the optimized algorithm for static lighting and shadowing.

Advantages

- Less fragments processed when rendering light
- Less fragments processed when rendering shadow volumes
- Static fake soft shadows for virtually no extra cost on a per-pixel basis
- Fake soft shadows helps to hide polygonal appearance of shadows
- The geometry throughput may be lower if more polygons lie in shadow than the amount of new polygons created through cutting geometry by shadow volumes.
- Potentially visible set helps to further lower fill rate usage (points 1,2)

Disadvantages

- Geometry throughput will in most cases be higher due to the polygons created when cutting the geometry by the shadow volumes. Especially for soft shadows due to the more complex penumbra wedges used for cutting
- CPU utilization may be higher due to the process of culling shadow volumes.
- Higher memory usage. Both client side (CPU) and server side (GFX Hardware). One light mesh per light or at least the extra vertices and faces introduced through clipping must be stored.

It's possible to view this technique as a way of trading a lower fill rate for a higher geometry throughput. Especially when using soft shadows the extra amount of splits will cause many new polygons but it's also at possible that the amount of new polygons inserted when clipping is lower than the amount of polygons not visible to the light but in the lights radius of influence. In this case the technique will lead to a lowered geometry complexity as well. In the demo scene used, the number of triangles sent to the graphics hardware was indeed lower when using static fake soft shadows than the traditional shadow volume algorithm. This was mainly due to many shadow volume polygons that could be saved using culling and non-capped shadow volume meshes.

In most engines static geometry is already of a considerably lower tessellation than dynamic meshes. The reason for this is that physics and collision detecting must often be performed on a per-polygon basis rather than on a bounding volume basis as can be done for dynamic meshes. In this case the increased geometry throughput is usually insignificant since the added geometry need not be used for physics or collision purposes but rendering only.

The increased CPU utilization of the technique is rather low. The extra power is used in the phase where dynamic objects are merged and shadow volumes are culled based on a frustum enclosing the dynamic objects. This process can be optimized using a suitable tree when performing shadow volume culling. Shadow volume meshes based on silhouette edges only (no capping geometry) also tend to have a lower complexity than the original mesh.

Last but not least is the memory issue. There's indeed a higher memory footprint than using a completely dynamic shadow volume algorithm. This is generally the case of structural optimizations, that performance is traded for memory (additional computed information). There are several optimizations that can be done to reduce the memory footprint and when compared to using the popular shadow and lighting cache optimization the difference shouldn't really be big.

In my opinion the disadvantages are a fair price to pay. Even though the evaluation was only performed on one single type of hardware setup and a rather limited set of different situations, the performance gain was apparent in all the tests performed. The theoretical minimal performance of the algorithm is almost equal to the dynamic shadow volume algorithm except for the extra CPU utilization from processing information about the dynamic objects. Unless the number of dynamic objects is very high this should not be an issue.

Appendix A: Rendered Images

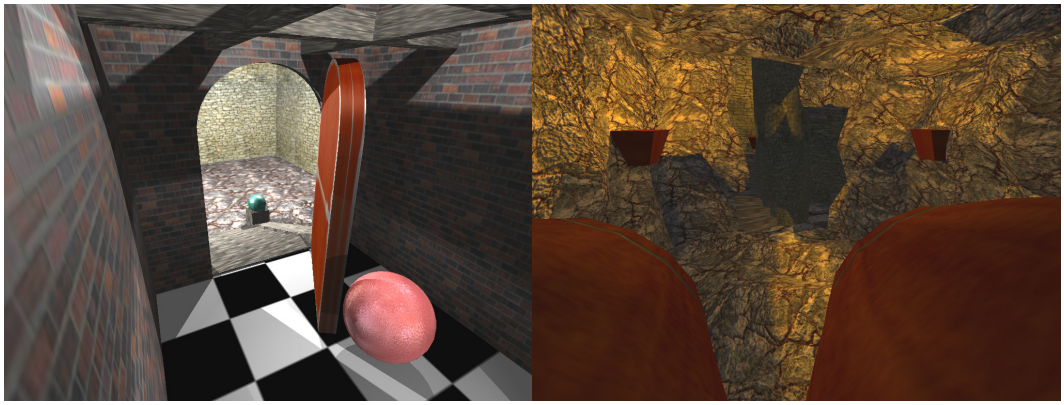

Figure 24 *Screenshot from a test and demo scene with 14 Lights and about 10000 polygons before shadow clipping*

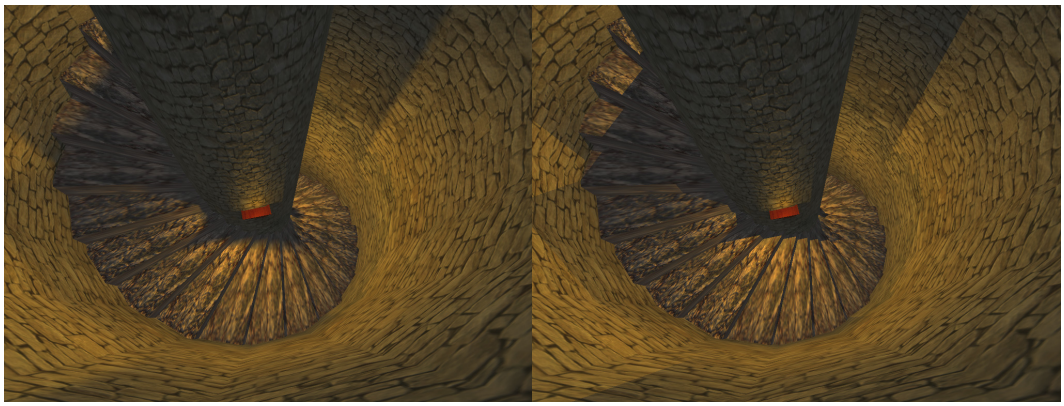

Figure 25 *Two screenshots of the same scene, they are taken with respectively without fake soft shadows enabled.*

References

T. Akenine-Möller and U. Assarsson. Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. *Eurographics Workshop on Rendering 2002.*

Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller, An Optimized Soft Shadow Volume Algorithm with Real-Time Performance, to *Graphics Hardware 2003*, pp. 33-40, p. 131, July 2003.

P. Beaudoin and J.Guardado. A Non-Integer Power Function on the Pixel Shader, *http://www.gamasutra.com/features/20020801/beaudoin_01.htm*, 2002

P. Bergeron. Shadow volumes for non-planar polygons, *Proceedings of the Conference on Graphics Interface* (May 1985), 417–418.

P. Bergeron. A general version of Crow's shadow volumes. *IEEE Comput. Graph. Appl. 6*, 9 (Sept 1986), 17–28.

B. Bilodeau and M. Songy, unpublished slides, *Creative Labs sponsored game developer conference*, Los Angeles, May 1999.

J. Blinn. Simulation of Wrinkled Surfaces, *Proceedings of SIGGRAPH 1978*, August 1978, pp. 286-292.

N.Chin, S.Feiner, Near Real-Time Shadow Volume Generation Using BSP Trees, Computer graphics, Volume 23, Number 3, July 1989.

J. Carmack. Unpublished correspondence, 2000

F. Crow. Shadow Algorithms for Computer Graphics, *Proceedings of SIGGRAPH*, 1977, pp. 242-248.

S. Dietrich. Per pixel lighting, Published online at developer.nvidia.com, September 2001.

C. Everitt, M. Kilgard. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering, Published on-line at developer.nvidia.com, March 2002.

C. Everitt, M. Kilgard. Optimized Stencil Shadow Volumes. *GDC 2003 presentation*. Published online at http://developer.nvidia.com/docs/IO/8230/GDC2003_ShadowVolumes.pdf, 2003

R. Fernando, S. Fernandez, K. Bala and D.P. Greenberg. Adaptive shadow maps. *Proceedings of SIGGRAPH 2001*, pp. 387-390, 2001

E. Haines. Point in polygon strategies, *Graphic gems IV*, ed. Paul Heckbert, Academic Press, p.24-46, 1994. Published online at http://www.acm.org/pubs/tog/editors/erich/ptinpoly

T. Heidmann. Real Shadows Real Time, *IRIS Universe*, Number 18, 1991, pp. 28-31.

M. Kilgard, C. Everitt, M. McGuire, J. Hughes and K. Egan. Fast, Practical and Robust Shadows, Published online at developer.nvidia.com, November 2003

References

M. Kilgard. Robust Stencil Shadow Volumes, *Cesa Developers Conference 2001 presentation*, published online at developer.nvidia.com, 2001.

M. Kilgard. A Practical and Robust Bump-mapping Technique for Today's GPUs. *Technical report, NVIDIA corporation*, February 2000. Available at www.nvidia.com.

E. Lengyel." The Mechanics of Robust Stencil Shadows." *Gamasutra, http://www.gamasutra.com/features/20021011/lengyel_01.htm*, 2003

Tomas Möller and Ben Trumbore, Fast, Minimum Storage Ray-Triangle Intersection. *journal of graphics tools*, vol. 2, no. 1, pp. 21-28, 1997

W. Policarpo. 3D Games Real-time Rendering and Software Technology, Volume one. *Book, First Edition* 2001.

P. Sen, M. Cammarano, P. Hanrahan. Shadow silhouette maps. *Proceedings of SIGGRAPH 2003*. Published online at http://graphics.stanford.edu/papers/silmap/silmap.pdf, 2003

M. Stamminger, G. Drettakis. Perspective Shadow Maps. *Proceedings of ACM SIGGRAPH 2002*, July 2002

A.Vlachos, G.James. Special effects with DirectX 9 - Composite shadows, *GDC 2003 presentation*. Published online at www.ati.com/developer, 2003

L. Williams. Casting Curved Shadows on Curved Surfaces. *Computer Graphics (Proceedings of SIGGRAPH 78)*, 12(3):270–274, August 1978. R. L. Phillips, editor.