Real-time shadows in 3D computer graphics

Hans E. Molin

Information Technology
Computing Science Department
Uppsala University
Box 337
S-751 05 Uppsala
Sweden
Abstract

This master’s thesis is a survey of commonly used methods to achieve shadows in 3D computer graphics. By evaluating the methods with regards to their speed, robustness and suitability to be integrated into a scene graph rendering toolkit which initially lacks shadow producing features, in this case the VRT (Virtual Reality Toolkit), the shadow mapping approach was chosen. This paper then describes the implementation of the shadow mapping approach into VRT.
Preface

This paper is the report of my master’s thesis project on the master of science of computer science program (Datavetenskapliga programmet) at Uppsala University, made for the Department of Information Technology, Division of Human-Computer Interaction.

The first part of this report consists of a survey of the most commonly used algorithms and an evaluation of their suitability for implementation in an existing 3D scene-rendering toolkit, in this case VRT (Virtual Reality Toolkit).

The second part of the report will describe the implementation itself. The choice of algorithm will be based on the earlier survey.

I would like to thank my family for their support, my supervisor professor Stefan Seipel for the opportunity to work with the thing I enjoy the most, 3D computer graphics, Mikael Erlandsson for assistance with the report, Lars Pettersson for various reflections on the subject, Daniel Wesslén for finding that elusive active texture unit bug and last but not least יהוה for everything.
## Contents

1 Introduction ............................................................................................................... 1  
1.1 Objective .............................................................................................................. 3  

2 Method ....................................................................................................................... 4  
2.1 Material ................................................................................................................ 5  

3 Survey of existing algorithms ................................................................................... 6  
3.1 Projected planar shadows ..................................................................................... 6  
   3.1.1 Advantages .................................................................................................... 8  
   3.1.2 Disadvantages ............................................................................................... 8  
3.2 Area subdivision .................................................................................................. 8  
   3.2.1 Advantages .................................................................................................... 9  
   3.2.2 Disadvantages ............................................................................................. 9  
3.3 Shadow volumes ................................................................................................10  
   3.3.1 Advantages .................................................................................................. 13  
   3.3.2 Disadvantages ............................................................................................. 13  
3.4 Shadows on curved surfaces .............................................................................. 13  
   3.4.1 Advantages .................................................................................................. 14  
   3.4.2 Disadvantages ............................................................................................. 14  
3.5 Shadow mapping ................................................................................................14  
   3.5.1 Advantages .................................................................................................. 15  
   3.5.2 Disadvantages ............................................................................................. 16  
3.6 Object-ID/Priority buffers .................................................................................. 16  
   3.6.1 Advantages .................................................................................................. 17  
   3.6.2 Disadvantages ............................................................................................. 17  

4 Design Issues............................................................................................................ 18  

5 Implementation ....................................................................................................... 20  
5.1 Types and structures .......................................................................................... 20  
5.2 Initialization ....................................................................................................... 21  
5.3 Shadowing parameter adjustment ...................................................................... 21  
5.4 Depth map generation ....................................................................................... 23  
5.5 Turning off shadowing ....................................................................................... 24  

6 Benchmark application .......................................................................................... 25  
6.1 Frame rate comparison ....................................................................................... 25  
6.2 Visual comparison ............................................................................................. 26  

7 Results ...................................................................................................................... 27  
7.1 Frame rates ......................................................................................................... 27  
7.2 Artifacts .............................................................................................................. 29  
7.3 Example VRT program ...................................................................................... 39  

8 Conclusion ............................................................................................................... 41  

9 Future work ............................................................................................................. 42  

Real-time shadows in 3D computer graphics
Contents

9.1 State change control ................................................................. 42
9.2 Eye space coordinate texture generation .................................. 42
9.3 Direct to texture render ......................................................... 42
9.4 Partial scene depth mapping .................................................. 43

References .................................................................................. 44
1 Introduction

“A shadow – a region of relative darkness within an illuminated region – occurs when an object totally or partially occludes the light.”[17]

While 3D computer graphics has always strived to become as realistic as possible, one aspect of the real world has often been omitted due to its computationally high cost or complexity, shadows. Even though complete and correct algorithms for shadows has existed for more than 25 years[1][16], developers have often chosen to implement a less costly but also a less accurate and less realistic alternative[17] or in some cases, no shadows at all.

Shadows are a very important spatial cue (Figure 1, Figure 2, Figure 3) and provide an increased sense of reality in a rendered scene. They help to determine the relative position of objects, particularly depth order and the height of objects above the ground plane. Since a shadow is basically a projection of the scene from an alternative viewpoint, cast shadows can also clarify the shape of an object and the position of a light source[7]. Most 3D computer graphics applications can benefit from using shadows, but since most real-time 3D computer graphics applications today use the projected synthetic-camera model approach, shadows are not unlike in ray-tracing generated automatically and have to be calculated separately. This separate calculation can generally be divided into two different approaches, analytical and image-based[5]. The analytical approach uses the knowledge of the representation of the geometrical data to calculate which parts of the rendered scene are shaded. The image-based approach renders the scene multiple times; storing the data projected unto the projection plane and using that data to determine which parts lie in shadow. Both approaches suffer from the additional computation on top of the computation done for rendering the scene, the analytical approach by adding or transforming polygons and determining the shaded areas, the image-based by rendering the scene multiple times. This computational overhead will lead to a lower frame-rate, but with today’s hardware, frame-rate is rarely a problem when rendering, it’s more of a limit of how many features you can implement and how detailed the rendered scene may be.

Generating shadows is a classical computer graphics problem and considerable research effort has been devoted to it[1][17]. In this report I will focus on hard-edged shadows cast by point sources. Soft shadows can, given a fast hard-edged shadow algorithm, be approximated by summarizing several point light sources. This is however still computationally too heavy to realistically be considered as an option.
1 Introduction

Figure 1: Without shadows one cannot precisely determine where the textured cube is located. It looks like it is right next to the left cube.

Figure 2: With shadows one can easily determine that the textured cube is located much closer to the observer than the left cube.

Figure 3: The same scene from a different angle and different light position.
1 Introduction

1.1 Objective

The original objective of this master’s thesis project was to do a survey of a number of shadow algorithms, trying to locate the major bottlenecks and devise an enhancement to these specific parts, resulting in an increased efficiency. It soon became evident that this goal was out of reach for a project that is supposed to last a total of six month, considering the more than 25 years of research and monumental amount of effort that has been devoted to research of various shadow algorithms. Considering plausible alternatives with the fact that most of the survey had been completed, a new objective was devised. The 3D scene graph library developed and used at the Division of Human-Computer Interaction, VRT (Virtual Reality Toolkit) lacks any internal shadow generation. Since the toolkit is designed to let the application programmer develop 3D applications without explicit calls to OpenGL or having to keep track of the specific representation of the geometrical data, generating shadows with VRT would be intricate to say the least. The new objective was to integrate an automatic shadow generation feature using the method best suited for this purpose based on the results of the survey.
2 Method

The process of researching and evaluating shadow algorithms naturally begins with the examination of previous work and research. In the more than 25 years of shadow generation research six algorithms can be considered the most common and widely used ones\[5\][17], not including ray-tracing: Projected planar shadows, Area subdivision, Shadow volumes, Shadows on curved surfaces, Object-ID/Priority buffers and Shadow mapping. These methods vary considerably in computational load and level of reality.

When evaluating these different shadow algorithms, the most obvious criterion is a low computational overhead, making it suitable for real-time rendering. But perhaps the most important requirement is how real or how honest[17] the algorithm is. The requirements that have to be fulfilled by an algorithm that produces “true” shadows are:

- The ability to cast shadows on other objects, not just a flat shadow receiving plane, like a floor or wall.
- Self-shadowing, an object should be able to cast a shadow onto itself. For example the arm should be able to cast a shadow onto the body.

Other important aspects to consider are:

- Omni-directional shadow casting, to be able to cast shadows in all directions, not just in a spotlight frustum[6].
- Window-space shadow determination. Accurate shadow determination, down to pixel level or lower if multisampling.
- Does not require modifications on the scene.
- Works on all scenes and all kinds of objects no matter what primitive (quads or triangles) the object is constructed by.

Unfortunately, no algorithm fulfills all of these criteria. To achieve a testing platform as broad as possible I chose two algorithms which fulfills the requirements of true shadows and as much of the remaining requirements as possible and at the same time being each others complement in respect to the requirements. These two are also among the most researched and optimized available: Shadow volumes and Shadow mapping.

To evaluate the performance of these two algorithms I had to implement them or modify an existing version. Very early in the evaluation the favor leaned towards shadow mapping due to a number of factors more thoroughly explained in the results part. The implementation of shadow mapping into VRT initially estimated as a smaller part of the project turned out to be the a lot more time consuming, due to the following:

In VRT a scene is constructed using a hierarchical tree structure, where every node has a model-view matrix that along with its parents’ matrices represent the transformations of the object connected to that node. For this to work VRT handles all drawing routines internally. Fitting shadows into this architecture meant modifying the basic drawing functions. Since shadow mapping works by mapping a shadow texture to every object in the scene and generating that texture by comparing positions
in a depth buffer generated from the shadow caster’s position and generating the texture coordinates automatically one has to modify the texture-matrix every time the model-view matrix is modified. The first step was than to locate all modifications to the model-view matrix and apply the same changes to the texture matrix. But I had at first not anticipated for the textures already applied to objects, this constituted a problem since the shadows itself are applied to all objects in the scene as a texture. This meant that I had to modify the algorithm to use multi-texturing, the first texture unit for the original textures and a second texture unit as the shadow texture. To test this I implemented it in a separate test program and it worked with some tricks. The same method did however not work as well with VRT as it had done with the test program. The reason was that I due to very sparse documentation on multi-texturing had assumed that if one uses auto generated texture coordinates that very texture unit had to be active when rendering the object for which the coordinates were generated. In hindsight it might seem a bit strange to assume, but my test program had worked using that approach. Eventually with some help I learned that this is not the case and managed to get shadow mapping with multi-texturing to work in VRT.

With the basics working I decided to improve the multi-texture blending which at this point used ordinary GL_MODULATE, which multiplies the values in the shadow texture which consists of texels with the value 1 or 0 with the result of the previous texture unit blending. The result of this is that illuminated areas are multiplied with one i.e. no change, while shaded areas are multiplied with zero, resulting in completely dark areas. The values in the shadow texture can be modified if the ARB_shadow_ambient extension is supported, but since none of the graphics card I’ve used had support for this feature I decided to use another approach. If instead of using GL_MODULATE, GL_INTERPOLATE might be used. Interpolate works like the S0 * S2 + S1 * (1 - S2). S0 to S2 are the sources from where you can get texel data. I chose to set S0 to GL_PREVIOUS which is the result of the previous texture blending which is for a none-textured object, its color and for a textured object the texel data if using GL_DECAL and the result of a previous blending if using GL_BLEND or GL_MODULATE. S1 is set to the shadow texture, while S2 is a constant of 0.9. This gives the effect of shaded areas being a bit darker and illuminated areas a bit lighter than without using shadows.

2.1 Material

All testing of algorithms where made on the same computer with the following specification:

- Operating system: Windows XP Pro, Service Pack 1
- CPU: Intel Pentium 3, 950 MHz
- Memory: 512 MB,
- Graphics card: Nvidia GeForce4 Ti 4200, 64 MB DDR memory, 128-bit memory bus, AGP 4x,
- Graphics driver: Detonator 45.23
- Compiler: Intel C++ Compiler Version 7.1
  Using compiler optimization flags: /O2 /Ob1 /Oy /GF /ML /GS /Gy /Gd
3 Survey of existing algorithms

The six most commonly used methods[5][17], not including ray-tracing, hybrid methods or modified scan-line algorithms like projecting polygons: Projected planar shadows, Area subdivision, Shadow volumes, Shadows on curved surfaces, Object-ID/Priority buffers and Shadow mapping can be divided into two major groups, analytical and image based. The analytical approach can be further divided into two groups[7]: Projection and Volumes.

Projection techniques project primitives away from the light source onto the surface of other primitives.

Volume techniques construct volumes by adding vertices to a scene or modifying the geometry of an object. The volume of an object represents the shadow it casts.

Image based techniques render the scene from the lights point of view and uses that image to determine what parts are shaded when rendering the scene from the eyes point of view.

The process of applying shadows to a scene can generally be divided into two parts; the shadow determination part and the actual rendering of shadows. The render a scene with shadows either by:

1. Start with a lit scene and subtract or modulate out light in the shadowed areas.
2. Start with an unlit scene and add in light contributions for each light in the scene, except where there is a shadow.

Most of the below described techniques can work in both ways, but the most common way is to lay down grey texture or polygons to indicate shadows. This works well with one light source but for several light sources the second approach has to be used to get correct results.

3.1 Projected planar shadows

Projected planar shadows was originally proposed by Blinn in 1988[1] as fake shadows. Today also referred to as planar shadows, I have chosen to refer to them as projected planar shadows for clarity.

Projected planar shadows is definitely the simplest and fastest method. It is suitable for hardware acceleration, but also the most inaccurate. The technique works by rendering objects that are supposed to cast shadows twice. The first time the object is rendered normally. The second time the model-view matrix is multiplied with a shadow caster matrix constructed (as seen in Figure 4) from the position of the light source and the receiving ground plane.
3 Survey of existing algorithms

```
dot = ground_plane[X] * light_position[X] +
ground_plane[Y] * light_position[Y] +
ground_plane[Z] * light_position[Z] +
ground_plane[W] * light_position[W];
shadow_matrix[0][0] = dot – light_position[X] * ground_plane[X];
shadow_matrix[1][0] = 0.f – light_position[X] * ground_plane[Y];
shadow_matrix[2][0] = 0.f – light_position[X] * ground_plane[Z];
shadow_matrix[3][0] = 0.f – light_position[X] * ground_plane[W];
shadow_matrix[0][1] = 0.f – light_position[Y] * ground_plane[X];
shadow_matrix[1][1] = dot – light_position[Y] * ground_plane[Y];
shadow_matrix[2][1] = 0.f – light_position[Y] * ground_plane[Z];
shadow_matrix[3][1] = 0.f – light_position[Y] * ground_plane[W];
shadow_matrix[0][2] = 0.f – light_position[Z] * ground_plane[X];
shadow_matrix[1][2] = 0.f – light_position[Z] * ground_plane[Y];
shadow_matrix[3][2] = 0.f – light_position[Z] * ground_plane[W];
shadow_matrix[0][3] = 0.f – light_position[W] * ground_plane[X];
shadow_matrix[1][3] = 0.f – light_position[W] * ground_plane[Y];
shadow_matrix[2][3] = 0.f – light_position[W] * ground_plane[Z];
```

Figure 4: Calculation of the shadow caster matrix for projected planar shadows.

The shadow object may be drawn in black or blended to achieve a more realistic result.
If blending is used you have the problem with double blending (Figure 5), that is when a part of the shadow is rendered twice resulting in areas darker than the rest of the shadow due to that it’s rendered and blended multiple times. To solve this, the stencil buffer may be used to prevent multiple drawing on the same pixel in the resulting projected image.

Another problem is shadows that extend beyond the edge of the shadow receiver (Figure 5). The solution is again to use the stencil buffer to prevent drawing of the shadow object.

Z-fighting is also a common problem (Figure 5). This occurs when the flat receiving plane and the shadow are located on the same depth. The solution is to raise the shadow slightly in Z direction.

Figure 5: Problems with projected planar shadows.
3.1.1 Advantages

The biggest advantage of the projected planar shadow algorithm is its speed. It is by far the fastest algorithm. The algorithm is also extremely easy to implement and has no requirements on the objects that casts the shadow, any type of object can cast shadows.

3.1.2 Disadvantages

The major drawback of this technique is that it can only cast the shadow onto a flat plane. If you want to be able to cast a shadow both onto a wall and a floor you must draw the shadow two times, i.e. once per plane. Because of this it does not fulfill any of the requirements for true shadows.

3.2 Area subdivision

The area subdivision technique, originally proposed by Nishita and Nakamae in 1974[12] and modified by Atherton, Weiler and Greenberg in 1978[1] is a two pass algorithm. In the first pass the scene is rendered from the lights point of view and uses a hidden surface removal algorithm.

The basic concept of a polygon sorting hidden surface removal algorithm is that all surfaces that lay behind each unique polygonal area and within its borders are removed. The algorithm involves four steps:

1. a rough depth sort, front to back
2. a 2D comparison of the currently most forward polygon to the remaining polygons.
3. removal of polygons that exist behind the template and within its borders
4. a recursive comparison when an error in the preliminary depth sort has occurred.

The core function of this algorithm is its polygon clipper. It considers two polygons at a time and clips the hidden polygon to create a new polygon, representing the visible part.

When rendering the scene from the lights point of view using this hidden surface polygon clipper the scene is divided into two parts: the parts visible from the light, marked as being lit and the not visible clipped parts, marked as being in shadow. The second pass renders the scene from the eyes point of view and renders the lit parts with the light turned on and the shaded parts with the light turned off.
3 Survey of existing algorithms

3.2.1 Advantages

The biggest advantage of this technique is that it divides the scene analytically into two groups of polygons. This makes it possible to use the geometrical data for other calculations. You can also render the two parts separately one with light and the other without light to get completely accurate shadows and lighting. Since the shadow calculation is done using the geometrical data, there is no aliasing problem. It supports self shadowing and all objects can shadow any other object.

3.2.2 Disadvantages

The preprocessing complexity of the algorithm is high, due to the polygon clipping process and since the shadow determination clips polygons we might have to consider a lot more polygons when drawing the scene compared to the original scene. Additionally, it is algorithmically difficult to come up with a numerically robust polygon clipper for the hidden surface removal phase that can handle convex and concave polygons with holes[17].

Figure 6: The area subdivision approach.
3.3 Shadow volumes

Crow proposed in 1977[4] an algorithm to generate polygons that represent the shadow volume of objects in the scene, then placing them into the rendering data structure as invisible objects (Figure 7).

The construction of the volumes can be achieved in several ways. The most robust way is to tessellate the object then construct an edge silhouette from the lights point of view.

The silhouette is constructed by first transforming the light into object space; compute the plane equation for every polygon in the model; for every polygon, determine if the object-space light position is behind or in front of the polygon’s plane; search for edges where polygons have opposite facing toward the light, these edges are possible silhouette edges[9].

The silhouette is then used for the construction of the shadow volume itself. The side of the volume is constructed by adding new polygons to the scene, aligned from the position of the light source back towards infinity or at least until clipped by the frustum.

The second approach called shadow volume extrusion is faster and can be achieved with the use of the graphics cards vertex shader (programmable vertex processor). It is not possible for the vertex shader to add vertices and construct new polygons in a scene, just modify existing vertices. Instead it uses the vertices of the object itself and stretches back-facing polygons to represent the shadow volume.
The basic shadow volumes approach is a 3-pass method, all rendered from the eyes point of view. The shadow determination is done in the first two steps to handle overlapping shadow volumes.

The stencil buffer is used as a counter of how many shadow volumes a corresponding pixel in the frame buffer is located in. The stencil buffer is a buffer that can be used to store data about pixels in the rendered scene. It can be used for example to restrict rendering of certain pixels.

The values in the stencil buffer are first set to the number of shadow volumes the eye is located in, i.e. zero if it’s outside of any shadow volume.

The first pass the scene is rendered with only the front-facing polygons of the shadow volumes, from the eyes point of view as seen in Figure 8. For every rendered pixel in the frame buffer, increase the corresponding stencil buffer value by one for every shadow volume polygon that the imagined line between the eye and the pixel passes through (enters a shadow volume.)

In the second pass the scene is rendered with only the back-facing shadow volume polygons as seen in Figure 9. For every pixel in the frame buffer decrease the corresponding stencil buffer value by one for every polygon that the imagined line between the eye and the pixel passes through (leaves a shadow volume.)

The stencil buffer becomes an indicator for every pixel in the scene whether it’s lit or shaded. If the value is zero it is lit, it has “left” the same amount of shadow volumes as it has “entered”. If the value in the stencil buffer is not zero it is shaded, it has “entered” more shadows than it has “left” (Figure 10).

In the final pass the scene is rendered using the stencil buffer as a light occlusion mask.
3 Survey of existing algorithms

Figure 8: First pass of shadow determination, rendering only front-facing polygons.

Figure 9: Second pass of shadow determination, rendering only back-facing polygons.

Figure 10: The result of the first two shadow determination passes.
3.3.1 Advantages

This approach supports both self shadowing and the ability for every object in the scene to shadow all other objects. Since the shadow calculation is done using the geometrical data there is no problem with aliasing. It can achieve omni-directional shadow casting, the ability to cast shadows in all directions without additional phases.

3.3.2 Disadvantages

The silhouette generation is computationally expensive. Additionally, if the objects are not well tessellated the tessellation phase can be costly. The construction of the volumes may require adding polygons to the scene and by that increasing complexity and computational load. If the graphics card has a vertex shader the volumes may however be constructed by extrusion, but this requires non flat objects.

3.4 Shadows on curved surfaces

Also known as fake shadows using textures or projective texture shadows. This approach proposed by Nguyen[12] can be seen as a budget version of shadow mapping or as an extension to light maps. It’s a simple two-pass algorithm which involves render to texture. It is similar to projected planar shadows.

In the first pass you clear the rendering destination texture to white. Then render the shadow caster in black from the lights point of view (Figure 11, Figure 12).

In the second pass the scene is rendered from the eyes point of view and the shadow texture is projected onto the receivers (Figure 13).

With this approach you can easily accomplish softer edges by using low resolution textures to render to and use some filtering like bilinear resulting in shadow edges that looks soft. On multi-texturing hardware you can use multiple textures and take multiple jittered bilinear samples to smooth out the edges even more.
3.4.1 Advantages

This approach can produce shadows on non-flat objects with a relatively low computational cost. It is the fastest of the algorithms with this ability. It is also relatively simple to implement.

3.4.2 Disadvantages

The main problem with this approach is the shadow casters themselves. If you treat them as receivers, they will always be completely shadowed and otherwise self-shadowing is impossible. There is also a problem of having multiple shadow casters in one scene. This technique is therefore limited to cast a shadow from one object onto a wall or a floor. Hence it does not fulfill the requirements of a true shadow technique.

3.5 Shadow mapping

This two-pass approach was originally proposed by Williams in 1978[16]. It is able to generate shadows for any object in a scene without knowing anything about its geometry.

In the first pass the scene is rendered from the lights point of view (Figure 14), just like with the Shadows on curved surfaces approach. But in this case we render not only the shadow caster, but the entire scene and instead of storing a black and white image we store the z-buffer. The texture holding the z-buffer is called a shadow map. Every value in the shadow map is the closest pixels distance from the light source as seen in Figure 15.
In the second pass the scene is rendered from the eyes point of view. And for every pixel in the rasterization; decide the pixels XYZ location relative to the light source. This location is calculated using the same projection matrix as was used when rendering the shadow map. Then compare the depth value on position XY in the shadow map with the rendered pixels distance from the light source (Z value.) If the Z value of the rendered pixel is larger than the value in the shadow map on the corresponding position we know that some object must be closer than the rendered pixel, and hence the rendered pixel is shaded. If the Z-value of the rendered pixel is approximately the same as the corresponding value in the shadow map it must be the same object that was rendered from the lights point of view and hence it’s visible from the lights point of view and it must be lit.

The most common way to render the shadows is to texture all objects in the scene and darken the shadowed areas, but the opposite is also possible, i.e. lighten the lit areas.

### 3.5.1 Advantages

Shadow mapping supports self shadowing and the ability to cast shadows from every object unto any object in the scene. It is a relatively fast approach. There is no need to alter the scene by adding additional polygons, modifying existing ones, tessellate objects or calculate object silhouette. Since it is a pure image based algorithm we don’t need to know anything about the geometrical data of any abject. There are a lot of optimization alternatives; you can alter the texture resolution, use different depth precision and use filtering on the texture. You can even achieve fake soft shadows by altering the depth buffer. Another great advantage is that one modern graphics card there is hardware support in OpenGL for comparing the texture R coordinate to a depth texture value in order to produce a boolean texture value. This makes the comparison in shadow mapping very fast.
3.5.2 Disadvantages

The biggest drawback of this approach is aliasing or sampling error. When projecting the coordinates from object space to light space you get small rounding errors that can give erroneous shadowing results. This can be corrected to some degree by biasing your depth values by pushing them farther to the light when performing the depth test. This problem is most is most obvious when the light source is located in or close to the plane of a polygon, when this occurs the polygon renders from the light point of view is stored using a much lesser amount of pixels than when rendered from the eyes point of view. This visual effect of this is lessened if the light source that causes the shading is located at the same place as the shadow mapping light source since it will be quite dark. Another issue is the resolution of the depth buffer. The size of the frustum when rendering the shadow map is directly linked to the resulting resolution of the depth, the larger the frustum the lesser the accuracy. To optimize the use of the depth buffer you may adjust the frustum to be 0 at the object closest to the light and 1 at the farthest object.

![Figure 16: Light source located close to the plane of the polygon.](image)

3.6 Object-ID/Priority buffers

A very similar approach to shadow mapping. The shadow calculation is divided into two passes.

The algorithm works by instead of storing a depth map like shadow mapping, it stores an object-ID buffer. It assigns a unique ID to each object in the scene, where every object is defined as something that cannot shadow itself. So, any convex object or piece of a convex object works.

The scene is rendered from the lights point of view and every objects ID is stored in a texture, known as the object-ID buffer. After rendering, every pixel in the texture will contain the ID of the object closest to the light.

In the second phase the scene is rendered from the eyes point of view. Similarly to shadow mapping you use the projection matrix from when rendering from the lights point of view to match the rendered pixels to the value in the object-ID buffer. If the rendered pixels ID match the ID in the buffer it must be lit since it was visible from the lights point of view. If the ID of the rendered pixel does not match the one in the buffer, some other object must be located between the rendered object and the light source and hence it must be shadowed.
We apply the shadows to the scene in the same way as with shadow mapping.

### 3.6.1 Advantages

Since this approach is so similar to shadow mapping it shares almost all advantages as long as non convex objects are broken down into convex objects, otherwise self shadowing is not possible. The advantage Object-ID/Priority buffers have over shadow mapping is that since it doesn’t use depth values the resolution of the depth buffer and the size of the frustum is not a problem.

### 3.6.2 Disadvantages

The disadvantage compared to shadow mapping is that you have to break down non convex objects into convex ones, which takes time. And it does suffer from a similar aliasing problem due to that you will not always project exactly onto the same shadow buffer pixel, causing a different ID value to be found instead. Another big disadvantage in speed compared to shadow mapping is that there is no hardware support for comparing ID values.
4 Design Issues

In order to achieve shadowing from all objects with the possibility of casting shadows onto every object in an arbitrary scene where the geometry and construction of an object is without any limitations we have to choose an algorithm with the ability to cast shadows from all objects onto all objects. Also objects have to shadow themselves. Area Subdivision, Shadow volumes, Shadow mapping and Object-ID/Priority buffers fulfills these requirements. Comparing the two analytical methods, Shadow volumes and Area subdivision, Shadow volumes is a lot faster. Similarly comparing Shadow mapping and Object-ID/Priority buffers, Shadow mapping is the faster one due to hardware support. We compare these two algorithms and evaluate their pros and cons with regards to implementation in an existing rendering toolkit like VRT.

Since objects can be constructed to be flat we cannot use shadow volume extrusion, so this comparison will be using ordinary shadow volume construction, by calculating a silhouette and extruding that backwards.

Since objects in VRT does not have any limitations and can be modified at runtime the tessellation of the objects for the shadow volume technique would have to be done while rendering, every time the geometry changes, which would cause a large computational load and a slow down of rendering. The tessellated geometrical data would have to be stored in memory along with the original geometrical data, resulting in a larger memory load.

If we compare the speed of the two algorithms it soon becomes evident that the shadow volume algorithm cannot become faster than the shadow mapping algorithm if we don’t use shadow volumes extrusion. If we take the simple example with one triangle composed by three vertices; The shadow mapping computational time will at most be twice the one of un-shadowed rendering, because the shadow map is either the same size or smaller than the rendered original scene. In the shadow volumes approach the computational time will always be larger, since for one triangle we will have to render the triangle itself and at least three sides composed of either 6 triangles or three quads. In the most common shadow volume version, called Carmack’s reverse[6] (also known as Z-fail) we have to render the end caps of the volumes. The sum is then to render at least two triangles and three quads, which is obviously slower than shadow mapping in every case. The only advantages shadow volumes have over shadow mapping is that it doesn’t suffer from sampling error and can achieve omni-directional shadowing. But this does not in my opinion make up for the disadvantages, especially considering that the technique does suffer from problems when eye is located in a shadow volume which in practice has proven to be difficult to solve even though a number of theoretical solutions has been suggested[12]. Additionally the problem with near and far clip planes that clip the volumes[12] should be considered as a disadvantage.

My conclusion is that the shadow mapping approach would be the best suited approach for implementation in VRT.
To use shadow mapping in VRT one has to use multi-texturing to apply the shadows to the scene. This is due to that objects should be able to be textured by other textures and also get shadowed. Multi-texturing is a core feature in OpenGL since version 1.3[15]. Multi-texturing is done in hardware and therefore the performance overhead for using multiple textures is very small.
5 Implementation

In order to implement shadow mapping using hardware support and multiple texturing into VRT, functions that enable, disable shadow rendering, sets the shadow map size, etc. are all located in vrt_shadow.c. All variables used when rendering is stored in the VRT context located in vrt_types.h.

5.1 Types and structures

As mentioned all variables used when rendering shadows are stored in vrtctx variable. If not specifically mentioned all boolean variables use 1 as true and 0 as false.

    int render_shadow;

Indicates whether shadow mapping is active.

    int render_shadow_this_pass;

Indicates that the scene is rendered for the purpose of being stored in the shadow map itself.

    int shadow_width;
    int shadow_height;

Is the width and height of the shadow map and has the same restriction as textures in OpenGL.

    int shadow_auto_resize;

Indicates whether the size of the shadow map should be automatically resized to the largest allowed size smaller or equal to the screen size.

    float shadow_polygon_scale;
    float shadow_polygon_bias;

The variables used for compensating for sampling error. Defaults to 1.1 and 4.0.

    float shadow_coordinate_scale;
    float shadow_coordinate_bias;

The texture offset variables, default 0.5 for both.

    double shadow_light_position[3];
    double shadow_light_look_at[3];
    double shadow_light_up[3];

Used to define the viewing transformation when rendering the shadow map. Defaults to {10, 10, 10}, {0, 0, 0} and {0, 1, 0}.
5 Implementation

M4 shadow_mvm;

Used to store the modelview matrix from when rendering the shadow map.

VRT_Light *shadow_light;

The location of the eye when constructing a shadow map can be attached to a light source, which is stored here.

int shadow_render_map;

This is a special variable that indicates when VRT should swap the backbuffer to the front buffer. This makes it easy to check the eyes location when rendering the shadow map. If 1 then render only the shadow map, if 2 then render both scene and map (very flickering), else render only scene. Defaults to 0.

5.2 Initialization

Before any shadows are applied to the scene you have to initialize the shadowing by calling:

int VRT_ShadowSwitchOn();

This function first checks if the system supports depth textures, multi-texturing and comparing depth texture coordinates. These all used to be extensions, but are since OpenGL 1.4 core features[15]. Still, since they are hardware dependent, a system might not support them. If any of these is not supported the function fails and returns 0. Otherwise it activates the second texture “layer” GL_TEXTURE1. Linear interpolation of the texture is used because it is fast and also gives great results. The texture is set to clamp to edge for the best result. The pixel unpack alignment is set to 4 for optimal speed on a 32 bit processor. The core feature of the algorithm GL_TEXTURE_COMPARE_MODE is set to GL_COMPARE_R_TO_TEXTURE, this essentially tells OpenGL to compare the values in the texture to the rendering R coordinate. We set the GL_TEXTURE_COMPARE_FUNC to GL_LEQUAL, this means; result = ( 1.0 if R less or equal to Dt, 0.0 if R greater than Dt). The texture coordinate generation is turned off for texture coordinates S, T, R and Q. GL_OBJECT_LINEAR is used as the function for determining the coordinates.

5.3 Shadowing parameter adjustment

There are a number of adjustments that can be made on the different variables used in shadow mapping. Most of the time there is no need to change any of these, but if it is necessary a number of functions are provided to facilitate the need.

int VRT_ExtensionSupported(const char *extension);

VRT_ExtensionSupported is used internally to check if the system supports shadowing, but can be used by to check for any extension on the system. It returns 1 if the extension is supported and 0 otherwise.

Real-time shadows in 3D computer graphics
void VRT_ShadowSetLookAt( double position[3],
double look_at[3],
double up[3]);

VRT_ShadowSetLookAt is probably the most useful function for the application
programmer using VRT. It sets the location of the shadow casting “light source”, the
position of the reference point and the direction of the up vector. It works similarly to
gluLookAt. If the shadow caster is connected to a VRT_Light any call to this function
is ignored.

int VRT_ShadowAttachToLight(VRT_Light *light);

Instead of specify the position of the shadow casting light source one may attach it to
a VRT_Light. This works for positional lights and spotlights. To detach from a light
source call the function with NULL as argument.

void VRT_ShadowSetDimension(int width, int height);

By default the shadow map resizes automatically to be equal to or the largest possible
permitted size lower than the drawing window size. The permitted values for a
shadow map are the same as on textures in OpenGL, and it has to be the same size or
smaller than the drawing window. With this function you can set the size manually.
The size of the width does not have to be the same as the height. If you use this
function the auto size feature which is on by default, will be disabled. If you send 0 as
width and height the function will set the size as the largest possible value smaller
than or equal to the window size.

void VRT_ShadowSetAutoResizeMap(int val);

VRT_ShadowSetAutoResizeMap controls the auto size feature. If enabled, it resizes the
shadow map every time the drawing window is resized. This feature is on by default.
To disable it send 0 as argument and to enable it send 1 as argument.

void VRT_ShadowSetIntensity(int val);

By default shadows are achieved by blending the shadowed areas with a darker
texture. The result is that shadowed parts are a bit darker than the lit parts. However
there is also the ability to get completely black shadows. Call the function with 0 to
activate black shadows, and 1 to get shadows that are blended.

void VRT_ShadowSetPolygonOffset( float polygon_scale,
                                    float polygon_bias);

To compensate for sampling error, polygon offset is used. It is possible to modify the
offset arguments with this function. The offset value $o$ is calculated by

\[ o = m \cdot \text{polygon}\_scale + r \cdot \text{polygon}\_bias \]

where $m$ is the maximum depth slope of the polygon and $r$ is the smallest value
guaranteed to produce a resolvable difference in window coordinate depth values.
Polygon_scale defaults to 1.1 and polygon_bias to 4.0.

void VRT_ShadowSetTextureOffset( float coordinate_scale,
float coordinate_bias);

Another possibility to tweak the algorithm is to modify the texture offset value. They both default to 0.5.

void VRT_ShadowDisplayShadowMap(int val);

VRT_ShadowDisplayShadowMap can be handy when trying to set up your scene and check whether the shadow casting light source is located correct. It controls the feature to swap buffers after rendering from the lights point of view enabling the possibility to view it in the frame buffer. If 1 is provided as argument the shadow map is rendered in the frame buffer, if 2 is provided both scene and map are rendered. As default it only renders the scene.

5.4 Depth map generation

A major part of the shadow mapping algorithm is the depth map generation. In VRT this is handled internally. If shadowing is enabled the main drawing function calls a wrapper function that behaves similarly to the ordinary scene rendering function, with the difference that it renders the scene twice. Before the first pass, the viewport is set to the same size as the depth map. The projection matrix is set to match the arguments sent to VRT_ShadowSetLookAt, or if the shadow generator is attached to a light source, the position of the eye and reference point of the scene when rendering from the light source point of view is set to match the position and direction of the light source. When rendering the scene the first pass all modifications to the projection matrix are ignored. After the rendering of the scene, the depth component of the buffer is copied to the texture in TEXTURE1 (i.e. the second texture environment, used for applying shadows to all objects in the scene), used as depth map.

Before rendering the scene in the second phase, we reset the projection matrix. And multiply the texture matrix with the projection matrix used in the first phase, to get accurate texture coordinate generation.

When rendering the scene the second time all modifications to the modelview matrix has to be applied to the texture matrix of texture environment 1 as well. This is due to that all texture coordinates for the texture used to apply shadows are generated automatically.

Even if we use the depth map to texture all objects in the scene, the resulting texture applied is not the depth value since we have turned on the feature compare_r_to_texture and set the comparing function to be less or equal. This means the resulting texture in texture environment 1 will be 1 if the r value is less or equal to the value in the texture and 0 otherwise. If we simply multiply this value with the result of all previous textures and coloring, we get the effect that all areas that are shadowed become completely black, and lit areas get the same color as without shadowing enabled. This is not very realistic, and one solution to this is that instead of multiplying the values in texture environment 1 we use interpolation and introduce a constant of 0.9.
5.5 Turning off shadowing

Turning off shadows in VRT is simple. Only call the function

```
VRT_ShadowTurnOff()
```

This disables the rendering of the scene twice and turns off automatic texture coordinate generation.
6 Benchmark application

6.1 Frame rate comparison

Evaluating the two most commonly used methods to achieve shadows; shadow volumes and shadow mapping in the most straightforward way is to implement an application that renders a scene and is able to produce shadows with both methods and measuring the frame rate. There are two major factors that affect the computational load; the resolution of the drawing area, due to the number of pixels drawn in the rasterization phase and the number of polygons drawn.

I have chosen to measure the frame rate with different resolutions. I have chosen not do a benchmark based on different number of polygons because as I will show, if shadow mapping is faster with one number of polygons it will be faster with any number of polygons. The reason for this is that the shadow mapping approach in the worst case scenario essentially works like rendering the scene twice resulting in half the frame rate. A worst case scenario being when the depth map has the same resolution as the rendered scene, texturing and lighting is not disabled during rendering of the depth map. There are additional costs for copying the depth values from the depth buffer and calculating the texture coordinates, but you also save computations by only rendering two times not three like in shadow volumes. So shadow mapping can be considered to be twice as computationally heavy or the same as rendering twice the amount of polygons as without shadows.

Shadow volumes however cannot in any circumstance be that efficient. If we use the most efficient and accurate version shadow volumes called Carmack’s reverse we have to draw all the polygons of the objects, plus all sides of the volume, including the end cap of the volume which is a copy of the objects silhouette. If we take one single triangle as an example we have to draw the triangle itself, three quads for the sides and one triangle as an end cap.

The shadow volumes approach also has to generate the silhouette of the object which is computationally quite heavy, we have to construct the volumes by extruding the silhouette along the line of the light to create the end cap and we have to render the scene three times compared to two times for shadow mapping.

So the shadow volumes approach can be considered to always have to draw more than twice the number of polygons than the object itself. Therefore comparing the two methods by rendering a scene with a different amount of polygons is redundant.

The performance benchmark application is a single program that implements both shadow mapping and shadow volumes. Shadow mapping is achieved by rendering the scene from the lights point of view and copying the frame buffer to a depth texture. When the final scene is rendered the R coordinate of rendered pixels are compared to the depth texture, just as described earlier. The shadow volumes version used is the same as the one used by Everitt and Kilgard in their paper “Practical and robust
stenciled shadow volumes for hardware-accelerated rendering” [6]. It uses the zfail or Carmack’s reverse.

I chose to render the same scene with 3 different resolutions and with both windowed and full screen mode for one resolution: 800x600 windowed mode, 800x600 full screen mode, 1024x768 full screen mode and 1280x1024 full screen mode.

For every resolution a number of different resolutions of the shadow mapping depth map were used.

For 800x600: 512x512, 256x256, 128x128 and 64x64 were used.
For 1024x768: 1024x512, 512x512, 512x256 and 256x256 were used.
For 1280x1024: 1024x1024, 512x512, 256x256 and 128x128 were used.

All resolutions were rendered; without shadows and without texturing, without shadows and with texturing, with shadow mapping with the different depth map resolutions, with shadow volumes without texturing and with shadow volumes with texturing. The reason for rendering with texturing both enabled and disabled is that most 3D applications today use texturing whether or not shadow mapping is used, and therefore I also present the result when texturing is enabled for a more relevant result.

6.2 Visual comparison

In addition to the frame rate benchmark program I used three additional programs to demonstrate artifacts that can occur when applying shadows to a scene.

The "md2shader" demo by Mark J. Kilgard is used to show artifacts when the eye is located inside a shadow volumes.

The "Fast, Practical and Robust Shadows" demo by Morgan McGuire et al. is used to show clipping plane artifacts.

Additionally I have programmed a shadow mapping demo to demonstrate rounding errors for shadow mapping.
7 Results

7.1 Frame rates

Figure 17, Figure 18, Figure 19 and Figure 20 show the frame rate for unshadowed rendering with texturing disabled, unshadowed rendering with texturing enabled, shadow mapping, shadow volumes with texturing disabled, shadow volumes with texturing enabled.

![Graph showing frame rates for different rendering techniques](image)

Figure 17: 800x600 windowed mode.
Figure 18: 800x600 full screen mode.

Figure 19: 1024x768 full screen mode.
It is clear that shadow mapping is faster than shadow volumes in every instance, even when shadow volumes has texturing disabled.

### 7.2 Artifacts

Neither of these two approaches is flawless. There are situations when they will produce artifacts. For shadow volumes there are mainly two instances that produce artifacts: when the eye is located in a shadow volume and when the near or far clipping plane clips the shadow volume.

The reason artifacts appear when the eye is inside a shadow volume is that the central shadow determining algorithm counts shadow volume edges between the eye and the rendered object. If the initial value in the stencil buffer is not set to the number of volumes the eye is located in the final resulting value in the stencil buffer will be erroneous (Figure 21, Figure 22). This artifact can be avoided as long as the algorithm has a way of correctly determining if the eye is inside of a volume and how many volumes it is inside.
Figure 21: Shadow volumes, eye outside of shadow volume.

Figure 22: Shadow volume artifacts when eye is located inside a shadow volume.
The problem with clipping planes that clip a shadow volume is due to the fact that everything outside of the clipping planes is not rendered. And if it’s not rendered it cannot be counted and hence the value in the stencil buffer will be erroneous. This occurs if there is a shadow volume edge between the eye and the near clipping plane (Figure 23, Figure 24) or a shadow volume edge beyond the far clipping plane where another edge of the volume is inside the clipping volume.

Figure 23: Shadow volumes.
Figure 24: Near clipping plane artifacts.
The problem with near clipping plane artifacts can be solved by counting the edges in reverse order like the Carmack’s reverse does (Figure 25). It counts the volume edges from the far clipping plane to the eye instead of from the eye to the far clipping plane.

Figure 25: Same scene as Figure 24, without artifacts.
Shadow mapping artifacts differ from the ones produced by the shadow volume approach, since shadow mapping is image based. The most common artifact is when rounding errors occur. This is especially noticeable when a polygon is located in the plane of the eye (Figure 26), this artifact will however be almost totally invisible if the OpenGL light source is located in the same place as the shadow mapping light source. The reason is that the shading will cause the polygon to be a lot darker.

Figure 26: Shadow mapping artifacts.
Another “artifact” is the chunky appearance of the shadows if the resolution in the depth map is too low. Figure 27, Figure 28, Figure 29 and Figure 30 are examples of the same scene with a resolution of 800x600 rendered with different depth map resolutions. A low resolution does not however have only negative consequences since it is much faster to render the depth map at a low resolution and a low resolution also gives the appearance of a much softer shadow edge.

Figure 27: Shadow mapping 64 x 64 depth map resolution.
Figure 28: Shadow mapping 128 x 128 depth map resolution.
Figure 29: Shadow mapping 256 x 256 depth map resolution.
Figure 30: Shadow mapping 512 x 512 depth map resolution.
7.3 Example VRT program

Using shadows in VRT is very simple. The following code is an example program that uses shadows. The only required thing to do is to add shadows is to call the function VRT_ShadowSwitchOn(), it can be done anytime in the program. The call to VRT_ShadowSetLookAt sets the location of the shadow casting light source. With this example we can see that the goal of easily achieving shadows in VRT is fulfilled. Figure 31 shows the results, notice the shadows the arm casts on the torso.

```c
int main(int argc, char *argv[])  
{  
double position[3] = {6, 6, -6}, lookat[3] = {0,0,0}, up[3] = {0,1,0};  
  /* initialize VRT Toolkit */  
  VRT_Init(&argc, argv);  
  /* chose a display type, usually default */  
  VRT_SetDisplay(VRT_DEFAULT);  
  /* build up your own scenery */  
  build_scene();  
  if(VRT_ShadowSwitchOn())  
    VRT_ShadowSetLookAt(position, lookat, up);  
  /* install a user-define callback function */  
  VRT_SetCallback((VRT_HookPtr)SimulationLoop);  
  /* decide for background color */  
  VRT_SetClearColor(0.7f,0.7f,8.0f,0.5f);  
  /* install and set a virtual camera */  
  VRT_SetDefaultCamera(6.0,6.0,6.0,0.0,0.0,0.0,0.0,1.0,0.0);  
  fov = 15.f;  
  VRT_SetCameraFieldOfView(fov);  
  /* start the simulation loop */  
  VRT_SimulationLoop();  
  /* shut down simulation server */  
  VRT_Close();  
  return 0;  
}
```
Figure 31: VRT program with shadows.
8 Conclusion

Shadow mapping is today clearly the most suitable algorithm for implementation in a scene graph rendering toolkit. Its advantages are mainly its robustness, speed and the fact that is relatively easy to implement. Shadow volumes however have a number of advantages like omni-directional shadow casting and lack of rounding errors. Because of this I predict that in the future shadow volumes will become the dominant algorithm, especially when more hardware support is introduced.
9 Future work

Shadow mapping is an immensely adaptable algorithm which can be extremely optimized. I will in this section mention some of the enhancements that can be made to the implementation of shadow mapping in VRT.

9.1 State change control

VRT has a feature that keeps track of whether the scene needs redrawing, that is if any object or the eye has moved. If the scene is changed the scene is redrawn. The same approach could be applied to the rendering of the shadows. Today the shadow is redrawn every time the scene is redrawn, but instead a separate state control variable could be introduced to keep track of whether any object has moved or the eye of the shadow map has moved.

9.2 Eye space coordinate texture generation

When calculating the texture coordinates for the shadow texture I calculate them in object space. According to Everitt[7] we can use eye space coordinates and reduce sampling errors.

9.3 Direct to texture render

There are several ways to render a scene into a texture.

- `glReadPixels()`, `glTexImage*()`, Slow.
- `glCopyTexImage*()`, Better.
- `glCopyTexSubImage*()`, Even Better.
- Render Directly to Texture, Eliminates “texture copy” – potentially optimal

I chose the `glCopyTexImage` because you can in a simple way copy the depth part of the scene. It is however possible to render directly to a texture, this removes one bottleneck in the shadow mapping algorithm. The drawback is that it requires the following extensions that might not be supported:

- `WGL_ARB_render_texture`
- `WGL_ARB_pbuffer`
- `WGL_ARB_pixel_format`
9.4 Partial scene depth mapping

An optimization to shadow mapping is to render only the objects that you want to cast shadows, when rendering the depth map. This reduces the time it takes to render the depth map, but also requires a flag for every object in the scene whether it should cast a shadow or not.
References