Fall 2010

Investigation into the feasibility of shadow generation on mobile graphic cards

Nicholas Maina Kiguta
University of Iowa

INVESTIGATION INTO THE FEASIBILITY OF SHADOW GENERATION ON MOBILE GRAPHIC CARDS

by

Nicholas Maina Kiguta

A thesis submitted in partial fulfillment
of the requirements for the Master of
Science degree in Electrical and Computer Engineering
in the Graduate College of
The University of Iowa

December 2010

Thesis Supervisor: Professor Jon Kuhl

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

____

MASTER'S THESIS

____

This is to certify that the Master's thesis of

Nicholas Maina Kiguta

has been approved by the Examining Committee
for the thesis requirement for the Master of Science
degree in Electrical and Computer Engineering at the December 2010
graduation.

Thesis Committee: ___________________________________
Jon Kuhl, Thesis Supervisor

____
Chris Wyman

____
David Andersen

To the Kigutas; I finally figured out how deep the rabbit-hole goes.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

> I would remind you O Painter! To dress your figures in the lightest colors you can, since, if you put them in dark colors, they will be in too slight relief and inconspicuous from a distance. And this is because the **shadows** of all objects are dark. And if you make a dress dark there is little variety between **the lights and shadows**, while in light colors there will be greater variety.
>
> Leonardo Da Vinci *ca* 1470

The use of shadows for establishing visual cues in imagery dates as far back as Leonardo Da Vinci's era when he invented 'Chiaroscuro' [1]. Chiaroscuro is a shading style that relates light, color and form in a way that approximates their scientific behavior by allowing depth and intensity to blend harmoniously. Leonardo elegantly used it in the famous painting of Mona Lisa. This style has since been used to 'bring images to life' by various artists and has found its way into computer generated imagery.

Much research has gone into the study of shadows and the role shadows play in the perception of three-dimensional world. Hubona et al [2] investigated the effect of object shadows in promoting 3D visualization. Their findings conclude that shadows aid in understanding not only the position but also the size of the occluder.

Kersten et al [3] conducted a psychophysical investigation that culminated in the following conclusions;

- The motion of an object's shadow overrides other perceptual biases such as a constant object size assumption.

- While a moving image patch can be easily identified as a shadow by the shadow darkness, in certain conditions, even unnatural shadow darkness can induce apparent motion in depth of an object.
- Interestingly, when shadow motion is caused by a moving light source, our visual system interprets this shadow motion as consistent with a moving object rather than a moving light source.

In the investigation on spatial relationship perception as it relates to shadow quality, Wanger [4] echoes the second point above. He shows that it is usually better to have an inaccurate shadow than none at all as the eye is fairly forgiving about the shape of a shadow. These findings suggest that in the quest for visual realism in computer-generated imagery, shadows should be included as much as is possible.

The last decade has seen a proliferation of shadow generation techniques, particularly soft shadows. This is largely due to the advances in graphics processing units (GPU's), which have made possible the implementation of all sorts of algorithms that were not feasible previously. These graphics cards have increased their computational power and therefore allowed more compute intensive algorithms to see the light of day. In 1990, Woo et al [5] provided a comprehensive survey of shadow generation techniques that were at the time, considered state of the art. The technology advancement that has since happened has made possible the generation of real-time 3D imagery that was unfathomable then. Chief among the beneficiaries of this advancement is the dynamic creation of soft-shadows in real-time. The concept of real-time and a further explication of soft-shadows will be detailed in subsequent chapters in this work.

A parallel development within the last decade has been a development of techniques intended to reduce power consumption in electronics. Aggressive measures have been developed to reduce power consumption and prolong battery life in hand-held electronics, resulting in devices that can run for a long time before being re-charged. The ideal situation would be an extremely efficient, low-power high performance system capable of meeting its intended design goal using the least amount of power. These two quests (low-power and high performance) seem to be at odds with each other and as is often the case in problem solving, a compromise solution is generally necessary.

This work investigates the feasibility of shadow generation on a memory and power constrained graphics card such as would be found on a low-end mobile device. In essence, the question is posed whether it is possible to generate 'good-looking' shadows on a device that has severe constraints in comparison to the commodity graphics cards on desktop computers. While 'good-looking' is a subjective criterion, for the purposes of this work, it is defined as both aesthetically pleasing and immersive. For instance, a shadow that has the right intensity but is misplaced may be aesthetically pleasing but not immersive. However, a shadow with an unnaturally dark intensity but correct placement is considered 'good-looking'. This paper is organized as follows:

A condensed view of the graphics pipeline is presented in chapter two. This is intended to provide the necessary background for the terms and concepts used in the rest of the paper. Readers already familiar with graphics pipelines can safely skip this chapter.

Chapter three defines shadows and the difference between hard and soft shadows. This is also where the state of the art algorithms for shadow generation are presented. Emphasis is placed on the algorithms that work in real-time so lengthy discussions of global illumination schemes such as ray tracing and radiosity based algorithms are omitted.

Chapter four details the problem statement. Here, the graphics card that was used in this study is presented and any assumptions made about the nature of this investigation are outlined. A detailed look at the various shadow generation techniques chosen is given in this chapter and the conditions under which the tests were conducted are also presented. Both the quantitative and qualitative findings of the chosen algorithms are explained, followed by suggestions/recommendations of which algorithm to use based on both context and the reported results.

Chapter five concludes this study with a summary of the study and discusses suggestions for future research.

CHAPTER 2

BRIEF REVIEW OF THE GRAPHICS PIPELINE

Webster's dictionary defines a pipeline as a route, channel or process along which something passes or is provided at a steady rate. As we shall see, the graphics pipeline is no exception. It has but one main function: to map a three-dimensional scene description into a two dimensional projection on the screen or viewing media. The scene description represents the geometry of the said objects in a mathematical form with which the pipeline can work. While most commodity graphics cards support quadrilateral polygons (quads), graphics units on handhelds restrict polygonal models to triangles.

Normally, other agents are needed to produce the final image, which is displayed on the viewing media. Such agents include, but are not limited to, a virtual camera, light sources, textures and shading equations. Perhaps rather surprising is the fact that complex realistic scenes can be generated using triangles with each triangle being oblivious to its immediate neighbor i.e. a triangle has no global spatial knowledge of its location relative to its neighbors.

It is henceforth assumed that the viewing media is the screen even though any output media may be used.

2.1 Fixed Function Graphics Pipeline

Here a simplified model of the fixed-function pipeline is introduced, so called because the various pipeline stages are not programmable/customizable This model preceded the architectures that are commonplace today but allows us to see the flow of data much more clearly. Figure 1 depicts this simplified pipeline as it applies to the Open

Graphics Library (OpenGL). OpenGL is a 3D application-programming interface (API) that provides the necessary abstraction to the actual graphics hardware[1]. What follows next is a brief discussion of the responsibilities of the depicted stages of the pipeline.

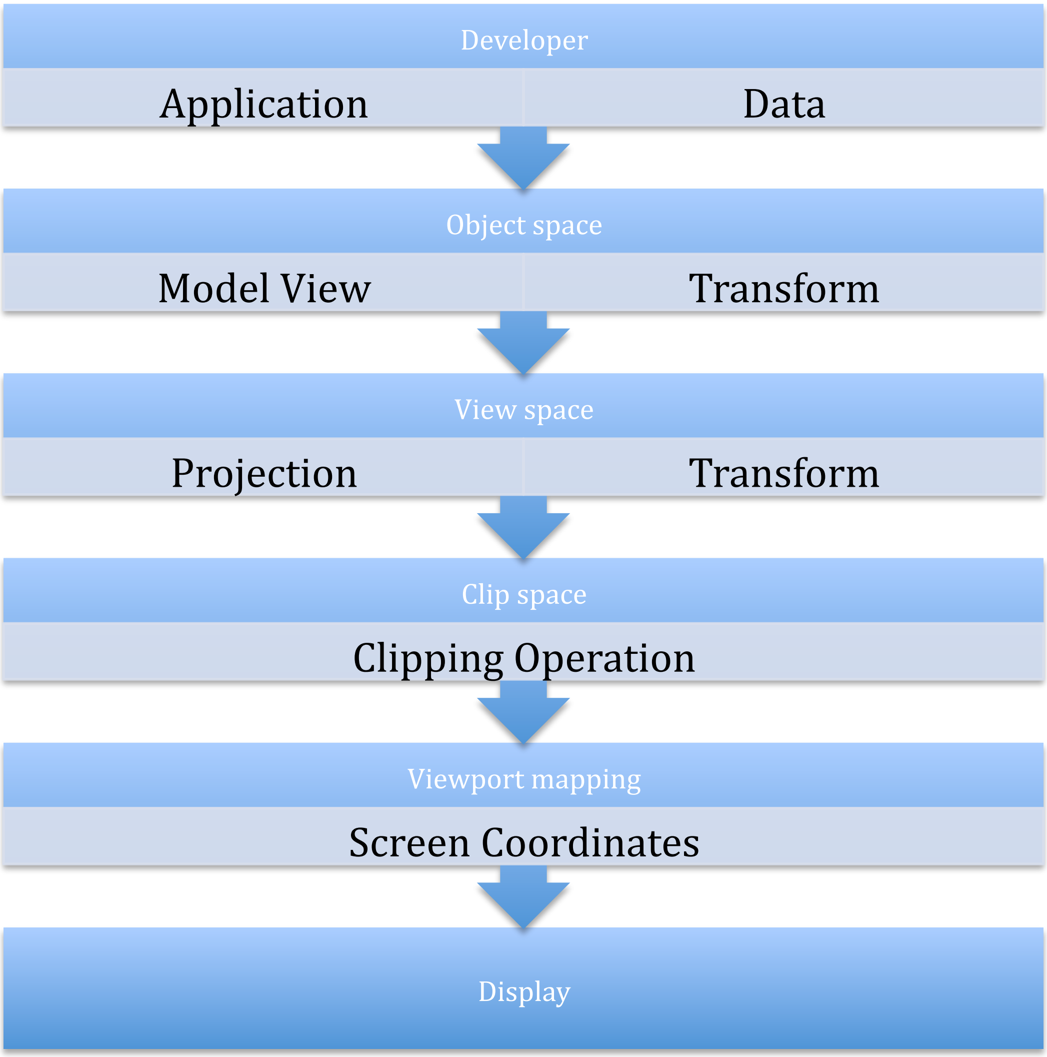


Figure 1 A basic fixed-function rendering pipeline.

[1] Direct3D is a 3D API from Microsoft ™. These two libraries are the most common in use today and while they may have slight differences in the sequence of operations to data and/or the representation of the coordinate systems used to represent the 3D world, the end result is the same.

2.1.1 Application

This section is the responsibility of the developer as it is executed on the CPU. Here, a developer organizes the data to be rendered as a collection of polygons (which in turn are composed of vertices in a virtual 3D world), chooses the various algorithms with which to manipulate this data, collects and processes any external events that relate to the application (e.g. keyboard presses and mouse clicks) and ultimately ensures that the outputs of this stage are valid rendering primitives i.e. points, lines and triangles. These primitives *could* eventually end up on the screen.

2.1.2 Model View Transform

Here, the rendering primitives undergo a series of transformations that change the underlying coordinate frame in which they are defined. Since individual objects are defined in their local coordinate spaces (object space), it is necessary to place them in the global world frame (object positioning). It is important to note that simply placing the objects in the world frame does not make them visible. It is often necessary to orient the models relative to the camera before we can see them. The camera, which is also placed and oriented in the world space, establishes a view volume within which all objects are seen (assuming they are not occluded). This view volume has the shape of a truncated pyramid with a rectangular base. The orientations needed to position the objects relative to the camera are elegantly carried out using matrix operations. OpenGL combines the object-to-world and world-to-camera transforms under a single matrix called the model-view matrix. In Direct3D, these two matrices are separate and are individually applied to the model. The following figure shows the transformations that take place in this section of the pipeline.

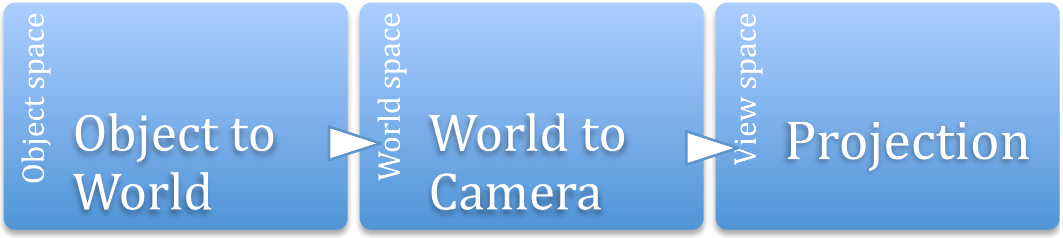


Figure 2 The transformations a model undergoes under the vertex stage.

Lighting is typically performed in view/camera space after the view matrix has multiplied all entities in the world.

2.1.3 Projection

This section performs a projection on the output of the previous section. Projection has the effect of transforming the view volume into a unit cube bounded by [-1, -1, -1] and [1,1,1][2] often referred to as the canonical view volume. There are many different kinds of projections but two commonly used ones are orthographic/parallel and perspective projections. The figure below shows an orthographic projection. Note how parallel lines remain parallel after projection.

[2] Direct3D maps the z values to [0, 1] instead of [-1, 1] as OpenGL does.

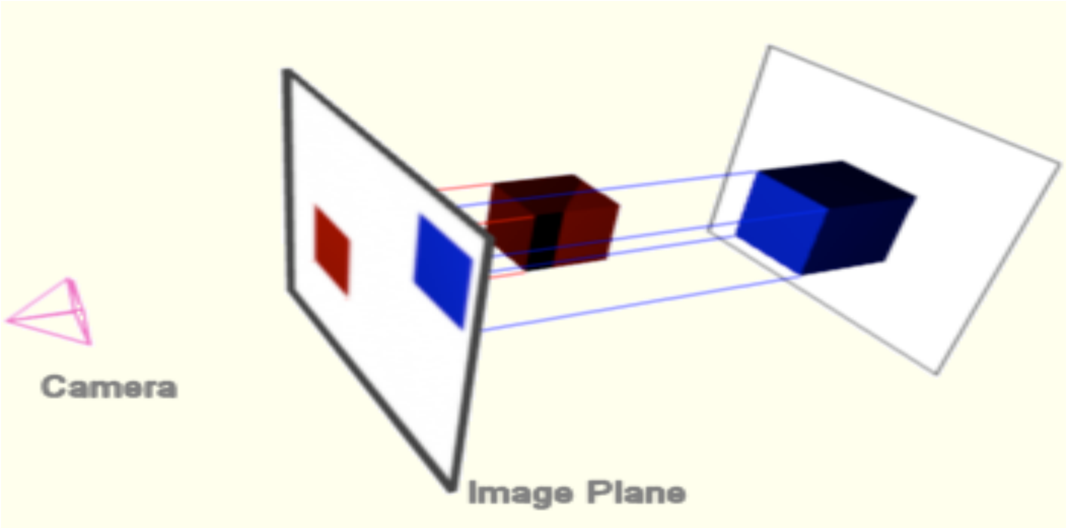


Figure 3 Orthographic projection projects parallel lines to parallel lines. Image courtesy of Burke [6]

Perspective projection mimics the way we perceive objects; the further out they are, the smaller is their projected size. Parallel lines in perspective projections seem to converge at the horizon. Figure 4 shows a perspective projection. Note that both projections project models from three dimensions to two dimensions.

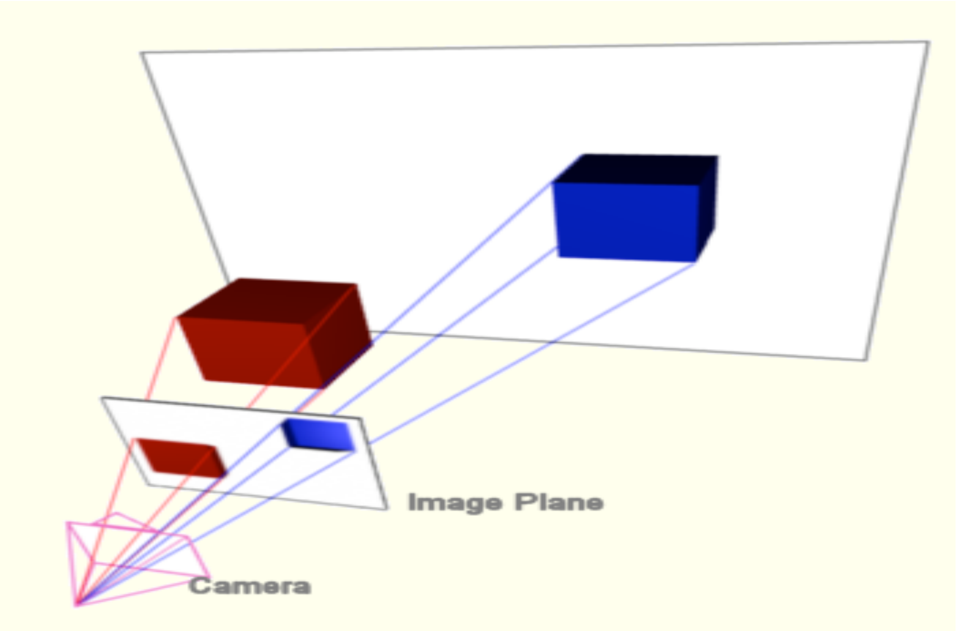


Figure 4 Perspective projection. Image courtesy of Burke [6]

2.1.4 Clipping

After the projection is performed, the vertices are said to be in clip space. Primitives that partially lie within the view volume require clipping to avoid possible division by zero (when the polygon lies on either side of the view plane and a vertex has a z coordinate of 0) or negative inversions. Clipping in this space is easier as polygons are trivially clipped against the unit cube.

2.1.5 Viewport mapping

After the primitives have been clipped, this stage maps the primitives to the screen coordinates (screen space) by translating and scaling the polygons by an appropriate matrix. The output of this stage is fed into the rasterizer, which is responsible for converting from two-dimensional vertices in screen space into the pixels on the screen.

Note that this simplified rendering pipeline has a lot more going on than has been described here. The next section briefly highlights the changes that have been made to the fixed-function pipeline.

2.2 Vertex and Fragment Processors

The above pipeline, while common to many rendering systems, became harder to use **consistently** especially as features multiplied in commodity graphics hardware. This is because these features were accessible only via a limited set of settings and switches, which also had to increase in complexity in tandem with the hardware complexity. The result was a complex API that was unable to fully represent the full flexibility of the new hardware at best and at worst, a complex interplay of the said settings and switches, which often interacted in confusing ways [7].

This led to the creation of 'shaders'; application provided code that runs on the graphics card, which replaced several of the most important fixed-function stages. Vertex and fragment (pixel) shaders, the subject of the next sections, afford developers the most flexibility in using the provided feature set and in directly controlling the operations that are applied in various stages throughout the rendering pipeline.

2.2.1 Vertex shader processor

This stage manipulates vertex data such as position, color and texture coordinates and cannot create new vertices. It is used to add special effects to objects in a 3D environment. Its output feeds right into the rasterizer. This corresponds to the combination of the model view, projection, clipping and viewport mapping in the fixed function pipeline. Note that since it replaces the model view fixed function stage, this processor is invoked on application-supplied vertices.

2.2.2 Fragment shader processor

This stage computes color and other pixel attributes. It is invoked on dynamically generated fragments and thus has no concept of application provided per-fragment attributes. Since this manipulates pixel attributes, a lot of image space algorithms are implemented using fragment shaders. Bump mapping, specular highlights, ***shadows*** and translucency are some of the things that can be easily implemented by a fragment shader.

CHAPTER 3

CURRENT SHADOW ALGORITHMS

Visible surface algorithms (VSA), as their name implies, are concerned with determining what geometric objects are visible from the camera's viewpoint. If an object A is in front of another object B along the line of sight of the camera, A is said to **occlude** B. and B is **occluded** by A. Shadow algorithms determine which faces can be "seen" from the light source. Thus shadow algorithms and visible surface algorithms are essentially the same [8]. Surfaces that can be seen from the light source are not in shadow while those that are occluded from the light are in shadow. The significance of the two algorithms being similar is that we can employ some algorithms from VSA in determining shadowed regions. In fact, one of the algorithms reviewed here does just that with successful results.

This investigation only considers shadow algorithms for point-light sources. Point light sources are sources without an areal extent and whose light irradiance emanates in all directions from a single point. Even though they do not exist in practice, they are easy to approximate and compute their effects in real time. We shall adopt the definition used by Moller et al and define real time as being 15 frames per second [9] or more.

What follows is a distinction between hard and soft shadows, after which we delve into the currently used shadow generation algorithms.

3.1 Hard shadows vs. soft shadows

As point light sources are implemented as points in space, objects either have a direct line of sight to the point light or they are occluded and no line of sight exists between the two. This results in a point being in shadow or being lit. This bivalent distinction naturally gives rise to hard shadows as depicted in the figure below.

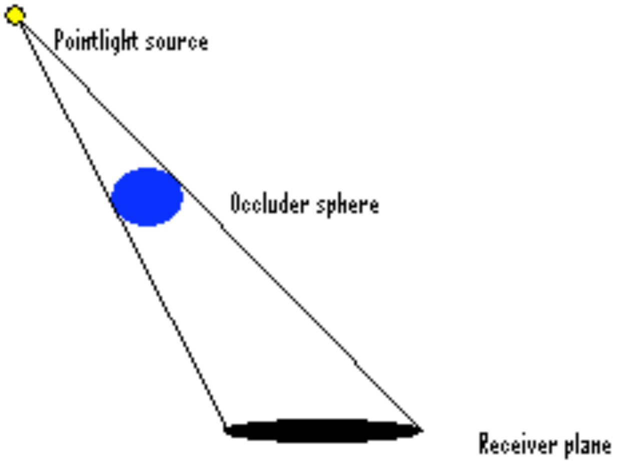


Figure 5 Hard shadows caused by point lights.

As can be seen from this figure, the receiver, which is defined as an object upon which a shadow is cast, is either fully lit or fully shadowed. Contrast this with soft-shadows, which more closely resemble the way shadows are cast in real life. Real light sources have a finite area from which irradiance is emitted. This extent has a perceptible influence on the geometry of the shadow projection as each point on the area light source contributes to the overall 'sum' of shadows on a receiver. In fact, this view of an area light source as a collection of point light sources has been used to generate soft shadows with much success.

The region of the shadow that is completely blocked from the area light source is referred to as the **umbra.** Partially illuminated areas on the receiver are called **penumbrae.** The following figure shows the same lighting setup with the point light replaced by an area light source.

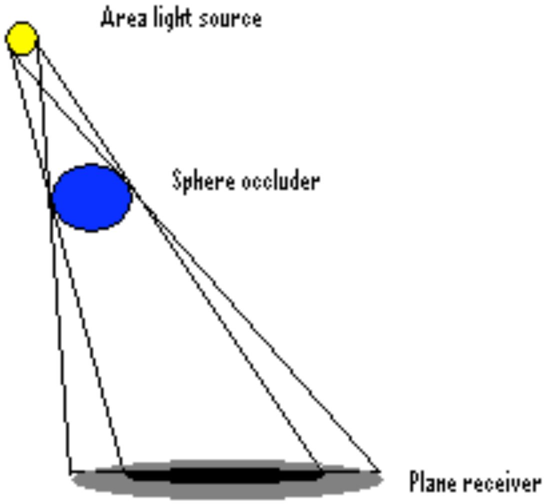


Figure 6 Area light source. Note the distinct shadow regions on the planar receiver. The dark region is the umbra while the grey region the penumbra.

Notice how the lighter shaded region encompasses the umbra. This is a direct result of the finite extent of the light source as the image shows. Area light sources are more challenging to model and the effects achievable by them are also a lot more compute intensive as compared to point light sources. For this reason, when real-time performance is expected, most applications use point lights instead and employ tricks to simulate area light sources.

Since soft shadows are generated **after** hard shadows have been generated, this study focuses on algorithms that produce hard shadows. Hasenfratz et al [10] provide an excellent survey of real time soft shadow generation algorithms for commodity graphics hardware.

We now proceed to look at the various shadow generation techniques that are currently in use. The first algorithm is simple to implement and has therefore been widely used. The remaining ones require special hardware features to implement but are also widely adopted.

3.2 Projection/Planar Shadows

Blinn [11] proposed a simple way to generate 'fake' shadows from a point or directional light. In his approach, an object's vertices are projected on a plane. Figure 7 below shows this method.

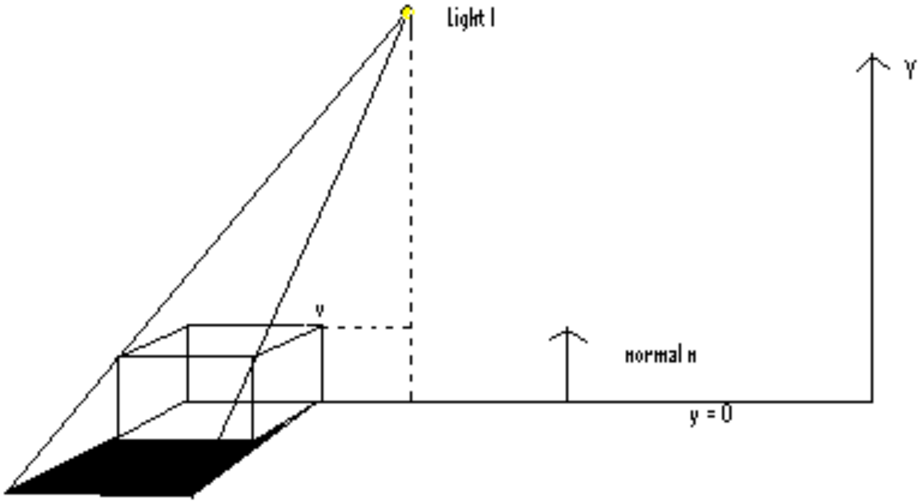


Figure 7 The projection of vertex *i* is analytically determined to obtain a shadow image on the ground plane.

In the above figure, the light source is located at position $L = (xl, yl, zl)$ and casts a shadow onto the plane $y = 0$. The similar triangles method is used to analytically solve for the point of projection as follows;

Let p be the projected point of vertex v (on the $y = 0$ plane). Then from similar triangles,

$$\frac{px - lx}{vx - lx} = \frac{ly}{ly - vy} \qquad (1)$$

$$px - lx = \frac{(vx - lx)ly}{ly - vy} \tag{2}$$

$$px = \left[\frac{(vx - ly)ly}{ly - vy}\right] + lx \tag{3}$$

$$px = \frac{lyvx - vylx}{ly - vy} \tag{4}$$

The z coordinate is obtained in a similar way. With the three projected vertex positions ($y = 0$), we get a new shadow vertex. This is done for all the vertices of the object and for all the lights in the scene. This process can be generalized to a matrix multiply which acts as a projection matrix M. The benefit of using a matrix is that graphics cards are highly optimized for matrix multiplication. Thus, the penalty associated with this shadow planar projection turns out to be minimal. Assuming a left handed system, this matrix M generalizes to

$$Mpo\text{int}-shadow = \begin{pmatrix} ly & 0 & 0 & 0 \\ -lx & 0 & -lz & -1 \\ 0 & 0 & ly & 0 \\ 0 & 0 & 0 & ly \end{pmatrix} \tag{5}$$

To project to any plane, we simply solve for matrix M that projects a vertex v down to p. Note that the above matrix requires a division by w (homogeneous division) to yield the correct vertices.

In most cases, we want to project shadows on the ground plane since lights are normally situated above objects in most scenes. However, we can use this projective idea to cast a shadow on any plane given by the normal-point form plane equation.

To render the shadow, we simply select the shadow casters and apply the above matrix on them. The resultant shadow object is then drawn with a dark color and no illumination. To avoid rendering the shadow below the plane, we draw the ground plane first, and then the shadow polygons with depth buffering turned off. This way, no depth comparisons are made and the shadow rests neatly on the surface of the receiver.

Note that we can also use this method to generate shadows from directional lights[3] The above derivation assumed a position for the light and for each vertex of the shadow caster, we determined the projected shadow vertex using similar triangles. For directional lights, we simply substitute the position for the direction of the light’s rays as shown below.

Given a point (vertex) on the object $P = (xp, yp, zp)$ and a directional light $L = (xl, yl, zl)$, the point will cast a shadow at $S = (xsw, 0, zsw)$. The projected point is derived using the implicit equation of a line $S = P - \alpha L$. Since $y = 0$, we can solve for the unknown alpha and use it to derive the other projected positions.

$$S = P - \alpha L \tag{6}$$

$$0 = Yp - \alpha Yl \tag{7}$$

$$\Rightarrow \alpha = \frac{Yp}{Yl} \tag{8}$$

$$xsw = xp - \left(\frac{Yp}{Yl}\right) xl \tag{9}$$

$$zsw = zp - \left(\frac{Yp}{Yl}\right) zl \tag{10}$$

Again, we can generalize this construction into a matrix M given by

$$Mdirshadow = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{-xl}{yl} & 0 & \frac{-zl}{yl} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{11}$$

[3] Directional lights are located at infinity and thus have no position per se. They are characterized by the direction of illumination and their rays are assumed to be parallel. The sun is a perfect example of a directional light.

This construction assumes a left-handed coordinate system as well. Note that the projected shadow from a directional light does not extend out as far as that from a point light. The following figure shows this concept.

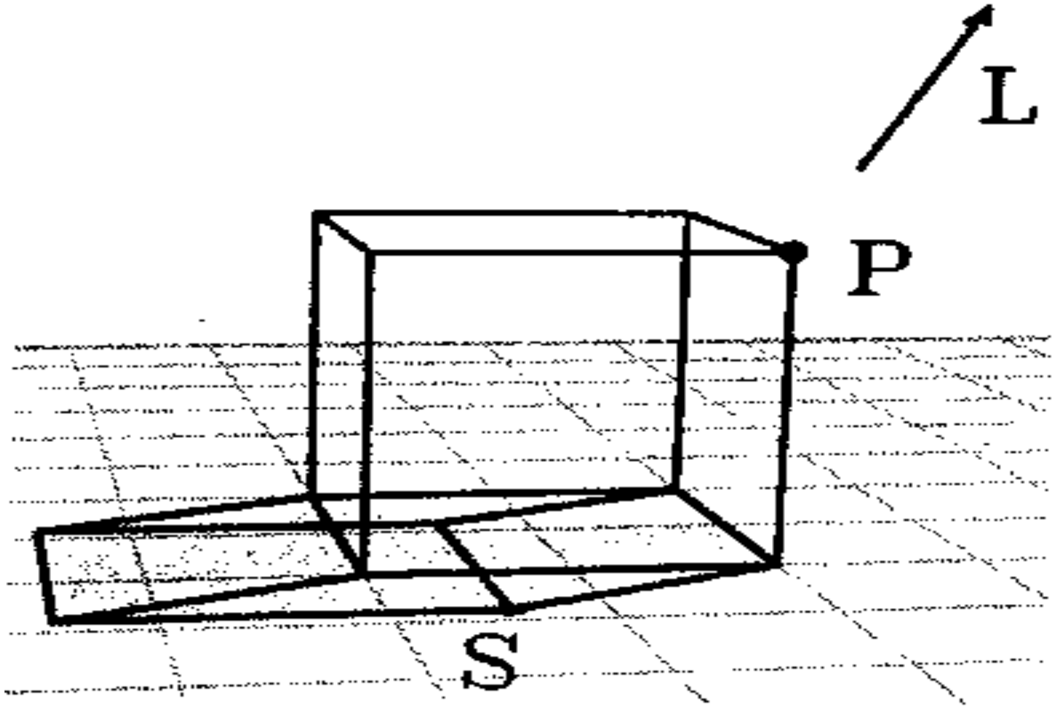


Figure 8 Planar shadow from a directional light. Image courtesy of Chris Bentley [12]

The figure below shows the shadow generated from a point light source. Notice the extent of the shadow on the ground plane.

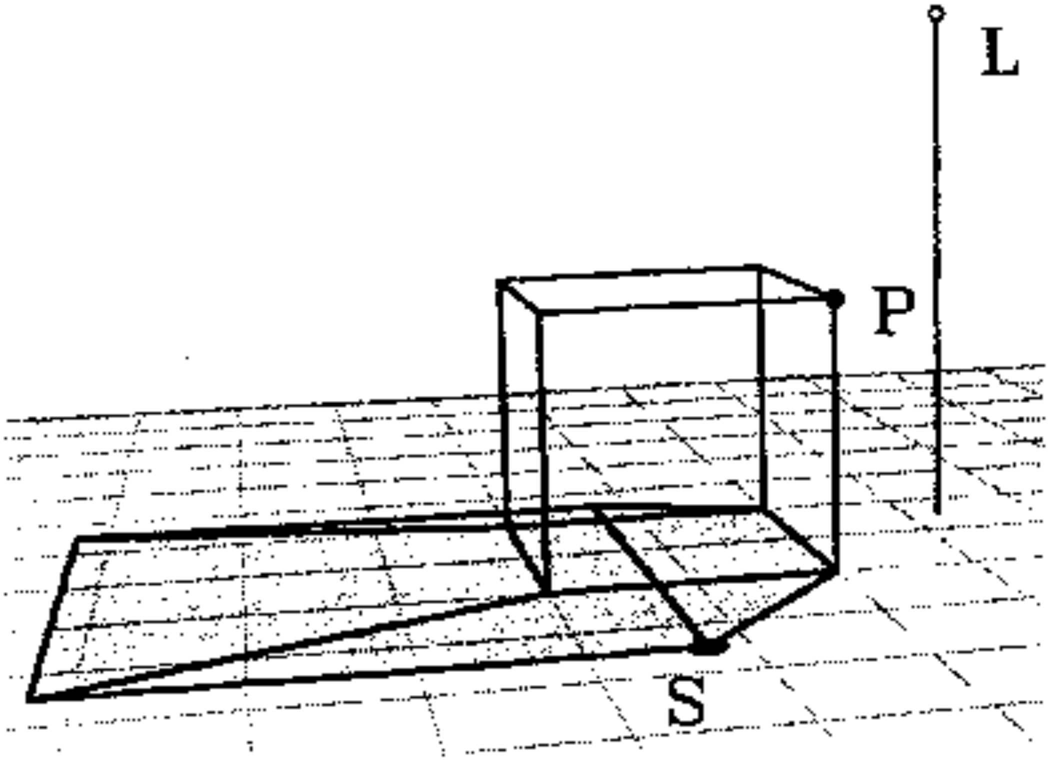


Figure 9 Planar shadow from a point light source. Image courtesy of Chris Bentley [12]

3.2.1 Pros and Cons

The planar shadow algorithm has been a favorite amongst developers for many years as it is easy to implement and results in shadows that are realistically cast on the planes. The disadvantage with this method is that shadows can only be cast on planes. This has the effect of limiting the realism in the scene as shadows in real life are cast on arbitrary geometry.

It is worth mentioning that further processing of the cast shadows can be done to blend them in with the environment. It is indeed possible to generate soft shadows by using planar projections as Heckbert and Herf [13] demonstrated. Gooch et al [14] also use a variant of planar shadows to obtain soft shadows. Both methods require multiple projections per object, which necessarily reduces the frame rate.

3.3 Shadow Mapping

This algorithm, proposed by Williams [15] in 1978, is a direct extension of the visible surface determination method applied to a light source. It is also known as the Z-Buffer shadow algorithm for reasons that will be clear shortly. It works as follows:

From the light's point of view, what is 'seen' is illuminated while the areas that are not seen (occluded areas) are in shadow. The technique requires two passes; the first pass renders the scene from the light's perspective and stores the visible areas depths in the z-buffer. This depth corresponds to the distance to the light source for those visible regions and is stored as a texture (also called a shadow map). Texture coordinates for the objects in the scene are then computed and associated with the object's vertices.

The second pass involves rendering the scene from the viewer's point of view and for each pixel, comparing the interpolated texture coordinate depth with the depth stored in the texture map from the first pass. If the former is greater, then this means that something occluded this fragment from the light and so it must be in shadow. Fragments that do not index the texture map (those that fall outside the depth map) are also treated as illuminated.

Note that the depth/shadow map must be updated any time there are changes to either the light or the objects in the scene, otherwise the application would obtain an incorrect depth value. Since shadows are not view-dependent, the viewer can move about without necessitating an update of the shadow map.

In practice, first the shadow map is obtained as described above. Then the scene is rendered from the viewer using ambient lighting[4] only. This ensures that even the shadowed areas have ambient lighting[5] A shadow testing step is then performed, which

[4] Ambient lighting is lighting that comes from all directions. In graphics, it is an attempt to model the light that is still available even when light sources are turned off. Our ability to still perceive light on an overcast day provides an example of ambient lighting.

[5] This is normally the case as shadows are never really purely black regions. Ambient lighting therefore enhances the realism of the scene.

compares the z-value in the Z-buffer with the shadow maps z-value (The shadow map's z-value is transformed from the light's coordinate system to the viewer's coordinate system). For each pixel, a degree of occlusion is stored that is later used when the scene is re-rendered with the full lighting equation. This value spans the range [0,1] and dictates the blending factor used in blending the pre-lit (ambient lit only) and the fully lit fragment (after the full lighting equation). Note that in most commodity graphics cards, enabling bilinear filtering enables hardware interpolation on this value resulting in a much softer transition from dark to light. This also reduces edge aliasing (jaggedness)[16].

As the shadow map only stores depth values, lighting, texture fetches and updates to the color-buffer can all be turned off to speed things up in the first pass. Figure 10 below shows two methods of implementing shadow maps: the top row shows a standard map constructed in world coordinates and the bottom row shows a perspective shadow map, constructed in clip-space.

3.3.1 Pros and Cons

This method has a few advantages, which makes it quite popular. For starters, it can be implemented entirely using general-purpose graphics hardware. It is guaranteed to work on most if not all graphics cards. Secondly, creating the shadow maps is relatively fast. In fact, the cost of building the shadow map is linear in the number of rendered primitives and the texture access time is constant [17].

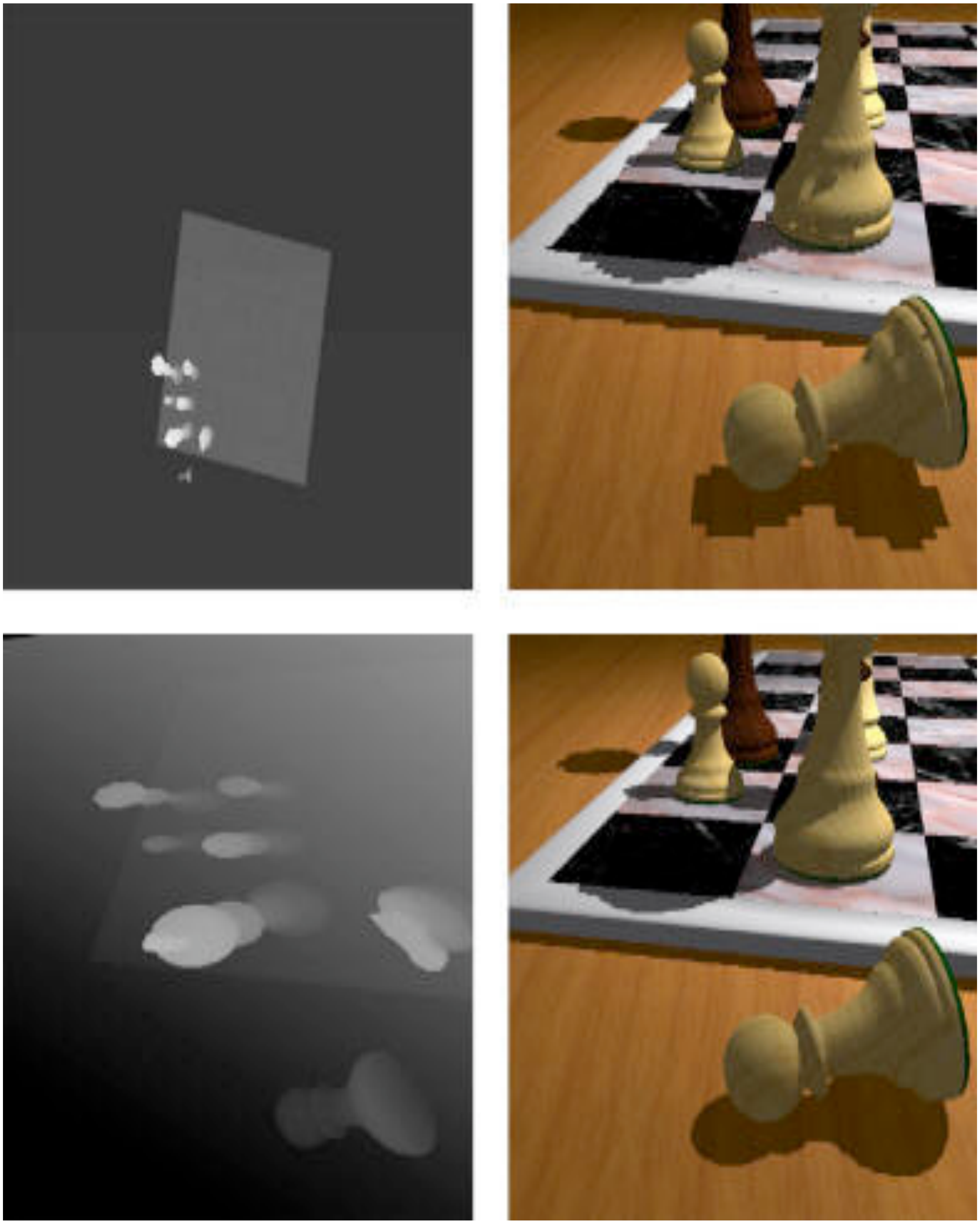


Figure 10 Perspective shadow mapping. On the left is the view of the scene from the light’s perspective. The white areas are closest to the light. On the right is the scene rendered with these shadow maps. Top row shows a standard shadow map while the bottom row shows a perspective shadow map.

This method also handles self-shadowing[6]. Since the scene and all the objects therein are shaded from the point of view of the light source, self-shadowing comes at no extra cost.

Despite these advantages, this method also has some drawbacks. First, even though most graphics cards support it, the quality of the shadows depends not only on the pixel resolution but also on the numerical precision of the Z-buffer. Therefore, it is subject to many sampling and aliasing problems, especially close to shadow edges. Recall that the shadow map is sampled per pixel during the depth comparison. This point sampling method is inherently imprecise and creates self-shadow aliasing in which a polygon is incorrectly considered to shadow itself [17]. The comparison fails because the light's stored depth value may be slightly lower than the surfaces depth value resulting in the classic under-sampling phenomenon known as moiré patterns. See figure 11 below.

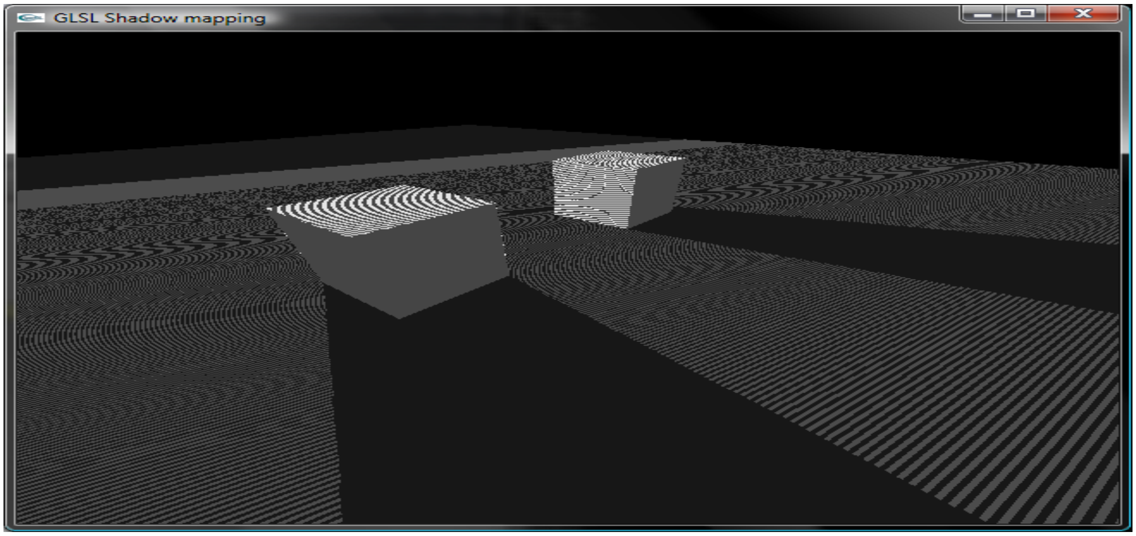


Figure 11 Moire patterns caused by depth inconsistencies between the stored light's value and the surfaces depth value.

[6] Self-shadowing allows non-static objects in a scene to cast shadows on themselves. This greatly enhances scene realism.

There are many suggestions for improving the aliasing problems discussed above. One idea as proposed by Hourcade et al [19] consists of storing object ID' in a priority buffer. Comparisons are then done against the stored ID's, lighting if they match and shading otherwise. They claim that this eliminates the self-shadowing problem.

Stamminger and Drettakis presented a different solution to reduce perspective aliasing on the generated output. Their method, perspective shadow maps [stammingger] performs the shadow map computation and the shadow test in normalized device coordinates after perspective transformation. After the perspective projection, the generated image is an orthogonal view onto the unit cube; therefore perspective aliasing due to the distance to the eye is avoided. Figure 10 shows the perspective shadow maps. The top row shows a standard shadow map generated in world coordinates from the light's viewpoint. The bottom row shows the perspective shadow map generated in clip space (post-perspective transform).

Percentage closer filtering as initially proposed by Reeves et al [20] has been used to demonstrate significant improvements on the generated output. This method has since been incorporated into commodity graphics cards.

Other ideas range from simply adding a bias to the light's stored value to setting the view frustums near plane as far away from the light source and the far plane as close to the light as possible (thereby increasing the precision of the Z-buffer).

A second disadvantage of this method is that shadow mapping cannot handle omni-directional lights. Since shadow mapping is built upon the premise of light 'looking at' a particular direction and then shadowing what is not visible, lights that 'look' at all directions would be infeasible to model with this technique.

Lastly, it requires at least two rendering passes (one from the light source and the other from the viewpoint).

3.4.Shadow Volumes

Heidmann [21] first implemented Crow's [22] original shadow volume idea in 1991 by using graphics hardware. This multi-pass algorithm can be thought of as being purely geometrical as it works by first determining the silhouette of the occluder as viewed from the light source, and then extending the pyramid formed by the light (apex) and the polygons edges to infinity. This extrusion of the polygon's silhouette along the light direction essentially forms a *shadow volume*. The basic premise is that all objects inside the shadow volume are considered to be in shadow and objects outside the shadow volume are lit. The figure below shows how the extrusions are performed starting from the light source and extending to infinity. The rectangle in this image is partially within the shadow volume formed from the silhouette of the sphere.

It should be noted that the shadow volume extrusion differs for different light sources. For point light sources, as the image below suggests, the extrusion diverges along the tangent of the silhouette edge in the lights 'view' direction. For directional lights, the extrusion converges to a point at infinity (recall that directional light sources have no position, simply a direction of illumination)

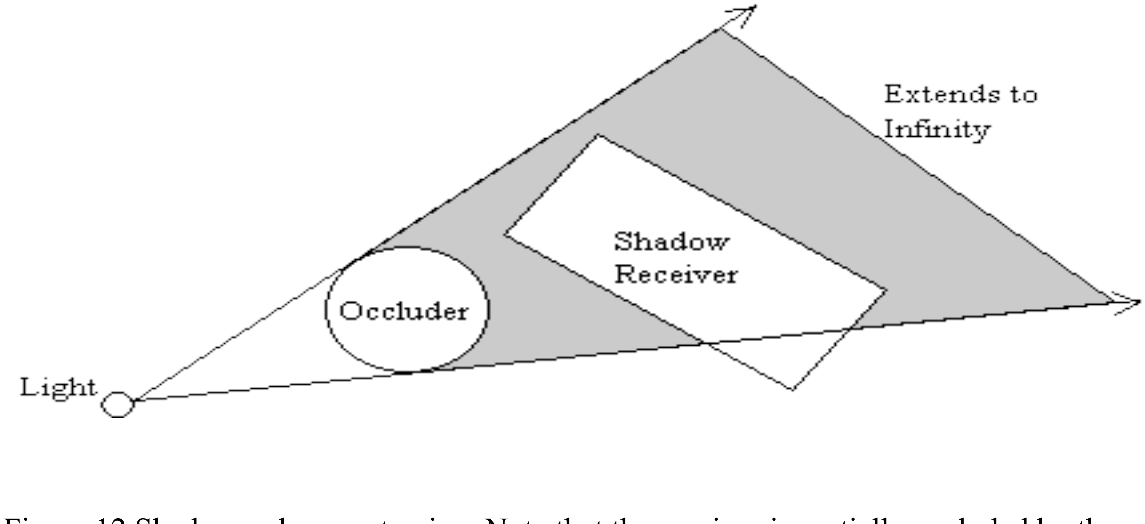


Figure 12 Shadow volume extrusion. Note that the receiver is partially occluded by the sphere upstream. Image courtesy Kwoon [23]

After the shadow volume has been generated, we can use the stencil buffer to keep track of whether an object is in shadow or not. To see how this works, consider viewing a scene and following a ray fired from the viewpoint to the object of interest (which should be displayed on the screen). While the ray is on its way to the object, we increment a counter each time it crosses a shadow volume face whose normal points to the viewer (front-facing face). This corresponds to incrementing a counter each time the ray goes into shadow. Similarly, we decrement a counter every time the ray crosses a back-facing shadow volume face. When we are done tracing the ray, the counter may have one of two values, zero or greater than zero[7]. If the counter is zero, the object is not in shadow whereas if it is greater than zero, it is considered occluded. The stencil buffer keeps track of the count and provides a simple way to check for occlusion. The figure below depicts this process.

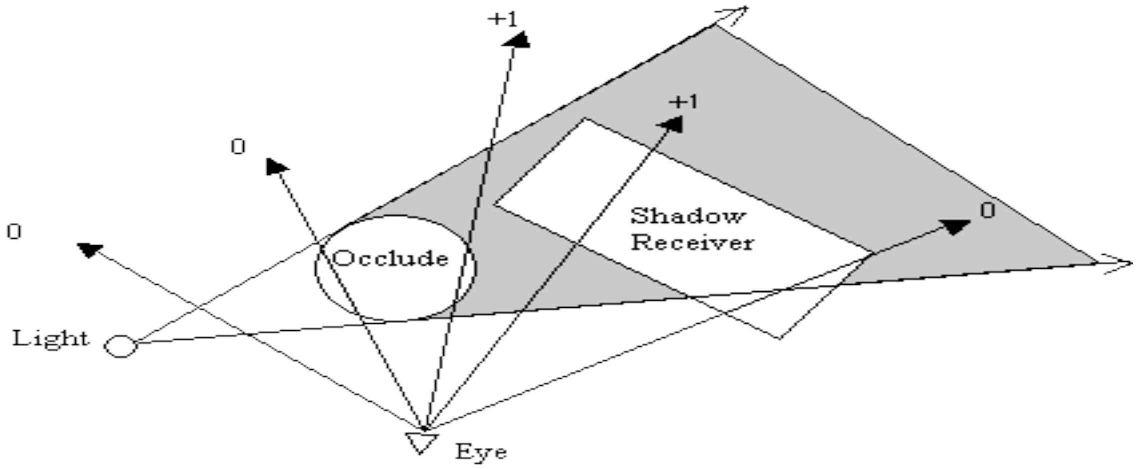


Figure 13 Stencil buffer counts. The fragments with a stencil count of zero are lit while those with a non-zero (but positive) count are considered in shadow. Image courtesy Kwoon [23]

[7] This is best-case scenario. If the viewer is located *within* a shadow volume, this does not work.

Note that the counting scheme works even there is more than one polygon that casts shadows. The actual shadow volumes are not rendered in the color buffer but rather in the stencil buffer and as previously mentioned, this happens in two passes. In the first pass, the front faces are rendered in the stencil buffer incrementing the count. In a second pass, the back faces are rendered decrementing it. Pixels in shadow are 'captured' between the front and back faces of the shadow volume, and have a positive value in the stencil buffer. This is also known as the z-pass method.

To render a scene using this technique, the following steps are followed:

- The scene is rendered with only ambient/emissive lighting[8].
- Then the shadow volumes are determined and rendered in the stencil buffer.
- The scene is then rendered illuminated with the stencil test enabled. Pixels with a stencil value of zero are updated while those with a positive non-zero value are left unmodified, keeping their ambient color.

The method just described works well for scenes in which the viewer is outside any shadow volume. As the figure 14 below suggests, the counts don't work if the viewer is inside a shadow volume. In this case, both rays end up with the wrong count. After exiting and entering a shadow volume, the left ray has an incorrect value of zero when it hits the object. This is because the stencil buffer was initially cleared to zero and therefore subsequent changes to the stencil buffer propagated this 'erroneous state'. In principle, the stencil buffer should be cleared to the number of shadow volumes the view starts inside (in this case, 1).

There is a far more elegant solution to this problem that was independently discovered by Bilodeau and Songy and by Carmack. This method reverses the counting order and starts by rendering **the back faces** and only incrementing the stencil buffer

[8] Emissive lighting is the self-illumination, which equally radiates from a surface in all directions. It is not dependent on the amount of ambient light in an environment.

should the **depth test fail**. The **front faces** are then rendered with the stencil buffer being decremented if the **depth test fails**. Tracing the ray backwards and using this approach indeed works as advertised. As an example on the ray from the left, after rendering the back face first, we find that the depth test fails (the ray is occluded by the object which has a smaller z-value) so the stencil buffer is incremented. The next two faces leave the count unchanged as the depth test passes for both, resulting in a stencil buffer count of 1.

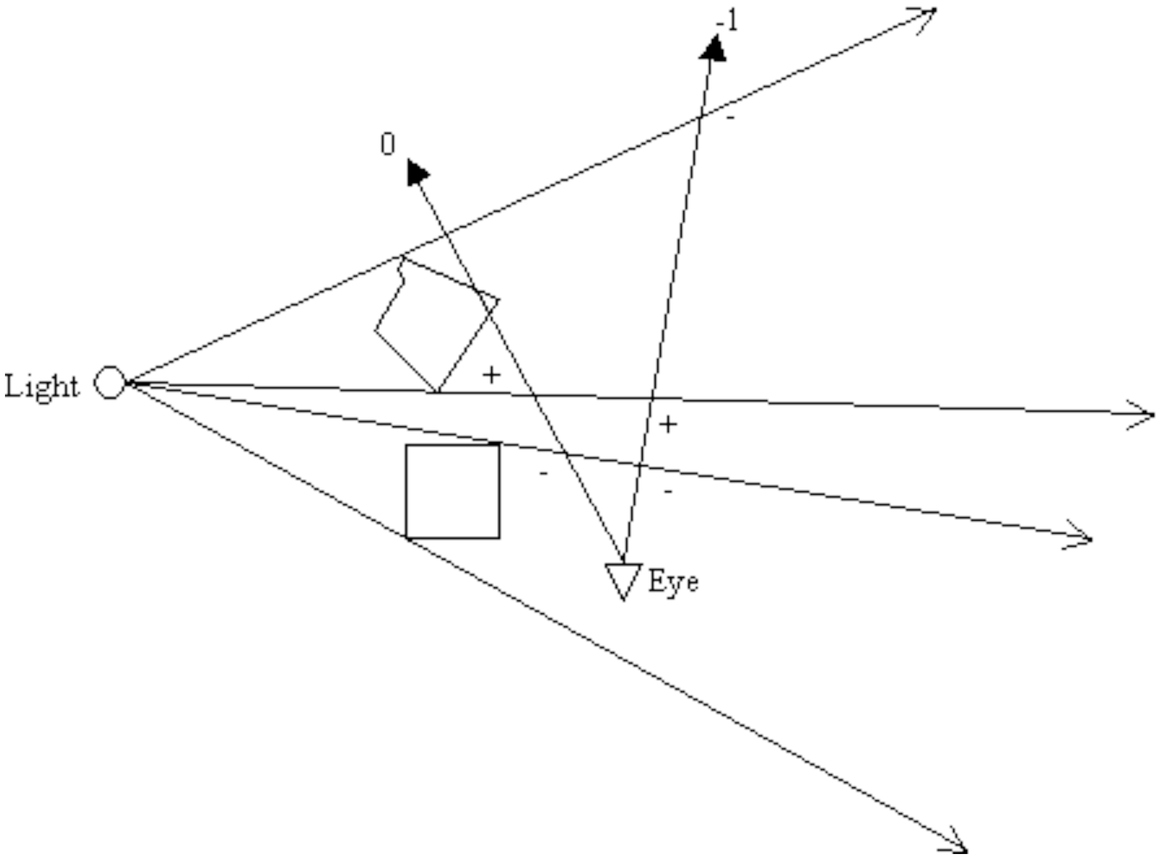


Figure 14 The z-fail method, also known as 'Carmacks reverse', works even when the viewer is insider a shadow volume. Image courtesy Kwoon [23]

3.4.1 Pros and Cons

This method has many advantages: First, it works for omni-directional lights. As it depends on extruded geometry to track whether a fragment is shaded or not, the light direction is of no consequence. This is an improvement over the shadow mapping method reviewed previously.

Secondly, it renders eye-view pixel precision shadows. It is not subject to the aliasing problems that plague the shadow-mapping method. This is a direct consequence of its construction. Lastly, it handles self-shadowing. All pixels on an object whose silhouette edges are being extruded and that fall within this shadow volume are not illuminated, resulting in proper self-shadowing.

While this method has garnered quite a following, it has its drawbacks as well. Most important of these is the fact that the computation time is highly dependent on the complexity of the occluders. Arbitrarily shaped objects take more time to extrude the edges for proper operation of the algorithm.

Perhaps tied to the first disadvantage is the pre-computation of the silhouettes required for the occluders. Note that this happens every time the geometry changes or the light changes. This determination of the object’s silhouette edges is compute-intensive and takes place on the CPU. The effect of this is a reduced frame rate.

Also, like the shadow mapping algorithm, this method requires at least two rendering passes, which again reduce the achievable frame rate.

On the hardware, rendering the shadow volume in the stencil buffer consumes the fillrate[9] of the graphics card. This necessarily also reduces the achievable frame rates.

[9] Fillrate refers to the number of pixels a video card can render and write to video memory in a second. Source Wikipedia.

3.5 Alternative methods

While the afore-mentioned algorithms are widely used, they are not the only way to generate shadows in a scene. Ray tracing determines the visibility of surfaces by tracing imaginary rays of light from the viewer's eye to the objects in the scene [8]. First developed by Appel [24], this method can be used to generate impressive pictures of shadows in complex environments. It has been used to generate shadows from point light sources to extended light sources with stunning results albeit at much slower frame rates. Radiosity methods model light sources as light emitting surfaces without any constraints on their geometry. These methods also support extended light sources (area light sources) and have also been used to produce visually stunning shaded scenes. These methods are collectively referred to as 'global illumination' methods as they take into account the indirect reflection and transmission of light at a point in determining the point's color [8]. Such methods fall outside the purview of this work and are not further investigated. For the interested reader, some excellent resources on this topic include [8], [25], and [26.]

CHAPTER 4

SHADOWS ON THE TEST GRAPHICS CARD

4.1 Problem Statement

4.1.1 The what

This thesis asks whether it is feasible to generate **good-looking** shadows **cheaply** (using the least amount of energy), in real-time and on a **constrained** graphics card as would be found on a low-end mobile device. As explained before, real-time frame rates are 15 frames/sec or more. For this study, good-looking shadows satisfy the following criteria, in the order specified:

1. The correct shadow placement on the scene.
2. The correct intensity.
3. The correct shape of the shadow

The correct placement of shadows means that an object should cast a shadow at or around the expected position relative to the caster and receiver. A shadow with the correct intensity and whose shape matches that of the caster would not be considered good-looking if it was well positioned on the receiver but several feet away from a caster that sits on a ground plane. In fact this would be confusing as the shadow could be misinterpreted for a separate object in the scene.

The correct intensity is ranked second provided the said intensity is sufficiently darker than the receiver or caster. A bright white spot that is correctly placed under a caster and on a receiver is neither immersive nor aesthetically pleasing and therefore does not meet the ‘good-looking’ objective.

As discussed earlier, Wanger [4] has demonstrated that the eye tends to overlook the shape of the shadow as long as a shadow exists. Having the correct shape of the object last in the above criteria allows for instance, a black circle that is correctly placed under a rectangular object in the scene to meet the ‘good-looking’ objective.

In this investigation, **cheap** refers to using the least amount of memory accesses to external memory. As Fromm et al demonstrates, external memory accesses are often the operation in a computer system that uses the most energy [29]. In low power processes, an off-chip memory access consumes more than an order of magnitude more energy than access to a small on-chip SRAM memory. As part of the objective is to minimize power consumption, algorithms that make heavy use of external memory are not considered cheap and therefore rank lower than those that minimize such accesses.

To get a sense for how **constrained** the test graphics card used in this study is, it is instructive to compare it to a typical desktop graphics card. The table below shows a comparison of the sections of a graphics card that would be useful in implementing image space effects e.g. generating shadows.

Test Card	**ATi Card**
1 Stream Processing Unit.	1600 Stream Processing Units.
1 Texture Unit.	80 Texture Units.
1 Stencil Pipeline.	128 Z/Stencil Pipelines.
1 Color Pipeline	32 Color Pipelines.
16 bit fixed point rendering.	128-bit floating point HDR rendering.
104700 Polygons/sec throughput.	850 M Polygons/sec throughput.

Table 1 Comparison of the test card with a typical graphics card (ATI Radeon™ HD 5870) [30].

The test card on the left is the card used in this study. The card on the right is a typical graphics card that would be found on a desktop computer.

As this table shows, the test card used does not have the resources that are typically available to developers for implementing complex shadow algorithms that require multiple passes. This limits the types of algorithms that can be employed to generate shadows. For instance, shadow mapping requires multiple passes to ‘cleanup’ the generated image and to reduce aliasing. While this works well on desktop graphics cards, these repeated passes may not meet the real time criterion outlined in the objective.

The lack of resources forces a compromise between **quality** and **quantity**. The quality of shadows generated on the test card will undoubtedly be inferior to those that can be generated on a desktop card. The idea is to trade this quality for increased performance, the latter of which is defined quantitatively. For purposes of this study, **quantity** has a two-fold definition:

1. Measured frame rates averaged over multiple runs (the higher the better).
2. Memory accesses to external memory (the lower the better).

While quality is a subjective term and therefore hard to quantify, the quality of the generated images is judged based on how well they meet the ‘good-looking’ criteria.

4.1.2 Methodology

To answer the question this thesis poses, the various shadow generation algorithms are run on a scene composed of a truck and a forklift, both of which are resting on a wooden platform. For all the tests, the scene is rendered from four different angles for two reasons: first, this showcases the algorithm’s qualitative results better (from all angles) and second, to get a better average cost per frame, the four runs are in turn averaged yielding a single rate that can be used for comparison to other algorithms.

For each algorithm, the sustained frame rates as well as the total accesses to external memory are recorded. Multiple runs are done per scene yielding high and low frame rate counts that in turn are averaged and it is this average that is recorded for analysis later. The same thing is done for memory accesses. The difference in frame rates and external memory accesses (high vs. low) is due to camera placement.

The conditions and assumptions for each run are given and finally the results are outlined, along with a brief summary of the pros and cons of the method. Finally, the individual results are merged for a tabular comparison of the various algorithms. It should also be noted that all the algorithms were tested on the same scene without employing back face removal[10].

What follows first is a formal introduction to the test graphics card that is used in the investigation.

4.2 The test graphics card

The card used is actually an emulation of the hardware in software. It has all the modules that the hardware should have and has been written to resemble the hardware exactly. This means that even in areas where software would have produced an efficient implementation, the design follows the hardware specification, which is outlined below.

- The use of fixed-point[11] arithmetic for all real-values.
- Two 16-bit depth-buffers (0.5 MB total).
- Two color buffers (24 bit color buffer with 8 bits left over for application use)
- A 2MB texture memory (external)
- Single rendering pipeline
- Hardware transform and lighting.

[10] Back face removal is the removal of polygons that are not 'facing' the viewer. These polygons will not be seen and therefore can be eliminated to speed up the rendering process.

[11] Fixed-point math like floating point math allows us to represent fractional values. However, in fixed-point, the underlying machinery is purely integer math!

The clock speed is not as important since the emulator is not cycle accurate. The depth buffer and color buffers are internal while the texture memory is **external**. The card also has access to video ram but like the texture memory, this is also external to the rendering core.

The planar projection method is the first algorithm tested. Its details are discussed below.

4.3 Planar projection on the test card

4.3.1 Planar projection test method.

First, all the object's colors are turned off (set to black) and the projective matrix applied to their vertices in world coordinates. These generated vertices are saved and await final rendering with the rest of the lit geometry. After rendering the receiver plane on which the caster rests, the depth buffer is turned off and the shadows are rendered. This eliminates z fighting and ensures the shadows are always resting on the receiver. Then the rest of the lit geometry is rendered after enabling the depth buffer.

The light used in this test is situated at $L = (x = 5, y = 45, z = -60)$ and revolves around the objects at $w = 2rads/frame$

4.3.2 Planar projection qualitative results

The first image below shows the scene rendered from the front, henceforth called the front face.

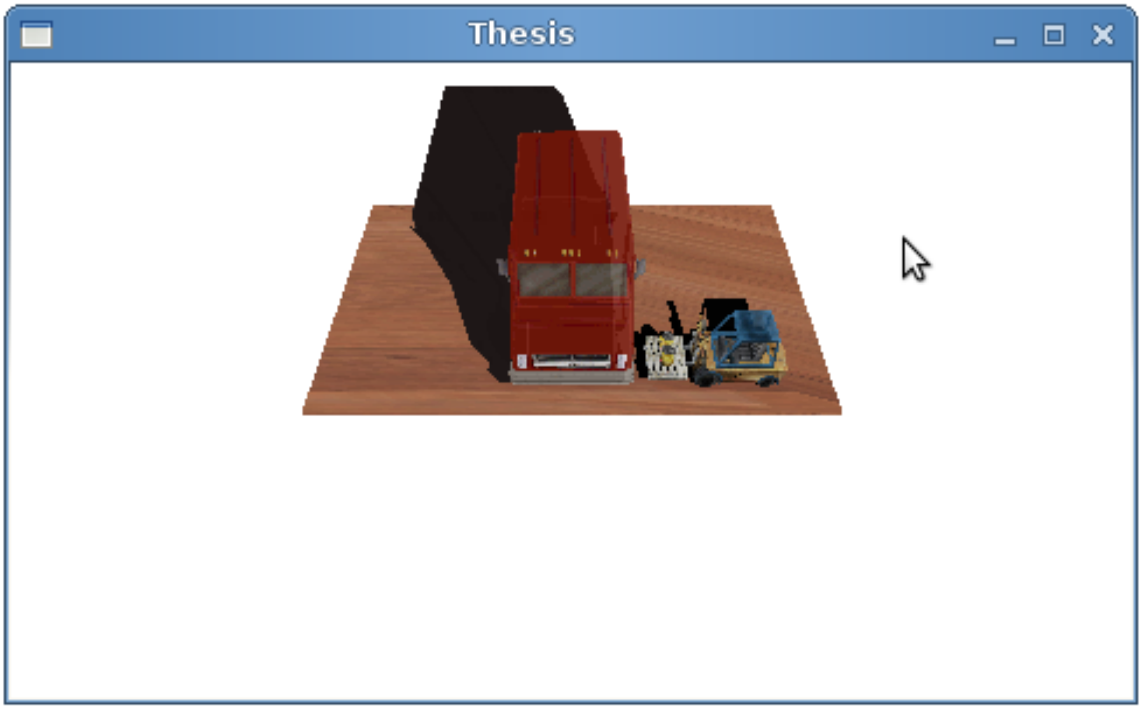


Figure 15 Planar projection 1. Front face screen-shot. Notice the overflow of the shadows on the ground plane.

In this scene, the light is positioned behind the objects and the shadows projected on the ground plane accordingly. Notice the overflow of the shadow on the ground plane. This shadow meets the ‘good-looking requirement as it is placed correctly and has both the tolerable intensity and the correct shape of the casters. While the overflow is a nuisance, it does not disqualify this shadow from consideration. This is an example of the compromise that was mentioned previously between quality and quantity.

The next image shows the same scene but the camera is now placed to the right of the objects (right face).

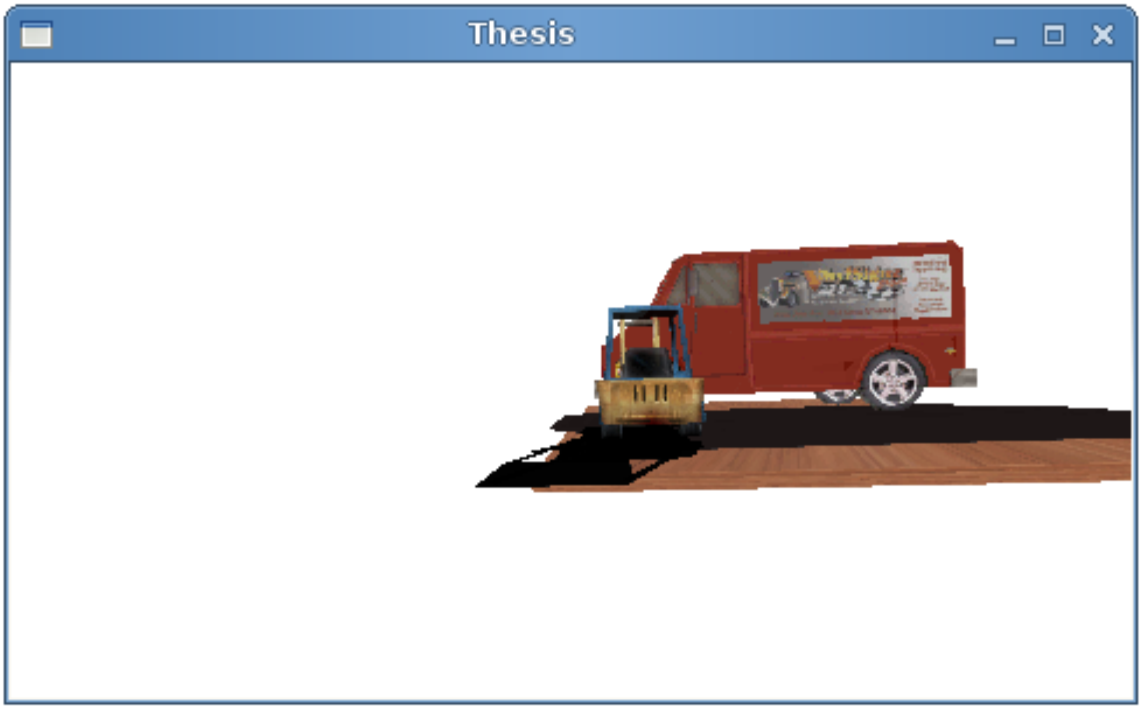


Figure 16 Planar projection 2. Right face of the scene.

The light is located above and to the left of the truck in this scene. Once again, we see proper placement of shadows and tolerable shadow intensity. The shape also looks correct even though it is hard to tell from the shadows position.

The next image shows the ‘back face’ of the scene. Here, the light is positioned above and to the right of the truck. In this image, the shadow overflow is more pronounced.

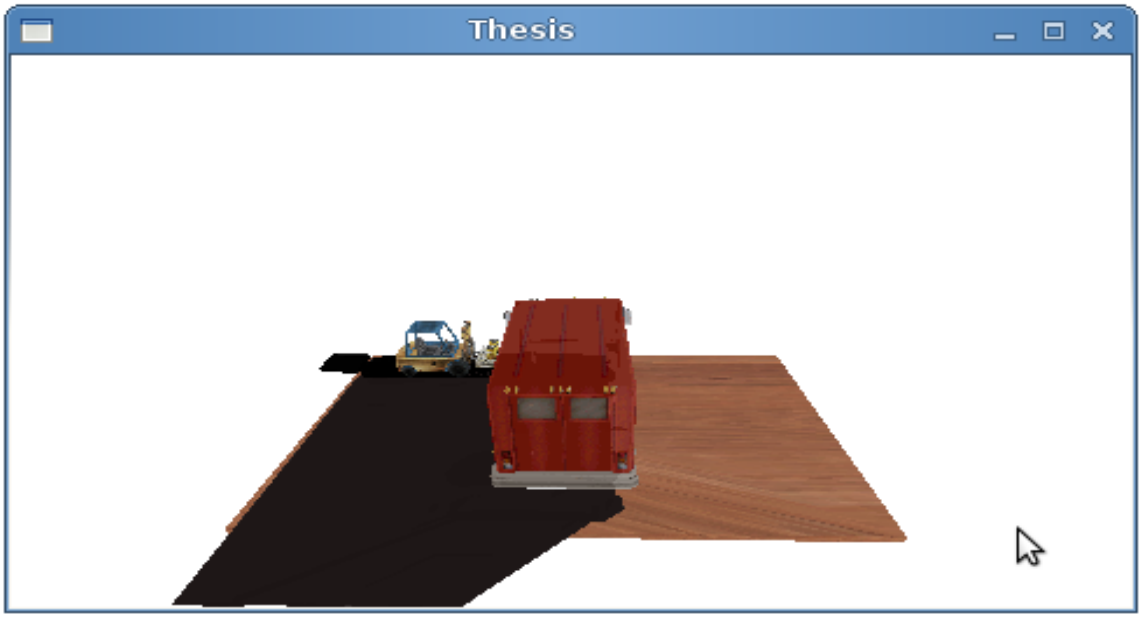


Figure 17 Planar projection (3). Back face of the scene. The light is to the right.

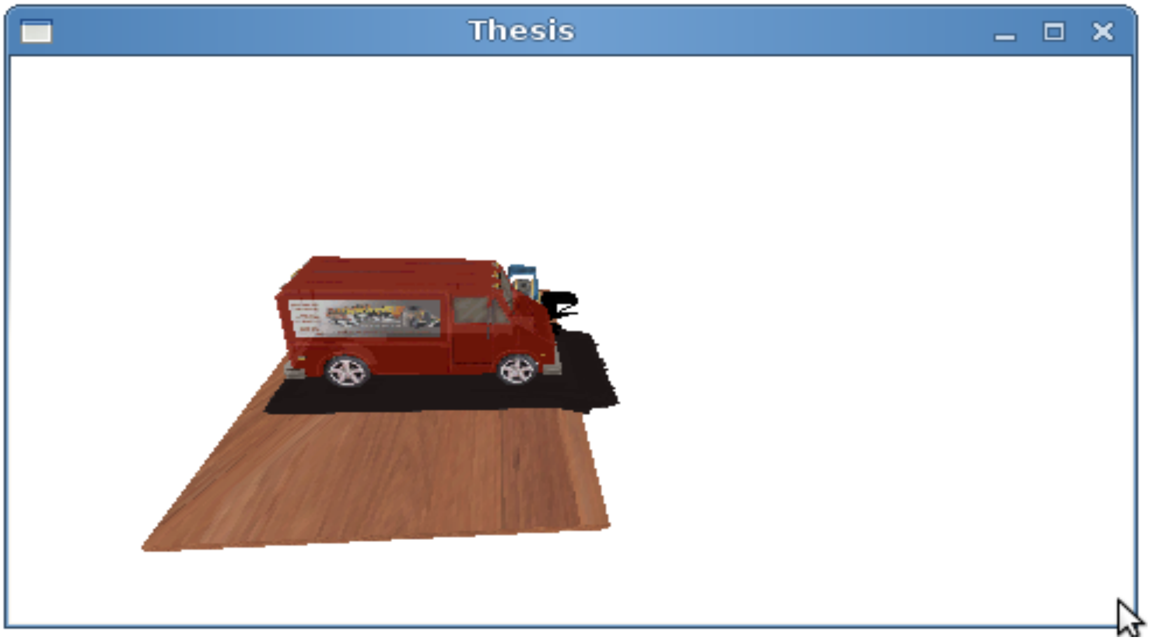


Figure 18 Planar projection (4). The left face of the scene. The light is behind the objects in the scene.

Figure 18 above shows the scene from the left side (relative to the front face). The light in this scene is above and behind the objects.

This next section discusses the quantitative results of running this algorithm. The number of polygons in the scenes shown is 3940.

4.3.3 Planar projection quantitative results

The table below shows the sustained frame rates, the external memory access count for all four views and their total averages for the scene. Since the planar method only accesses texture memory for the objects when the object color is being determined, it exhibits the least amount of texture fetches and should serve as the base case for the other algorithms. This also implies that it should run the fastest since accessing memory has some associated latency, and this method minimizes such accesses.

Planar Method	**Polygon Count**	**Highest Frame Rate**	**Lowest Frame Rate**	**Highest Memory Access Count**	**Lowest Memory Access Count**
Front Face	3490	30	28	33000	31985
Right Face	3490	33	31	31000	30670
Back Face	3490	33	30	32300	32000
Left Face	3490	32	31.5	32000	32460
Averages		**32**	**30.125**	**32075**	**31779**
			Scene Frame Rate Average		**Scene Memory Access Average**
			31.06		**31927**

Table 2 Planar projection statistics.

The table above shows that on average, there are roughly 32,000 texture access calls for this scene. The average sustained frame rate is about 31 frames/sec. The different scene faces show similar statistics with minor differences resulting from the inexact positioning of the camera.

4.3.4 Planar projection summary

This method non-discriminately projects all the objects onto a receiver plane. It does this using the least amount of resources and therefore runs fast. As mentioned before, the above results should be used as a base case with which the other algorithms are compared. It is also easy to implement, as it only requires a projection matrix, and the shadows it generates meet the established objective.

A drawback to this method is the restriction that shadows are cast on planes. This means that self-shadowing is not possible with this method, as the caster cannot cast a shadow on itself. Also the overflow of the shadows on the receiver can give the illusion that another plane exists outside the main receiver even when that is not the case.

Another drawback to this method is that the shadow has to be rendered for each frame, even though the shadow may not change (shadows are view-independent so their shapes do not change with different viewpoints).

4.4 The shadow mapping approach

4.4.1 Shadow mapping test method

First, the color buffer is disabled (writes to the buffer are disabled), and then the scene is rendered from the light’s viewpoint. The depth buffer resulting from this is saved as a texture in texture memory. Then texture coordinates are generated for all the objects in the scene and the scene is rendered normally. For every pixel, two texture fetches are done to resolve the color and the shade. A fudge factor (alpha) is used to ‘raise’ the light’s z-value (offset’s the stored z-value in the shadow map) to avoid aliasing. In these runs, it is set to $\alpha = 0.001$. This user-programmable value provides the best-looking results and was arrived at iteratively.

The light used in this test is also situated at $L = (x = 5, y = 45, z = -60)$ and revolves around the objects at $w = 2rads / frame$

4.4.2 Shadow mapping qualitative results

Since the shadow mapping algorithm works by shading what is not visible to the light, it should generate the most accurate shadows of all the algorithms. This suggests that from a qualitative standpoint, its results should serve as the base case.

This first image shows the front-face with the light situated at the back of the scene.

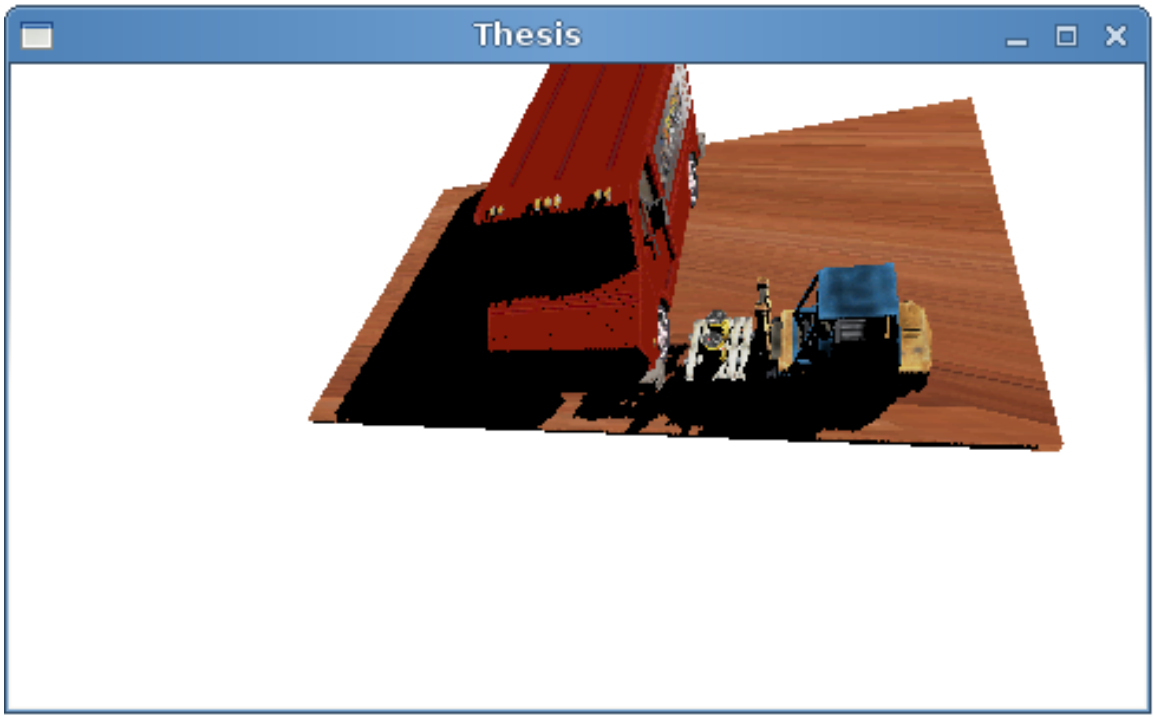


Figure 19 Shadow mapping front face (1). In this scene, the light is behind the objects and to the right. Notice the intricate shadows on the receiver plane.

Notice the correct self-shadowing of the objects in the above figure. This is one of the reasons this algorithm is widely used. The random dark spots seen on the truck show the inaccuracies associated with point sampling.

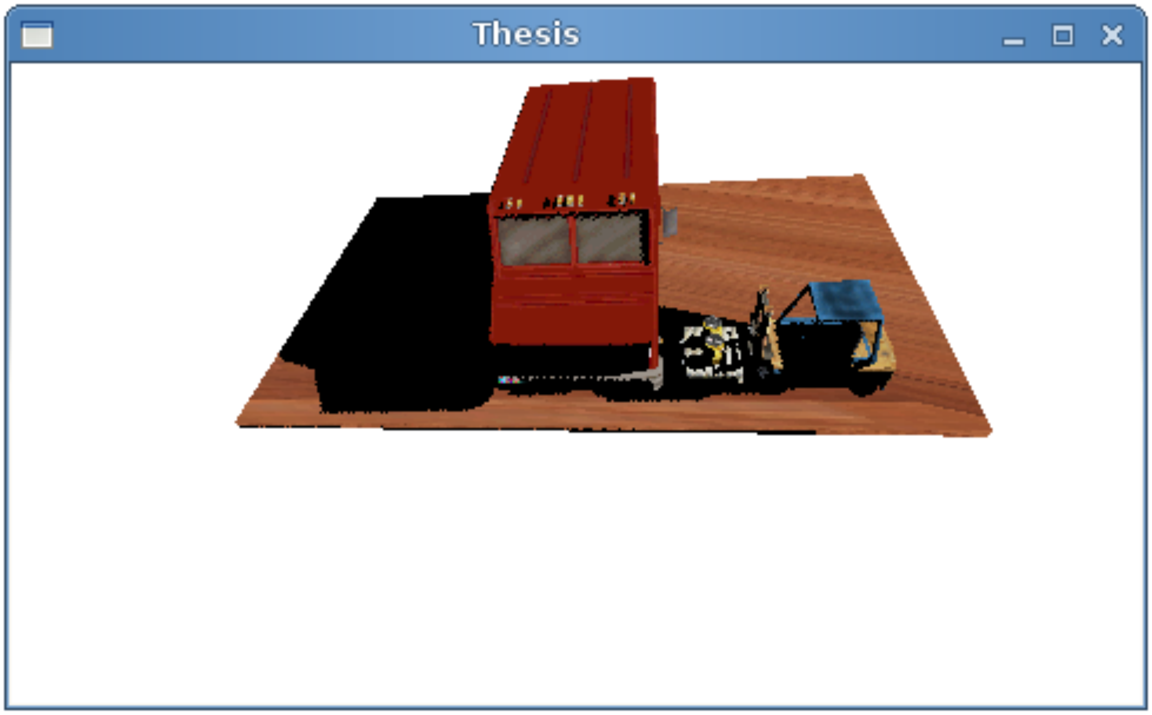


Figure 20 Shadow mapping front face (2). Notice the self-shadowing of the plane at the edges.

Figure 20 above shows a different front-face view with the camera slightly less elevated. In this figure, the light is in front of the objects and to the right. Notice that there is no overflow of the shadows on the receiver plane. This is a consequence of its construction; for all objects in the scene, only what the light does not 'see' is shaded.

The image below shows the right face scene. Here the light is positioned to the left of the scene and further back (to the right of the truck). This demonstrates correct shadow placement, correct intensity and the correct shadow shape. Also visible in this image is the jagged outline on the edge of the receiver plane, due to aliasing.

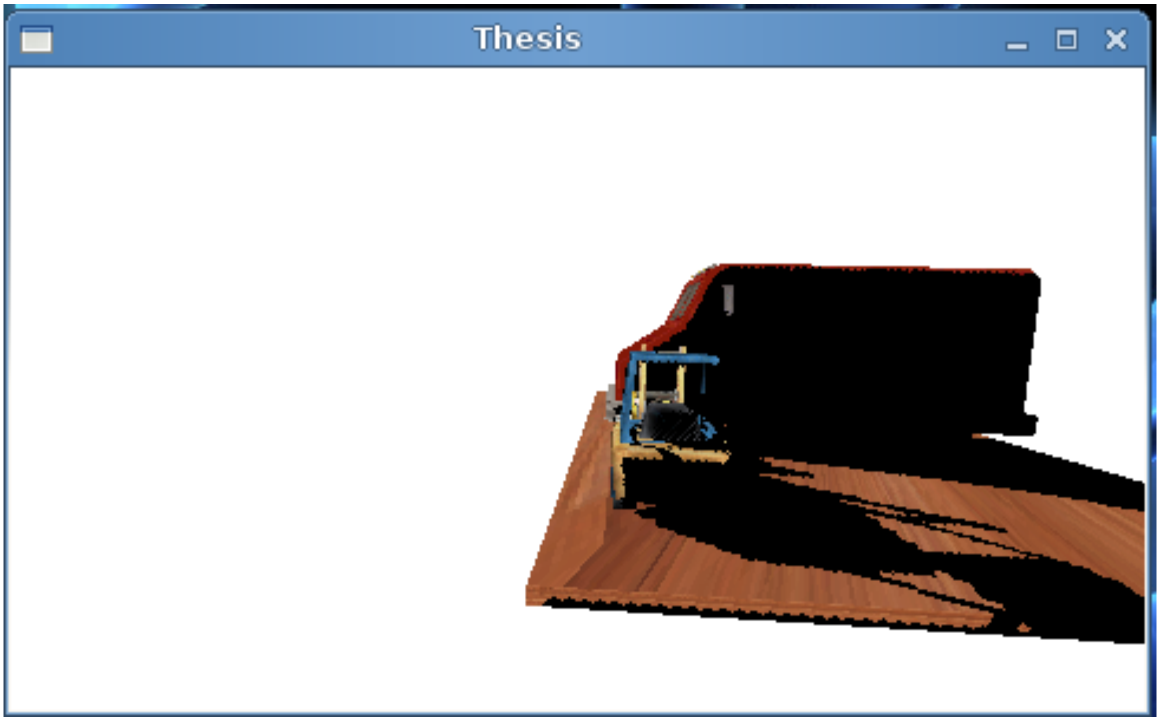


Figure 21 The right face scene. All three criteria are met as seen in this image.

The next two images show the back face view of the scene. Figure 22 shows a problem that is inherent in using shadow maps; shadow continuity glitches. The continuity problem occurs mainly when the shadow map quality changes significantly from frame to frame due to the motion of the eye or light. In this case, it is due to the light, which moves counterclockwise and is in front of the objects. Figure 23 shows the correct shadow placement and self-shadowing.

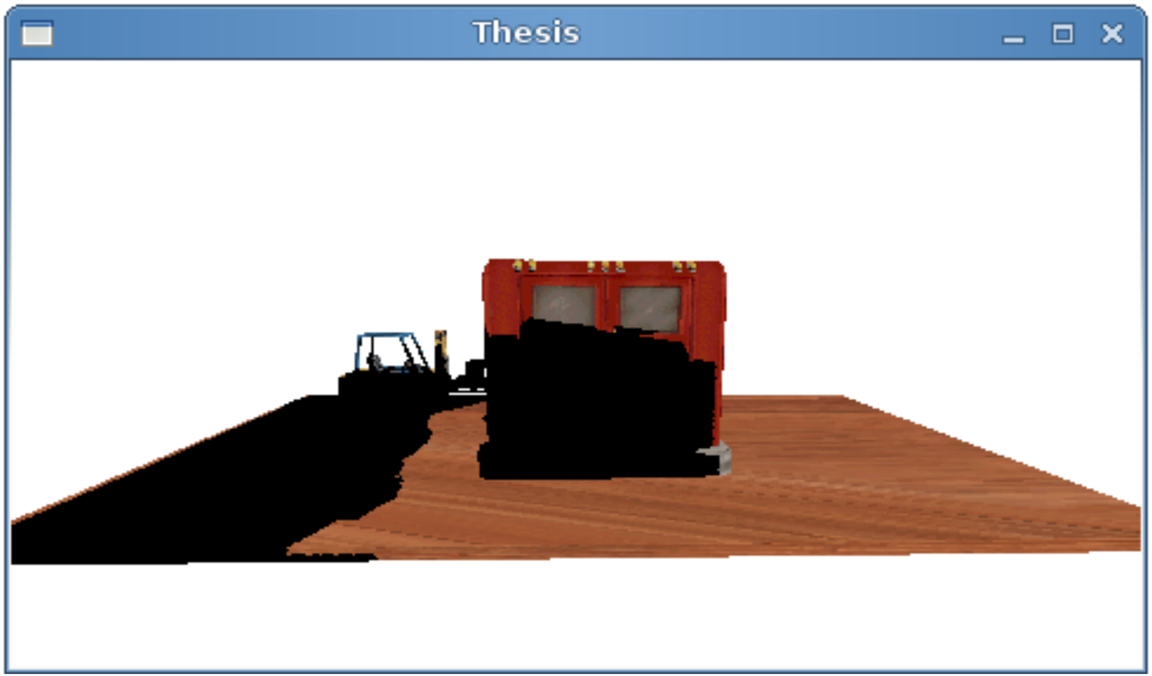


Figure 22 Back face scene (1) Notice the shadow continuity from the forklift shadow.

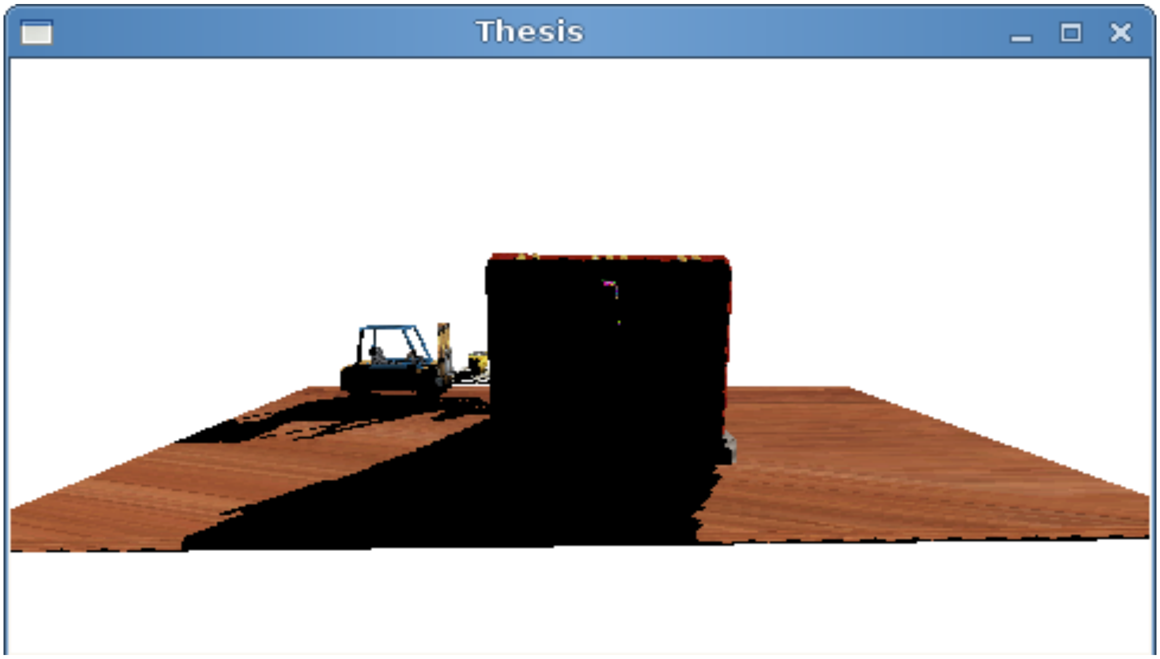


Figure 23 Back face scene (2). Proper shadow placement on both the receiver and shadow-casting object.

The next image shows the left scene screenshot. The left side of the truck is shadowed since the light is situated above and to the left of the truck.

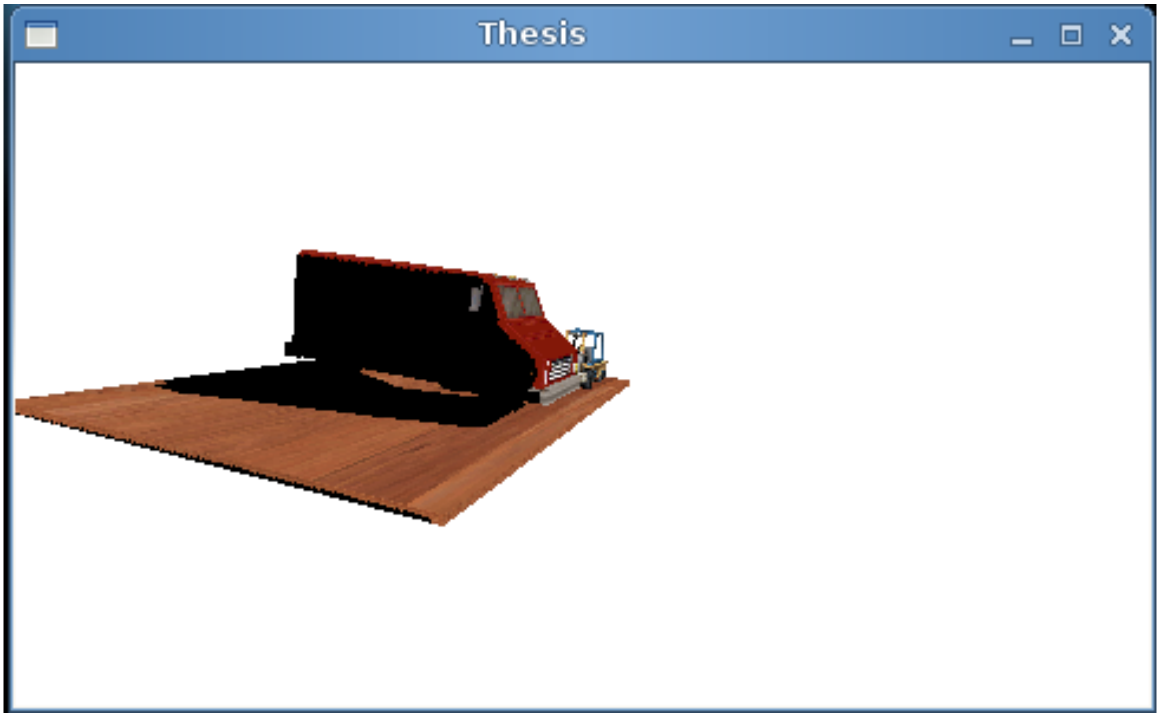


Figure 24 The left face scene. This image also shows proper placement and self-shadowing.

Despite the fudge factor, this next image shows that aliasing is still unavoidable. Notice the alternating dark bands on the side of the truck. There is also quite a bit of shadow continuity by the forklift.

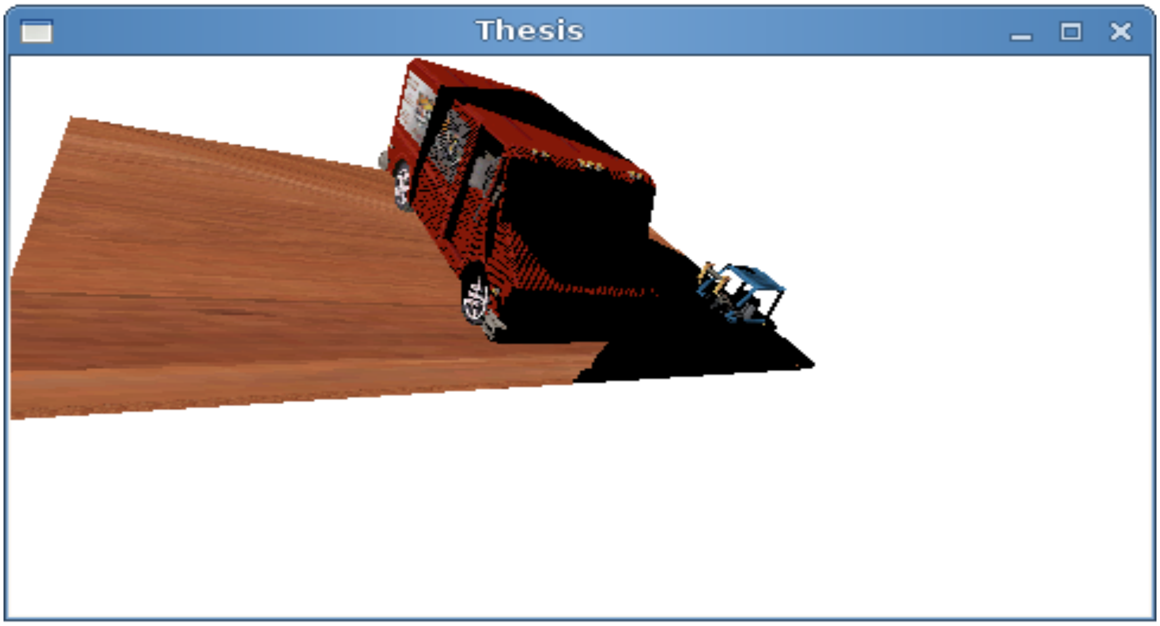


Figure 25 Shadow mapping . Notice the moiré patterns caused by aliasing.

4.4.3 Shadow mapping quantitative results

Shadow Mapping	Polygon Count	Highest Frame Rate	Lowest Frame Rate	Highest Memory Access Count	Lowest Memory Access Count
Front Face	3940	26	23.7	117288	107234
Right Face	3940	26	23	114185	109123
Back Face	3940	25.6	23.4	109370	103897
Left Face	3940	26.7	24	111985	103123
Averages		**26.07**	**23.53**	**113207**	**105844**
			Scene Frame Rate Average		**Scene Memory Access Average**
			24.8		**109526**

Table 3 Shadow mapping quantitative results.

This table shows that almost 110,000 texture fetches are made per frame. This is an increase by almost 3.5 times over the planar projection method. The frame rate has also dropped at 24.8 frames/sec compared to 31 frames/sec from the planar projection method.

Once again, the discrepancies in statistics collected between the different scenes may be explained by relative camera positioning. However, there does not appear to be a significant difference that would affect the reported results.

4.4.4 Shadow mapping summary

Shadow mapping generates shadows that easily meet the established criteria. In addition to that, self-shadowing, which enhances the visual realism of the scene, comes for free with this method. The fact that it can be implemented on this test card is perhaps a testament to its algorithmic complexity. While the frame rate dropped compared to the planar method, 24.8 frames/sec still meets the real-time criterion and is thus encouraging considering the objects were not back-face culled.

Having said that, it has a few drawbacks; First, this method is very expensive from an external memory access perspective. As the table shows, it makes 3.5 times as many texture calls as the planar projection method. This makes this method unattractive for any useful work on a graphics card that has the kinds of constraints that were presented earlier. Secondly, from a qualitative standpoint, the shadow continuity problem and the aliasing problem may be distracting in an interactive environment. A simple solution to the shadow continuity problem is to increase the shadow map size (texture memory), which may not always be feasible. The aliasing problem, which is caused by under sampling, has a natural solution of increasing the sampling per pixel. Again, this may not be a viable solution given the constraints of a limited mobile graphics card. The last major drawback is its multi-pass nature. Rendering the scene from two vantage points necessarily impacts the achievable frame rates.

4.5 The shadow volume approach

Shadow volumes require the creation of shadow geometry (object silhouette edges), which is CPU intensive. Even after such extrusions, the shadow volumes can be quite large in screen space requiring significant fill time. This makes this approach infeasible on the test card if we stick to the real-time requirement established earlier. For this reason, this method was not implemented.

4.6 The Hybrid approach

4.6.1 The hybrid test method

This next method combines some features from the two algorithms thus covered. The shadow from a preselected caster is generated from the light point but rather than indexing the stored depth, the actual shadow is stored as a texture and is indexed using the current pixel index. The resulting color is then used to modulate the scene. The free 8 bits of the color buffer are used as a stencil buffer to prevent the overflow of the projected shadow on the receiver. This method necessarily takes two passes: one to generate the shadows from the light point (simple projection) and the second pass to modulate the fully lit scene. In the first pass, writes to the color buffer and the depth buffer are turned off. The stencil buffer is cleared and the stencil test is enabled and set to increment when the receiver is rendered. Then the shadows are projected and the stencil buffer is used to stencil the projection only where the count is greater than zero. In this way, the projected shadows are kept within the confines of the receiver. The second pass simply fetches the texture value (‘shadow map’) and blends it with the calculated color in the color buffer.

This test is run under the same conditions as the other tests, with the same light positioning.

4.6.2 The hybrid approach qualitative results

Figure 26 below shows the use of the stencil buffer to restrict the shadows to the receiver plane. The shadow from the truck would normally overflow the receiver plane but is here restricted as explained above.

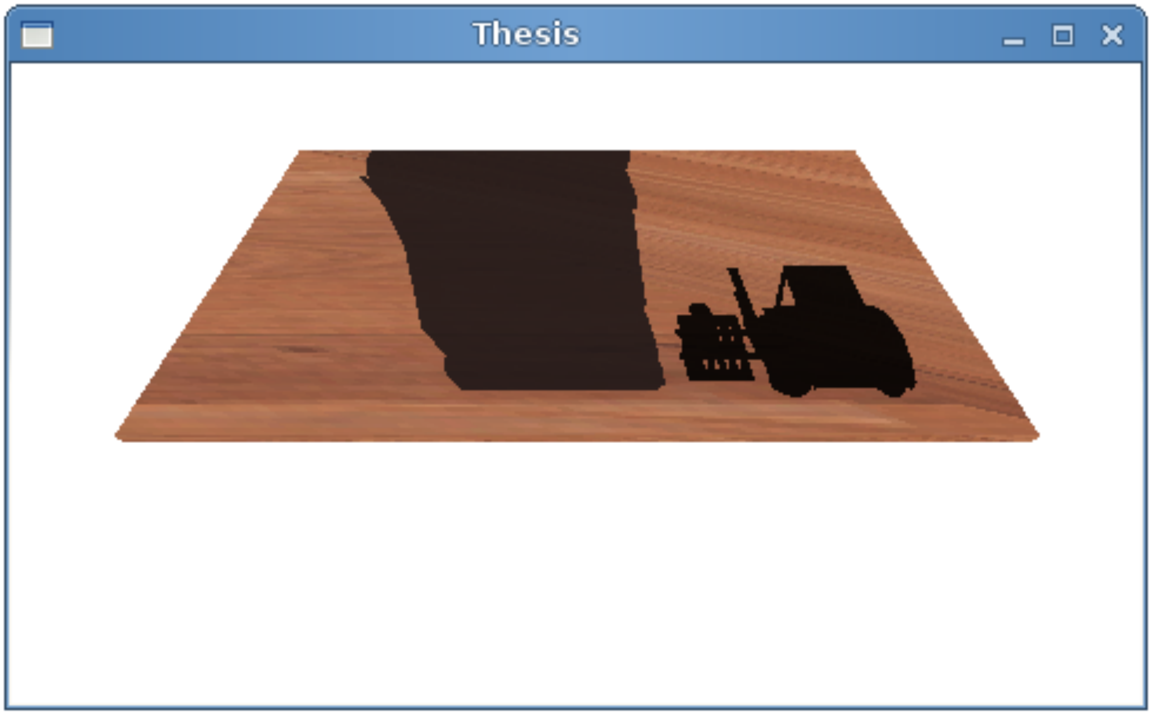


Figure 26 Hybrid approach. Notice the stenciled shadows on the receiver plane.

The front face scene is shown below. The light's position can be gauged by inspecting the correct shadow placement. Since the shadows satisfy all three criteria outlined in the objective, this method shows that it meets the 'good-looking' requirement.

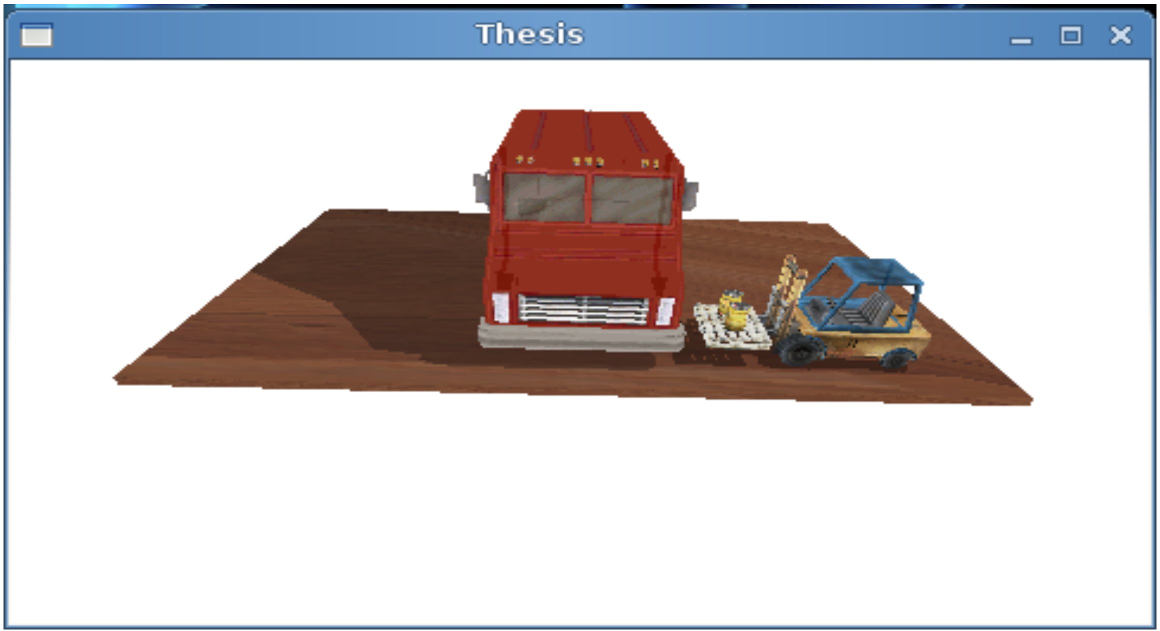


Figure 27 Hybrid approach front face scene. The shadows are placed correctly on the receiver.

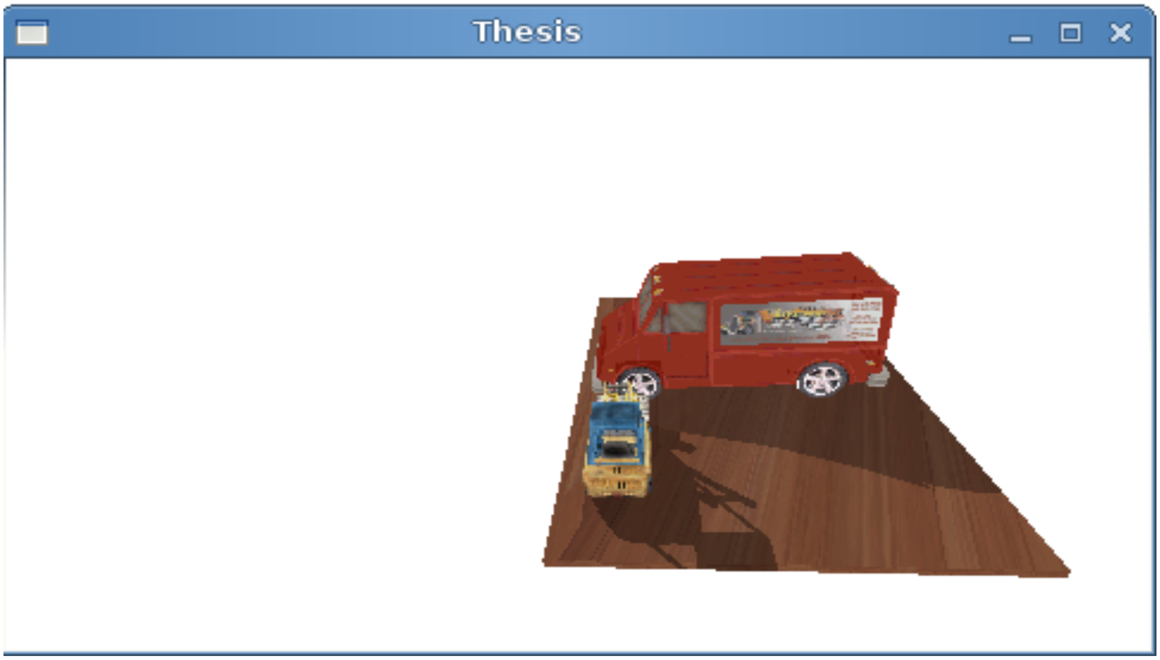


Figure 28 Hybrid approach right face scene.

The next screenshots show the back and left faces. The shadows on these images meet the established requirement as well.

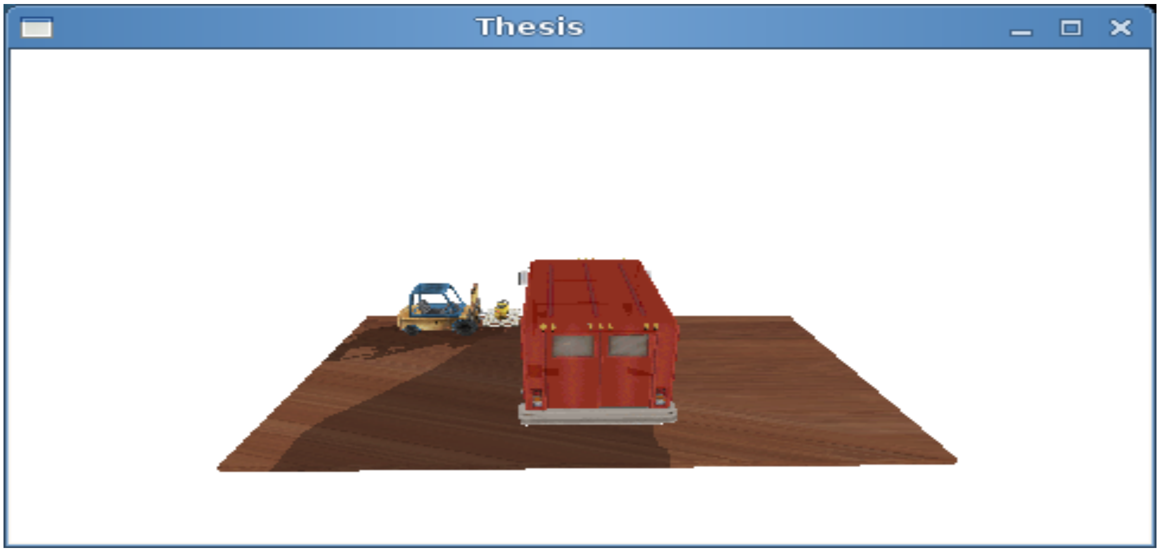


Figure 29 Hybrid approach back face.

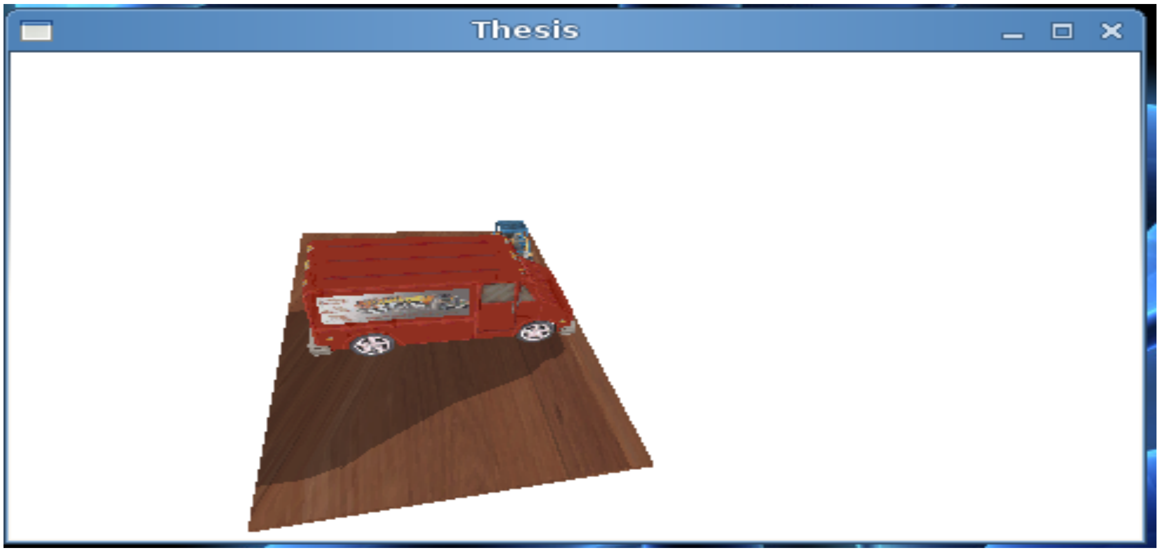
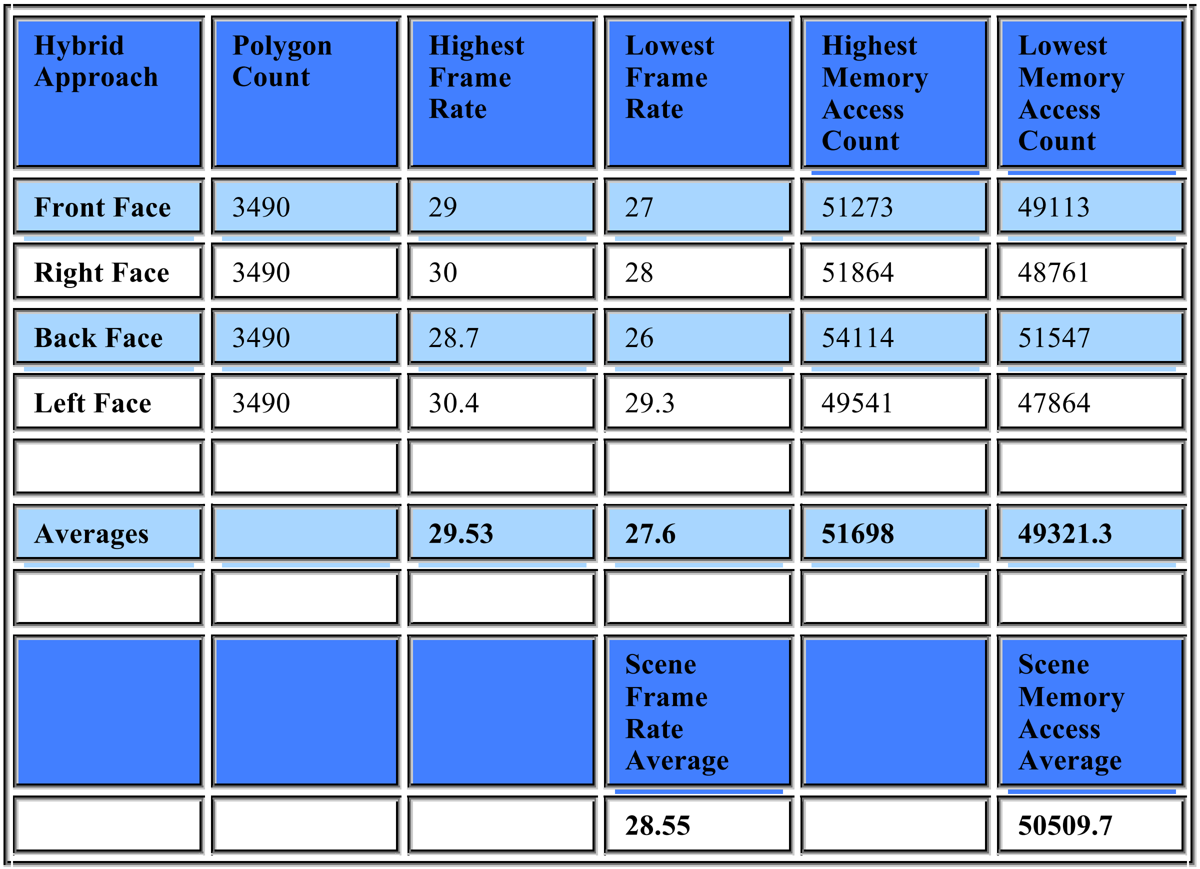


Figure 30 Hybrid approach left face.

The statistics from this test are discussed below.

4.6.3 Hybrid approach quantitative results



Hybrid Approach	Polygon Count	Highest Frame Rate	Lowest Frame Rate	Highest Memory Access Count	Lowest Memory Access Count
Front Face	3490	29	27	51273	49113
Right Face	3490	30	28	51864	48761
Back Face	3490	28.7	26	54114	51547
Left Face	3490	30.4	29.3	49541	47864
Averages		**29.53**	**27.6**	**51698**	**49321.3**
			Scene Frame Rate Average		**Scene Memory Access Average**
			28.55		**50509.7**

Table 4 Hybrid approach statistics.

Even though both the shadow mapping and hybrid approaches index texture memory for shading, table 4 above shows that this method makes less than half as many texture access calls compared to the shadow mapping approach. This large difference can be explained as follows; shadow maps are applied to the entire scene i.e. all the objects in the scene have texture coordinates that are used to index the shadow map to determine the degree of occlusion. The hybrid approach however, only accesses the texture for the **preselected receivers.** For the scenes shown, only the ground plane was the receiver and therefore only when shading the plane was an extra texture fetch made. The difference this makes in frame rates is noticeable at 28.55 frames/sec compared to 24.8 frames/sec from the shadow mapping algorithm.

Compared to the planar projection, this method makes slightly more than 1.5 times as many texture fetches and runs slower (approx 9% slower). However, the quality of the shadows is much better.

4.6.4 Hybrid approach summary

This method inherits some good traits from the other two algorithms; it is easy to implement and uses existing functionality on the test card. The shadows it generates are stenciled to prevent overflowing the receivers thereby making it more attractive compared to the planar projection. At 28.5 frames/sec, the performance from this method, though lower than that from the planar algorithm, is still relatively fast. The shadows generated from this method are also of good quality and are not plagued by the aliasing problems inherent in shadow mapping. As table 4 shows, the texture access count is considerably less than that of shadow mapping. This is a major plus as reducing external memory accesses sits well with the objective of reducing power consumption. Also, since this method is based on the planar projection, it can support multiple lights using the same texture. Contrast this to shadow mapping where each light generally requires a different shadow map. Lastly, this method avoids rendering the same shadows every frame when lights are stationary (which is mostly the case in games) since the shadows are stored in a texture.

It has the same drawback as the planar algorithm; it is restricted to casting shadows on planes therefore self-shadowing is not a possibility. A second drawback is that it requires the developer to choose/preselect shadow casters and receivers prior to rendering.

4.7 Summary

The algorithms investigated thus far generate shadows that meet the qualitative (good looking) requirement that was established in the beginning of this chapter. Their respective frame rates are well above the 15 frames/sec real-time criterion but they differ significantly when it comes to external memory accesses. From a texture access perspective, the planar projection is superior to the rest of the algorithms as it only fetches a texel[12] during the color determination stage of rendering. It also has the highest frame rate at 31 frames/sec, which makes it very attractive for implementation on the test card.

The hybrid algorithm ranks second from a quantitative perspective. Its texture access pattern is about 1.5 times that of the planar projection and has a reduced frame rate of 28.5 frames/sec compared to 31 frames/sec for the planar algorithm. However, the quality of shadows produced from this method is superior compared to the planar algorithm. It avoids multiple renderings of the same shadow objects by storing the shadows in a texture and indexing the said texture when shading. It also uses a stencil buffer to prevent shadow overflow on the receiver, making the shadows more realistic.

Shadow mapping performs the poorest of all the algorithms quantitatively. At about 25 frames/sec (24.8), this method lags the group and displays the most expensive texture access pattern. At about 110,000 texture accesses per frame, this method ranks highest from a power consumption standpoint. The shadows it generates are also prone to aliasing and shadow continuity, which may have a distracting effect in an interactive environment. However, it does generate shadows that exhibit the self-shadowing phenomenon, which makes it attractive. Considering the real-time criterion of 15 frames/sec or higher, it might be concluded that there is enough leeway to ‘cleanup’ the generated shadows (perhaps by employing percentage closer filtering), given the

[12] Texel is short for texture element.

recorded 25 frames/sec frame rate. The table below consolidates the average statistics from all the algorithms.

Method	Highest Frame Rate Avg.	Lowest Frame Rate Avg.	Scene Frame Rate Avg.		Highest Memory Access Avg.	Lowest Memory Access Avg.	Scene Memory Access Avg.
Planar Approach	**32**	**30.125**	**31.06**		**32075**	**31779**	**31927**
Hybrid Approach	**29.53**	**27.6**	**28.55**		**51698**	**49321.3**	**50509.7**
Shadow Mapping	**26.07**	**23.53**	**24.8**		**113207**	**105844**	**109526**

Table 5 Average frame rates and memory access counts from all the algorithms.

The type of algorithm selected is highly dependent on the context. For example, if most objects hover in the air with the ground being the only receiver, then planar projection would be ideal since it is fast and has the least amount of external memory accesses. However, for games with highly complex scenes composed of many hierarchical receivers, the shadow algorithm requires the least amount of intervention and would therefore be the best solution. Its shadows would need to be post-processed to be useful if there is enough bandwidth to allow this to be done in real time. The hybrid algorithm would be the preferred solution for scenes between the two extremes mentioned. Caching the shadows as a texture avoids the unnecessary work of re-projecting the object’s vertices on the receiver planes if the lights are stationary. Its shadows do not need post-processing and in fact look good. This method is preferable over the planar projection if a scene has multiple lights due to the reuse of shadows.

These are merely suggestions and not rules and the algorithm used may perhaps depend more on power consumption limitations rather than shadow quality.

CHAPTER 5
CONCLUSION

Shadows enhance visual realism in a 3D scene. Their inclusion in a scene goes a long way in giving the viewer the sense of immersion. Much research has gone into the generation of shadow algorithms and graphic cards are now able to render realistic-looking shadows in real time. As has been shown, this is also feasible on a low-end mobile graphics card. As always, there is a trade-off between quality and quantity. More so than their desktop counterparts, mobile graphic cards can exploit the dynamic nature of their applications to hide the inconsistencies associated with a low quality shadow algorithm in favor of increasing battery life.

We have seen a few shadowing techniques, discussed their pros and cons and analyzed their run-time performance on a constrained graphics card. No-one shadow generation technique fits all sizes/situations but the knowledge gained from the findings of this investigation can allow a developer to choose which technique suits their needs.

Reducing power consumption in mobile graphic cards does not necessarily mean reducing the quality of shadows. The shadow mapping technique fairs quite well compared to the hybrid approach and its output can be ‘cleaned up’ to produce stunning shadows.

5.1 Future research

More efficient shadowing techniques from a standpoint of memory access are still needed but it appears we are headed in the right direction. Implementing the graphics card used in this test in hardware would provide for more tangible results and shed some more light on the actual power dissipation when running the tests.

REFERENCES

[1] Leonardo Da Vinci. Cordex Urbinas. 1490. 1,2

[2] Geoffre S. Hubona, Phillip N. Wheeler, Gregory W. Shirah and Matthew Brandt. The role of object shadows in promoting 3D visualization. *ACM transactions on Computer-Human Interaction,* 6(3):214-242, 1999. 1,2

[3] Daniel Kersten, Pascal Mamassian, and David C. Knill. Moving cast shadows and the perception of relative depth. *Perception,* 26(2):171-192, 1997. 1,2

[4] Leonard Wanger. The effect of shadow quality on the perception of spatial relationships in computer generated imagery. *Computer Graphics (Interactive 3D Graphics 1992),* 25(2):39-42, 1992, 1,2

[5] Andrew Woo, Pierre Poulin and Alain Fournier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications,*10(6):13-32, November 1990. 1,2

[6] Robert Burke. Creating a Page Template. www.rab3d.com June 2010.

[7] James Van Verth, Lars M. Bishop. Essential Mathematics for Games and Interactive Applications. Morgan Kauffman Publishers, 2008, 276-277

[8] Foley, van Dam, Feiner, Hughes. Computer Graphics:Principles and Practice 2nd Edition in C. Addison Wesley Publishing Company,1997,744-745.

[9] Tomas Akenine-Moller, Eric Haines, Naty Hoffman. Real-Time Rendering 3rd Edition. A K Peters Ltd. 2008

[10] J-M. Hasenfratz, M. Lapierre, N. Holzschuch and F. Sillion. A survey of Real-time Soft Shadow Algorithms. 25(4):753-774, 2003.

[11] Jim Blinn. Me and My (Fake) Shadow. 8(1):82-86, 1998.

[12] Chris Bentley. Two Shadow Rendering Algorithms. http://web.cs.wpi.edu/~matt/courses/cs563/talks/shadow/shadow.html June 2010.

[13] Paul S. Heckbert and Michael Herf. Simulating soft shadows with graphics hardware. Technical Report CMU-CS-97-104, Carnegie Mellon University, January 1997. 9, 17, 18

[14] Amy Gooch, Peter Shirley and Richard Riesenfeld. Interactive Technical Illustration. *Proceedings 1999 Symposium on Interactive 3D Graphics* 31-38, 1999

[15] L.Williams. Casting Curved Shadows on Curved Surfaces. SIGGRAPH 78, 270-274.

[16] Randima Fernando and Mark J. Kilgard. The Cg Tutorial.*The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003, 257.

[17] Eric Haines, Tomas Moller. Real-Time Shadows.

[18] Stamminger M and Drettakis Perspective shadow maps. In Proceedings of ACM SIGGRAPH 2002, ACM Press/ACM SIGGRAPH.

[19] J-C Hourcade and A. Nicolas. Algorithms for antialiased cast shadows. *Computers & Graphics*. 9(3):259-265, 1985.

[20] W.T Reeves, D.H Salesin and R.L. Cook. Rendering Antialiased Shadows with Depth Maps. SIGGRAPH 87, 283-291

[21] Tim Heidmann. Real shadows, real time. In *Iris Universe*, volume 18, pages 23-31. Silicon Graphics Inc., 1991.

[22] F.C Crow. Shadow Algorithms for Computer Graphics. SIGGRAPH 77, 242-247.

[23] Hun Yen Kwoon. The theory of stencil shadow volumes. www.gamedev.net. June 2010.

[24] A. Appel. Some Techniques for Shading Machine Renderings of Solids. SJCC, 1968, 37-45

[25] Phil Dutre. Advanced Global Illumination 2nd Edition. A K Peters, 2006

[26] Kevin Suffern. Ray Tracing from the Ground Up. A K Peters, 2007

[27] Pascal Mamassian, David C. Knill and Daniel Kersten. The perception of cast shadows. *Trends in Cognitive Sciences,* 2(8):288-295, 1998.

[28] Tomas Akenine Moller, Jacob Strom Graphics for the Masses: *A Hardware Rasterization Architecture for Mobile Phones*, 22(3):801-808, 2003

[29] Perissakis Fromm, S. Cardwell, N. Kozyrakis, C. McCaughy, Patterson, Anderson and Yelick K. The Energy Efficiency of IRAM Architectures. In *24th Annual International Symposium on Computer Architecture,* ACM/IEEE 327-337, 1997

[30] AMD. ATI Radeon ™ HD 5870 GPU Feature Summary. July 2010.