FAST SELF-SHADOWING USING OCCLUDER TEXTURES

A Thesis

by

CHRISTOPHER RYAN COLEMAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2006

Major Subject: Visualization Sciences

FAST SELF-SHADOWING USING OCCLUDER TEXTURES

A Thesis

by

CHRISTOPHER RYAN COLEMAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Frederic Parke
Committee Members,	Nancy Amato
	Vinod Srinivasan
Head of Department,	Mark Clayton

December 2006

Major Subject: Visualization Sciences

ABSTRACT

Fast Self-Shadowing Using Occluder Textures. (December 2006)
Christopher Ryan Coleman, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Frederic I. Parke

A real-time self-shadowing technique is described. State of the art shadowing techniques that utilize modern hardware often require multiple rendering passes and introduce rendering artifacts. Combining separate ideas from earlier techniques which project geometry onto a plane and project imagery onto an object results in a new real-time technique for self-shadowing. This technique allows an artist to construct *occluder textures* and assign them to *shadow planes* for a self-shadowed model. Utilizing a graphics processing unit (GPU), a vertex program computes *shadowing coordinates* in real-time, while a fragment program applies the shading and shadowing in a single rendering pass. The methodology used to create shadow planes and write the vertex and fragment programs is given, as well as the relation to the previous work. This work includes implementing this technique, applying it to a small set of test models, describing the types of models for which the technique is well suited, as well as those for which it is not well suited, and comparing the technique's performance and image quality to other state of the art shadowing techniques. This technique performs as well as other real-time techniques and can reduce rendering artifacts in certain circumstances.

ACKNOWLEDGMENTS

I would first like to thank my wife, Heather, for all of her support. I would also like to thank Dr. Parke, Dr. Amato, and Vinod for their patience and support as committee members. I would also like to thank Chris Henderson and Scott Davidson at MultiGen-Paradigm who allowed me to use this topic to complete my degree requirements and were very helpful in allowing time off, machine access, and general encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Motivation

Recent advances in real-time computer graphics center around programmable graphics hardware for the personal computer. Modern graphics processing units (GPUs) are programmable, allowing for vertex programs to compute vertex positions and texture coordinates of a polygon, and fragment programs to compute color for the fragments that result from rasterization of that polygon. Vertex and fragment programs, both individually and combined, are commonly referred to as shaders. One of the features made possible by this new hardware is shadowing at high frame rates. However, state of the art shadowing techniques that utilize modern hardware often require multiple rendering passes and introduce rendering artifacts.

Frame rates are determined by computation time for a single frame. Real-time dynamic shadowing techniques are often the most expensive aspect of rendering a frame. This is because most shadowing techniques require multiple rendering passes to pre-process shadowing information for the frame, and then apply that information. The entire scene may have to be rendered twice, often cutting the frame rate in half. Pre-processing shadowing information means determining whether each location on a model is in or out of shadow. However, it is possible to take advantage of knowledge about the shape of a model to simplify the shadow computations allowing self-shadowing to be applied to a model in a single pass. Many models with complex geometry can be approximated by a set of grayscale images containing occlusion information about the model that can be applied to any light source.

1.2 Overview

A method for approximating shadow casting geometry as an image that is then projected onto a 3D model is developed in this work. A vertex program computes texture coordinates and a fragment program performs texture look-ups and applies the shadowing effect. This technique allows an artist to construct *occluder textures* and assign them to *shadow planes* for a self-shadowed model. A vertex program computes *shadowing coordinates* in real-time, while a fragment program applies the shading and shadowing in a single rendering pass.

One advantage of this self-shadowing technique is that a separate rendering pass is not required. Instead, an artist is allowed to pick the salient features of the model to be captured in the occluder textures while

---

This thesis follows the style and format of the *IEEE Transactions on Visualization and Computer Graphics*.

creating the model. In addition, the resolution of the shadow textures can be carefully chosen to avoid pixelization artifacts. Finally, the shadow plane textures and the vertex and fragment shader code required to interpret the textures can be encapsulated within the model's definition. This adds dynamic shadows to the model without requiring the entire scene to be processed for shadows. The combination of artist determined shadow textures and single-pass rendering allows for high quality self-shadowing with minimal performance impact.

2. PRIOR WORK

State of the art techniques in real-time shadowing take advantage of advances in graphics hardware and specialized graphics language extensions. A recent survey [10] of real-time shadowing techniques lists stencil shadow volumes, shadow maps (also called depth-map shadows), and perspective shadow maps (PSMs). All of these techniques support self-shadowing as well as optionally casting shadows onto other objects of arbitrary shape. An advantage of these techniques is that they can be applied globally without any prior knowledge of the object(s) they are being applied to. Another advantage is that some steps of these techniques can be hardware accelerated, through use of the stencil buffer, or extensions that allow a depth pass to be captured as a texture, or extensions that allow for sophisticated filtering of that texture [4][8][10]. The disadvantages are that multiple render passes are required which aversely impact performance and artifacts can result from resolution mismatches, depth precision issues, or polygonal boundaries.

Earlier techniques that did not require graphics language extensions include projected planar shadows and projective texturing. These techniques are simple and fast, but do not support self-shadowing. The technique presented here is similar to projected planar shadows and projected texturing, but is designed for self-shadowing and is meant to compete with stencil shadow volumes and shadow maps in both image quality and performance.

Some common terminology for shadowing techniques is required. The object that casts a shadow is referred to as the occluder. Its polygonal geometry is the *occluder geometry*. Occluder geometry blocks, or occludes, the light from reaching *shadow-receiving geometry*. For self-shadowing, the occluder geometry can also be shadow-receiving geometry, such that polygons from one part of an object may be casting shadows onto another part of the same object. It is assumed in this thesis that individual polygons are flat, and therefore cannot cast shadows onto themselves.

Some OpenGL terminology is required as well. Existing shadowing techniques make use of different hardware-accelerated rendering buffers, all of which together are referred to as the *frame buffer*. A typical frame buffer configuration might consist of a color buffer, depth buffer, and a stencil buffer. The color buffer is where the final rendered color image is written. The depth buffer contains depth information that is useful while the scene is being rendered. The distance of each rasterized fragment of a polygon is computed and stored in the depth buffer as the polygons are being rendered. Finally, the stencil buffer is simply a buffer where a bitmask can be written while selected objects render. It is similar to the color buffer, except that the rasterized fragment writes a bit-wise value into the buffer. The simplest stencil

buffers contain only a single bit per-pixel – either "on" or "off". Fragments can both test the stencil buffer to find out if a fragment should be rejected from all buffers – color, depth and stencil, or accepted and optionally write a bitmask to the stencil buffer. The stencil buffer is often used to query whether a particular pixel in the framebuffer has already been written for a particular frame.

Rendering can be single-pass, or require multiple render passes. *Single-pass rendering* means that the color values are rendered into the frame buffer as each polygon in the scene renders. By contrast, in dual-pass rendering the entire scene is rendered, sometimes from a different point of view, capturing some information in the framebuffer, and then the entire scene is rendered again to fill in the final color values.

Finally, textures, texture mapping, and texture coordinates are terms used in this thesis in the context of two-dimensional texture mapping. A *texture* is simply a buffer of two-dimensional raster data. The data might contain color information, depth information, or simply grayscale intensity values. The two-dimensional image is mapped to three-dimensional geometry through the assignment of two-dimensional *texture coordinates* at every vertex of every polygon. Sometimes, the texture coordinates are directly created by an artist and stored for each vertex of an object. Sometimes, texture coordinates are generated algorithmically through OpenGL language extensions, or through code in a vertex shader.

2.1 Projected Planar Shadows

One technique for rendering real-time shadows simply projects the shadow casting geometry onto a flat plane using a specialized projection matrix [9]. Rendering of a frame consists of rendering several layers back to front in a single pass [7]. First, the shadow receiving object is rendered. For example, in Figure 1 below, the first thing rendered would be the grassy ground plane. Second, the shadow casting object is projected onto the plane of the shadow receiving object (often a floor, wall, or ceiling) using a projection matrix. The projected geometry is rendered flat-shaded in a dark color, sometimes partially transparent. The stencil buffer can be used to ensure that the object only darkens the underlying pixels one time to avoid over-darkening where the shadow casting geometry might overlap itself. The stencil buffer can also be used to make sure that the shadow does not project off the edge of the shadow receiving object. In Figure 1 below, the aircraft shadows would be rendered second. The image on the right in Figure 1 is shown in wireframe to make it clear that the aircraft shadows are actually just the entire geometry for the aircraft with the positions of the vertices flattened onto a plane and the color of the aircraft set to a dark, partially transparent color. The stencil buffer was used in this image to avoid rendering the shadow multiple times at the same pixel location since the aircraft will have many overlapping polygons when it is rendered flattened onto a plane. Finally, the rest of the scene, including the shadow casting objects, is rendered with its normal shading. In Figure 1, the last step is to render the aircraft with their normal vertex

positions and shading. All of the layers are rendered in a single pass, resulting in the final image that contains the projected planar shadows.

The advantage of projected planar shadows is that the technique can be applied to cast shadows from any arbitrary object with a minimal performance impact as long as the object's geometry is not overly dense. The density of an object refers to the number of polygons used to represent that object. If an object consists of a few hundred polygons, then rendering the projected planar shadows will only require a few hundred more polygons in the scene. Rendering with projected planar shadows is equivalent to rendering the shadow-casting objects twice, once to show the object and once for its shadow. A typical scene will be composed of several thousand polygons, while only a few objects near the observer will cast shadows. The disadvantages are that shadows can only be cast on flat surfaces, self-shadowing is not supported, shadows will have sharp, polygonal boundaries, and the shadows will "pop" from one shadow receiver plane to another as the shadow casting object and/or light source move. Figure 1 shows that projected planar shadows flatten the occluder geometry onto a plane as can be seen in the shadow wire-frames in the image on the right.

Figure 1. Projected Planar Shadows.

2.2 Stencil Shadow Volumes

Stencil shadow volumes are another real-time technique for computing shadows. The core elements for this technique have been around since long before the stencil buffer hardware acceleration for the technique [2]. The hardware accelerated technique makes use of the stencil buffer through a two-pass rendering approach that uses volumes constructed from the edges of shadow casting geometry projected in the direction of the light source to determine if other surfaces are occluded from the light source [5]. Shadow volume geometry is algorithmically constructed each frame. First, the silhouette edges of an occluder object from the point of view of the light source are determined. Then, the object's geometry is extruded along the direction of the light, breaking the object in half along the silhouette edge and creating

new geometry between the two halves. The result is a closed *shadow volume*, where objects that fall within the volume are in shadow for the light source.

The stencil shadow volume technique can be thought of as tracing a ray from the eye position into the scene for each pixel and keeping track of entering a shadow volume by incrementing the stencil bits, and then decrementing the stencil bits if the ray exits the shadow volume. The ray stops when it hits a surface in the scene. A non-zero value in the stencil buffer means that the pixel is in shadow.

To implement the technique, first the color, depth, and stencil buffers are cleared, where the stencil buffer is cleared to a value of 0. In the first pass, the scene is rendered with ambient and emissive lighting components only while rendering to the color and depth buffers. This lays down the depth values for the frame and the "in-shadow" color for the entire scene. Next, a polygonal representation of the shadow volumes from the shadow casting objects in the scene is rendered into the stencil buffer where front facing polygons increment the stencil bits, and back facing polygons decrement the stencil bits only when the fragments from the shadow volume geometry pass the depth test. At this point, the stencil buffer is 0 everywhere that is not within a shadow volume, and greater than 0 if the pixel is in shadow. In the second rendering pass, the entire scene is rendered again, this time with diffuse and specular lighting enabled while checking the stencil buffer to see if each fragment is in or out of shadow. If the pixel is not in shadow, the diffuse and specular lighting contributions are added to the ambient and emissive components already captured in the frame buffer. Since the depth buffer already contains values from rendering the entire scene for the first pass, the depth test must be set such that fragments only render if the depth is equal to the value in the depth buffer. Otherwise, the depth buffer would have to be cleared again between the first and second passes, which is an expensive fill operation. The final result is a frame where objects appear lit, except for the pixels that were flagged in the stencil buffer as in-shadow.

One advantage of stencil shadow volumes is that shadow boundaries are computed with pixel or even sub-pixel accuracy. In addition, self-shadowing of the shadow casting object is supported. Finally, the technique can be hardware accelerated since most modern graphics cards include hardware acceleration for the stencil buffer.

One disadvantage of stencil shadow volumes is the large pixel "fill" penalty for the two pass approach. "Fill-hit" refers to the number of times each pixel in the framebuffer is computed and the varying complexity of that computation, especially if fragment shaders or multiple textures are used. For this technique, every pixel in the scene is filled at least twice. Also, dynamic silhouette computations can prove expensive and place restrictions on the types of geometry allowed. The volumes produced require

the generation of geometry every frame, as well as the fill-hit for rendering that shadow volume geometry into the stencil buffer. A shadow volume might consume the entire view from the eye position and therefore fill the entire stencil buffer. Another disadvantage is that the technique is difficult to use in a practical and robust manner, especially since the near and far clip planes can "slice open" the shadow volumes when the volume intersects the near or far clip plane [5]. The image quality for stencil shadow volumes is fairly good, but, like projected planar shadows, the shadows produced have sharp, polygonal boundaries. Stencil shadow volume shadows sometimes "pop" with low polygon counts because the silhouette computation used to determine the edges from which shadow volumes are extruded will "pop" from one set of edges to another. Finally, stencil volume shadows are not compatible with partially transparent occluder objects.

2.3 Projective Texturing

Projective texturing is capable of producing texture coordinates that project an image onto an object. An often used analogy is that of a slide projector casting an image onto a surface [3][11]. Shadows can be simulated using projective texturing by rendering a shadow casting object flat-shaded in black on a white background from the point of view of the light source. This "silhouette" image is then projected from the light source onto other objects by computing texture coordinates for the objects using a special projection matrix. The projection matrix transforms object-space or eye-space vertex positions into a set of homogeneous texture coordinates [3]. It is helpful to think of this transformation as moving from one coordinate system to another using matrix transformations, followed by a perspective division [6]. Texture coordinate generation is available in OpenGL to create projective texture coordinates on the fly given the description of the texture plane. Note that the silhouette image must be recomputed any time the shadow casting object or the light source moves. In Figure 2 below, an image of the letter "A" is projected onto a plane. The image used for projection is applied to a plane shown at the location of the light source. Projection is perpendicular to that plane along the light direction.

Figure 2. Projective Texture.

An advantage to using projective texturing is that projective textures can cast shadows onto arbitrary geometry, and are not limited to casting onto flat surfaces. It is also a very fast technique to render. A disadvantage is that this projective texture shadowing technique cannot be used to self-shadow objects.

2.4 Shadow Maps

Williams described in 1978 how to cast curved shadows onto curved surfaces using a depth pass from the point of view of the light source [12]. This technique, sometimes called "depth map shadows," or simply "shadow maps" has long been supported in software rendering packages such as Pixar's Renderman, and can now be hardware accelerated [10][4]. Shadow mapping performs shadow determination using a depth map created from the point of view of the light source. On a first pass, the scene is rendered from the point of view of the light source capturing the depth values only. On a second pass, the scene is rendered from the observer's point of view. Projective texturing is used to automatically generate texture coordinates for all shadow-receiving objects to look up in the depth map. The distance from a point on the shadow receiving object to the light source is compared to the values in the depth map to determine if the point is in shadow. If the distance is less than the one stored in the depth map, then the point on the object can be seen by the light. If the distance is greater than the one stored in the depth map, another object must be occluding the light source and therefore the point is in shadow. The point on the shadow receiving object is rendered either lit or in shadow, and the final fragment color is written into the frame buffer.

Today, shadow maps can be hardware accelerated. OpenGL extensions exist that can capture depth buffer values into a texture, perform eye-linear projective texture coordinate generation, and perform a "shadow comparison" texture look-up. Eye-linear texture coordinate generation is the specific OpenGL extension through which projective texturing is implemented in real-time applications because it allows for texture coordinates to be generated given a point and a normal to define a plane in eye-space. A texture matrix can be used to position the model vertices such that the plane aligns with the light source. The shadow comparison texture lookup is another extension which compares a distance value to the depth value captured in a depth map [4].

One advantage of shadow mapping is that self-shadowing is supported. Additionally, transparent occluder geometry and alpha texturing are supported through the alpha test, which allows for only a single bit of alpha. A fragment in the first pass either writes to the depth buffer, or does not. Smooth alpha blending is not allowed in the depth buffer, so the edges of occluder geometry cannot support smooth transparency. Fortunately, a single bit of transparency is typically enough to capture the shape of shadow casting geometry. Finally, shadow mapping can be supported for any shadow casting model and shadow receiving model without restrictions on the geometry.

The disadvantages of shadow mapping include performance and image quality issues. Shadow mapping is a two pass approach, which impacts performance. The depth pass must be regenerated any time the light source or any shadow casting object moves in any way. Usually the depth pass is computed every frame. Shadow artifacts can lower image quality. Shadow map shadows may show pixelization artifacts due to pixel resolution mismatches for the observer's view frustum and the frustum used by the light source to generate the depth pass. Depth images are frequently computed at very high resolution to try to avoid pixelization artifacts, resulting in a further performance penalty. Shadow maps are also prone to depth precision errors, which require the use of a polygon offset that can never be exactly correct [4]. Polygon offset is an OpenGL setting that modifies the depth buffer value computed by one or more depth units. In this manner, the occluder geometry can have its depth biased to avoid casting shadows over the entire object due to precision errors. Finally, shadow maps require a frustum to be specified from the light source, and therefore do not support omni-directional lights. The depth pass covers a rectangular region of the scene as seen from the light source, rarely casting shadows over the entire scene. Frequently in real-time applications, shadows are only computed right around the eye-point to avoid having to make the depth image too large.

As mentioned above, the capture of a depth map from the perspective of the light source requires a view frustum. Most often this frustum is orthographic, beginning at the exact location of the light source. However, there is often a mismatch in resolution between the depth pass and the observer's point of view of the scene resulting in pixelization artifacts. Perspective shadow maps (PSMs) attempt to alleviate the resolution mismatch by creating shadow maps in post-projective space using a perspective viewing frustum that is located in front of, or behind the location of the light source. In the perspective shadow map technique, nearby objects become larger than farther ones, giving greater emphasis to objects nearer the observer [8]. It is mentioned here because PSMs represent the latest research in shadow maps to try and reduce image artifacts associated with the technique.

2.5 Pre-Computed Shadows and Shading

It should be noted that pre-computed methods for shadowing and shading exist. These are typically less dynamic in nature. Light maps, global illumination methods, pre-computed radiance transfer (PRT), pre-computed ambient occlusion, and dynamic ambient occlusion all provide combinations of shading and shadowing that approach cinematic quality rendering in real-time applications. Both pre-computed ambient occlusion and dynamic ambient occlusion involve calculating an "accessibility value," which is the percentage of the hemisphere above each surface point not occluded by geometry [1]. In addition, the "bent normal" can be computed which is the direction of least occlusion for each surface point. The

advantage of these techniques is that they often provide a more accurate simulation of real-world lighting and an added level of realism. The disadvantages are that they can be very complex to implement for an arbitrary scene containing an arbitrary number of objects and are very performance intensive to compute and update in real-time. These techniques typically provide better shading than direct shadowing, and are often used in conjunction with other techniques. They require multiple passes to approximate shadowing.

3. OBJECTIVES

The main goal of this thesis was to accomplish self-shadowing for certain kinds of models at high frame rates in a single rendering pass. The occluder texture technique evolved from two older techniques which did not support self-shadowing, projected planar shadows and projective texturing. Utilizing modern hardware, self-shadowing can be accomplished with a simpler, faster technique that is the combination of these two. This technique should be better than other techniques that use modern hardware (stencil shadow volumes, shadow maps, and perspective shadow maps) because it should only require a single rendering pass and should not introduce distracting shadowing artifacts into the image. Because it is a single pass technique, the occluder texture technique should be faster. Further, this technique should have fewer rendering artifacts because the edges of shadows that intersect with the shadow casting geometry should have floating-point precision, and because it gives control for the look of the shadow to the artist.

4. METHODOLOGY

The occluder texture technique allows an artist to construct *occluder textures* and assign them to *shadow planes* for a self-shadowed model. A vertex program computes *shadowing coordinates* in real-time, while a fragment program applies the shading and shadowing in a single rendering pass. The proposed technique can be understood by closely examining projective texturing. See Figure 3 below.

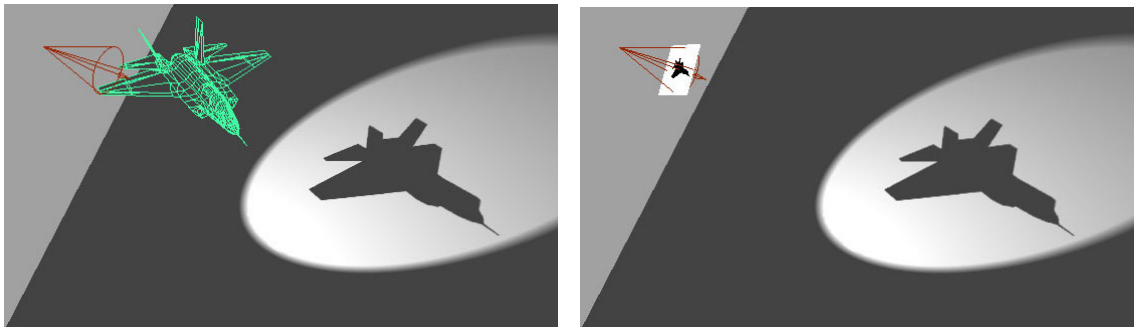


Figure 3. a) F-35 Example of a Cast Shadow. **b) Projective Texture Shown at the Light Source.**

In Figure 3, an F-35 model (highlighted in green) appears to be casting a shadow onto another surface. This image was created by rendering the airplane from the point of view of the light source, creating a grayscale "silhouette" image of the F-35 that was then used as a projected planar shadow, just like the letter "A" in Figure 2 in the section on Projective Texturing. In Figure 3b, the shadow image can be seen with its projection from the light source. As mentioned earlier, the projective texture technique is very fast to render, but requires a shadow image be re-computed any time the light source or the shadow casting object moves, and this technique cannot be used for self-shadowing of the shadow casting object.

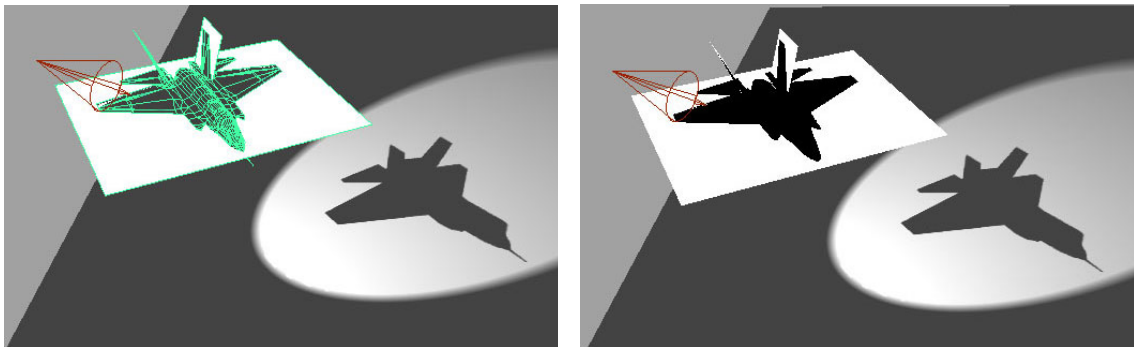


Figure 4. a) F-35 Intersecting Shadow Planes. **b) Shadow Planes are Not Positioned or Aligned with the Light Source.**

In Figure 4, the occluder texture technique of using *shadow planes* that are chosen during modeling is shown. The shadow planes are not aligned with or captured from the point of view of the light source. Shadow planes modeled with *occluder textures* can cast shadows onto other scene elements, or onto the shadow casting object itself. Occluder textures are static in nature, and do not have to be re-computed at run time. Figure 4a shows shadow planes intersecting the F-35 model. Figure 4b shows the light source and the shadow planes with their occluder textures applied, while the F-35 is hidden to show that the shadow planes and occluder textures are casting the shadow. Note that the shadow planes can be used to self-shadow the F-35 and not just cast shadows onto a ground plane as with projected texturing described in Section 2.4.

The shadowing technique proposed is similar to planar projected shadows in that the shadow receiving geometry is projected onto a plane. However, in this case, it is the shadow receiving geometry projected rather than the shadow casting geometry. The plane the shadow receiving geometry is cast onto is the cross-section plane of the shadow casting geometry. The proposed method is also similar to projective texturing, except that the direction of projection is rarely perpendicular to the plane of the occluder texture. The silhouette images created for occluder textures are similar to the silhouette computations performed for stencil shadow volumes. Finally, this approach has something in common with pre-computed techniques in that occluder textures themselves must be modeled or pre-computed.

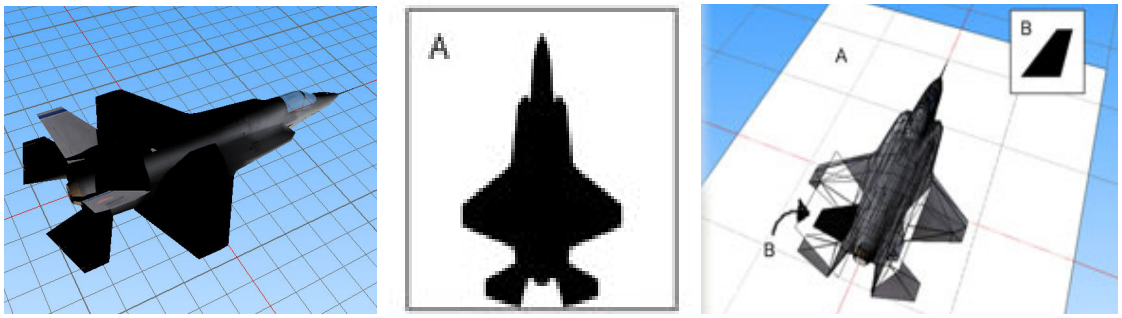


Figure 5. F-35 Self-Shadowing from Three Shadow Planes and Two Occluder Textures.

Figure 5 consists of three separate images. The left image shows a polygonal model of an F-35 with self-shadowing. The center image shows an occluder texture for the shadow plane that approximates the horizontal surfaces of the aircraft. The image is labeled “A” to illustrate which shadow plane the occluder texture matches with in the image on the right. The right image shows three shadow planes which cut through the F-35. Also in the image on the right is a second, smaller occluder texture labeled “B” which is used for the two shadow planes that intersect the tail fins. Because this technique takes advantage of rigid, flat surfaces of a model, not all models are well suited for this technique. The advantage of this technique

is that often shadows could be approximated in a far simpler manner if assumptions can be made about the object that is casting the shadow. The modeler has to make decisions about the salient shadow-casting features of a model, and choose appropriate shadow planes to model those features. Shadow plane textures can even be built directly into the modeling process, resulting in a self-contained "feature" of a model.

The core idea of the technique involves finding a texture coordinate, or *shadowing coordinate*, for each vertex of the shadow receiving geometry for each of the shadow planes. One way to visualize this is to draw a line from the vertex in the direction of the light source finding intersections with each of the shadow planes.

4.1 Computing the Intersection

We start from a parametric equation for a three dimensional line:

$$\boldsymbol{p}(s) = \boldsymbol{p_0} + s\boldsymbol{d} \tag{1}$$

where $\boldsymbol{p_0}$ is the originating point for the line, $\boldsymbol{d}$ is the direction vector of the line, and s is a parameter which can be used to generate different points $\boldsymbol{p}$ on the line [9].

A vertex shader is executed for every vertex of a polygonal object. The vertex shader's job is to generate texture coordinates for the object. For example, let's take a vertex on the tail of the airplane through the process of generating a texture coordinate for the shadow plane of the tail fin. First, a ray is traced from the vertex, along the direction to the light source, intersecting with the shadow plane. Applying equation (1) to create a line in the light direction $\boldsymbol{L}$ from a vertex on the tail of the airplane $\boldsymbol{p_0}$ results in equation 2 below. To keep matters simple, a directional light source is assumed.

$$\boldsymbol{p}(s) = \boldsymbol{p_0} + s\boldsymbol{L} \tag{2}$$

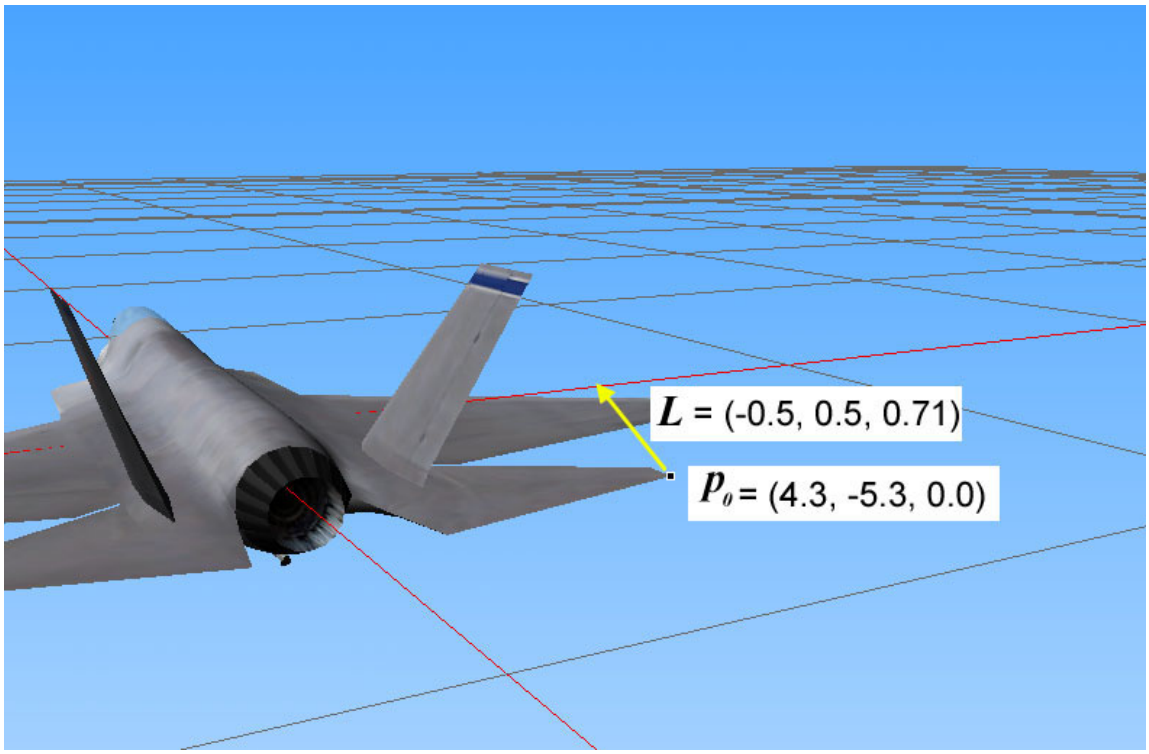


Figure 6. A Vertex on the Tail and the Direction to the Light Source.

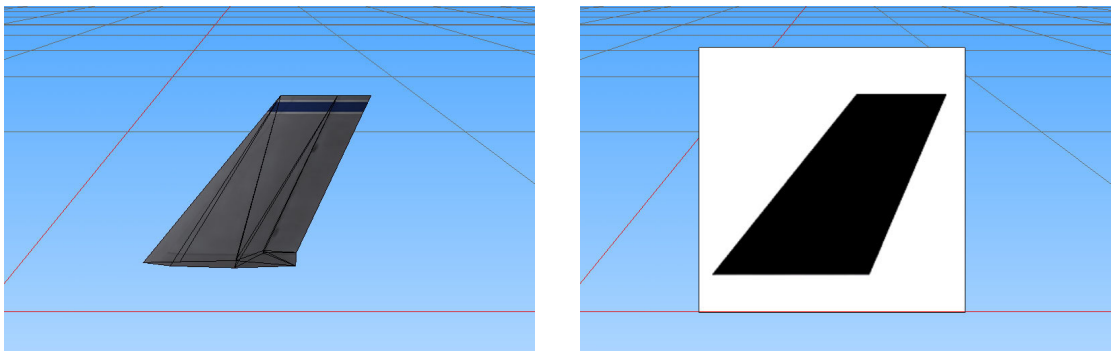


Figure 7. The Tail Fin Geometry and a Simplified Occluder Texture.

The key to the vertex shader is the math to intersect a line with a plane, similar to projected planar shadows. First, we have defined a line with an end point ($\boldsymbol{p}_0$) at the current vertex in object-space traveling in the direction toward the light source ($\boldsymbol{L}$), which must be transformed into local object-space. Figure 6 shows how the parameters for this line would appear in object-space. Next, we define the shadow plane from its origin ($\boldsymbol{o}$) and normal ($\boldsymbol{n}$) as determined by the modeler to cut an occluder geometry in half. Figure 7 shows how occluder geometry can be approximated by an occluder texture on a shadow plane.

Now we can define the intersection point in terms of the parameter s for the line $\boldsymbol{p}(s)$. The intersection point is called $\boldsymbol{p}(s_I)$. See Figures 8 and 9.

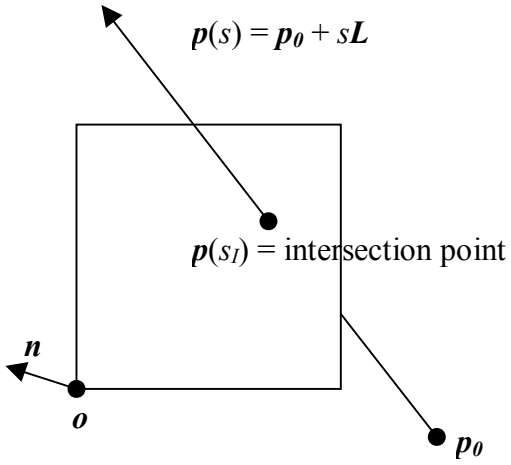


Figure 8. A Schematic Diagram of the Intersection Variables.

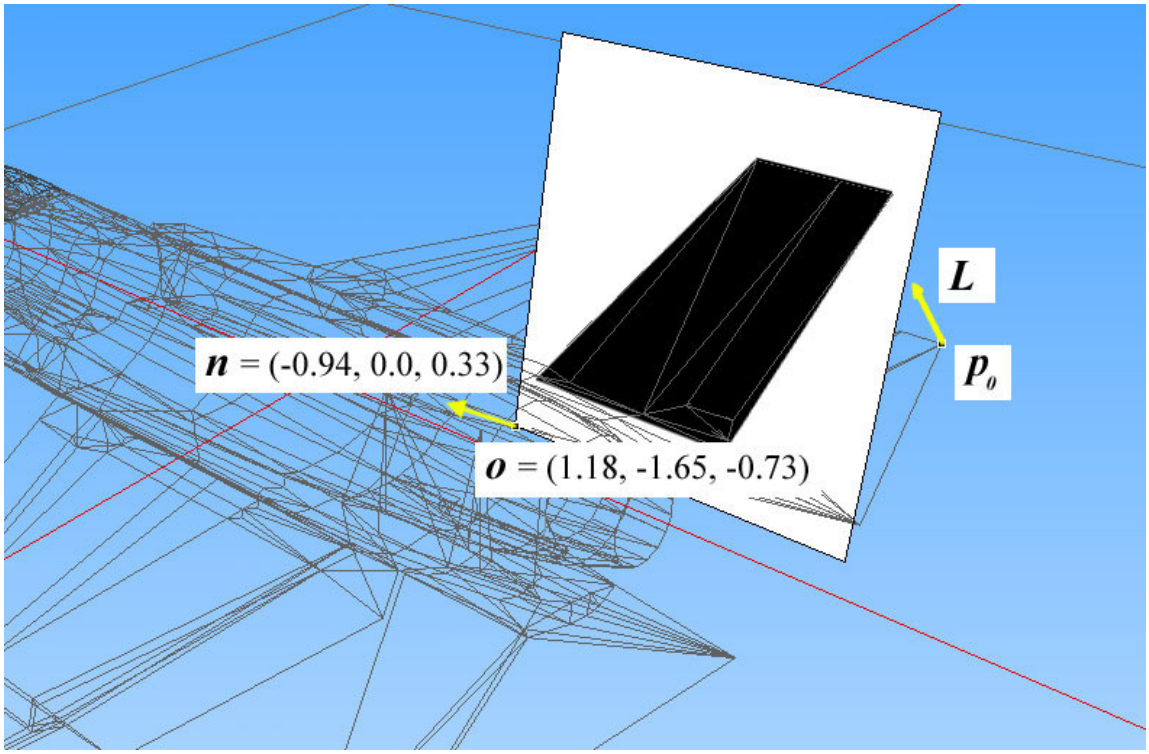


Figure 9. The Occluder Texture Rendered on the Shadow Plane for the Right Tail Fin.

A vector from the shadow plane's origin (***o***) to the intersection point ***p***(s_I) can be found by just subtracting the two vectors: ***p***(s_I)-***o***. This new vector is perpendicular to ***n.*** In Figure 10, a side-view of the intersection shows that a vector from the intersection point to the origin for the shadow plane is perpendicular to the normal for that plane. This means that the dot product between ***n*** and ***p***(s_I)-***o*** is 0.

Figure 10. A Side-View of the Intersection Variables in the Shadow Plane.

$$0 = \boldsymbol{n} \bullet (\boldsymbol{p}(s_I) - \boldsymbol{o}) \tag{3}$$

Substituting the definition of ***p***(s) from equation (2),

$$0 = \boldsymbol{n} \bullet (\boldsymbol{p}_0 + s_I \boldsymbol{L} - \boldsymbol{o}) \tag{4}$$

A little algebraic rearranging,

$$\begin{aligned} 0 &= \boldsymbol{n} \bullet ((\boldsymbol{p}_0 - \boldsymbol{o}) + s_I \boldsymbol{L}) \\ 0 &= \boldsymbol{n} \bullet (\boldsymbol{p}_0 - \boldsymbol{o}) + s_I (\boldsymbol{n} \bullet \boldsymbol{L}) \\ -s_I (\boldsymbol{n} \bullet \boldsymbol{L}) &= \boldsymbol{n} \bullet (\boldsymbol{p}_0 - \boldsymbol{o}) \\ s_I (\boldsymbol{n} \bullet \boldsymbol{L}) &= -\boldsymbol{n} \bullet (\boldsymbol{p}_0 - \boldsymbol{o}) \\ s_I (\boldsymbol{n} \bullet \boldsymbol{L}) &= \boldsymbol{n} \bullet (\boldsymbol{o} - \boldsymbol{p}_0) \end{aligned} \tag{5}$$

Solving for s_I:

$$s_I = \frac{\boldsymbol{n} \bullet (\boldsymbol{o} - \boldsymbol{p}_0)}{\boldsymbol{n} \bullet \boldsymbol{L}} \tag{6}$$

Note that if the shadow plane aligns with one of the major axes of the object's space, equation 6 simplifies considerably. For the "z-up" shadow plane, where $\boldsymbol{o}$ = (0,0,0) and $\boldsymbol{n}$ = (0, 0, 1), equation 6 simplifies down considerably to:

$$s_I = \frac{-\boldsymbol{p}_{0z}}{\boldsymbol{L}_z} \tag{7}$$

Whether using equation 6 or equation 7, substituting s_I into equation (2) results in a position in object space on the surface of the shadow plane. The next step is to derive texture coordinates from the three dimensional object-space position $\boldsymbol{p}(s_I)$ just computed by using an up vector and right vector for the shadow plane. Visually, the projection of the positions onto the shadow planes looks like Figure 11 below.

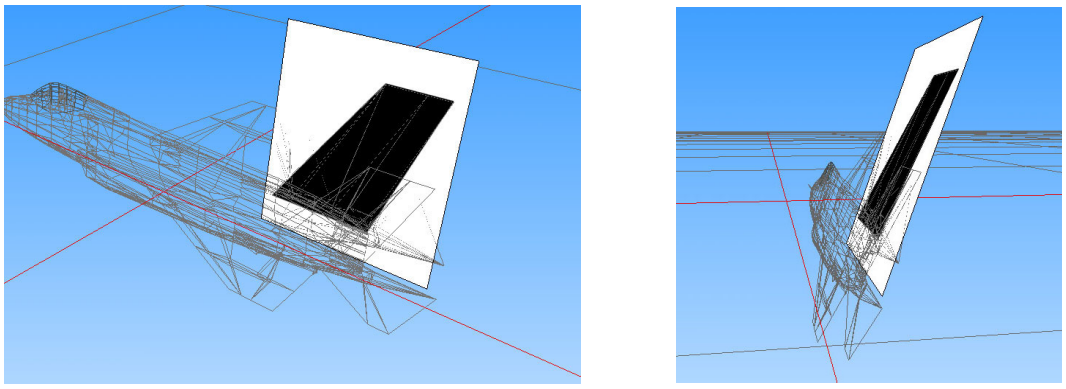


Figure 11. The Shadow Receiving Geometry Projected onto the Shadow Plane.

To compute the shadow coordinates, simply find the vector $\boldsymbol{o} - \boldsymbol{p}(s_I)$, compute the dot product with the ***up*** and ***right*** vectors, and normalize by width and height. See Figure 12 and equation 8 below.

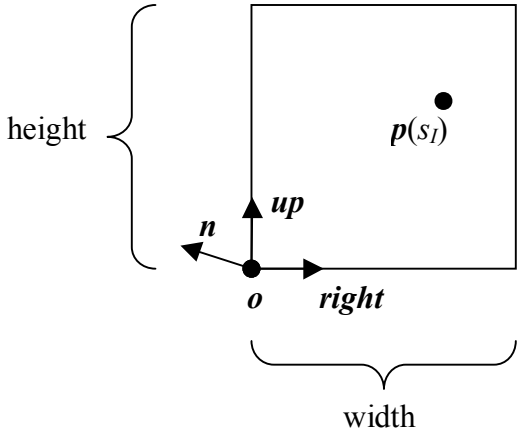


Figure 12. Computing Shadow Coordinates from the Intersection Point $p(s_I)$.

$$texcoord_x = \frac{\boldsymbol{right} \bullet (\boldsymbol{o} - \boldsymbol{p}(s_I))}{\text{width}}$$
$$texcoord_y = \frac{\boldsymbol{up} \bullet (\boldsymbol{o} - \boldsymbol{p}(s_I))}{\text{height}} \qquad (8)$$

Finally, the sign of the parametric value (s_I) determines whether or not the shadow plane is between the vertex's position and the light source. Projection works in both the positive and negative directions. A shadow projected from a shadow plane that is not between the shadow receiving vertex and the light source is called a false shadow. To avoid false shadows, the sign of (s_I) is passed to the fragment shader so that it can determine whether or not to look-up into the occluder texture for each shadow plane.

4.2 Writing the Shaders

The vertex shader has the job of projecting the vertex positions of the model into the shadow plane. The equations in Section 4.1 are implemented in the vertex shader with the specific values for ***o***, ***n***, ***up***, ***right***, width, and height encoded for each shadow plane of the object. The light direction (***L***) varies at run-time and $\boldsymbol{p}_0$ is simply the incoming vertex position in object-space. The fragment shader has the job of accessing the occluder textures to see whether the fragment is in shadow.

4.3 Creating the Shadow Planes and Occluder Textures

Some attention should be given to choosing the shadow planes in the first place. The goal is to pick the minimum number of planes required to cast shadows. This technique is limited by the number of hardware

texture units available and the number of texture lookups that can be performed in the fragment shader while maintaining high frame rates. The goal is not to produce perfect shadowing from all surfaces of the object, but instead to provide enough of the obvious shadows well enough to fool the eye into believing that the object's rendering is shadowed correctly.

This technique can be thought of as a first-order approximation of shadowing, so the most important shadow casting surfaces must be considered, rather than all surfaces that can potentially cast shadows. Small surfaces can sometimes be ignored because they typically cast small shadows, and the eye doesn't immediately notice the lack of a small shadow when there are larger shadows present. Rounded and spherical surfaces can usually be ignored because shading gives a good indication of the shape of rounded surfaces without the need for self-shadowing of that surface. For example, a sphere does not actually cast shadows onto itself. Large, flat, protruding surfaces should definitely be included when choosing shadow planes. Large surfaces, such as the spout on a teapot, or the tail fins of an aircraft will obviously cast shadows onto an object. See Figure 13 below.

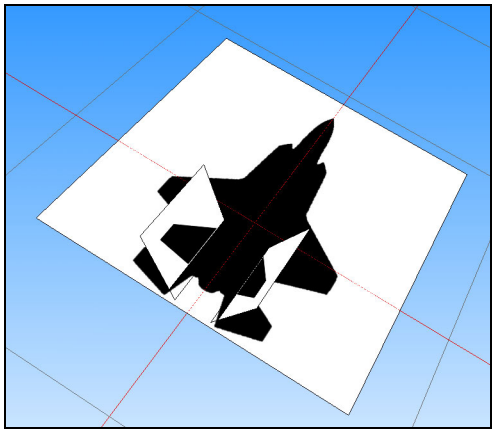


Figure 13. F-35 Shadow Planes in Isolation.

In Figure 13, the wings, fuselage, and horizontal stabilizers (basically, everything on the X-Y plane) are chosen as a single shadow plane. The F-35 model is well suited for this technique because so many of its shadow casting surfaces can be approximated with a single shadow plane that cuts the airplane into top and bottom halves. Next, the two vertical tail fins are chosen to be shadow casters, which require two more shadow planes. The landing gear, or the little surface that sticks up just behind the cockpit, or even the silhouette of the man in the cockpit could have been chosen as additional shadow planes. However, the three planes already chosen are sufficient to provide visual cues for self-shadowing with the most "bang for the buck."

Shadow planes are modeled just as any other part of the model, with the exception that they are flagged as invisible, non-rendering surfaces. The occluder textures are created by an artist using Photoshop. It is useful to render an orthographic view of the model or the portion of the model pertinent to the occluder texture with lighting disabled, flat-shaded, black in color, on a white background as a starting point for the artist. The artist has complete control over the look of the shadows, what portions of the object cast shadows, and the resolution of the occluder textures.

After shadow planes and occluder textures are modeled, the values for the origin of the shadow plane, its normal, and vectors along its width and height are constants that can be hard-coded into the vertex shader to compute shadow coordinates as discussed in previous sections. Alternatively, a future implementation of this technique could write the vertex and fragment shaders in a more generic manner such that these constants could be parameters passed into the vertex and fragment shaders. The occluder texture is assigned to a texture stage for the model. The vertex and fragment shaders are also assigned to the model. The result is a self-shadowed model whose shadows change immediately as light direction or orientation of the model itself changes.

5. IMPLEMENTATION

To implement and test this approach, a set of test models was chosen. First, a model that is well-suited for this technique was chosen. The F-35 model used in previous sections was chosen as a good candidate for this technique because several of its surfaces can be combined into a single shadow plane, and it has the large protruding surfaces of the vertical tail fins that are nearly planar and can easily be represented by a shadow plane and occluder texture. Shadow planes were selected, occluder textures were created, and the vertex and fragment shader source code specific to this model was written. The process and the decisions made at each step are described below.

First, one of the vertical tail fins was chosen to try the theory out. The geometry of the tail fin was isolated and rendered into a 256x256 image where the tail fin geometry was rendered black on a white background. The shadow plane was modeled as "hidden" geometry with the occluder texture applied. The edges of the silhouette in the occluder texture were visually found to match closely with the occluder geometry of the tail fin. Then, Photoshop was used on that image to extend the shape of the tail fin downward so that when the occluder texture was applied to the shadow plane in the modeling package, the black silhouette penetrated down into the object rather than just touching the edge of the object. This is required because the discreet, raster nature of the silhouette edge in the occluder texture cannot precisely match the geometric edge, which is computed with floating-point precision. Most of the silhouette extension will not actually ever be seen, but it ensures that projected shadows will reach the edge of the occluding geometry. See Figure 7 in Section 4 to see the tail fin and its occluder texture.

A simple vertex shader to compute the shadow coordinates using the math from Section 4 was written. The origin, normal, and horizontal and vertical axes of the shadow plane for the tail fin were hard-coded as constants into the equations in the vertex shader rather than passing the values as parameters to a more generic shader. This allowed the shader to be bundled with the model and did not require any input parameters not already provided by the run-time. Appendix A contains the vertex shader source code. A fragment shader to perform the occluder texture lookup and modify the color of each shadowed fragment was written (see Appendix B). The shaders and textures were applied to the model, and the results observed. The shadow from the tail fin cast correctly onto the fuselage of the aircraft and the tail fin also shadowed itself successfully.

At this point, the model was un-shadowed, except for shadows cast from one of the vertical tail fins. For the F-35 model, the other vertical tail fin has exactly the same shape. The vertex and fragment shaders

were modified to compute a second set of shadow coordinates for the same occluder texture. The model now had two shadow planes utilizing a single occluder texture.

The aircraft with the two tail fins casting shadows looked good for most times of day, but it was obvious that the bottom side of the aircraft was not getting shadows from the wings and horizontal tail stabilizers. A third shadow plane that cut the entire aircraft into a top half and bottom half was chosen. An occluder texture was created by rendering the aircraft in an orthographic view as before, but the vertical tail fin geometry had to be removed so that it did not show up in the silhouette. The image was brought into Photoshop to clean up holes where the tail fins used to be. Also, while modeling the shadow plane, some of the vertices of the aircraft had to be subtly modified to make it possible to cut the airplane exactly along the z=0 plane. The rest of the work in this thesis used the model with slightly modified vertex positions. If the original aircraft geometry had been required, all of the vertex positions of the modified aircraft could be stored into a set of 3D texture coordinates for the original aircraft. The vertex shader would have to be changed to operate on the positions stored in the texture coordinate for the model, rather than the actual positions of the vertices.

The result was a self-shadowed model that looked reasonably shadowed, rendered in a single pass. The rendering performance was very high, and will be described in detail in Section 6. The image quality was very good because the shadow planes exactly bisect the aircraft and the math performed in the vertex shader to calculate shadow coordinates results in floating-point precision for the intersection with the shadow plane. The raster nature of the silhouette in the occluder texture does result in "jaggies" where the texture is stretched onto the aircraft, but the resolution of the occluder texture was chosen high enough that the artifacts are minimal unless the eye point is directly over the edge of the shadow from very close range. Enabling bilinear filtering on magnification of the texture also minimizes the artifacts. No extra shadows in unshadowed areas were observed, and the shadows edges reached exactly up to (and even inside) the shadow casting geometry. See Figure 14.

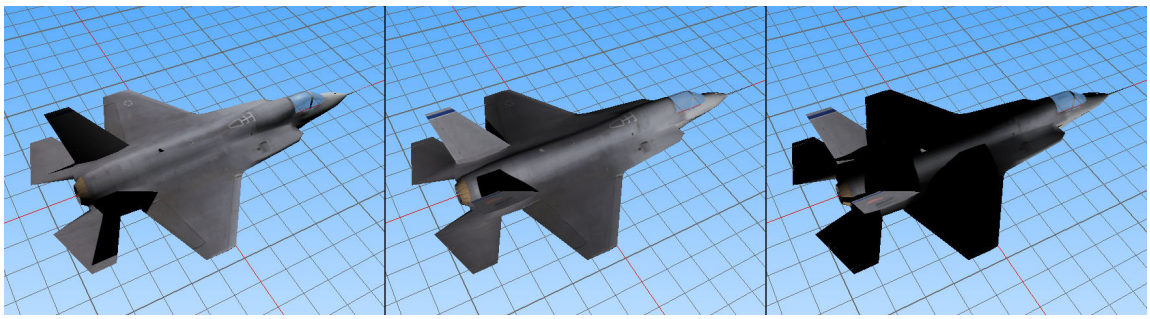


Figure 14. The F-35 Self-Shadowed Model Viewed Under Different Light Conditions.

The second model attempted was a teapot. See Figure 15. The teapot was chosen because it is a fairly well-known example of a difficult shape for shadowing. While an aircraft had proven an ideal model for this technique, the teapot proved to be more of a challenge. The aircraft has several areas of planar geometry that can easily be represented by a shadow plane and an occluder texture. The teapot, on the other hand, has large, non-planar protrusions such as the spout, top, and handle that cannot easily be represented by a plane.

Figure 15. Teapot Without Shadows.

The handle seemed the simplest to represent as a plane that would cut through with a "C" shaped silhouette, but that plane only produces shadows when the light source is perpendicular to the handle. If the light source is above, the shadow becomes infinitely thin and disappears. The thickness of the handle should actually cast a shadow straight down. The handle would require multiple shadow planes to handle all lighting conditions.

Similarly, I attempted to model the shadow from the spout with a single shadow plane that cut along the same axis as the shadow plane for the handle. Shadows when the light source was above produced the same flawed result. The spout would require more than one plane to represent its shadows when the light was coming from different directions. Finally, the top of the teapot could not be represented by a single shadow plane for the same reason as the handle and spout.

Since each of these parts would require multiple shadow planes, the teapot would require several occluder textures. The number of texture lookups required per fragment would reduce the overall performance of the technique. Also, some graphics hardware limits the number of textures applied to any polygon to as few as four textures. Luckily, the bulk of the teapot is round, and as described in Section 4 cannot cast shadows onto itself. Interestingly, the front of the teapot can only receive shadows from the spout. The back of the teapot can only receive shadows from the handle. The top of the teapot can only receive

shadows from the top of the teapot. This allowed the teapot to effectively be cut into three separate objects to apply this technique to. Any one section of the teapot would only have to consider shadow planes for that section. This required three different sets of vertex and fragment shaders, shadow planes, and occluder textures. Implementing the technique for this one model required the work of applying the technique to three separate models.

First, three shadow planes were chosen for the top of the teapot. Two perpendicular planes cut the geometry vertically with the same silhouette, and a third plane intersected the widest point of the knob on the top of the teapot with a circular silhouette. See Figure 16 below.

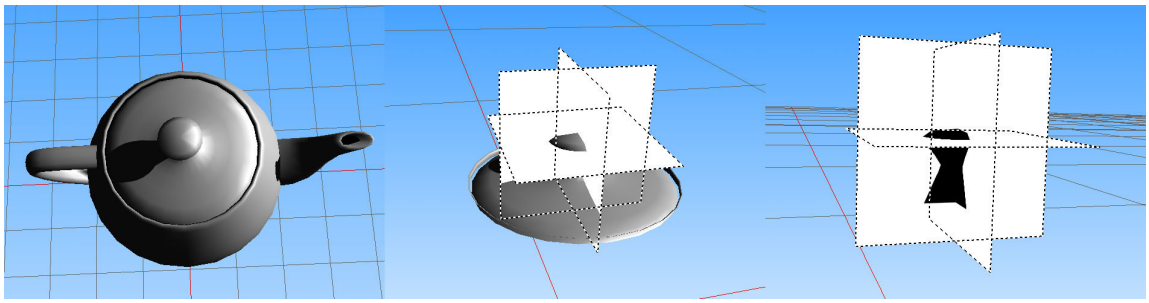


Figure 16. The Top of the Teapot and its Shadow Planes.

Second, the spout was attempted. I first tried to use two intersecting shadow planes, hoping to represent the thickness of the handle, as well as its side profile. Unfortunately, the shadows appeared strange when the light source direction was nearly straight down the handle. The "X" pattern of the shadow planes was apparent in the shadows that they cast on the object. Also the shadows rarely reached the edge of the occluder geometry of the spout. See Figure 17.

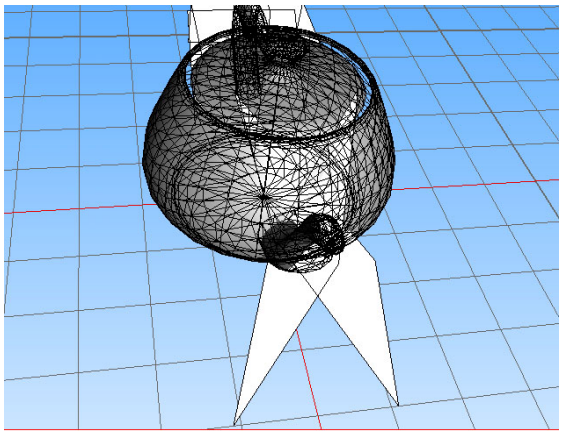


Figure 17. The Spout's Unsuccessful Shadow Planes.

To correct this, three planes were chosen, similar to the way the top was represented as shadow planes and occluder textures, except on an angle. The horizontal profile, vertical profile, and a "cap" to avoid seeing the "X" pattern at the tip of the shadow were used. There are still angles for which the thickness of the spout does not project correctly, and the shadows do not reach all the way up to the occluder geometry. The spout does not self-shadow. See Figure 18.

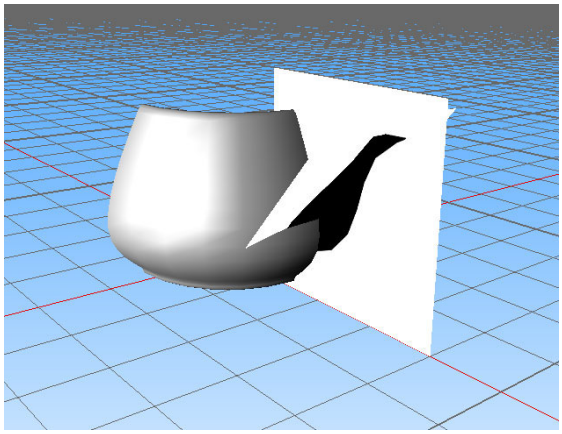


Figure 18. The Spout's Shadow Planes.

Finally, the handle proved the hardest challenge of all. I first attempted to model the handle with four shadow planes, one plane producing the "C" shaped silhouette, plus three to capture the shape and width of the handle. Unfortunately, while the shadow planes can be placed with extreme precision, the occluder textures are a discrete, raster representation of a cross-section of the geometry and it is very difficult for the textures to overlap where the texels exactly touch, but do not extend past one another. See Figure 19 below.

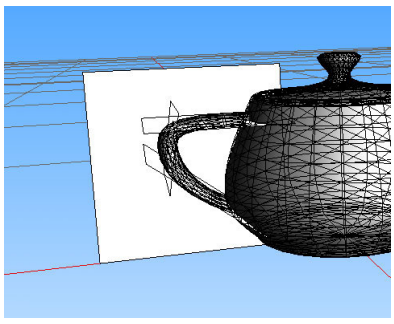


Figure 19. The Handle's Unsuccessful Shadow Planes.

The three planes for the handle produced nice looking shadows when the light was directly overhead, but if the light was just slightly off, the overlapping nature of the three planes became apparent as part of the shadow would appear to extend too far. Also, the shape of the shadow was extremely segmented as the three segments approximating the hundreds of polygons of the handle were very apparent. To shadow the handle, the shadow planes would have to be modeled directly end to end, and more segments would be required. The best results were obtained using five segments to represent the handle and no occluder textures at all. Instead, the occluder texture was assumed to be solid black so that if the texture coordinates were in the range [0,1], then the fragment is considered to be in shadow. See Figure 20 below for the shadow planes used to represent the handle.

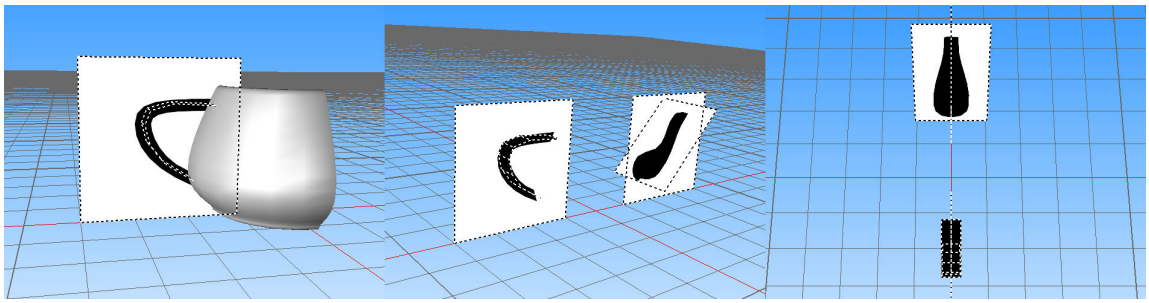


Figure 20. The Handle's Shadow Planes.

Overall, the teapot is not well-suited for this technique. I was, however, able to successfully model shadow planes and occluder textures that produce reasonable shadows. Figure 21 shows the teapot from various angles with the resulting shadows.

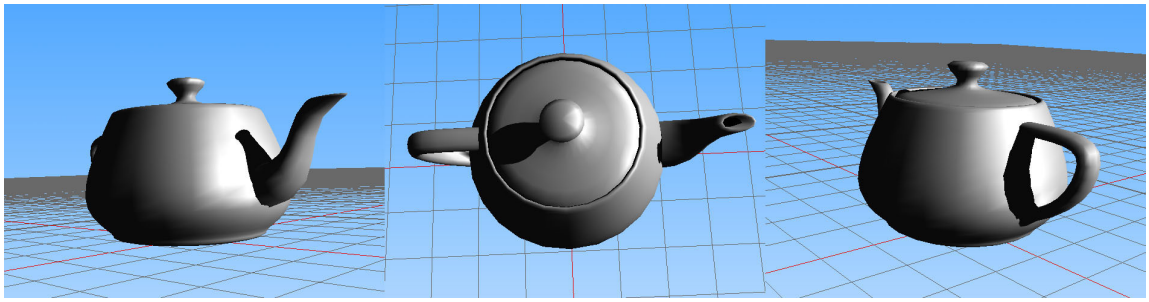


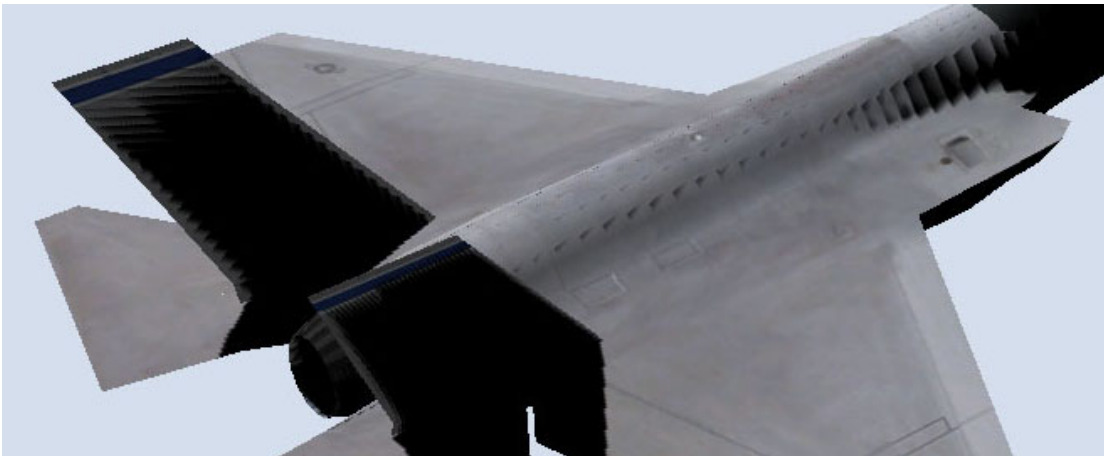
Figure 21. The Teapot with Shadows.

6. EVALUATION AND RESULTS

6.1 Image Quality Evaluation

To evaluate the occluder texture technique, I compared the images produced and performance to the shadow map technique, which is independent of modeling steps, as described in Nvidia's white paper [4]. The same scene was rendered with the same resolution (1024x1024) from a fixed eye position on the same hardware for both techniques. The graphics hardware was an NVIDIA 7900 GTX. The results are shown below.

Occluder Texture Technique



Shadow Map Technique with Artifact Close-up

Figure 22. F-35 Image Quality Comparison.

Figure 22 shows the image quality of the two techniques for the F-35 model. The shadows produced by the shadow map technique could not reach the top of the tail fins due to depth precision errors. Also, the shadows at the front of the plane appear blocky. This aircraft should not have air intakes resembling gills. Instead, a severe artifact caused by the stretching of the projected depth texture across the curved surface exaggerates the depth precision and texture resolution problems associated with shadow maps making it appear as if this aircraft has gills. In this case, the occluder texture technique produces superior image quality results due to the relatively low number of noticeable artifacts.

Occluder Texture Technique **Shadow Map Technique**

Figure 23. Teapot Image Quality Comparison.

Figure 23 shows the image quality of the two techniques for the teapot model. Comparing the images for the teapot reveals the problem inherent to shadow maps, shadow acne. The depth buffer does not have enough accuracy to perfectly reproduce shadowing, and the depth values are stretched across the model, further compounding the errors. The curved surface of the right half of the teapot shows an artifact similar to the "gills" on the F-35. The blocky nature of the artifact on the curved surface is due to the angle of the light source and the stretched projection of depth values. One depth value on the edge of the curved surface from the point of view of the light source may be stretched across a large vertical region of the curved body of the teapot from the point of view of the observer. Overall, the shadows can have blocky areas and rarely reach the edges of shadow casters.

I believe that the occluder texture technique compared favorably in image quality. One measure of image quality is the lack of noticeable artifacts. The most noticeable artifact for the occluder texture technique is

that shadows are missing from some parts of the model. For the F-35, the missing shadows are rarely noticed unless in a side by side comparison as is being made here. For the teapot, the shadows from the lip of the teapot onto the handle are noticeably missing, as are the shadows on the interior of the handle. Shadow maps, on the other hand, will capture all of the shadows for a model, but often have many rendering artifacts which lower the image quality. The occluder texture technique might miss some shadows, but the shadows that it does produce are relatively artifact free.

6.2 Performance Evaluation

Comparing performance proved challenging because the graphics hardware could perform both techniques at well over 1000 Hz for a single instance of either model. Multiple models had to be added to the scene to be able to compare performance numbers. A fixed eye position was used, as well as fixed locations for each of the models to provide the same conditions for testing both techniques. Figure 24 shows the rendered scene for 200 models.

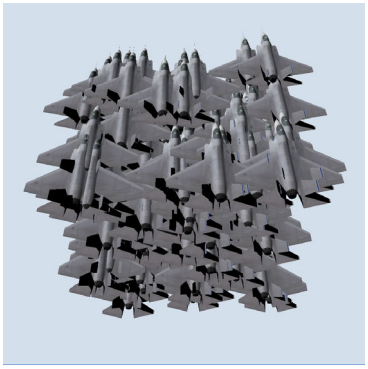


Figure 24. 200 instance of the F-35 used in Performance Comparison.

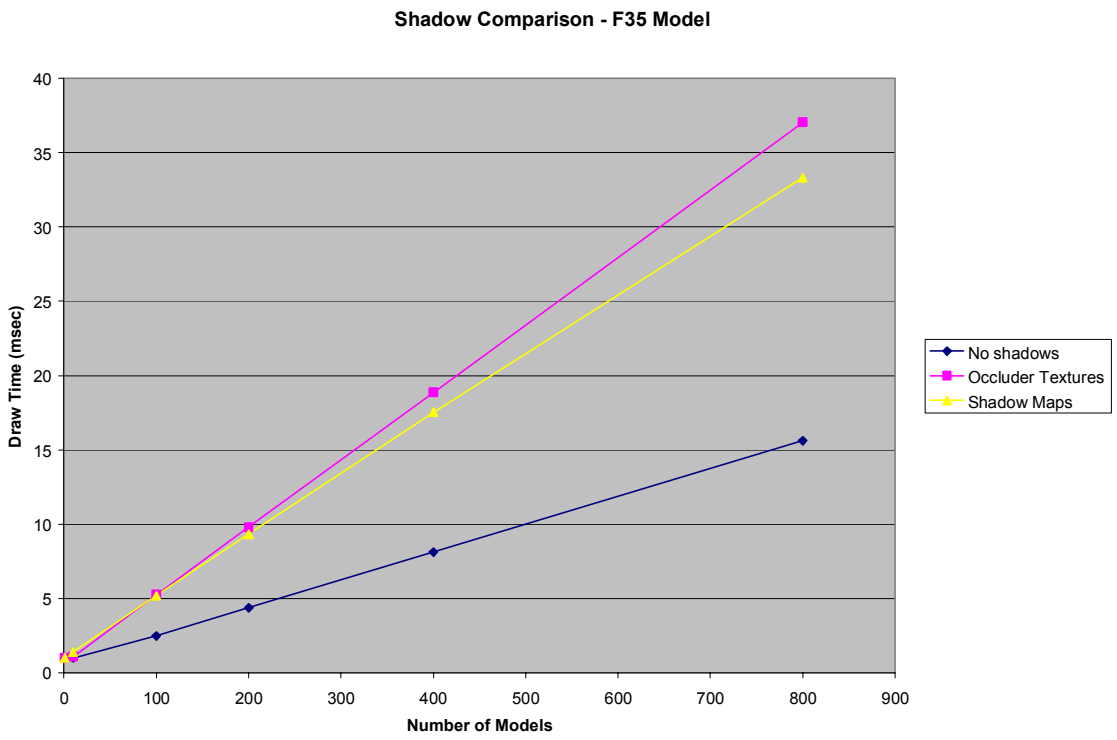


Figure 25. F-35 Performance Comparison.

Figure 25 shows a chart of the draw times required for varying numbers of the F-35 model randomly spread in the field of view. The F-35 produced surprising results. Although the occlusion texture technique is single pass, it only manages to match the performance of the shadow map technique which requires two passes. The F-35 is considered an ideal model for the technique because it is simple to model with shadow planes and occluder textures. The technique did not result in improved performance, however.

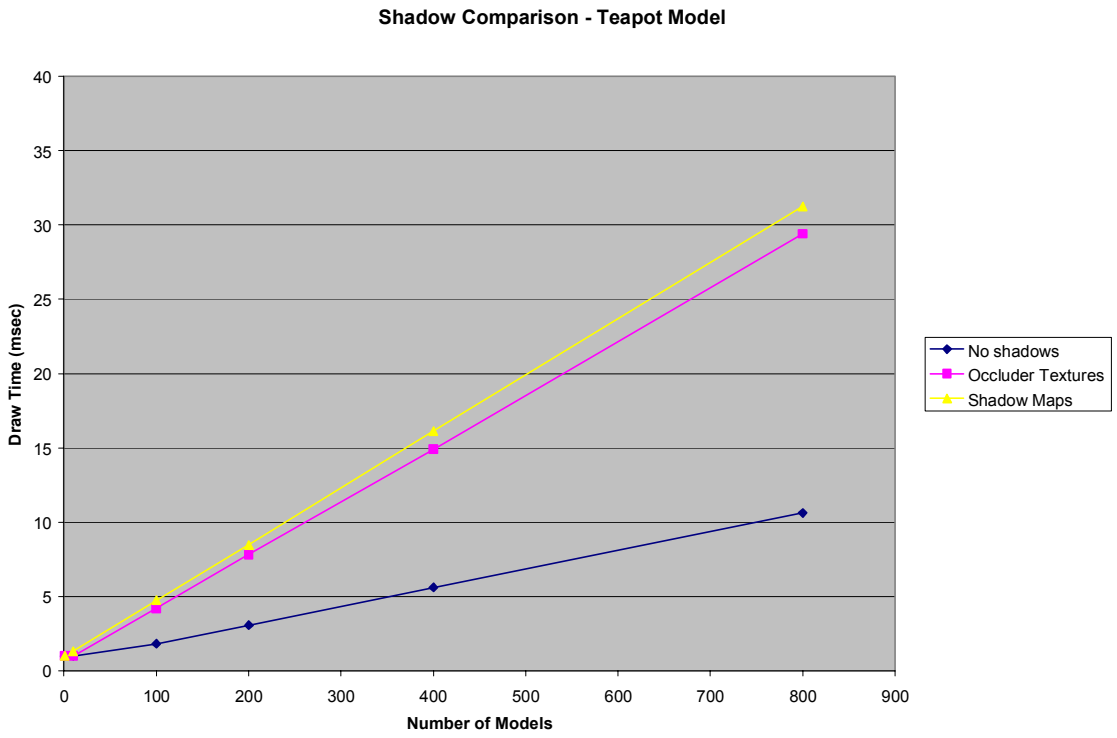


Figure 26. Teapot Performance Comparison.

Figure 26 shows a chart of the draw times required for varying numbers of the teapot model. As you can see, again the occluder texture technique has nearly exactly the same performance characteristics as the shadow map technique. Amazingly, even though the teapot required many more shadow planes, the performance did not suffer as expected, but rather performed slightly better than the shadow map technique for this model.

The unexpected performance results warranted further investigation. Performance testing is a tricky subject because there are so many variables involved. For graphics applications, a performance bottleneck may exist in one of three places. First, making OpenGL calls which pass data from the CPU to the GPU can create a bottleneck referred to as being *CPU limited*. Second, vertex processing can become expensive, or there may be too many vertices in the scene causing an application to be *vertex processing limited*. Finally, texture lookups and fragment processing can become the bottleneck when rendering an object. This is referred to as being *fragment processing limited*, or "fill" limited.

To find the bottlenecks, the F-35 test case with 100 copies of the model was chosen because it has enough copies of the model to be able to measure performance differences, but not so many copies that the scene

is dominated by overlapping models that would cause portions of the screen to be "filled" multiple times. Recall from the graph in Figure 24 that the occluder texture technique performed only slightly worse than the shadow map technique for this test case, which renders the scene from the fixed eye position. An image of a similar rendering can be seen in Figure 25. CPU limiting was tested by sending the draw commands for the F-35 in *immediate mode*. This means calling OpenGL API to pass all of the data for every vertex of every instance of the F-35 model one at a time. For example, calling glBegin(GL_TRIANGE_STRIP), followed by many calls to glVertex3f() and glNormal3f(). Immediate mode rendering is known to cause CPU limited bottlenecks. In the test scenario, both the occluder texture technique and the shadow map technique dropped severely in framerate, although the occluder texture technique did not drop as much as the shadow map technique. The occluder texture technique was favored in performance because it is a single pass technique, only requiring the draw commands to be issued once, while shadow maps require all of the draw commands to be issued twice. This interesting result proves that single pass rendering techniques, such as occluder texture shadows, are preferred in applications that are CPU limited.

The rendering mode was switched back to display lists, which greatly reduce the number of calls from the CPU to the GPU using the graphics API. Thousands of vertices and normals can be cached in a display list, which allows future calls to render those vertices to be made by simply referencing a display list ID. With display lists enabled, the CPU bottleneck was once again eliminated and performance returned to its earlier state.

Next, fragment processing limits were tested by moving the eye position closer to one of the F-35 instances, looking straight down on it, and scaling all of the F-35 instances up by a factor of ten. In this manner, the entire screen is filled multiple times by multiple F-35 surfaces. This test did cause a fragment processing, or fill bottleneck. In this case, the shadow map technique performed noticeably better because the technique only requires one texture lookup per-fragment and no conditionals (if statements) when rendering the shadows for an object. The occluder texture technique on the other hand, required three texture lookups per fragment for the F-35 test object. This test proves that when fragment processing is the bottleneck, the shadow map technique performs better than the occluder texture technique.

Finally, to test for vertex processing bottlenecks, two approaches were used. First, the original eye position was restored and rather than scaling the model up by a factor of ten, it was scaled down by a factor of one-hundred. This results in the same amount of vertex processing as the original test case, but with fewer fragments and therefore less fragment processing. The shadow map technique improved in performance, but the occluder texture technique did not. This means, for the test case chosen, the occluder texture

technique is vertex processing limited, while shadow maps are fill limited. To improve the performance of the occluder texture technique, the math that performs the shadow coordinate computation could be further optimized. For example, every vertex is currently computing the light direction local model-space. Instead, the light direction vector could be computed once per instance of the F-35 and passed into the vertex shader as a parameter. This would reduce vertex processing, alleviating the bottleneck for the technique, and allow the occluder texture technique to outperform the shadow map technique for the test case used in this thesis.

To test this hypothesis of reducing vertex processing to achieve better performance for the occluder texture technique, the vertex shader was simplified. Shadow coordinate computation was replaced with constant shadow coordinates, and the light direction vector was no longer transformed into the local model space or even used. The result was as expected. The occluder texture technique no longer functioned, but the performance was noticeably superior to the shadow map technique. Even when computing only a single shadow coordinate (rather than all three), performance was better for the occluder texture technique than the shadow map technique. This means that on a model by model case, depending on the number of shadow planes, and the complexity of the shadow coordinate computations, either the shadow map technique or the occluder texture technique may yield superior performance. One of the possibilities for future work for this thesis is to improve the performance of the shadow coordinate computation and reduce the possibility of being vertex processing limited.

6.3 Results

- The occluder texture technique produces self-shadowing.
- The occluder texture technique renders in a single pass.
- Shadow planes and occluder textures were easily modeled for the F-35 test model.
- The occluder texture technique was surprisingly only equal in performance to the shadow map technique for the F-35 test model. The difference between a two pass approach and a single pass approach was negated by the extra work being done in the single rendering pass and the relatively little work done in the first pass for the shadow map approach. For the test case used here, the occluder texture technique was vertex processing limited, so an improvement in shadow coordinate computation would allow the technique to perform better.
- The occluder texture technique creates higher quality images than the shadow map technique for the F-35 test model because of the lack of distracting artifacts.

- Shadow planes and occluder texture were modeled for the teapot (a non-ideal test model) after much trial and error. The occluder texture technique can be applied to almost any shape if it can be broken into smaller parts of the model that do not shadow one another.
- The occluder texture technique performed well, actually performing slightly better than shadow maps for the teapot model.
- The occluder texture technique creates higher quality images than the shadow map technique for the teapot test model. However, the teapot does show artifacts in the shadows because of the low number of shadow planes that are attempting to represent the teapot geometry. Some of the shadows, especially the handle, appear segmented. Also, the shadow map technique allows different areas of the teapot to cast shadows onto other areas.

Overall, both techniques have their advantages and disadvantages. For self-shadowing of aircraft, the occluder texture technique is probably preferred. For shadows cast by that aircraft onto the ground, shadow maps or projective shadows would be preferred. A single application could certainly use both techniques to get the best of each.

7. CONCLUSION AND FUTURE WORK

This research enables future work in at least three areas: applying the technique to less “ideal” models, improving performance, and automating the production of shadow planes and occluder textures. The proposed technique can be applied to a wider range of models by allowing more shadow planes. Methods for computing intersections with multiple shadow planes simultaneously could be derived. Also, multiple occluder textures might be combined into a single texture through texture atlases, three dimensional textures, or just using the red, green, blue, and alpha channels of a texture independently. Improving the performance of the shadow coordinate calculation may be possible by using texture matrices to perform the projection. Similarly, pre-aligning the model to an axis and storing the results as a texture coordinate for simplified projection may further improve performance. Methods for automated choice of cut-planes for the shadow planes could be investigated, and methods for automated generation of occluder textures could also be explored.

Also, hardware makers could introduce further improvements in the hardware to help with hardware acceleration of this technique. Both shadow maps and stencil shadow volumes are taking advantage of hardware improvements designed specifically for those techniques. For the occluder texture technique, performance for texture lookups could be improved, where several textures could be “fetched” in parallel since the occluder texture technique often requires many texture lookups. Also, improved conditionals (if statements) in the fragment processing could avoid some texture lookups altogether. Once any of the occluder texture lookups has determined that a fragment is in shadow, the other occluder textures do not have to be queried. Finally, the hardware could improve the vertex processing, which was the bottleneck for this test case, by always providing the light direction vector in model-space as a part of the state available to shader writers without having to transform the vector for each vertex in the shader code.

REFERENCES

[1] M. Bunnell, "Dyanmic Ambient Occlusion and Indirect Lighting," *GPU Gems 2*, pp. 223-233, 2005.

[2] F. Crow, "Shadow Algorithms for Computer Graphics," *Proc. ACM SIGGRAPH '77*, pp. 242-248, 1977.

[3] C. Everitt, "Projective Texture Mapping," technical report, NVIDIA Corp., http://developer.nvidia.com/object/Projective_Texture_Mapping.html, 2001.

[4] C. Everitt, "Shadow Mapping," technical report, NVIDIA Corp., http://developer.nvidia.com/object/shadow_mapping.html, 2001.

[5] C. Everitt and M. Kilgard, "Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering," technical report, NVIDIA Corp., http://developer.nvidia.com/object/robust_shadow_volumes.html, 2002.

[6] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed., Reading, Mass.: Addison-Wesley Publishing. 1997.

[7] M. J. Kilgard, "Creating Reflections and Shadows Using Stencil Buffers," Online Presentation, http://www.opengl.org/resources/features/StencilTalk/index.htm, 2000.

[8] S. Kozlov, "Perspective Shadow Maps: Care and Feeding," *GPU Gems*, pp. 217-244, 2004.

[9] T. Moller and E.Haines, *Real-Time Rendering*. Natick, Mass.: A K Peters, Ltd. 1999.

[10] A. Rege, "Shadow Considerations," technical report, NVIDIA Corp., http://developer.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_shadows.pdf, 2004.

[11] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli, "Fast Shadows and Lighting Effects Using Texture Mapping," *Proc. ACM SIGGRAPH '92*, pp. 249-252, 1992.

[12] L. Williams, "Casting Curved Shadows on Curved Surfaces," *Proc. ACM SIGGRAPH '78*, pp. 270-274, 1978.

APPENDIX A

EXAMPLE VERTEX SHADER SOURCE

```
//-----------------------------------------------------------------------
// appin - the per-vertex attributes to be passed in from the
//         application using standard OpenGL conventions
//-----------------------------------------------------------------------
struct appin
{
  float4 position  : POSITION;
  float3 normal    : NORMAL;
  float4 color0    : COLOR0;
  float4 texcoord0 : TEXCOORD0;
};

//-----------------------------------------------------------------------
// vertout - the per-vertex attributes to be passed to the
//           fragment shader, interpolated across the triangle
//-----------------------------------------------------------------------
struct vertout
{
  float4 position  : POSITION;
  float4 color0    : COLOR0;
  float4 texcoord0 : TEXCOORD0;
  float4 texcoord1 : TEXCOORD1;
  float4 texcoord2 : TEXCOORD2;
  float4 texcoord3 : TEXCOORD3;
  float4 ambient   : TEXCOORD4;
};

//-----------------------------------------------------------------------
// f35ShadowVertexShader - the main program
//-----------------------------------------------------------------------
vertout f35ShadowVertexShader( appin IN )
{
  vertout OUT;

  // glstate is a pre-defined Cg parameter that queries the OpenGL state
  // machine directly. For a full set of available parameters, see the
```

```
// Cg Toolkit User's Manual from NVidia.

const float4x4 modelViewProj = glstate.matrix.mvp;
const float4x4 modelView = glstate.matrix.modelview[0];
const float4x4 modelViewInv = glstate.matrix.inverse.modelview[0];
const float4x4 modelViewIT = glstate.matrix.invtrans.modelview[0];
const float4 lightDirection_eye = glstate.light[0].position;
const float4 lightDiffuse = glstate.light[0].diffuse;
const float4 lightAmbient = glstate.light[0].ambient;
const float4 materialAmbient = glstate.material.ambient;
const float4 materialDiffuse = glstate.material.diffuse;

// compute directional lighting in eye-space

float3 N_eye = normalize( mul( (float3x3)(modelViewIT), IN.normal ) );
float cos_ang = max(dot(N_eye, lightDirection_eye.xyz), 0.0);
float3 diffuse = cos_ang * materialDiffuse.rgb * lightDiffuse.rgb;
// pass the directional color and ambient color to the fragment shader

OUT.color0 = float4(IN.color0.rgb * diffuse, IN.color0.a);
OUT.ambient = lightAmbient * materialAmbient;

// transform the vertex for clipping

OUT.position = mul( modelViewProj, IN.position );

// pass texture coordinates for the base texture

OUT.texcoord0 = IN.texcoord0;

// compute shadow coordinates for the first shadow map

// compute the intersection of a line with the z=0 plane
// the line begins at vertex point and travels along lightDirection
// computed in model-space

float3 lightDirection_model =
           normalize( mul((float3x3)(modelViewInv), lightDirection_eye.xyz));

// find the intersection point in model-space
```

```
// remember that for the z=0 plane, sI = -pos.z/light.z

float2 intersection = IN.position.xy +
          (-IN.position.z/lightDirection_model.z)*lightDirection_model.xy;

// convert to a texture coordinate using
// V0 = (-7.5, -5.315, 0.0)
// N = (0, 0, 1)
// and size of shadow plane = 15.0m on each side

float2 normalizedTexCoord = (intersection + float2(7.5, 5.315)) / 15.0;

// the z coordinate is used to determine if this part of the aircraft is
// closer to the light source than the shadow plane (and should not be
// shadowed), or behind the shadow plane, and should be shadowed.

float zTexCoord = lightDirection_model.z*IN.position.z;
OUT.texcoord1 = float4(normalizedTexCoord, zTexCoord, 1.0);

// compute shadow coordinates for the right tail fin shadow map

// intersection of a line with the shadow plane
// V0 = (0.899235, -0.676381, -0.706742)
// N = (-0.917594, 0.0, 0.397518)
// line begins at vertex point P0 and travels along lightDirection
// computed in model-space

float3 N = float3(-0.917594, 0.0, 0.397518);
float3 V0 = float3(0.899235, -0.676381,  -0.706742);
float3 V0minusP0 = V0 - IN.position.xyz;
float numerator = dot(N, V0minusP0);
float denominator = dot(N, lightDirection_model);
float sI = numerator / denominator;

// find the intersection point in model-space

float3 intersectionVec = lightDirection_model.xyz * sI;
float3 intersectionRTail = IN.position.xyz + intersectionVec;

// convert to a texture coordinate
```

```
// size of shadow plane = 4.1218186m on each side
// the normalized up vector for this tail fin's shadow plane
// is = (0.3975177, 0.0, 0.91759448)

float3 distanceVec = intersectionRTail - V0;
float3 upVec = float3(0.3975177, 0.0, 0.91759448);
OUT.texcoord2.x = - distanceVec.y / 4.1218186;
OUT.texcoord2.y = dot( distanceVec, upVec ) / 4.1218186;

// the z coordinate is used to determine if this part of the aircraft is
// closer to the light source than the shadow plane (and should not be
// shadowed), or behind the shadow plane, and should be shadowed.

OUT.texcoord2.z = dot(-intersectionVec, lightDirection_model);

// compute shadow coordinates for the left tail fin shadow map

// intersection of a line with the shadow plane
// V0 = (0.899235, -0.676381, -0.706742)
// N = (0.917594, 0.0, 0.397518)
// line begins at vertex point P0 and travels along lightDirection
// computed in model-space

N = float3(0.917594, 0.0, 0.397518);
V0 = float3(-0.899235, -0.676381,  -0.706742);
V0minusP0 = V0 - IN.position.xyz;
numerator = dot(N, V0minusP0);
denominator = dot(N, lightDirection_model);
sI = numerator / denominator;

// find the intersection point in model-space

intersectionVec = lightDirection_model.xyz * sI;
float3 intersectionLTail = IN.position.xyz + intersectionVec;

// convert to a texture coordinate
// size of shadow plane = 4.1218186m on each side
// the normalized up vector for this tail fin's shadow plane
// is = (-0.3975177, 0.0, 0.91759448)
```

```
  distanceVec = intersectionLTail - V0;
  upVec = float3(-0.3975177, 0.0, 0.91759448);
  OUT.texcoord3.x = - distanceVec.y / 4.1218186;
  OUT.texcoord3.y = dot( distanceVec, upVec ) / 4.1218186;

  // the z coordinate is used to determine if this part of the aircraft is
  // closer to the light source than the shadow plane (and should not be
  // shadowed), or behind the shadow plane, and should be shadowed.

  OUT.texcoord3.z = dot(-intersectionVec, lightDirection_model);

  return OUT;
}
```

APPENDIX B

EXAMPLE FRAGMENT SHADER SOURCE

```
//--------------------------------------------------------------------
// vertout - the per-vertex attributes to be passed to the
//           fragment shader, interpolated across the triangle
//--------------------------------------------------------------------
struct vertout
{
  float4 position  : POSITION;
  float4 color0    : COLOR0;
  float4 texcoord0 : TEXCOORD0;
  float4 texcoord1 : TEXCOORD1;
  float4 texcoord2 : TEXCOORD2;
  float4 texcoord3 : TEXCOORD3;
  float4 ambient   : TEXCOORD4;
};

//--------------------------------------------------------------------
// pixout - the output of the fragment shader
//--------------------------------------------------------------------
struct pixout
{
  float4 color : COLOR;
};

//--------------------------------------------------------------------
// f35ShadowFragmentShader - the main program
//--------------------------------------------------------------------
pixout f35ShadowFragmentShader( vertout IN,
                      uniform sampler2D colorTEX   : TEXUNIT0,
                      uniform sampler2D shadowTEX  : TEXUNIT1,
                      uniform sampler2D shadowTEX2 : TEXUNIT2 )
{
  pixout OUT;

  // lookup the base texture color

  float4 texColor = tex2D( colorTEX, IN.texcoord0.xy );
```

```
  // perform the shadow plane lookups
  // this model has 3 shadow planes

  // the texcoords for shadow lookup need to be clamped [0,1]
  // setting the texture wrap to clamp would work but shows a strange
  // performance problem on the latest nVidia drivers.

  float shadowColor = tex2D( shadowTEX, saturate(IN.texcoord1.xy) );
  float shadowColor2 = tex2D( shadowTEX2, saturate(IN.texcoord2.xy) );
  float shadowColor3 = tex2D( shadowTEX2, saturate(IN.texcoord3.xy) );

  // apply the diffuse lighting

  float4 diffuseColor = IN.color0 * texColor;

  // test whether this fragment is in front of or behind the shadow map

  float shadowPercent = (IN.texcoord1.z < 0.0) ? shadowColor : 1.0;
  float shadowPercent2 = (IN.texcoord2.z < 0.0) ? shadowColor2 : 1.0;
  float shadowPercent3 = (IN.texcoord3.z < 0.0) ? shadowColor3 : 1.0;
  shadowPercent *= shadowPercent2;
  shadowPercent *= shadowPercent3;

  // this final fragment color is the diffuse lighting with shadowing
  // plus the ambient lighting contribution

  float3 outcolor = diffuseColor.rgb * shadowPercent
                  + texColor.rgb * IN.ambient;

  OUT.color = float4(outcolor, diffuseColor.a);

  return OUT;
}
```

VITA

Name: Christopher Ryan Coleman

Address: MultiGen-Paradigm, Inc., 1301 W. George Bush, Suite 120, Richardson, TX 75080

Email Address: chriscoleman@sbcglobal.net

Education: B.S., Computer Science, Texas A&M University 1999
M.S., Visualization Sciences, Texas A&M University, 2006