Improving Shadows and Reflections via the Stencil Buffer

Mark J. Kilgard [*]
NVIDIA Corporation

"Slap textures on polygons, throw them at the screen, and let the depth buffer sort it out."

For too many game developers, the above statement sums up their approach to 3D hardware-acceleration. A game programmer might turn on some fog and maybe throw in some transparency, but that's often the limit. The question is whether that is enough anymore for a game to stand out visually from the crowd. Now everybody's using 3D hardware-acceleration, and everybody's slinging textured polygons around, and, frankly, as a result, most 3D games look more or less the same.

Sure there are differences in game play, network interaction, artwork, sound track, etc., etc., but we all know that the "big differentiator" for computer gaming, today and tomorrow, is the look. The problem is that with everyone resorting to hardware rendering, most games look fairly consistent, too consistent. You know exactly the textured polygon look that I'm talking about.

Instead of settling for 3D hardware doing nothing more than slinging textured polygons, I argue that cutting-edge game developers must embrace new hardware functionality to achieve visual effects beyond what everybody gets today from your basic textured and depth-buffered polygons.

Two effects that can definitely set games apart from the crowd are better quality shadows and reflections. While some games today incorporate shadows and reflections to a limited extent, the common techniques are fairly crude. Reflections in today's games are mostly limited to "infinite extent" ground planes or ground planes bounded by walls. As far as such reflections go, don't expect to see reflections on isolated surfaces such as a reflective tabletop or reflective stair steps. Shadows have worse limitations. Shadows in today's games are often not much more than dark spots projected on the floor. As far as shadows go, don't expect the shadow to be properly combined with the ground texture, don't expect to see shadows cast onto arbitrarily shaped objects, and don't expect to see shadows from multiple light sources.

What this tutorial describes is a set of techniques for improving the quality of shadows and reflections using the per-pixel stencil testing functionality supported by both OpenGL and

---

[*] Mark graduated with B.A. in Computer Science from Rice University and is a System Software Engineer at NVIDIA. Mark is the author of *OpenGL Programming for the X Window System* (Addison-Wesley, ISBN 0-201-48359-9) and can be reached by electronic mail addressed to mjk@nvidia.com

OpenGL implementation	Stencil bits supported
Most software implementations (Mesa, MS OpenGL, SGI OpenGL)	8
3Dlabs Permedia II	1
SGI Indigo^2 Extreme	4
SGI Octane MXI	8
ATI RAGE 128	8 (32-bit mode only)
NVIDIA RIVA TNT	8 (32-bit mode only)
SGI Onyx^2 InfiniteReality	1 or 8 (multisampled)

Table 1. Stencil bits supported by selected OpenGL implementations.

Direct3D. From its inception, OpenGL required stenciling support. Direct3D incorporated stenciling more recently in its DirectX 6 update.

Using texturing, fog, or depth buffering is pretty straightforward. However, if you just read the available documentation on stenciling, you'll probably be left scratching your head and wondering what in the world to do with it. Hence, this tutorial.

Stenciling is an extra per-pixel test and set of update operations that are closely coupled with the depth test. In addition to the color and depth bit-planes for each pixel, stenciling adds additional bit-planes to track the stencil value of each pixel. The word "stenciling" often gives those unfamiliar with per-pixel stenciling a pretty limited impression of the functionality's potential. Yes, you can use the stencil buffer simply to reject fragments outside some stenciled 2D region of a window (the way a stencil is used when painting letters onto a surface in the real world). However the actual stencil testing functionality is quite a bit more powerful than implied by the common usage of the word.

A better way to think of per-pixel stenciling is that stenciling is a means to "tag" pixels in one rendering pass to control their update in subsequent rendering passes. Consider how that can be applied to shadows. Given a scene with a hard shadow, pixels making up the scene can be considered either "inside" or "outside" of the shadow. Assume that we can tag each pixel in the scene appropriately. Once the pixels are tagged, then configure stenciling to *only* update the pixels tagged "inside the shadow" and re-render the scene with the blocked light source *disabled.* Then re-configure stenciling to update only pixels tagged "outside the shadow," enable the light source, and re-render the scene again. This simple description leaves several issues unexplained. For example, how do you go about "tagging" pixels as being inside or outside of a shadow? This issue and all the rest will be addressed in subsequent sections. Keep in mind the approach described above is just one way to use the "tagging" power of per-pixel stenciling.

Silicon Graphics introduced per-pixel stencil testing hardware over a decade ago with its VGX high-end graphics workstation.[1] Since that time, the price of stencil hardware has dropped by well over three orders of magnitude (Moore's law in action!) and is soon to become a ubiquitous feature of PC gaming systems. The Permedia II from 3Dlabs was the first mass-market 3D chip to support hardware stencil. As of last year, the RIVA TNT from NVIDIA was the first mass-market 3D chip to support a full 8-bit stencil buffer. The installed base of RIVA TNT systems is over 4 million and growing. More recently announced chips such as the Rage

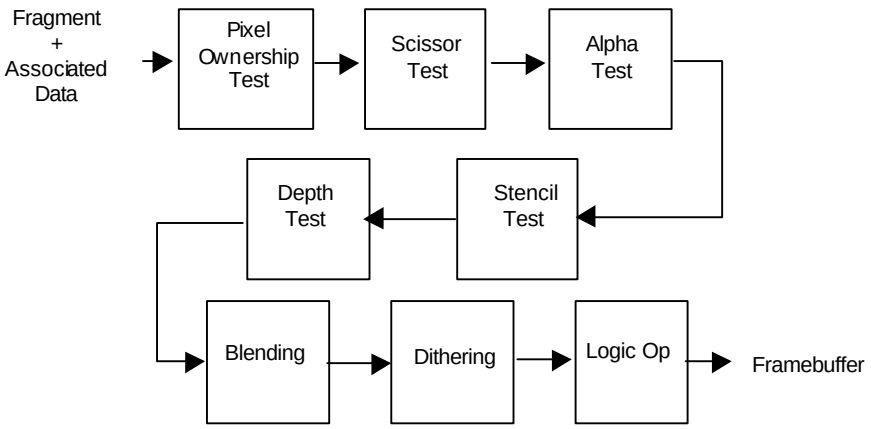


Figure 1. Per-fragment operation sequence.

128 from ATI also support a full 8-bit stencil buffer. With both major 3D APIs fully supporting hardware stenciling and the major 3D chip companies designing in hardware stencil support, you can expect stencil hardware to become nearly ubiquitous for new 3D hardware by Christmas 1999.

This tutorial will explain what per-pixel stencil testing is and how to use it. Section 1 overviews the per-pixel stenciling testing functionality from the programmer's point of view. Section 2 explains the basic non-stenciled approach to rendering reflections, while Section 3 shows how to improve reflection effects with stenciling. Section 4 explains the basic non-stenciled approach for rendering planar projected shadows, while Section 5 show how stenciling can improve such shadows. Section 6 explains a more powerful stencil-based shadow volume algorithm permitting volumetric shadow regions instead of simple projections of shadows onto planes. Section 7 provides a brief synopsis of several more applications for stenciling. Section 8 concludes.

1. Per-pixel Stencil Testing

Explaining how stencil testing works is a little like explaining a Swiss Army knife. The basic stencil testing functionality sounds pretty mechanical, but it is difficult to appreciate and use stencil testing for shadows and reflections until you understand its basic functionality. Please bear with me for the remainder of this section. To keep things simple, the subsequent examples of programming the stencil rendering state use OpenGL commands (rest assured that Direct3D provides the identical stenciling functionality just with a considerably more complex and cumbersome API).

Stencil testing assumes the existence of a stencil buffer so any explanation of how stencil testing works is predicated on understanding what the stencil buffer is. The stencil buffer is similar to the depth buffer in that the stencil buffer is a set of non-displayable bit-planes in addition to the color buffers. In the same way a depth buffer associates a depth value with every pixel for a window supporting a depth buffer, a stencil buffer associates a stencil value with every pixel for a window supporting a stencil buffer. And just as when depth testing is

enabled, the depth values are used to accept or reject rasterized fragments, when the stencil test is enabled, the frame buffer's stencil values are used to accept or reject rasterized fragments.

The first step to using stenciling is requesting a window that supports a stencil buffer. Typically, you specify the minimum number of bits of stencil you require because an implementation may be limited in the number of stencil bits it supports. In Win32 OpenGL programs, this means using `ChoosePixelFormat` or `DescribePixelFormat` to find a pixel format supporting a stencil buffer. OpenGL Utility Toolkit[2] (GLUT) users have it easy.[*] Here is an example of how to create a double buffered window supporting both stencil and depth buffers in GLUT:

```
glutInitDisplayString("stencil>=1 rgb depth double");
glutCreateWindow("stencil buffer example");
```

Nearly all software implementations of stencil support 8-bit stencil buffers. Hardware implementations vary in the number of bits of stencil they support, but one, four, or eight bits are the common configurations as shown in Table 1. Compliant OpenGL implementations are required to support at least one bit of stencil, even if the implementation is forced to resort to software rendering to support stenciling. Though functionally identical, hardware stenciling is typically at least an order of magnitude faster than software stenciling. Interactive stencil applications typically require hardware support for adequate performance, but fortunately, hardware stencil buffers are becoming standard-issue with new graphics hardware designs, even in the consumer PC space.

In particular, I expect that the 8-bit stencil buffer configuration will become a standard 3D accelerator feature. The reason has to do with the size of 32-bit memory words as much as anything else. In previous years, memory capacities, memory cost, and bandwidth limitations meant that PC graphics subsystems settled for 16-bit color buffers and 16-bit depth buffers. Unfortunately, this compromise dictates limited color and depth precision that undermines overall scene quality. Still this compromise allows two pixels to fit compactly into two 32-bit words. Cheaper and denser memories and smarter designs now permit current and future generations of PC graphics subsystem to support deeper frame buffers that support two 32-bit words per pixel. Such a "3D true color" frame buffer stores 8 bits of red, green, blue, and alpha in one of the pixel words. More importantly though, while 16-bit depth buffers often lack sufficient depth precision, 32-bit depth buffers end up with far more precision than applications typically ever require. So instead of a 32-bit depth buffer, 8 bits of the word are better utilized as stencil buffer, leaving a 24-bit depth buffer with still generous depth buffer resolution.

---

[*] The OpenGL Utility Toolkit (GLUT) is a freely available library for writing portable OpenGL examples and demos. GLUT supports both the X Window System (for Linux and Unix workstation users) and Win32 (for those addicted to Microsoft's tyranny). You can download GLUT from the Internet from http://reality.sgi.com/opengl/glut3/glut3.html

The stencil value itself is an unsigned integer, and as will be described later, increment, decrement, comparison, and bit-masking operations can be performed on the integer values held in the stencil buffer. This is in contrast to depth values that are typically thought of as fixed-point values ranging from zero to one. Also, the only operation typically performed on a depth buffer value is a comparison with another depth value; unlike stencil values, depth values are not masked, incremented, or decremented.

Typically at the beginning of rendering a scene, the stencil buffer is cleared to particular program-specified value. In OpenGL, setting the stencil buffer clear value and clearing the color, depth, and stencil buffers is done like this:

```
glClearStencil(0);                                    // clear to zero
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT);
```

Again like the depth test, the stencil test can either be enabled or disabled, and the result of the test determines if the rasterized fragment is discarded or is further processed. When enabled, the stencil test is performed for each and every rasterized fragment so the stencil test is referred to as *per-fragment* (or *per-pixel* depending on your point of view). When the stencil test is enabled and the test fails, the rasterized fragment is discarded. Otherwise, either because the stencil test passes or because stencil testing is disabled, the fragment continues to be processed. In OpenGL, enabling and disabling the stencil buffer is done like this:

```
glEnable(GL_STENCIL_TEST);
glDisable(GL_STENCIL_TEST);
```

You can think of the stencil test as a hurdle just like the depth test, alpha test, or scissor test that fragments must overcome before getting the opportunity to update the frame buffer. The stencil test is performed after the alpha test and before the depth test as shown in Figure 1. While the ordering of these operations may seem a bit arbitrary, the precise ordering of these per-fragment operations is crucial for the algorithms described later.

The depth test, alpha test, and stencil test each perform a comparison. The depth test compares the fragment's depth value to the pixel's depth buffer value. The alpha test compares the fragment's alpha value to an alpha reference value. Note that the alpha reference value is not fetched from any per-pixel buffer, but rather is a piece of rendering engine state. The stencil test compares the pixel's stencil buffer value to a stencil reference value that, like the alpha test reference value, is a piece of rendering engine state. Fortunately, these three per-fragment tests share the same set of eight comparison functions: **never**, **always**, **less than**, **less than or equal**, **greater than**, **greater than or equal**, **equal**, and **not equal**.

The stencil comparison is actually a bit more powerful than a straight comparison because before the comparison is made, both the stencil buffer value and the reference value are bit-wise ANDed with a stencil comparison bit-mask. This means the comparison can be limited to specific bits allowing the stencil bit-planes can be treated as a whole, individually, or as a

subset. In OpenGL, the stencil comparison function, the stencil reference value, and the stencil comparison bit-mask are set as follows:

```
glStencilFunc(GL_EQUAL,  // comparison function
              0x1,       // reference value
              0xff);     // comparison mask
```

When the alpha or scissor test fails, the fragment is simply rejected. When the alpha or scissor test passes, processing of the fragment continues. In neither case are there any side effects. The depth test is a bit more involved. If the depth test fails, the fragment is rejected without any side effect. However, when the depth test passes, the pixel's depth value is replaced with the incoming fragment's depth value, assuming that the depth buffer write mask permits updates. The stencil test is more complicated. The stencil test has three distinct side effects depending on whether: (1) the stencil test fails, (2) the stencil test passes *but* the depth test fails, or (3) the stencil test passes *and* the depth test passes (or the depth test is not enabled).

Not only are there three different per-fragment side effects, but the programmer can specify the operation of each of the three side effects. There are six standard stencil operations: **keep**, which leaves the pixel's stencil value unchanged; **replace**, which replaces the pixel's stencil value with the current reference stencil value; **zero**, which clears the pixel's stencil value; **increment**, which adds one to the pixel's stencil value and clamps the result to the stencil buffer's maximum value; **decrement**, which subtracts one from the pixel's stencil value and clamps the result to zero; and finally **invert**, which inverts the bits of the pixel's stencil value. [†] When a pixel's updated stencil value is written back to the stencil buffer, the stencil write bit-mask is applied so that only bits set in the stencil mask are modified. In OpenGL, the stencil operations and the stencil write mask are set as shown in this example:

```
glStencilOp(GL_KEEP,    // stencil fail
            GL_DECR,    // stencil pass, depth fail
            GL_INCR);   // stencil pass, depth pass
glStencilMask(0xff);
```

One warning about the stencil write mask: In OpenGL, when you clear the stencil buffer, the stencil write mask is applied during the clearing of the stencil buffer. This is useful if you want to preserve values in the stencil bit-planes across clears, but it can also cause confusion if you expect the clear operation to set every pixel's stencil value to the stencil clear value. If you

[†] Actually, Direct3D added two additional (and rather dubious) stencil operations: **increment wrap**, which adds one to the pixel's stencil value and wraps to zero when the stencil buffer's maximum value is incremented, and **decrement wrap**, which subtracts one from the pixel's stencil value and wraps to the stencil buffer's maximum value when zero is decremented. The same functionality exists as an OpenGL extension proposed by Intergraph and NVIDIA called `EXT_stencil_wrap` that adds two new enumerants: `GL_INCR_WRAP_EXT` (0x8507) and `GL_DECR_WRAP_EXT` (0x8508). Both Mesa 2.1 and the Release 2 RIVA TNT OpenGL drivers support the `EXT_stencil_wrap` extension.

State Description	Initial value	State update command	State query token
Stenciling enable	GL_FALSE	glEnable glDisable	GL_STENCIL_TEST
Stencil function	GL_ALWAYS	glStencilFunc	GL_STENCIL_FUNC
Stencil compare mask	All 1's	glStencilFunc	GL_STENCIL_VALUE_MASK
Stencil reference value	0	glStencilFunc	GL_STENCIL_REF
Stencil fail operation	GL_KEEP	glStencilOp	GL_STENCIL_FAIL
Stencil depth fail operation	GL_KEEP	glStencilOp	GL_STENCIL_PASS_DEPTH_FAIL
Stencil depth pass operation	GL_KEEP	glStencilOp	GL_STENCIL_PASS_DEPTH_PASS
Stencil write mask	All 1's	glStencilMask	GL_STENCIL_WRITEMASK

Table 2. OpenGL stencil state with initial values, update commands, and get tokens.

want to clear *all* the stencil bit-planes, make sure to call `glStencilMask(~0)` before calling `glClear`.

The OpenGL state is for stencil testing is summarized in Table 2.

For completeness, OpenGL also supports reading, writing, and copying the stencil buffer contents with the `glReadPixels`, `glDrawPixels`, and `glCopyPixels` calls. If the *format* parameter to these calls is `GL_STENCIL`, they operate on the stencil buffer values. Also, keep in mind that in OpenGL, the stencil test (like all the per-fragment operations) applies to *all* fragments whether they are generated by geometric primitives (points, line, and polygons) or image primitives (bitmaps and images).

2. Review of Planar Reflections without Stencil

In the creative imagination of Lewis Carroll, a looking glass can open up a gateway to a make-believe land that defies traditional logic. In the real world however, mirrors are not so mysterious. Essentially, a mirror flips the image of reality through its plane.

That actually turns out to be pretty good intuition for how to interactively render a planar reflection.[3] In practice, this amounts to a pretty straightforward algorithm that is easily expressed in OpenGL:

1. Load the modelview matrix with the view transform. For example, given the eye location, the center of viewing, and up direction vector, the view transform can be set as follows:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye[0], eye[1], eye[2],
          center[0], center[1], center[2],
          up[0], up[1], up[2]);
```

2. Push the modelview matrix:

```
glPushMatrix();
```

3. Multiply the current modelview matrix by a reflection matrix that reflects the scene through the plane of the mirror. Consider the special case where the mirror happens to lie in the z=0 plane. This special case reflection is a simple scaling matrix that negates the Z coordinate:

```
glScalef(1.0, 1.0, -1.0);
```

   For an arbitrary plane, a 4 by 4 matrix can be constructed to reflect through an arbitrary plane as detailed in Appendix A.

4. If using back face culling, cull front faces instead. The reflective transformation flips the front/back orientation of transformed polygons.

```
glCullFace(GL_FRONT);
```

5. Draw the scene, but be careful to render only objects on the reflective side of the mirror plane. Otherwise, objects behind the mirror that should properly be obscured by the mirror will be rendered as if they are actually in front of the mirror plane. This issue will be discussed in more detail later.

6. Resume normal back face culling and undo the reflection matrix.

```
glCullFace(GL_BACK);
glPopMatrix();
```

7. Optionally, to give the appearance of a semi-reflective surface such as polished marble, or simply a dull or dirty mirror, a textured planar surface coincident with the mirror plane can be blended on top of the reflection rendered in Step 5. For example:

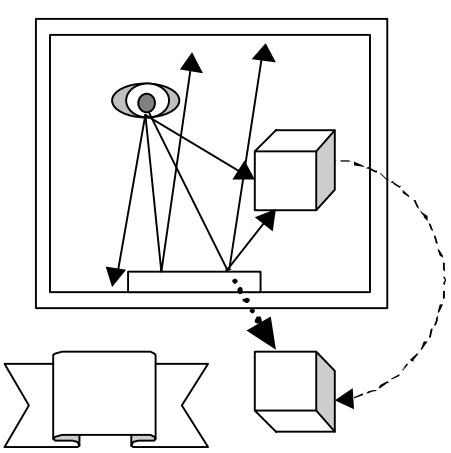
```
glEnable(GL_BLEND);
glBlendFunc(GL_ONE, GL_ONE);                    // additive blending
renderMarbleTexturedMirrorSurfacePolygons();
glDisable(GL_BLEND);
```

   Even if the mirror surface is not blended with a semi-reflective surface, it is important to render the mirror surface polygons into the depth buffer to ensure that when the scene is rendered again in the next step that objects properly are occluded by the mirrors. For example:

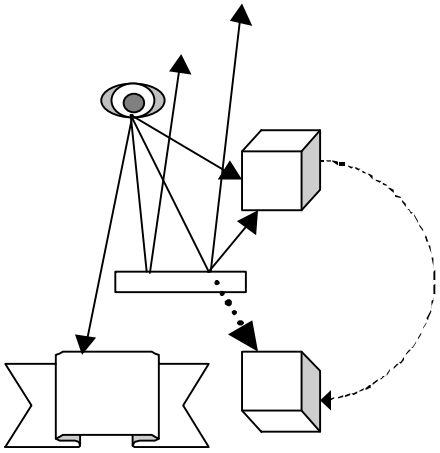
```
glColorMask(0,0,0,0);           // disable color buffer updates
renderMirrorSurfacePolygons();  // update depth buffer with the mirror surface
glColorMask(1,1,1,1);           // re-enable color buffer updates
```

8. Finally, now render the unreflected scene.

This type of rendering algorithm is often referred to as *multi-pass* because the sequence of operations involves rendering the scene or portions of the scene multiple times. Notice that steps 5 and 8 each render the scene but with different modelview transformations.



Constrained reflection: Mirror bounded by walls so no way exists for the eye to "look behind" the wall to see the ribbon, though box "in front" of the mirror also appears reflected by the mirror.



Unconstrained reflection: Free-standing non-infinite mirror presents the opportunity to "look behind" the mirror plane to see some of the ribbon, though the mirror also partially occludes a portion of the ribbon.

Figure 2. Distinguishing between constrained and unconstrained planar reflections.

While the above approach does work, but it has several limitations. The first limitation is that there must be no way to "look behind" the plane of the mirror. One way to accomplish this is to assume an "infinite" planar reflective surface where there is just no way to look around the mirror since the mirror is infinite. More typically, when the above-described technique is used, well placed walls, floors, and ceilings are used to occlude any visibility beyond the mirror plane. For the technique to work, the reflection must be a *constrained reflection* as shown in Figure 2 so that it is simply not possible to "look behind" the mirror plane.

While 3D games can often accommodate these restrictions by limiting where mirrors can be placed, simple room configurations such as an open doorway and a mirror on the same wall break the assumption.

Also consider what happens when multiple mirrors appear in the same scene. Note that when two or more mirrors are present in the same scene, you probably want to be careful to avoid situations that can create infinite recursive reflections. For example, two parallel mirrors facing each other such as in a barbershop. Note that a multi-pass reflection algorithm does not "go into an infinite loop" in such a situation. The best such an implementation can do is to render some finite number of nested reflections and thereby only poorly approximate the true reality of infinitely recursive reflections.

But infinite reflections are not always present. Consider a two-tiered reflective marble floor. Because the two floor surface planes have the identical planar orientation, the two sections of floor can not reflect each other, but each tier has its own noticeably distinct reflection. The problem is that the above technique is really only designed to render a single reflection. Other configurations of multiple mirrors such as a periscope also do not create infinite recursive reflections. Imagine a hallway with two mirrors at each end at 45° angles so you can peer

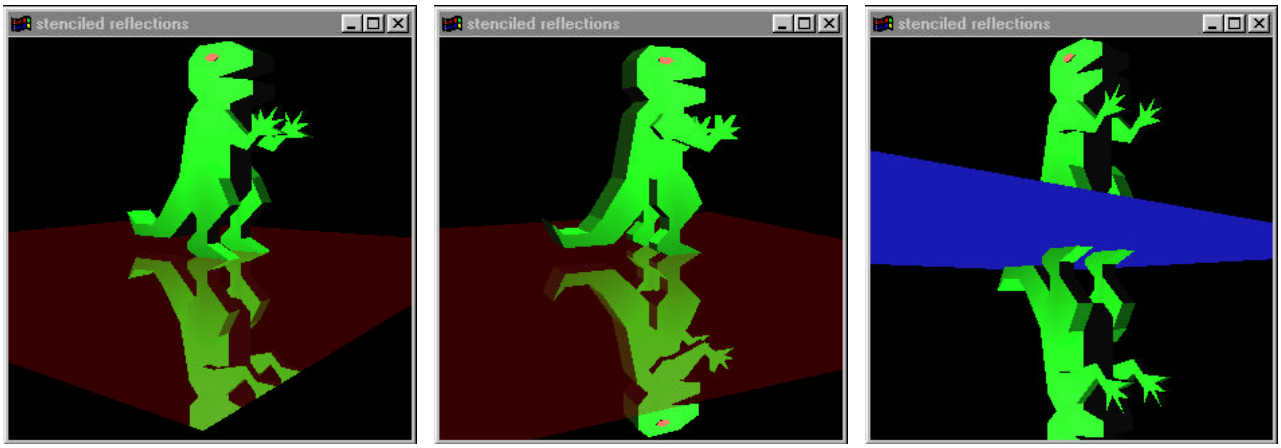


Figure 3. The left image shows the proper use of stenciling to limit the reflection to the reflective surface. The middle and right images do not perform any stenciling. Notice how in the middle image the head peeks beyond the surface. The right image demonstrates that the reflection really is just the object flipped through the mirror plane.

down the hallway, but without the risk of being shot since there is no direct line of sight (assuming no laser weapons!).

Even in the case of infinitely recursive reflections, some finite number of bounces can be rendered before giving up. Often, it only takes several bounces to create the hall-of-mirrors ambiance. The point is that more interesting scenarios are possible when a reflection algorithm can handle multiple unconstrained planar reflective surfaces.

3. Improving Planar Reflections with Stencil

With stenciling, the basic algorithm from the previous section can be enhanced to handle multiple unconstrained planar reflective surfaces. The crucial innovation is using the stencil buffer to “tag” each mirror surface with a unique identifier. Then, when a particular reflection is rendered, the enhanced algorithm only updates pixels matching the tag of the particular reflective surface.

The left image in Figure 3 demonstrates the benefit of stencil testing to limit the reflection to just the mirror surface upon which the reflection should be visible. The center and right images demonstrate the artifacts that can occur when not using stenciling.

The enhanced reflection algorithm is very similar to the algorithm described in Section 2, but with these modifications:

- The window should be allocated with a stencil buffer. When clearing the frame buffer at the beginning of the scene, also clear the stencil buffer to zero.

- For each mirror in the scene, the application must maintain a per-mirror data structure that contains the non-overlapping and co-planar polygons that compose the mirror surface. For

a standard rectangular mirror, the world space coordinates of four vertices are sufficient. From the vertices of any mirror polygon, the plane equation for the mirror plane can be derived. In the code fragments below, the variable `thisMirror` is assumed to point to the data structure maintaining the current mirror's polygon set and plane equation.

- Clear the color, depth, and stencil buffers. Then render the scene, excluding the mirror surfaces, with depth buffering but without stencil testing.

```
glClearStencil(0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFER_BIT);
glEnable(GL_DEPTH_BUFFER_BIT);
glDisable(GL_STENCIL_TEST);
drawScene();                        // draw everything except mirrors
```

- Then for each mirror, perform the following sequence:

  1. Set up stenciling to write the value 1 into the stencil buffer when the depth test passes. Also disable writes to the color buffer. Then draw the mirror's polygons.

     ```
     glEnable(GL_STENCIL_TEST);
     glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
     glStencilFunc(GL_ALWAYS, 1, ~0);
     glColorMask(0,0,0,0);
     renderMirrorSurfacePolygons(thisMirror);
     ```

     This step "tags" all the mirror's visible pixels with a stencil value 1. Depth testing prevents occluded mirror pixels from being tagged.

  2. With the color buffer writes still disabled, set the depth range to write the farthest value possible for all updated pixels and set the depth test to always pass. Also, set the stencil test to only update pixels tagged with the stencil value 1. Then draw the mirror's polygons.

     ```
     glDepthRange(1,1);                      // always
     glDepthFunc(GL_ALWAYS);                 //  write the farthest depth value
     glStencilFunc(GL_EQUAL, 1, ~0);         // match mirror's visible pixels
     glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP); // do not change stencil values
     renderMirrorSurfacePolygons(thisMirror);
     ```

     This step resets the depth buffer to its cleared maximum value for all the mirror's visible pixels.

  3. Restore the depth test, color mask, and depth range to their standard settings:

     ```
     glDepthFunc(GL_LESS);
     glColorMask(1,1,1,1);
     glDepthRange(0,1);
     ```

     We are now ready to render the reflection itself. The pixels belonging to the reflection

are still tagged with the stencil value 1, and these same pixels also have their depth values cleared to the farthest depth value. The stencil test remains enabled and only updating pixels with the stencil value 1, i.e. those pixels on the visible mirror. And the **less than** depth test will ensure subsequent rendering to the mirror pixels will determine visible surfaces appropriately.

4. Rendering the reflection requires reflecting the scene through the mirror plane, but we must also be careful to only render objects on the reflective side of the mirror. Therefore, we establish a user-defined clip plane to render only objects on the reflective side of the mirror plane. The reflection itself is accomplished by concatenating the appropriate reflection matrix to the modelview matrix so that everything is reflected through the mirror plane. Because the reflection flips the sense of back and front facing polygons, the cull face state is reversed. Then the scene is rendered.

```
GLfloat matrix[4][4];
GLdouble clipPlane[4];

glPushMatrix();
  // returns world-space plane equation for mirror plane to use as clip plane
  computeMirrorClipPlane(thisMirror, &clipPlane[0]);
  // set clip plane equation
  glClipPlane(GL_CLIP_PLANE0, &clipPlane);
  // returns mirrorMatrix (see Appendix A) for given mirror
  computeReflectionMatrixForMirror(thisMirror, &matrix[0][0]);
  // concatenate reflection transform into modelview matrix
  glMultMatrixf(&matrix[0][0]);
  glCullFace(GL_FRONT);
  drawScene();                                // draw everything except mirrors
  drawOtherMirrorsAsGraySurfaces(thisMirror);  // draw other mirrors as
                                               // neutral "gray" surfaces
  glCullFace(GL_BACK);
  glDisable(GL_CLIP_PLANE0);
glPopMatrix();
```

Now the mirror's reflection is rendered correctly. Other mirrors visible in the reflection of this mirror are rendered gray so at least they properly occlude objects behind them even if they do not reflect objects correctly. Immediately following, we sketch a recursive algorithm that handles reflections of reflections.

5. Finally, we reset to zero the stencil value of all the mirror's pixels so the pixels are not confused with another mirror's pixels while rendering the reflections of subsequent mirrors. Also update the depth buffer to reflect the mirror's proper depth so this mirror may properly occlude any subsequent mirrors. Do not update the color buffer during this step.

```
glColorMask(0,0,0,0);
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);
glDepthFunc(GL_ALWAYS);
renderMirrorSurfacePolygons(thisMirror);
glDepthFunc(GL_LESS);
glColorMask(1,1,1,1);
```

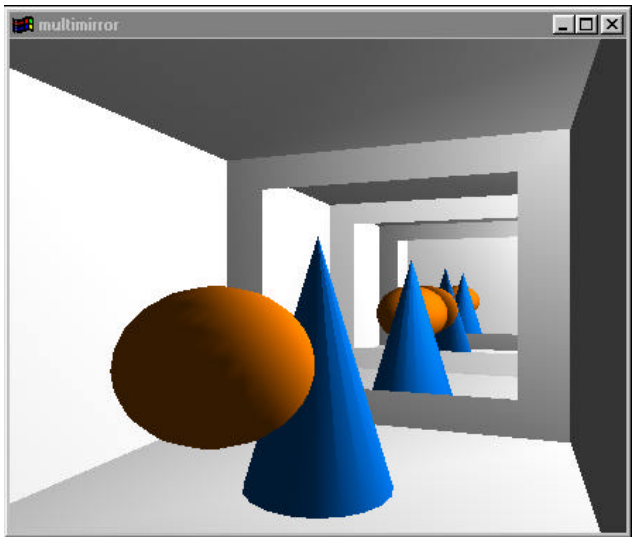


Figure 4. Multiple recursive reflections in a room with two parallel mirrors.

Or instead of not updating the color buffer, as discussed earlier, update the color buffer by blending the reflection in the frame buffer with a surface texture to simulate a semi-reflective surface such as polished marble.

The above algorithm cleanly handles multiple mirrors, though not reflections of reflections. But the algorithm can be further extended to handle such reflections of reflections by making the algorithm support a finite level of recursion. The algorithm presented above simply iterates through each mirror and tags the pixels visible on the given mirror, then renders the mirror's reflected view to the appropriately tagged pixels. Only two stencil values (0 and 1) are used in this process.

A recursive algorithm takes advantage of the **increment** and **decrement** stencil operations to track the depth of reflection using a depth-first rendering approach. Diefenbach[4,5] describes such a recursive reflection algorithm in detail. Figure 4 shows the sort of hall-of-mirrors effect that is possible by recursively applying stenciled planar reflections.[‡] To help you better understand what is actually being rendered, Figure 5 shows the total geometry rendered from a bird's eye view without any stenciling.

One issue to consider when implementing recursive reflections, and can even be an issue with a single reflection, is that reflections, particularly deep recursive reflections, tend to require a far clipping plane that is much further away than what is ordinarily required.

[‡] The source code for `multimirror.c` is available in the `progs/advanced97` directory of the GLUT 3.7 source code distribution. David Blythe wrote the example.

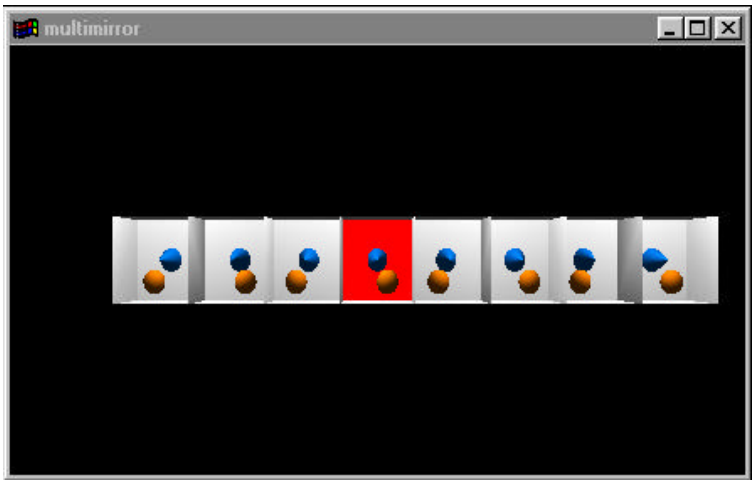


Figure 5. Bird's eye view of the `multimirror` scene with reflected re-renderings but without using stenciling. The middle darker floor indicates the actual room.

Assuming every mirror can reflect every other mirror (the worst case would be a circle of mirrors all facing each other), the maximum number of scene rendering passes required to render ***n*** mirrors and ***r*** reflection bounces is:

$$1 + n\sum_{i=1}^{r}(n-1)^{i-1}$$

For 3 mirrors and 2 bounces in the worst case, 10 renderings of the scene are required. Consider however that a scene with 5 mirrors and 4 bounces in the worst case requires 426 renderings of the scene. The problem with recursive planar reflections is that the number of rendering passes to render the scenes with lots of mirrors and bounces quickly become prohibitive for interactive use. Even for a small number of mirrors or bounces, it is worthwhile to explore more efficient culling strategies when re-rendering the scene for each reflection. The naïve approach of re-rendering the entire scene for each reflection is a source of extreme inefficiency. By tracking the geometry of each mirror and each bounced reflection, an application can determine a conservative reflection frustum to help bound what objects must be rendered in each reflected rendering pass. In addition to culling objects against such a reflection frustum for each re-rendering, recursive reflections can be limited to mirrors within each reflection frustum.

At this point, it is worth mentioning another approach to planar reflections that uses texture mapping.[6] This alternative renders each reflection through the reflected viewpoint and copies the image to a texture and then textures the mirror surface appropriately. This approach does not require stenciling. Quake 3 is expected to use the textured approach.

Using stenciling or texturing for reflections are two alternatives with different tradeoffs. Obviously, the stenciled approach requires a stencil buffer which is less commonly supported than hardware texturing today. The texturing approach though requires the copying of frame

buffer results into texture memory which is generally fairly expensive. Also if the reflection texture resolution does not reasonably match the resolution of the reflection in screen space, texture-sampling artifacts can result. In the stenciling approach however the entire reflection computation takes place in the frame buffer so there is no pixel copying and sampling artifacts are not an issue. Because a reflection texture can be warped when rendering the mirror surface, the texturing approach may be adaptable to rendering reflections on curved surfaces. This adaptability is not a unique advantage because Ofek and Rappoport[7] have implemented a successful stencil-based algorithm for interactive reflections on curved surfaces. In the textured approach, because reflections are copied to textures, the reflection images can be reused across multiple frames. One reason to do this is to amortize the overhead of copying the reflected image into a texture. However reusing reflection textures introduces both incorrect reflections if the view changes and temporal lags in the reflections.

Either approach for rendering reflections involves the awkward issue of lighting. Typically, lighting calculations for interactive graphics use a simple *local lighting model*. In a local lighting model, the lit appearance of a surface involves only the parameters of the surface (such as the surface normal, position, material, and texture) and the parameters of a small number of point and global light sources. In the real world, light interacts with *everything* so the local lighting model assumption is only a very simplified approximation. Mirrors reflecting light and objects casting shadows are two important effects ignored by a local lighting model. Computer graphics researchers refer to algorithms that attempt to model how light interacts with the totality of a scene as *global illumination* techniques. While global illumination techniques such as ray tracing support a broader set of light interactions including reflections and shadows, such techniques in general are far too slow for use in interactive applications.

It is fair to refer to the reflection techniques discussed in this section and the shadow techniques presented in subsequent sections as *pseudo-global illumination* techniques. True global illumination techniques really model the lighting interactions creating shadows and reflections, but the techniques in this tutorial are arguably more based on clever application of 3D geometry principles and per-pixel control of pixel update than any authentic attempt to model the true behavior of light. The strength of the pseudo-global illumination techniques described in this tutorial is that existing 3D graphics hardware directly accelerates them.

Inspecting Figure 5 again will help you see the sort of problems this approach creates. Notice that the sphere and cone appear to be illuminated less on their front left side. As can be inferred from the shading, OpenGL's standard local lighting model is applied using a light source in the upper right corner towards the back of the room. And this lighting effect is repeated in every reflection. But with mirrors on two walls and a light source in the room, is it physically plausible that the sphere and circle would be illuminated from only one direction? Indeed one of the reasons barbershops put mirrors on both walls is to increase the overall ambient light so that the barbers have plenty of light to see what they are cutting. It probably cuts down on the incidence of lacerated ears.

One solution is to reflect light sources through mirrors as well as objects. This introduces additional virtual light sources in order to model how the mirror reflects the light. Adding virtual light sources can help many situations, particularly when there are only one or two mirrors, but

because this adds more light to the scene, it can easily leads to overly bright scenes (energy is not conserved). Adjusting the light source parameters such as attenuation can help balance the lighting, but this is not a complete substitute for true global illumination. The easiest and cheapest solution is to simply add more global ambient light to the scene to account for the additional reflected light. In the presence of shadows, reasonable handling of lighting gets even more complicated. Diefenbach has presented an extensive treatment of many of these lighting issues.[5]

4. Review of Planar Projected Shadows without Stencil

Shadows are created wherever light is blocked. And this occurs all the time. Indeed shadows, more so than reflections, are vitally important to our appreciation and understanding of the 3D nature of the world. Unfortunately, the local lighting models supported by OpenGL, Direct3D, and other interactive graphics interfaces do not support shadows except to the very limited extent that surfaces facing away from a particular light source get no diffuse or specular contribution from the light. This self-shadowing effect is due only to the orientation of the surface with respect to the light and not any blocking of illumination by a shadowing object.

The planar projected shadow algorithm is a well-known, easy-to-implement graphics trick for casting the shadows of arbitrary objects onto planes.[8] The technique and its mathematical basis are described by Blinn in a classic column titled “Me and My (Fake) Shadow.”[9]

Given the plane equation for a ground plane and the homogenous position of the light, a 4 by 4 matrix called the *planar projected shadow matrix* can be constructed that projects 3D polygons onto the specified ground plane based on the light source position. If you transform a polygonal object by the planar projected shadow matrix, all its polygons are piled on the ground plane much like a shadow. Thinking of the shadow as a pile of projected polygons will give you the right intuition, but keep in mind the resulting polygons are co-planar so the pile has no height.

The trick is to construct the matrix then concatenate the matrix with the modelview matrix. Then the shadow is rasterized into the ground plane by simply re-rendering the object. In many ways, this is analogous to what we did with the reflection matrix in the previous two sections except we project instead of reflect. This algorithm is easily expressed in OpenGL:

1. Load the modelview matrix with the view transform. For example, given the eye location, the center of viewing, and up direction vector, the view transform can be set as follows:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(eye[0], eye[1], eye[2],
          center[0], center[1], center[2],
          up[0], up[1], up[2]);
```

2. Given a light position and plane equation for the ground plane, and assuming the object is between the plane in the light (otherwise there would be no shadow cast on the plane),

enable the light source, and draw the object and the ground plane with depth buffering:

```
// local light location (LX,LY,LZ)
GLfloat lightPosition[4] = { LX, LY, LZ, 1.0 };
// A*x + B*y + C*z + D = 0
Glfloat groundPlaneEquation[4] = { A, B, C, D };

glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
drawObject();
drawGroundPlane();
```

3. Push the modelview matrix:

```
glPushMatrix();
```

4. Given the light source position and ground plane equation, construct a shadow matrix with the `shadowMatrix` routine detailed in Appendix B. Then, multiply the current modelview matrix with the shadow matrix.

```
GLfloat matrix [4][4];

// Compute matrix based on light position and ground plane
// equation.  See Appendix B.
shadowMatrix(&matrix[0][0], lightPosition, groundPlaneEquation);
glMultMatrixf(&matrix[0][0]);
```

5. Unless we do something, the ground plane polygons rendered in step 2 and any shadow polygons projected to the ground plane will be very nearly co-planar. This causes problems when depth buffering because the ground plane polygons and the projected shadow polygons will have almost identical depth values. Unfortunately, because of limited numerical precision, the determination of whether shadow polygon fragments are "in front of" ground plane fragments or not is ill-defined and can vary from view to view and even within a frame. To help disambiguate things, use OpenGL's polygon offset functionality[§] to nudge the shadow polygon fragments slightly nearer:

[§] Using polygon offset is superior to a commonly used alternative. Instead of polygon offset, we could tweak the ground plane equation slightly so that instead of exactly matching the plane used to draw the ground plane polygons in `drawGroundPlane`, the shadow matrix is generated using a plane equation that is slightly elevated relative to the true ground equation. We want to raise the shadow plane just enough to disambiguate the depth values for the shadow and ground plane, but not enough so that the translation is noticeable. This works pretty well, but it is nearly impossible get it to work right for all cases. The problem is that vertex coordinates do not readily correspond to units of depth buffer precision. Different views and depth buffer resolutions may require more or less tweaking. And if the user gets the opportunity to look almost exactly parallel and level with the plane, the fact that the shadow is really floating over the ground plane may become apparent. Polygon offset is a better solution because it shift only the final depth buffer values for the fragments and the offset is expressed in terms of depth buffer units.

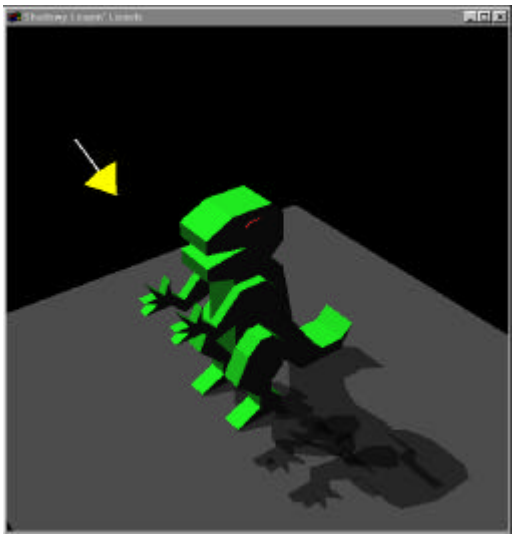


Figure 6. Planar projected shadows blended with the ground plane. Notice the obvious dark patches due to double blending. When animating, these dark spots jump around because they are a function of the shadowing object's depth complexity with respect to the light.

```
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 2.0);          // add 2 depth buffer precision units
                                    // plus slope factor
```

This way the shadow will always be slightly nearer and therefore visible.

6. Disable lighting, set the current color to be dark gray, and draw the object again.

```
glDisable(GL_LIGHTING);
glColor3f(0.25, 0.25, 0.25);        // dark gray
drawObject();                       // draw object as planar projected shadow
```

This rasterizes the object's shadow right on top of the ground plane.

7. Cleanup the rendering state.

```
glDisable(GL_POLYGON_OFFSET);
glPopMatrix();
```

The planar projected shadow technique works fairly well, but it has several limitations. Obviously, these shadows can only be cast onto planes and not on arbitrary objects. Likewise, the resulting shadows generated have hard edges, though in the real world, shadows are "soft" and fall of gradually. These are complex issues that will be addressed again in Section 6.

Accepting projected planar shadows for what they are, the technique suffers from a few unfortunate deficiencies. Consider what happens if the shadow falls on a textured surface. As described above, the shadow is always a uniform dark gray. Ideally, the shadow polygons

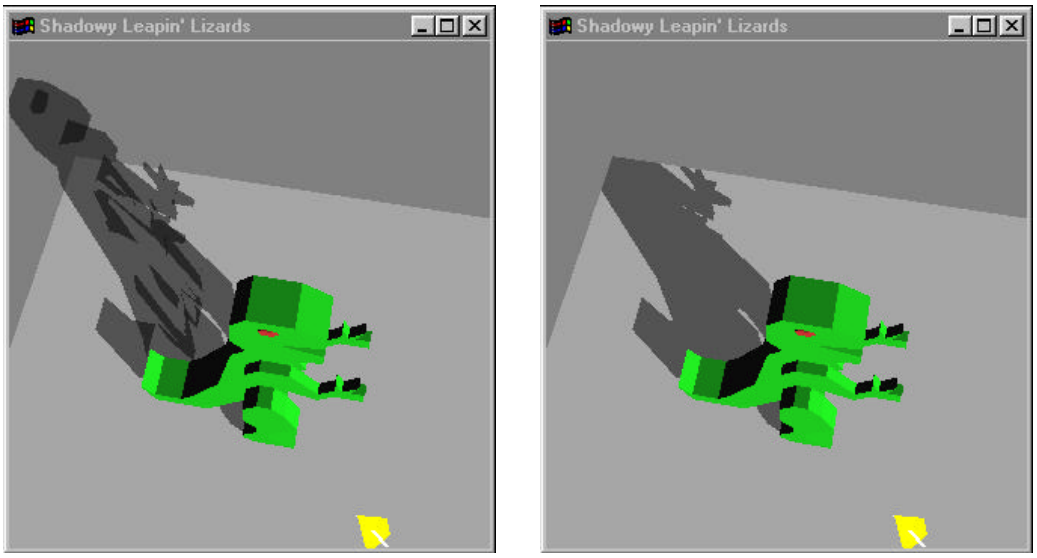


Figure 7. Real shadows do not float in mid-air. Yet the left image's shadow incorrectly extends out beyond the ground surface, and the image also suffers from double blending. The right image uses stenciling to correct both the mid-air shadow and the double blending.

should be drawn just as the actual ground plane surface would look if the shadowed light was disabled. But that is more involved than it sounds. Just disabling the light source when projecting the shadow does not work. All the surface normals required for lighting are squashed by the shadow projection. Likewise, if the ground plane is textured to look like a stone floor, any texture coordinates assigned to the object's vertices are jumbled once projected.

At the very least, instead of a constant dark gray region for the shadow, it would be nice to blend the shadow with whatever texture the ground plane surface has. Unfortunately, this is not that easy. The problem is that when the shadow matrix projects the polygons of an object onto a plane, pixels may be updated more than once. This means that a particular shadowed pixel may be blended multiple times, but blending multiple times will leave a shadow that is too dark. Real shadows cannot leave an object doubly dark! This problem is known as *double blending*. Figure 6 shows what this double blending looks like.

Many current 3D games simply use planar projected shadows and blend with the ground despite the double blending problems. Indeed, when I last flipped through a current computer gaming magazine, I saw an embarrassing number of screen snapshots from shipping games that suffer from double blended shadows. Apparently today's game developers believe that double blended, blotchy shadows are better than no shadows at all. What these game developers should learn though is that double blended shadows can be eliminated on graphics hardware supporting stencil.

Projected planar shadows also share an issue with planar reflections. If the ground surface is non-infinite or is not carefully bounded by walls, the planar projected shadow can extend

beyond the proper boundary of the ground plane. The left image in Figure 7 shows this problem.

5. Improving Planar Projected Shadows with Stencil

The planar projected shadow technique can be enhanced using stencil testing. As demonstrated in the right image in Figure 7, both the double blending issue and the problem of limiting the shadow to an arbitrary ground plane surface can be solved with stenciling.

The enhanced technique uses stencil testing to assign a unique non-zero stencil value to the pixels belonging to the ground plane. Then draw the ground plane polygons like this:

```
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, uniqueStencilValue, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
drawGroundPlane();
glDisable(GL_STENCIL_TEST);
```

Then replace step 6 of the original algorithm with the following:

```
glDisable(GL_LIGHTING);
glEnable(GL_BLEND);                  // enable blending to
glBlendFunc(GL_DST_COLOR, GL_ZER0);  //  modulate existing color
glColor3f(0.5, 0.5, 0.5);            //  by 50%
glDisable(GL_DEPTH_TEST);            // depth testing not necessary!

// Only update pixels tagged with the uniqueStencilValue and
// reset to zero the pixel's stencil value once updated.
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_EQUAL, uniqueStencilValue, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_ZERO);

drawObject();                   // draw object as planar projected shadow

// restore basic modes
glDisable(GL_BLEND);
glDisable(GL_STENCIL_TEST);
glEnable(GL_DEPTH_TEST);
```

Only pixels tagged with the ground plane's unique stencil value will be updated when the shadow is rendered. Blending is enabled, but to avoid double blending, when a pixel is updated, the stencil value is reset to zero so subsequent pixel updates will fail the stencil test and be discarded.

Another benefit of using stencil is that the use of polygon offset in step 5 of the original algorithm is unnecessary. This is advantageous because other multi-pass rendering techniques require the ability to re-visit exactly the same pixels as a previous rendering pass (by using the **equal** depth test) and using polygon offset to tweak depth values will break such algorithms. The depth test is actually disabled during the stencil-based rendering of the

projected shadow. Because the ground plane is drawn after the shadowing object, any region of the ground plane obscured by the object itself would not be tagged with the ground plane's stencil value because the initial ground plane rendering would fail the depth test for those pixels. (In general though, when multiple ground planes and other objects are drawn in arbitrary orders, the stencil test must be used to zero the stencil value of pixels rendered by objects that occlude previously rendered ground planes.)

The stenciled planar projected shadow technique presented so far just reduces the light source's intensity with a modulating blend. But as pointed out earlier, real shadows do not simply dampen the illumination from a light source; the illumination from an obscured light source is actually *blocked*. Using a blend to modulate the shadowed region of a surface illuminated by a light in an earlier rendering pass is not the same as actually re-rendering the shadowed region with the shadowed light source disabled. The former dampens the light; the later blocks it.

While the modulating blend is adequate for quick-and-dirty rendering, in cases where better lighting fidelity is desirable, a further improvement can better model the actual shadowed appearance. Below is the procedure for improving the appearance of shadows on planar surfaces.

When selecting an otherwise unused stencil value for tagging the planar surface's shadow, also ensure that one plus the unused stencil value is also otherwise unused. When rendering the projected shadow, instead of blending with the pixels on the planar surface, increment the pixel's stencil value and *do not* update the color buffer. Then re-render the planar surface again with lighting enabled but the blocked light source disabled and use stenciling to replace only pixel's with the incremented stencil value. For example:

```
glPushMatrix();
  // apply the planar projected shadow matrix
  shadowMatrix(&matrix[0][0], lightPosition, groundPlaneEquation);
  glMultMatrixf(&matrix[0][0]);

  glDisable(GL_BLEND);                              // no blending!
  glDisable(GL_DEPTH_TEST);                         // no depth testing needed
  // accept only pixels matching the planar surface's stencil value
  glStencilFunc(GL_EQUAL, uniqueStencilValue, ~0);
  // increment the stencil value if accepted
  glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);
  glColorMask(0,0,0,0);
  drawObject();                  // draw object as planar projected shadow
  glColorMask(1,1,1,1);
glPopMatrix();

glEnable(GL_LIGHTING);           // enable lighting
glDisable(GL_LIGHT0);            //  but disable the shadowed light source;
                                 //  global ambient light and light from
                                 //  other light sources is still computed
// accept only pixels matching the planar surface's incremented stencil value
glStencilFunc(GL_EQUAL, uniqueStencilValue+1, ~0);
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);
```

```
// re-render shadowed ground plane pixels with the light disabled
drawGroundPlane();
```

Note that when incrementing the stencil value of a shadowed pixel, the increment can only occur once during the rendering of the planar projected shadow because once the stencil value is incremented, subsequent pixel updates for the pixel will fail the stencil test. There is no risk of double incrementing a pixel!

Variations of the above technique can be used to generate shadows from multiple shadowing objects that each block multiple light sources. An object blocking the light of multiple light sources will create multiple shadows. If these multiple shadows are cast onto a plane, then zero, one, several, or all of the light sources may illuminate different regions of the plane. Properly computing the lighting effect for every region of multiple planar surfaces can be accomplished as follows:

1. Clear the stencil buffer to zero and assign each planar surface an otherwise unused stencil value. One plus each otherwise unused stencil value should also be otherwise unused.

2. Enable lighting and all the light sources.

3. Without stenciling enabled, render all the objects other than the planar surfaces supporting shadows with all the light sources enabled. Do not render the planar surfaces yet.

4. For each planar surface, render the planar surface with all light sources disabled. Note that any global ambient light will still illuminate the surface somewhat. Also during this rendering pass, tag the planar surface pixels with the planar surface's unique stencil value.

5. Set the global ambient light in the scene to zero. Subsequent rendering passes will add in the unblocked illumination from each light source one light at a time, so we want to ensure that the global ambient light is added only once.

6. For each planar surface:

   For each light source:

   - Push the modelview matrix.

   - Using the planar projected shadow matrix for the current light source and current planar surface, render all the shadowing objects in between the surface plane and the light source (clip planes can make this determination automatic) so that stencil values of shadowed pixels on the current planar surface are incremented. Do not enable depth testing or update the color buffer during this pass.

   - Pop the modelview matrix.

   - Enable just the current light source and render all the planar surface's polygons. Set up stencil testing to only update the planar surface's non-incremented stencil value.

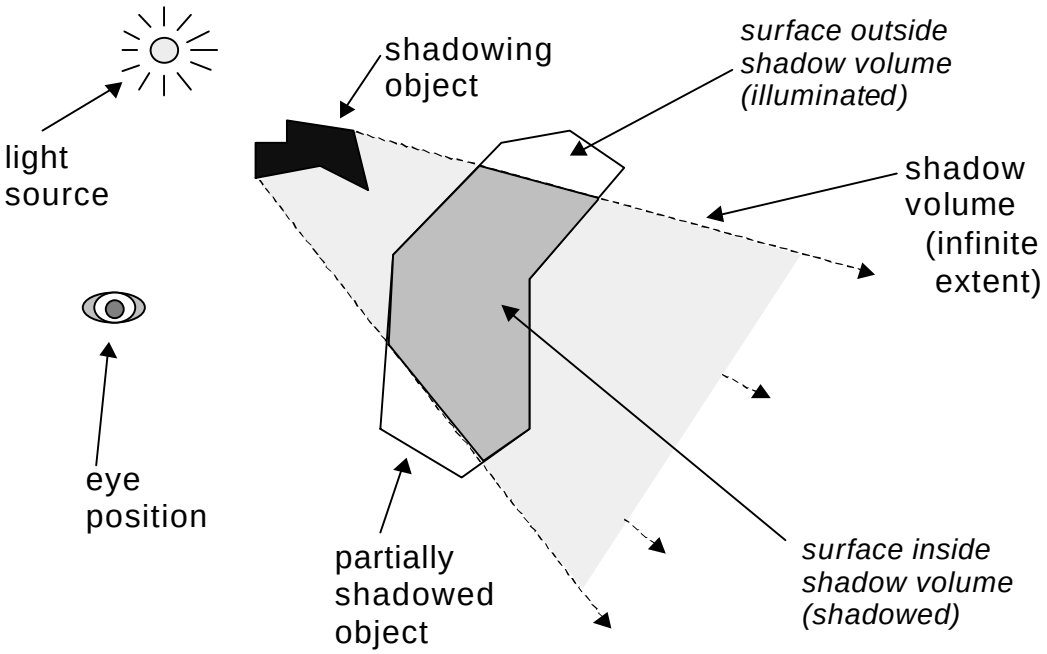


Figure 8. A two-dimensional slicing view of a shadow volume.

This will only update pixels on the surface *and not* in the light source's shadow region. When updating the color buffer, use additive blending to add the illumination from this light to the planar surface's illuminated pixels. Also use stencil testing to increment the stencil value of all the illuminated pixels.

- On the first time completing the above four sub-steps for a particular planar surface, all the stencil values for the planar surface will be left incremented. When repeating the above four sub-steps for the surface's next light source, perform a decrement instead of an increment and swap matching the "non-incremented" and "incremented" stencil value. For each subsequent light source, keep reversing the sense of increment and decrement and the "incremented" versus "non-incremented" stencil value from the sense of the last step.

7. Restore the global ambient light.

This heavily multi-pass stenciled algorithm can render shadows on multiple distinct planar surfaces cast by multiple objects blocking multiple light sources. While powerful, the algorithm is still limited in that shadows are only cast onto planar surfaces and, in particular, objects do not cast shadows on each other.

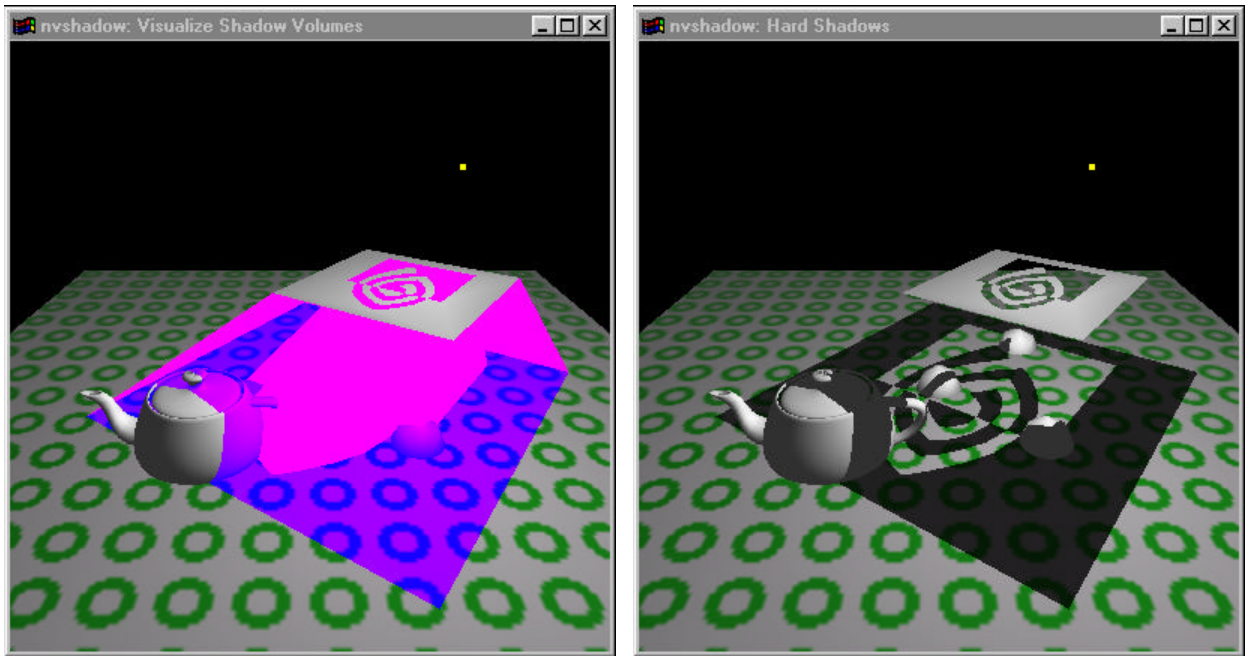


Figure 9. The left image shows a visualization of the shadow volume for the NVIDIA logo cutout. The right image shows how result of using the shadow volume to determine what pixels are within the shadow and outside the shadow.

6. Volumetric Shadows with Stenciled Shadow Volumes

Casting shadows on arbitrary non-planar surfaces requires an approach that is more powerful than the projected planar shadow trick. In the real world, the shadow cast by an object blocking a light is a *volume*, not merely some two-dimensional portion of a plane.

Over two decades ago, Crow[10] proposed a class of shadow algorithms[11] that model shadow regions as volumes. These algorithms operate in two stages. The first stage computes the *shadow volume* formed by a light source and a set of shadowing objects. The second stage then determines whether or not a point on a surface is illuminated by the light source or is instead shadowed due to one or more of the shadowing objects. This determination is based on whether or not the point on the surface is inside or outside of the shadow volume. If the point is outside the shadow volume, the point on the surface is illuminated by the light source, but if the point is inside the shadow volume, the point is not illuminated by the light source and so is considered shadowed.

Figure 8 shows a two-dimensional slicing of a shadow volume. Surfaces of the partially shadowed object that are outside the shadow volume are illuminated by the light source, while surfaces of the partially shadowed object that are inside the shadow volume are shadowed from the light source. The left image in Figure 9 shows a visualization of the shadow volume cast by the NVIDIA logo cutout in the scene. The right image uses the stenciled shadow volume discussed here to shadow pixels within the shadow volume and illuminate pixels

outside the shadow volume. Note that the shadows are properly cast onto curved surfaces such as the teapot.

6.1. The basic stenciled shadow volume algorithm

The first stage of the shadow volume process is constructing a shadow volume given a point or directional light source and a set of shadowing objects. We will come back to the issue of how to construct shadow volume. For now, assume that you know the *polygonal boundary representation*[**] for the shadow. A polygonal boundary representation is a set of non-intersecting polygons that cleanly divides 3D space into two regions: the inside and outside of the volume that the polygons bound.

Note that shadow volumes are typically "open" or infinite volumes. Notice that in Figure 8, the shadow volume extends infinitely away from the light source. The fact that shadow volumes are infinite is not a practical concern though because we are interested only in the region of the shadow that is viewable in the current view frustum, a finite region. The intersection of an infinite shadow volume and a finite view frustum is a finite region.

Given a polygonal boundary representation for the shadow volume, the stencil buffer can be used to determine whether surface pixels are inside or outside of the shadow volume.[4,5,12,13,14]

In the discussion that follows, only a single light source is considered though the shadow volume concept can be applied to multiple lights sources though each light source requires its own shadow volume.

The first step clears the depth, color, and stencil buffers, then renders the scene with the possibly shadowed light source enabled. An important result of this first step is to initialize the depth buffer with the depth of every pixel's nearest fragment. These nearest fragments correspond to the visible surfaces in the scene. The following OpenGL commands show what to do:

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);                        // enable light source
glEnable(GL_DEPTH_TEST);                    // standard depth testing
glDepthFunc(GL_LEQUAL);
glDepthMask(1);
glDisable(GL_STENCIL_TEST);                 // no stencil testing (this pass)
glColorMask(1,1,1,1);                       // update color buffer
glClearStencil(0);                          // clear stencil to zero
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
renderScene();
```

Because the light source is enabled during this stage, there are no shadows with respect to the light source yet. To properly account for the shadowed regions, the third step will re-render

[**] The jargon *b-rep* is often used in computational geometry and computer-aided design literature to abbreviate the term boundary representation.

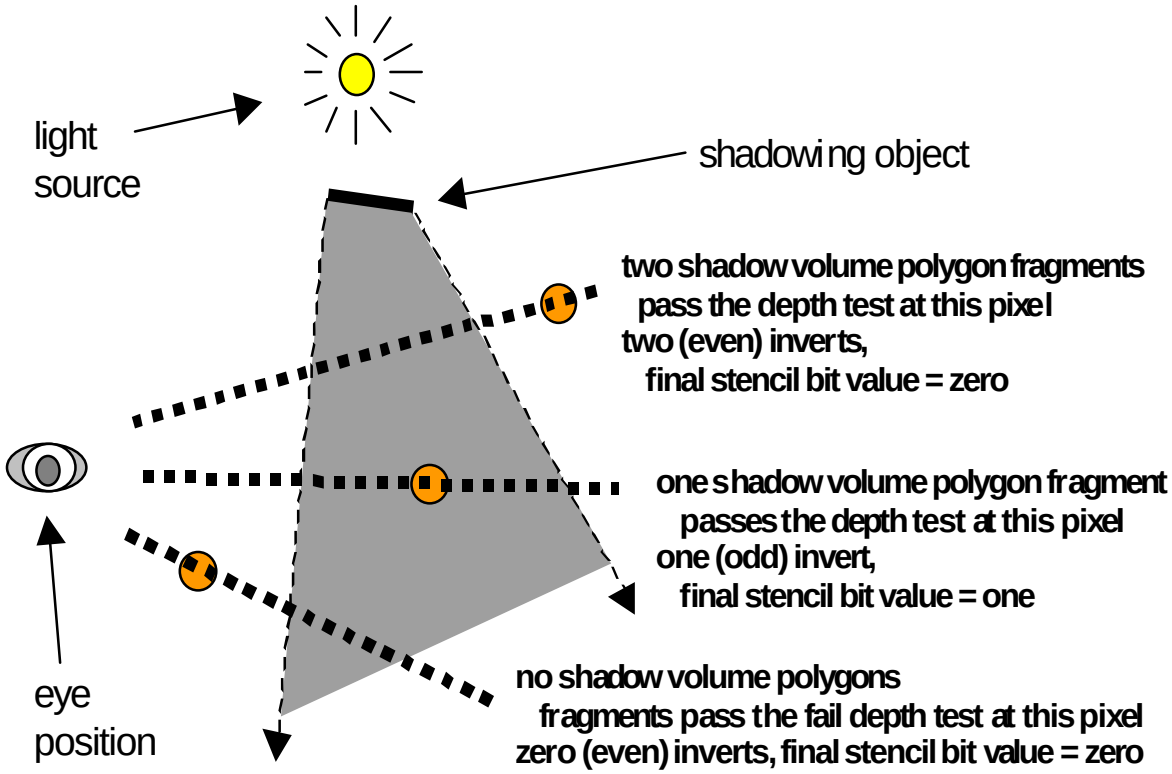


Figure 10: Two-dimensional view of counting enters and exits from a shadow volume to determine if inside or outside of the shadowed region.

any and all shadowed pixels with the light disabled. But discussing the third step is getting ahead of things.

The second step is to determine for each and every pixel in the frame buffer using stencil testing whether the visible fragment (that is, the closest fragment at each pixel as determined by depth buffering in the first step) is either inside or outside the shadow volume. We accomplish this by rendering the polygonal boundary representation of the shadow volume into the scene. While rendering the shadow volume's polygonal boundary representation, OpenGL's per-fragment state is configured as follows:

```
glDisable(GL_LIGHTING);                    // do not waste time lighting
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glDepthMask(0);                            // do not disturb the depth buffer
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 0, 0);
glStencilMask(0x1);                        // just write least significant
                                           //   stencil bit
```

```
glStencilOp(GL_KEEP, GL_KEEP, GL_INVERT);   // invert stencil bit if depth pass
glColorMask(0,0,0,0);                       // do not disturb the color buffer
```

The idea is that whenever a rendered fragment is closer than the depth of the visible fragment at a given pixel (that is, the pixel's depth value determined by the first step), invert a bit in the stencil value for that pixel, but do not disturb the depth or color buffers.

Now, draw all the polygons in the shadow volume's boundary representation like this:

```
glDisable(GL_CULL_FACE);
drawShadowVolumePolygons();
glEnable(GL_CULL_FACE);
```

Note that we disable face culling during this step.[††] It is critically important that we render *both* front and back facing polygons during this step as will be explained.

What is accomplished in this second step is not obvious. Assume that the eye point (that is, the viewer location) is outside the shadow. Then once step two is complete, the least significant stencil bit of every pixel is *one* if the particular pixel is shadowed with respect to the shadow and *zero* if the pixel is lit with respect to the shadow volume. On the other hand, if the eye point is inside the shadow volume, the opposite is true; that is, *one* means outside the shadow and *zero* means inside the shadow.

To understand why this is true, think about each polygon in the shadow volume's polygonal boundary representation as an opportunity to leave or enter the shadow volume. Recall the assumption that the boundary representation never intersects itself. This means if you are in the inside of the shadow volume and you cross the shadow volume boundary, you will then be on the outside. Likewise, if you are outside and you cross the shadow volume boundary, you will then be on the inside. Given a point that is known to be inside or outside the shadow volume, the even or odd count of how many times shadow volume polygons are crossed in getting to some other point indicates whether the other point is inside or outside the shadow volume.

For the moment, assume the eye point is not inside the shadow volume. For a given point in a scene, consider the line segment directly connecting the given point and the eye point. How many times does the line segment intersect any of the polygons in the shadow volume's boundary representation? If the number is odd, the given point is in the shadow volume, hence shadowed; otherwise, the number is even and the given point is outside the shadow volume, hence lit.

Think through the cases of zero, one, and two interesections with the eye point outside the shadow volume. In the case of zero interesections with the shadow volume boundary, the given point *must* be outside the shadow volume since the eye point is outside the shadow volume. In the case of one intersection, since the eye point is outside the shadow volume, the

[††] If face culling was already disabled, skip the disable and enable of face culling.

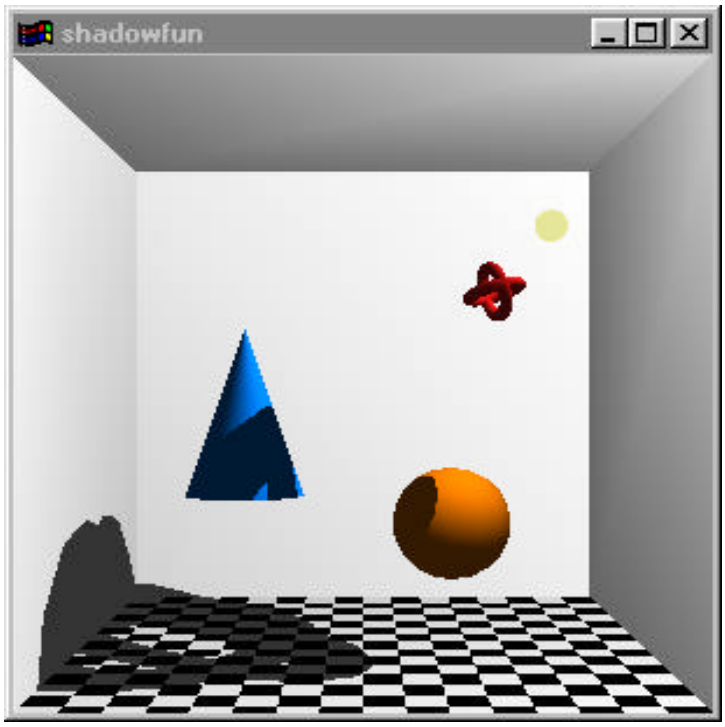


Figure 11. Scene from `shadowfun` example.

given point must be inside the shadow volume to account for the one intersection. In the case of two interesections, there must be an entry into the shadow volume, but likewise there must also be an exit from the shadow volume, so the given point must be outside the shadow volume. Figure 10 visualizes these situations in two dimensions.

Now that you have some intuition for counting the even or odd number of enters and leaves to the shadow volume boundary, consider what the OpenGL commands for step two accomplish via stencil testing. The per-fragment state will invert a pixel's stencil bit when a polygon covers the pixel *and* is in front of the pixel's depth value. Neither the depth buffer or color buffer are updated in this process; *only* the stencil buffer is updated during step two.

Note that since we are just determining evenness or oddness, a single bit of stencil is enough state to determine if we are inside or outside the shadow volume.

Once the second step has tagged pixels as inside or outside the shadow volume, the third and final step re-renders the scene with stencil testing configured to update only pixels tagged as inside the shadow volume. To make sure the shadowed light does not light updated pixels, the shadowed light is disabled. For this step, issue OpenGL commands as follows:

```
glEnable(GL_LIGHTING);                  // use lighting
glDisable(GL_LIGHT0);                    //   just not the shadowed light
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_EQUAL);                   // must match depth from 1st step
glDepthMask(0);
```

```
glEnable(GL_STENCIL_TEST);                  // and use stencil to update only
glStencilFunc(GL_EQUAL, 0x1, 0x1);          //   pixels tagged as
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);     //   "in the shadow volume"
glColorMask(1,1,1,1);
renderScene();
```

Setting the depth test to **equal** means that only fragments with depth values matching the depth values from the first pass will update the frame buffer. Additionally, the stencil test accepts only those pixels with the least-significant stencil bit set to one (that is, those tagged in the second step as being with the shadow volume). Disabling light source zero means that the shadowed light source does not contribute to the lighting of the pixel. However, lighting in general is still enabled so contributions from other light sources or any global ambient light will still affect the pixel.

6.2. Determining the shadow volume

Once you have a polygonal boundary representation for a shadow volume, the stenciling approach in the last section is straightforward to apply. However, the last section put off explaining how to compute the shadow volume for a given light source and a set of shadowing objects. The reason for this is that computing a correct shadow volume, in general, turns out to be an involved problem.

In its abstract formulation, the determination of a shadow volume is very similar to the object-space hidden surface elimination problem. While the object-space hidden surface elimination problem determines the set of closest unobscured polygons as viewed from an eye point, the shadow volume determination problem must determine the set of closest unobscured (unshadowed) polygons from a light source.

The object-space hidden surface elimination problem is complex enough that nearly all interactive graphics systems use image-space hidden surface elimination algorithms such as the depth buffer instead.[‡‡,15,16,17] While the shadow volume computation stage of the shadow volume techniques requires object-space computations to construct the shadow volume, the second stage uses image-space stencil testing. The complete stenciled shadow volume

---

[‡‡] In the same way that image-space hidden surface elimination algorithms such as depth buffering are simpler than object-space hidden surface removal techniques, image-space shadow algorithms can be simpler than shadow techniques such as shadow volumes that require object-space computations. Shadow mapping techniques such as those described by Williams[15] and Reeves, Salesin, and Cook[16] avoid the problem of object-space shadow volume construction by operating in image-space. However, shadow mapping techniques have other problems due to sampling and filtering. Because shadow mapping techniques are similar to texture mapping techniques, shadow mapping hardware support may become a viable alternative to stenciled shadow volumes as a means to hardware accelerate shadow rendering in the future. The Silicon Graphics RealityEngine and InfiniteReality graphics subsystems provide support for shadow mapping today though neither has adequate filtering or sampling suitable for good quality shadows. These machines support shadow mapping via the `SGIX_depth_texture` and `SGIX_shadow` OpenGL extensions.[17] Unfortunately, most existing texture mapping graphics hardware lacks the critical additional functionality required to support shadow mapping.

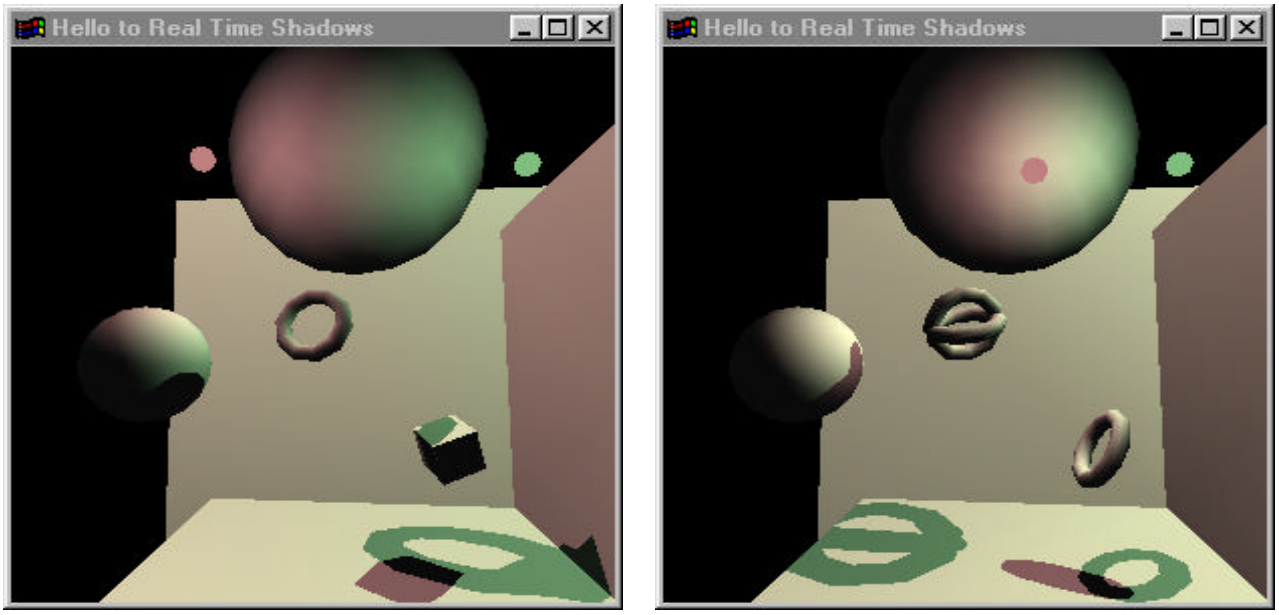


Figure 12. Two scenes from `hello2rts` example. Note overlapping shadows and shadows from two light sources.

technique is therefore a hybrid approach that relies on both object-space and image-space stages.

Instead of presenting some foolproof object-space algorithm for shadow volume construction that would work in all cases, yet would be too slow for use in an interactive application, the approach here is to present techniques for exact shadow volumes in a common special case and fast approximation of shadow volumes for harder cases.

6.2.1. The special case of a planar cut-out

Consider the shadow volume cast by a single triangle. The shadow volume is formed by the triangle itself, and then three infinite polygons projected away from the light source. As mentioned earlier, the polygons projected away from the light source need only to extend sufficiently beyond the view frustum. Then standard OpenGL view frustum clipping will clip the polygons to the view frustum automatically.

The basic idea of extending a triangle into its shadow volume can be generalized for arbitrary *n*-sided non-intersecting polygons. Appendix C details a routine called `extendVolume` that renders a loop of quads that extend from a shadowing polygon defined by a sequence of co-planar vertices in a direction away from a specified light position. An additional parameter indicates whether the light position is either a 3D position (for a local or positional light) or a 3D direction (for an infinite or directional light). A final parameter supplies a scaling factor to determine how far to scale the extended vertices. This parameter should be sufficiently large to guarantee that the extended vertices will fall outside of the view frustum.

The polygon defined by the sequence of co-planar vertices passed to `extendVolume` is logically part of the shadow volume too. However, because this polygon is assumed to be a

part of the scene and therefore rendered into the depth buffer during step one of the algorithm in Section 6.1, it is not necessary to render the polygon itself during the rendering of shadow volume polygons in step two.

A set of non-intersecting co-planar polygons is referred to as a *planar cut-out*. Common shadowing objects such as a wall with multiple windows or a latticework can be approximated by planar cut-outs. The NVIDIA logo in Figure 9 is an example of a cut-out. Because all the vertices of a planar cut-out are defined to be co-planar, a cut-out has the nice property that it never shadows itself. (Think of the cut-out as infinitely thin.) Also, because the cut-out is defined by a set of polygons, it can be extended into a shadow volume by simply calling `extendVolume` for each independent polygon making up the cut-out.

Advanced: While beyond the scope of this paper to explain fully, the static 2D geometry for a cut-out can be extruded from the Z=0 plane to the Z=1 plane. As a consequence of the *Fundamental Theorem of Projective Geometry,*[§§] there exists a projective transformation that will transform static geometry such as an extruded cut-out into a given world space coordinate system. This projective transformation can be parameterized by the light position, cut-out position and orientation, and some distance to a plane parallel to the cut-out plane. Like any projective transformation, the required transformation from cut-out space to world space is defined by a unique 4 by 4 matrix.

Once this 4 by 4 matrix is determined, this matrix can be concatenated with OpenGL's modelview matrix. Then the static extruded 2D geometry for the cut-out as described above can be projected automatically as part of OpenGL's projective coordinate transformation process. Updating the shadow volume when the light source or cut-out position or orientation changes is simply a matter of computing an updated 4 by 4 matrix. Because the extruded cut-out geometry is static, it is suitable for compilation into an OpenGL display list. Moreover, when graphics hardware accelerates OpenGL's coordinate transformation, the per-vertex computations required to extend the shadow volume away from the light are off-loaded to the dedicated graphics hardware. Computing the necessary 4 by 4 matrix is similar to the problems of view reconstruction or camera calibration. Solving for the 4 by 4 matrix requires the solution to 15 simultaneous linear equations with 15 unknowns. (A 4 by 4 matrix has 16 variables, but if the matrix represents a projective transformation, one of the 16 values is dependent on the other 15).

This is the approach used in the example shown in Figure 9. For further background on this topic, consult Chapter 6 of Penna and Patterson's useful book on projective geometry.[18]

[§§] The *Fundamental Theorem of Projective Geometry* says that if A, B, C, D, and E are five points in projective three-space, no four of which are co-planar, and P, Q, R, S, and T are five points in projective three-space, no four of which are co-planar, then there is one and only one projective transformation taking A to P, B to Q, C to R, D to S, and E to T.

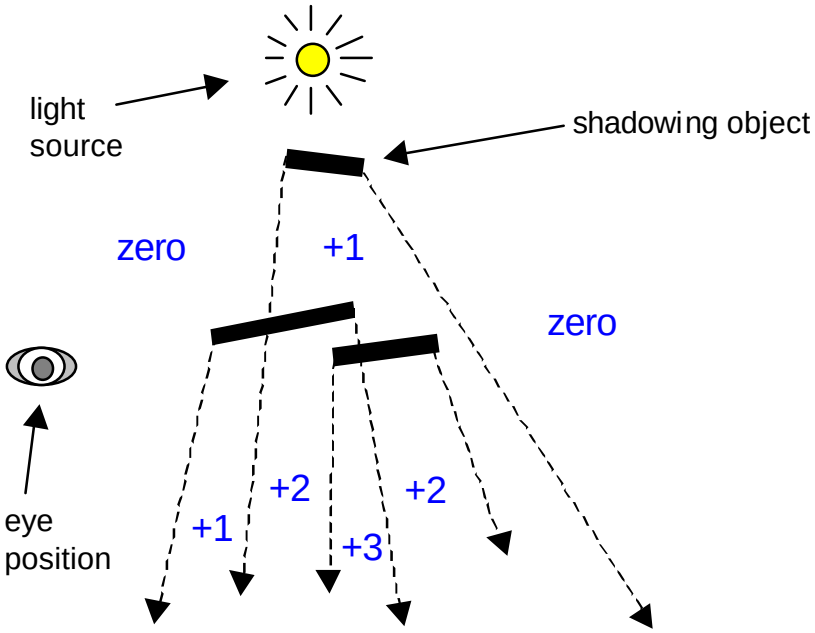


Figure 13. Counting is used to track being inside and outside of the union of multiple shadow volumes. A zero count is outside of shadow. A positive count is within the shadow.

6.2.2. Projecting objects flat to make cut-outs

The simplicity of a cut-out can be applied to more complex geometry by projecting a complex object into a 2D plane. The idea is to flatten the complex geometry into a simple cut-out. Of course, this will not exactly match the true shadow volume from the real geometry. But it can make for a good, fast approximation. This simplifying approach is similar to the way a sprite or billboard is sometimes used in 3D games to replace the geometry of a more complex object. The true shadow volume of a complex object is replaced with the simpler shadow volume of a cut-out approximating the complex object.

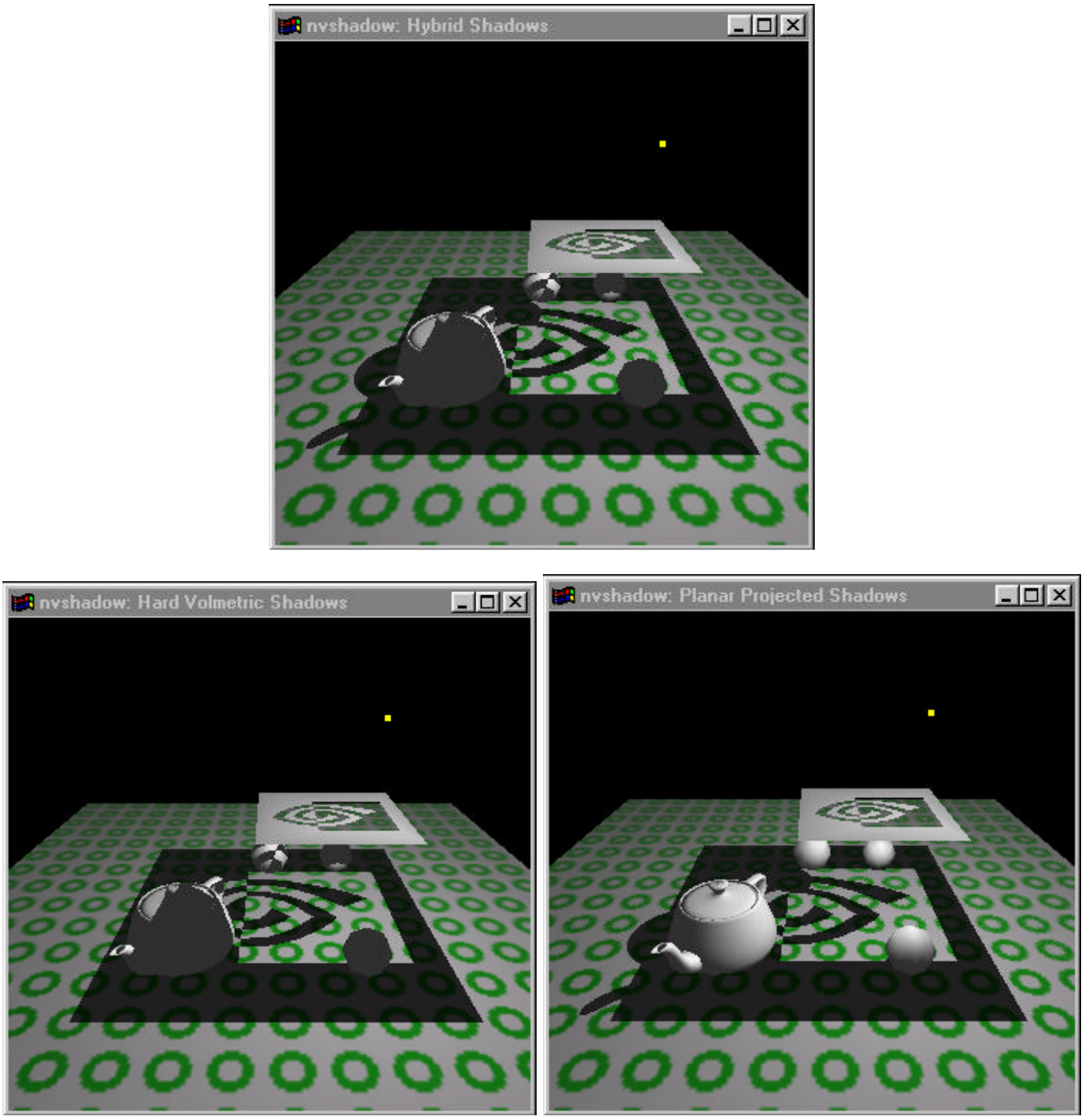


Figure 14. The top image combines the stenciled shadow volume approach with the projected planar shadow approach. The left image uses the stenciled shadow volume approach only. The right image uses the planar projected shadow approach only. Notice that the top image has both the shadow of the teapot spout on the floor and the shadow from the logo on the teapot too.

Once the 3D geometry is projected flat into a plane, lots of polygons are likely to overlap. A cut-out requires reducing this 2D pile of polygons to its silhouette boundary. The OpenGL Utility (GLU) library's polygon tessellator is well suited to determine the silhouette boundary for the 2D-projected object. If you use the GLU tessellator, you probably want to choose the `GLU_TESS_WINDING_NONZERO` winding rule option to `gluTessProperty`. You may also find the `GLU_TESS_BOUNDARY_ONLY` option useful if you do not need the polygon boundary tessellation itself. Most often though, you will want both the silhouette boundary and the polygonal tessellation of the silhouette to cap off the top of the cut-out shadow volume (in the case of an approximating cut-out, you cannot rely on the cut-out geometry to cap exactly the shadow volume). In this case, you can use the `GLU_TESS_EDGE_FLAG` callback to determine whether edges are exterior, and therefore on the boundary, or interior.

The `hello2rts` and `shadowfun` OpenGL stencil shadowing examples[14] use the approach of approximating the shadow volume from the 2D silhouette cut-out of objects. These programs use OpenGL's feedback mode to transform and capture the vertices of the object for constructing a silhouette and eventually a shadow volume. Figure 11 shows a scene from `shadowfun`. Figure 12 shows two scenes from `hello2rts`. Note that the `hello2rts` scene has multiple objects casting shadows from multiple light sources. This is accomplished by adding the contribution of each light excluding shadowed regions into the scene. Each light-object shadow interaction is modeled separately so two objects and two lights require four shadow volumes.

One temptation that should be avoided when creating silhouettes and shadow volumes is using a coarser polygonal representation for the object when constructing the shadow volume. Because shadows, particularly shadows from local light sources, tend to be larger than the object itself, the coarseness of the tessellation is magnified in the shadow. Indeed, a common problem with shadows cast by a curved object such as a sphere is that its enlarged shadow gives away the object's true polygonal basis.

Tessellating the silhouette boundary of an object projected into 2D is not a cheap operation. There are ways to minimize the cost of tessellating a silhouette. If the object's geometry is closed and all the edge and face information is available, you can minimize the set of edges considered by the tessellator. For a closed polygonal representation, only edges that share a face facing the light and a face facing away from the light will be candidates for silhouette edges.

6.2.3. Generic shadow volume construction and usage issues

There are a number of common problems associated with building shadow volumes with which you should be aware. These can cause problems for shadow volume techniques. Models that appear closed may not in fact be closed due to poor modeling. Sometimes polygons in models are non-planar. It can be difficult to determine whether the eye is in the shadow or not. In cases where an object is very nearly planar with a shadow volume plane, the object may pop in and out of the shadow during animation. Shadow volumes may not interact well with techniques that use OpenGL's polygon offset functionality to offset depth values. Refer to Bergeron[11] for suggestions about resolving these difficulties.

A significant performance issue with shadow volumes is the need to re-render the scene multiple times. One way to minimize the overhead due to multiple renderings is to bound the shadow region to a conservative shadow frustum. An application that can efficiently cull its scenery to what is visible from the view frustum can use this same culling approach to also cull against the conservative shadow frustum.

Additionally, rendering the shadow volume boundary polygons can be a significant consumer of rasterization and pixel fill rate resources. The more intricate the shadow volume, the more time must be spent rendering its polygons. And keep in mind that a small object can cast a very large shadow.

The near clipping plane can cause problems when rendering the shadow volume boundary polygons. You will need to figure in the near clip plane's clipping effect into the shadow volume if shadow volume polygons fall in the pyramidal region between the eye point and the view frustum's near clip plane.

So far, the discussion has quietly ignored the issue of shadows cast by semi-transparent objects. Unfortunately, shadows due to semi-transparent object are a more complex topic than they might seem at first consideration. Consider that a glass of white wine in bright sunlight. The shadow from the glass of wine can actually contain bright spots known as caustics because of how the light has refracted and focused itself within the shadow. These types of lighting effects can be modeled with ray tracing techniques, but such effects are difficult to render with today's available interactive graphics hardware.

6.3. Counting Enters and Leaves with Stencil Testing

There is an alternative to using the **invert** stencil operation to track both enters and exits of a shadow volume with a single operation. The alternative is to use the **increment** stencil operation to track shadow volume enters and the **decrement** stencil operation to track exits. Assuming the viewer is outside the shadow volume, if the stencil count is greater or equal to one, a pixel is within the shadow, but if the count is zero, a pixel is outside the shadow.

This "counting" approach has the advantage that the shadow volume does not have to be reduced to its polygonal boundary representation as required by the inverting approach. The inverting approach requires the elimination of any "internal" boundaries that might result from unioning two or more shadow volume boundary representations. However with the counting approach, the union of multiple intersecting shadow volumes can be handled even though the constituent shadow volumes overlap each other. Overlapping shadow volumes create the possibility of "entering" the combined (unioned) shadow volume twice without leaving, but counting properly tracks multiple enters and exits. Inverting a single bit is not as powerful as counting. Figure 13 shows in 2D how counting can be used for multiple shadow volumes.

With the counting technique, the hard problem of determining the unioned polygonal boundary representation for a complex shadow volume can be skipped. In the counting approach, you can represent the shadow volume for the entire scene as simply the union of the shadow

volumes extended from each and every polygon in the whole scene. The shadow volume for each polygon is easy to construct with the `extendVolume` routine in Appendix C.

The counting approach can be implemented in OpenGL like this:

```
glDisable(GL_LIGHTING);                  // lighting not needed
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glDepthMask(0);                          // do not disturb depth buffer
glColorMask(0,0,0,0);                    // do not disturb color buffer
glStencilMask(~0u);
glEnable(GL_CULL_FACE);                  // use face culling

glCullFace(GL_BACK);                     // increment for front facing fragments
glStencilOp(GL_KEEP, GL_KEEP, GL_INCR);  //   that pass the depth test
for (i=0; i<numPolygonsInScene; i++)     // for every polygon in the scene
  renderShadowVolumeForPolygon(i);       //   call extendVolume for ith polygon

glCullFace(GL_FRONT);                    // decrement for back facing fragments
glStencilOp(GL_KEEP, GL_KEEP, GL_DECR);  //   that pass the depth test
for (i=0; i<numPolygonsInScene; i++)     // for every polygon in the scene
  renderShadowVolumeForPolygon(i);       //   call extendVolume for ith polygon
```

By using OpenGL's polygon face culling mode, just the front-facing shadow volume polygons are rendered while incrementing the stencil value of pixels that pass the depth test. Then, in a second pass, just the back-facing shadow volume polygons are rendered while decrementing the stencil value of pixels that pass the depth test. As viewed from the eye point, front-facing shadow volume polygons are "entries" into the shadow volume, while back-facing shadow volume polygons are "exits" from the shadow volume.

As promising as this approach appears, the approach has a significant practical problem when used for interactive graphics. The polygons of a shadow volume tend to be large in terms of the number of pixels they cover. While the fragments generated by rendering the shadow volumes are invisible, they still take time to process. Rendering a set of large shadow volume polygons for every polygon in the scene will very significantly increase the cost of rendering the entire scene. Also note that, unlike in the invert approach, we render the shadow volume polygons in two passes. While face culling ensures that each polygon is actually only rendered once, every polygon must still be transformed twice. The up front simplicity of the counting approach has a high hidden cost due to increased transformation, rasterization, and pixel update requirements when naively rendering the unioned shadow volume polygons.

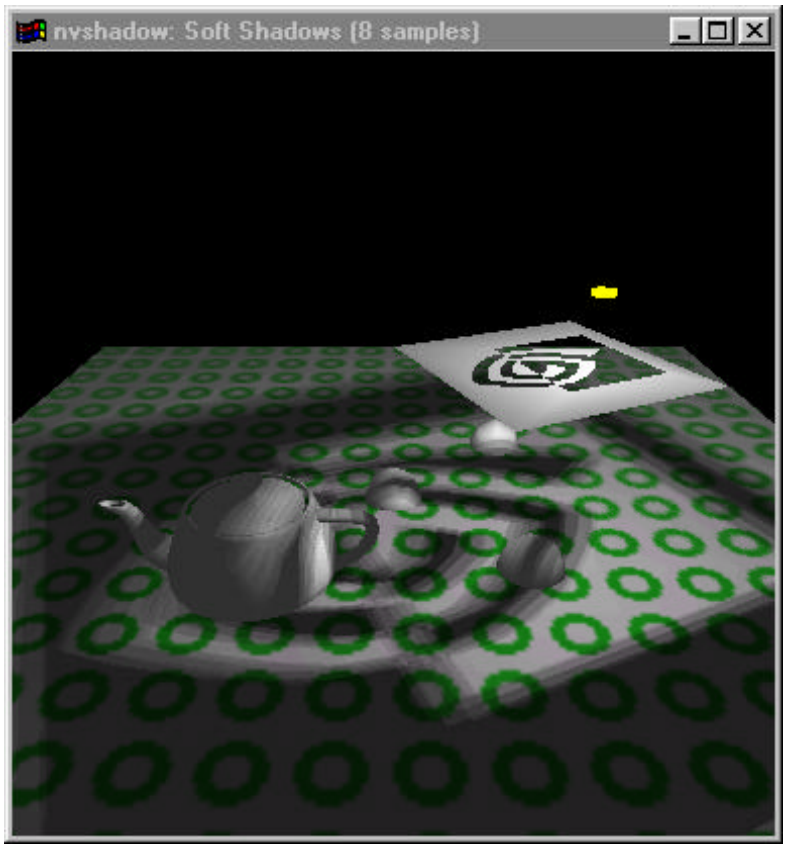


Figure 15. Soft shadows using multiple shadow volumes to simulate a non-point light source. There are banding artifacts. More shadow volumes would reduce the banding, but increase the rendering time too.

Another problem with this approach is that if the polygons in the scene do not seam together exactly, cracks may appear in the shadow. Admittedly, this is merely a modeling issue, but it still ends up often being an issue in practice.

The invert approach makes for more expensive object-space silhouette boundary computations, but this is often cheaper than the pixel update overhead of extending a shadow volume for each and every polygon in the scene. Additionally, the invert approach requires a single bit of stencil so that other stencil bits can be used for other purposes. The counting approach requires enough stencil values to accommodate the deepest nesting of shadow volume entries which is difficult to know up front. My own practical experience is that that the pixel update overhead of the counting approach is far too high for its practical use in interactive graphics.

In practice, hybrid shadow approaches can be utilized to balance the costs of various approaches. For example, if you refer back to the left image in Figure 9, you will notice that while the NVIDIA logo casts a shadow on the ground and the balls and teapots in the scene,

the balls and teapot themselves do not cast a shadow on the ground. The problem is that that shadow volume only accounts for the NVIDIA logo. *The shadows generated by a shadow volume are only as good as the shadow volume used to generate them.* If the shadow volume included the teapot and balls, the shadows from these objects would be included in the scene. Unfortunately, constructing such a complex shadow volume would be expensive and difficult. However, it is easy to construct the shadow volume for just the NVIDIA logo since the logo is a cut-out.

Because the scene in question has a flat ground plane, a good compromise to improve the overall appearance of the shadows in the scene without greatly increasing the rendering overhead is using projected planar shadows as described in Section 5. The planar projected shadows can account for the ground shadows cast by the teapots and balls. The top image in Figure 14 show the resulting combined effect, while the left and right images show the independent contributions of the shadow volume and projected planar shadow approaches.

6.4. Soft Shadows with Shadow Volumes

So far, all the consideration of shadows has been those cast from point or directional light sources. Shadows in the real world are often the result of area light sources. This accounts for much (but not all) of the "softness" associated with shadow boundaries. This soft region of a shadow is known as the penumbra.

A brute force way to approximate the appearance of soft shadows with stenciled shadow volumes is to model an area light source as a collection of point light sources. Brotman and Badler[19] used this approach to generate their soft shadows.

Figure 15 shows an example of this approach. The image uses 8 clustered point light sources to approximate a small area light source. A distinct shadow volume is extended from the NVIDIA logo for each of the 8 point light sources. The scene is first rendered with no lights. Then each shadow volume is rendered into the stencil buffer, each assigned to a unique stencil bit. Each of the 8 point light sources is then positioned, enabled at an eighth the brightness of the combined light, and rendered with stencil testing to only update pixels outside the light's particular shadow volume. Additive blending is used to accumulate the contribution of each light. Because blending is used, we must be careful to set the tested stencil bit if we update the pixel to avoid the possibility of double blending.

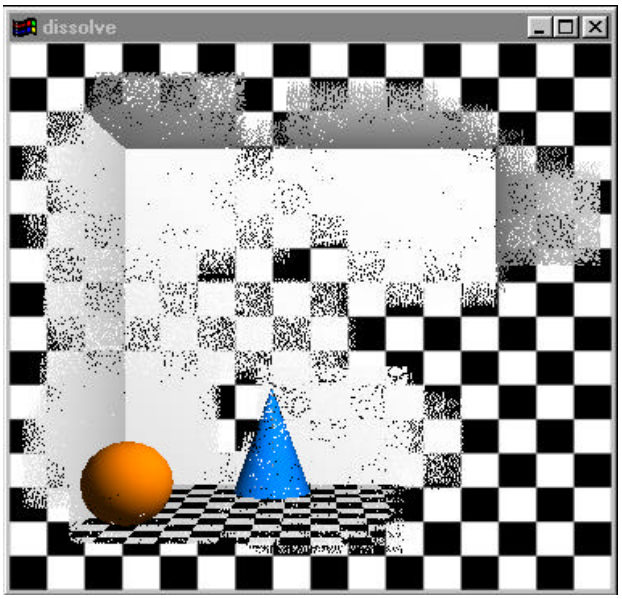


Figure 16. Stenciled "digital dissolve" effect. The checker board pattern and a 3D scene with a sphere and a cone are merged into a single image. The stencil buffer pattern controls which scene updates which pixels.

This approach has its limitations. The softness of the shadow depends on an adequate number of samples. Unfortunately, the time to render the scene increases linearly with the number of samples used to approximate an area light source. Also unsightly banding artifacts are introduced if not enough samples are used

If you use enough shadow volume samples, you may run out of frame buffer precision to accumulate all the lighting contributions. The extended precision of a hardware accumulation buffer[20] can alleviate this problem though.

While a nice idea, this approach to soft shadows is expensive enough that it is not viable for interactive rendering on today's graphics hardware.

7. Other Applications for Stenciling

The applications for stencil testing go well beyond reflections and shadows. This section briefly describes other interesting rendering techniques using stencil testing.

- **Digital dissolve effects.** A "dissolve" pattern in the stencil buffer can combine two scenes on a per-pixel basis. For example, one scene may be rendered to update only pixels with a stencil value of zero. A second scene may be rendered to update only pixels with a stencil value of one. By shifting the stencil value distribution from all 0's to all 1's and rendering

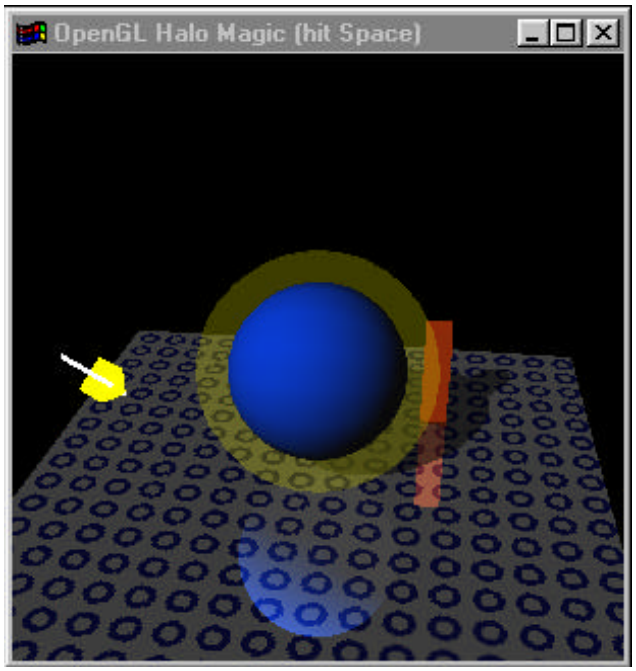


Figure 17. Stenciled magic halo effect. Note that the magic halo is combined with a stenciled reflection and shadow.

both stencil-tested scenes, a smooth digital dissolve can be achieved. Figure 16 shows an example of this effect.[***]

- **Magic haloes.** A magic halo extends a glow around an object, but the halo does not obscure the object itself (that's what makes it a "magic" halo).[21] A game may use a halo to indicate that a particular character or object has a special capability or power.

  The halo effect can be accomplished by first rendering the object with a particular stencil value. Then apply a scale transformation centered at the object's center to the modelview transform. Render the object's geometry a second time (scaled larger this time) but color the object based on the intended glow color (instead of the object's normal color and texture), and use stenciling to *not* update pixels tagged with the particular stencil value used to render the object the first time. This way the halo does not obscure the object. Finally, undo the scale transformation. Optionally, use blending to blend the halo with the background behind the object (use stenciling to avoid double blending as necessary). This works best when the object is relatively convex. However, in the case of a non-convex object, break the object into convex pieces and extend the halo around each piece. Figure 17 shows an example of such a stenciled magic halo applied to a sphere.

[***] The source code for `dissolve.c` is found in the `progs/advanced` directory of the GLUT 3.7 source code distribution. Tom McReynolds wrote the example.

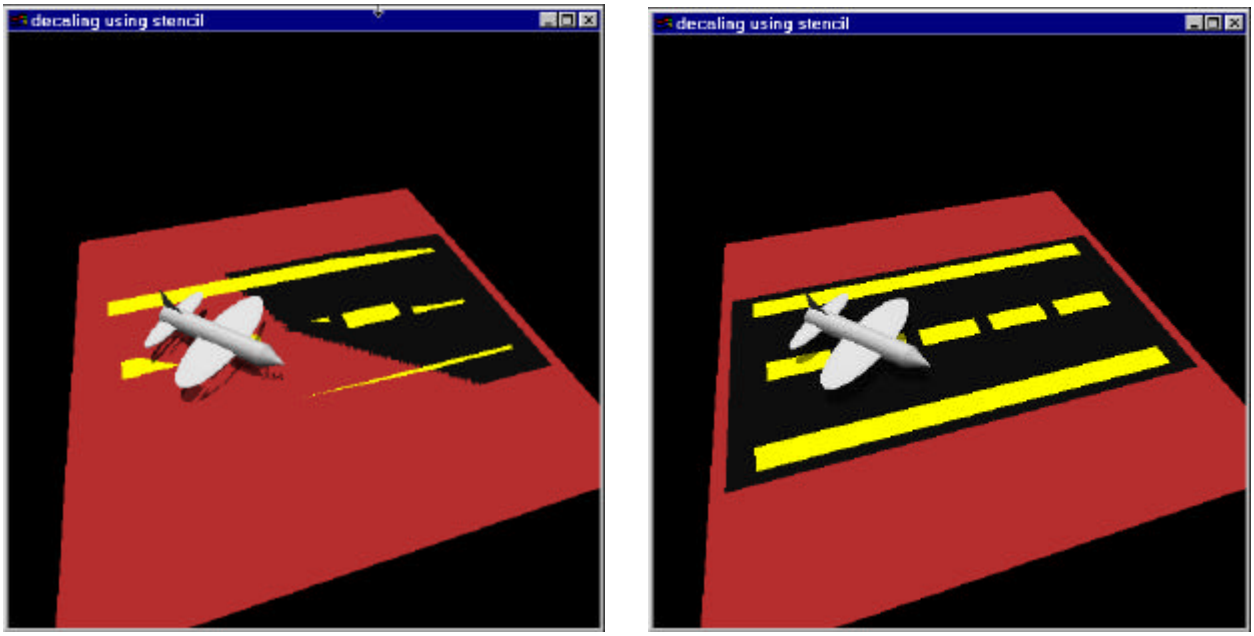


Figure 18. The left image suffers from "Z fighting" artifacts due to the co-planar nature of the runway markings and the airplane's shadow. The right image uses stencil testing to render correctly the co-planar geometry.

I've recently seen a Quake 2 mod that tries to accomplish the basic idea of a magic halo around certain characters, but without stencil. The scaling approach described above is attempted, but without the benefit of stencil. Not only is there unsightly double blending of the halo, but the yellow halo covers and obscures the entire character (instead of just being a glow around the character). The effect is something akin to seeing a guy walking around covered in yellow radioactive snot. Stencil could make this look much cooler.

- **Co-planar geometry.** Artifacts due to rendering co-planar geometry crop up in lots of situations. For example, rendering airplane runways often requires adding runway markings that should be drawn exactly co-planar to the runway surface. These sorts of situations create the potential for what is known as "Z fighting" where ordering of depth buffer values for co-planar or nearly co-planar surfaces varies depending on the rasterization parameters. The net effect is an indeterminate depth ordering because co-planar polygon fragments change their visibility from frame to frame and from pixel to pixel.

  The planar projected shadows discussed in Section 5 are another common case where co-planar geometry occurs. Polygon offset can help in some situations, but sometimes it is important that the depth buffer reflect the true (non-offset) depth values for the geometry in a scene.

  Stencil testing can resolve the ambiguity of co-planar polygons. By rendering the first co-planar polygon with a particular stencil value, subsequent co-planar polygons can use

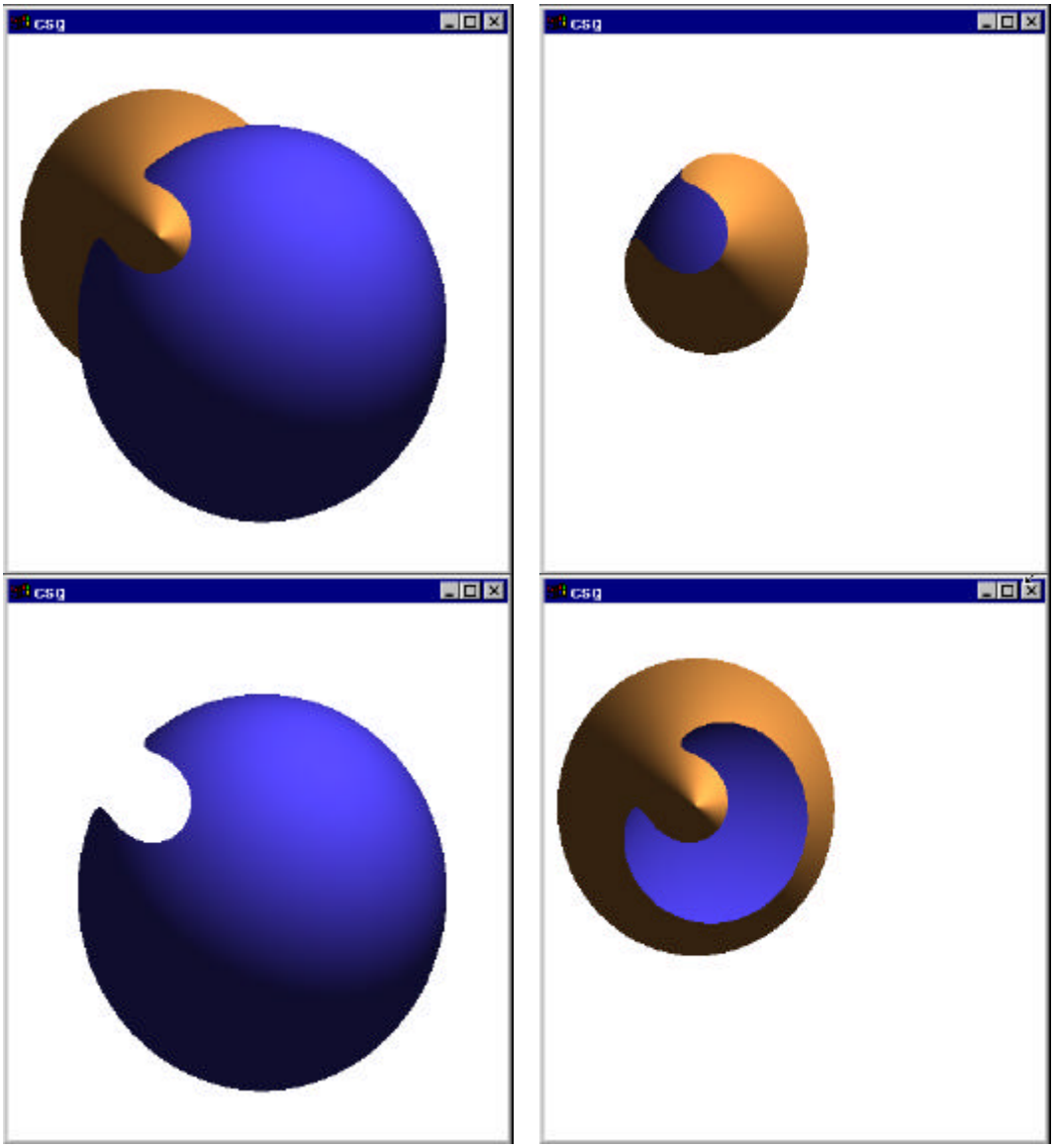


Figure 19. Four Constructive Solid Geometry configurations. Upper left: A or B. Upper right: A and B. Lower left: A minus B. Lower right: B minus A.

stencil testing to match the first polygon's stencil value without depth testing. Figure 18 demonstrates how stencil testing can help improve situations involving co-planar geometry.

- **Constructive Solid Geometry.** Computer-aided design and 3D modeling applications often require the ability to construct new solids by unioning, intersecting, and subtracting existing solids. This is commonly called Constructive Solid Geometry or CSG for short. While an object-space solution is eventually required, fast visualization of CSG models is important for interactive designing CSG-based objects.

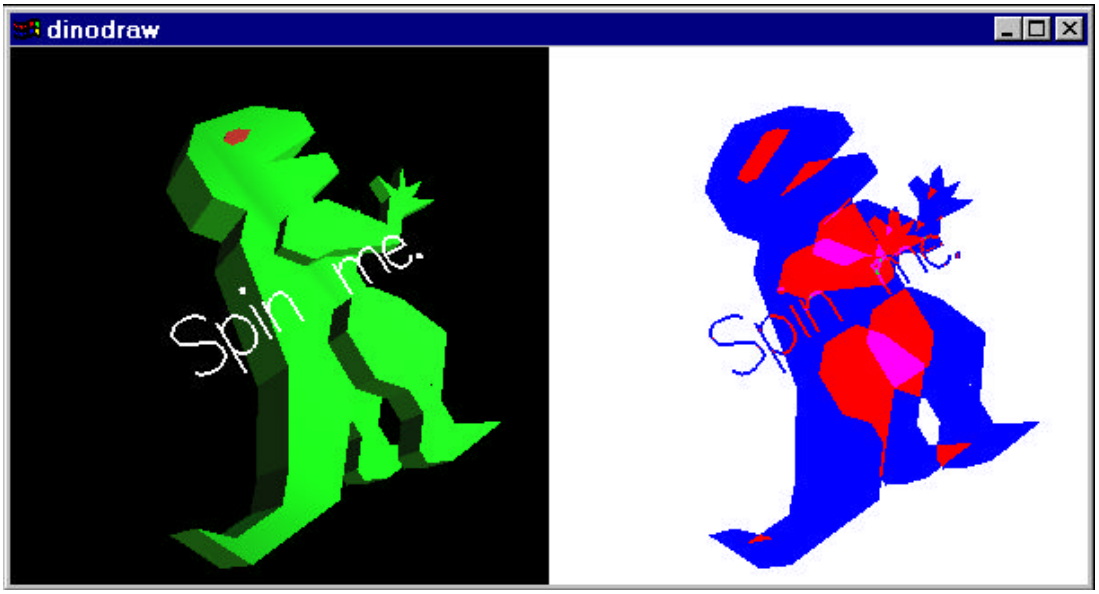


Figure 20. The left dinosaur is drawn normally. The right dinosaur uses stencil testing to visualize the depth complexity of the dinosaur.

Stencil testing can accelerate the visualization of CSG models.[22,23,24] Figure 19 shows some examples of stenciled CSG rendering.

- **Visualizing Depth Complexity.** For exploring performance issues in interactive 3D rendering applications, it is often useful to visualize the depth complexity of a given scene.[25] By incrementing the stencil value of a pixel each time it is updated by a fragment, the stencil buffer can reflect the depth complexity of the rendering scene. Then, the stencil buffer can be displayed in pseudo-color to visualize the scene's depth complexity.

  A hardware stencil buffer makes this fast enough for interactive visualization and animation of a 3D game or application's depth complexity. Figure 20 shows an example of visualizing depth complexity with stencil testing.

8. Conclusions

Stencil testing provides 3D programmers a versatile means to control per-pixel updates. Stenciling improves the quality of planar reflections and planar projected shadows. Stenciled shadow volumes provide a means to render volumetric shadows. For further information about rendering shadows, I recommend the survey of shadow algorithms written by Woo, Poulin, and Fournier.[26]

Keep in mind that stenciling is useful for a host of rendering effects beyond better quality reflections and shadows.

Stencil testing is already available in mass-market PC graphics hardware such as the RIVA TNT, and because both OpenGL and DirectX support stencil testing, 3D developers can expect that stenciling will quickly become a standard graphics hardware feature. 3D developers that use stencil testing can provide a richer, higher-quality visual experience.

A. Construct a Reflection Matrix given an Arbitrary Plane

The following routine `mirrorMatrix` constructs a 4x4 matrix suitable for passing to `glMultMatrixf` that reflects coordinates through a plane defined by a point p on the plane and a normalized vector v perpendicular to the plane.[27]

```
void mirrorMatrix(GLfloat m[4][4],  // OUT: resulting matrix
                  GLfloat p[3],     // IN: point on the plane
                  GLfloat v[3])     // IN: plane perpendicular direction
{
  GLfloat dot = p[0]*v[0] + p[1]*v[1] + p[2]*v[2];

  m[0][0] = 1 - 2*v[0]*v[0];
  m[1][0] =   - 2*v[0]*v[1];
  m[2][0] =   - 2*v[0]*v[2];
  m[3][0] = 2*dot*v[0];

  m[0][1] =   - 2*v[1]*v[0];
  m[1][1] = 1 - 2*v[1]*v[1];
  m[2][1] =   - 2*v[1]*v[2];
  m[3][1] = 2*dot*v[1];

  m[0][2] =   - 2*v[2]*v[0];
  m[1][2] =   - 2*v[2]*v[1];
  m[2][2] = 1 - 2*v[2]*v[2];
  m[3][2] = 2*dot*v[2];

  m[0][3] = 0;
  m[1][3] = 0;
  m[2][3] = 0;
  m[3][3] = 1;
}
```

Here is an example using `mirrorMatrix` to reflect a scene through a particular plane:

```
Glfloat planePoint[3] = { 3.0, 0.0, -1.0 };
Glfloat planeVector[3] = { 0.65, -0.34, 0.6843 };  // normalized
Glfloat matrix[4][4];

glPushMatrix();
  mirrorMatrix(matrix, planePoint, planeVector);
  glMultMatrixf(&matrix[0][0]);
  drawScene();
glPopMatrix();
glDrawScene();
```

B. Construct a Planar Projected Shadow Matrix given a Point and a Plane Equation

The following routine shadowMatrix constructs a 4x4 matrix suitable for passing to `glMultMatrixf` that projects coordinates onto the specified plane (the ground plane) based on the homogeneous point (the light source position).

```
void shadowMatrix(Glfloat m[4][4],
                  GLfloat plane[4],
                  GLfloat light[4])
{
  GLfloat dot = plane[0]*light[0] + plane[1]*light[1] +
                plane[2]*light[2] + plane[3]*light[3];

  m[0][0] = dot - light[0]*plane[0];
  m[1][0] =     - light[0]*plane[1];
  m[2][0] =     - light[0]*plane[2];
  m[3][0] =     - light[0]*plane[3];

  m[0][1] =     - light[1]*plane[0];
  m[1][1] = dot - light[1]*plane[1];
  m[2][1] =     - light[1]*plane[2];
  m[3][1] =     - light[1]*plane[3];

  m[0][2] =     - light[2]*plane[0];
  m[1][2] =     - light[2]*plane[1];
  m[2][2] = dot - light[2]*plane[2];
  m[3][2] =     - light[2]*plane[3];

  m[0][3] =     - light[3]*plane[0];
  m[1][3] =     - light[3]*plane[1];
  m[2][3] =     - light[3]*plane[2];
  m[3][3] = dot - light[3]*plane[3];
}
```

C. Given a Light Source, Extend a Shadow Volume from a Polygon

The following routine `extendVolume` renders with OpenGL a loop of quads extended from an n-side polygon with co-planar vertices `v` away from a light. In the case of a local (positional) light, `localLight` should be non-zero, and then the `lightPosition` is considered a 3D position. In the case of an infinite (directional) light, `localLight` should be zero, and then `lightPosition` is treated as a 3D direction. The `extendDistance` parameter should be a sufficiently large positive value that ensures that the extended vertices are always extended beyond the view frustum. The polygon itself is *not* rendered by this routine.

```
typedef GLfloat POSITION[3];

void extendVolume(int n, POSITION v[],
                  int localLight, POSITION lightPosition,
                  float extendDistance)
{
  POSITION extendedVertex, extendDirection;
  int i;

  // If light source is infinite (directional)...
  if (!localLight) {
    // compute direction opposite from light source direction.
    extendDirection[0] = -lightPosition[0];
    extendDirection[1] = -lightPosition[1];
    extendDirection[2] = -lightPosition[2];
  }

  glBegin(GL_QUAD_STRIP);
  // If light source is local (positional)...
  if (localLight) {
    for (i=0; i<n; i++) {
      glVertex3fv(v[i]);
      // Compute per-vertex direction from vertex away from the light source.
      extendDirection[0] = v[i][0] - lightPosition[0];
      extendDirection[1] = v[i][1] - lightPosition[1];
      extendDirection[2] = v[i][2] - lightPosition[2];
      // Compute extended vertex.
      extendedVertex[0] = v[i][0] + extendDirection[0] * extendDistance;
      extendedVertex[1] = v[i][1] + extendDirection[1] * extendDistance;
      extendedVertex[2] = v[i][2] + extendDirection[2] * extendDistance;
      glVertex3fv(extendedVertex);
    }
    // Repeat initial 2 vertices to close the quad strip loop.
    glVertex3fv(v[0]);
    extendDirection[0] = v[0][0] - lightPosition[0];
    extendDirection[1] = v[0][1] - lightPosition[1];
    extendDirection[2] = v[0][2] - lightPosition[2];
    extendedVertex[0] = v[0][0] + extendDirection[0] * extendDistance;
    extendedVertex[1] = v[0][1] + extendDirection[1] * extendDistance;
    extendedVertex[2] = v[0][2] + extendDirection[2] * extendDistance;
    glVertex3fv(extendedVertex);
  // otherwise, light source is infinite (directional)...
```

```
  } else {
    for (i=0; i<n; i++) {
      glVertex3fv(v[i]);
      // Compute extended vertex.
      extendedVertex[0] = v[i][0] + extendDirection[0] * extendDistance;
      extendedVertex[1] = v[i][1] + extendDirection[1] * extendDistance;
      extendedVertex[2] = v[i][2] + extendDirection[2] * extendDistance;
      glVertex3fv(extendedVertex);
    }
    // Repeat initial 2 vertices to close the quad strip loop.
    glVertex3fv(v[0]);
    extendedVertex[0] = v[0][0] + extendDirection[0] * extendDistance;
    extendedVertex[1] = v[0][1] + extendDirection[1] * extendDistance;
    extendedVertex[2] = v[0][2] + extendDirection[2] * extendDistance;
    glVertex3fv(extendedVertex);
  }
  glEnd();
}
```

Here is an example using extendVolume to render the shadow volume for a triangle:

```
POSITION triangle[3] = { { 0, 0, 0 }, { 1, 0, 0}, { 0, 1, 0 } };
POSITION lightPosition = { 1, 1, 7 };

glDisable(GL_LIGHTING);
extendVolume(3, triangle,
  1, lightPosition,                    // local light
  100000.0);                           // big positive number
glEnable(GL_LIGHTING);
```

---

[1] Kurt Akeley and Tom Jermoluk, “High-Performance Polygon Rendering,” *SIGGRAPH ’88 Proceedings*, pp. 239-246, August 1988.

[2] Mark J. Kilgard, *OpenGL Programming for the X Window System*, Addison-Wesley, 1996.

[3] Tim Hall, “A how to for using OpenGL to render mirrors,” a posting to *comp.graphics.api.opengl* (a Usenet newsgroup), August 1, 1996. http://reality.sgi.com/opengl/tips/TimHall_Reflections.txt

[4] Paul Diefenbach and Norman Badler, “Pipeline rendering: Interactive Refractions, Reflections, and Shadows,” *Displays: Special Issue on Interactive Graphics*, 15(3), pp. 173-180, 1994. ftp://ftp.cis.upenn.edu/pub/diefenba/displays.ps.Z

[5] Paul Diefenbach, *Pipeline Rendering: Interaction and Realism through Hardware-based Multi-pass Rendering*, Ph.D. dissertation, University of Pennsylvania, 1996. ftp://ftp.cis.upenn.edu/pub/ircs/technical-reports/96-25.ps.Z

[6] Tom McReynolds, David Blythe, and cohorts, “Reflections and Refractions,” *Programming with OpenGL: Advanced Rendering*, SIGGRAPH course notes, pp. 85-93, 1997.

[7] Eyal Ofek and Ari Rappoport, “Interactive Reflections on Curved Surfaces,” *SIGGRAPH ‘98 Proceedings*, pp. 333-342, 1998.

[8] Thant Tessman, “Casting Shadows on Flat Surfaces,” *IRIS Universe*, winter: 16, 1989.

[9] Jim Blinn, “Me and My (Fake) Shadow,” *IEEE Computer Graphics and Applications*, January 1988. Reprinted in *Jim Blinn’s Corner: A Trip Down the Graphics Pipeline*, 1996.

[10] Frank Crow, “Shadow Algorithms for Computer Graphics,” *Computer Graphics*, 11(2), pp. 442-448, summer 1977.

[11] Phillippe Bergeron, “A General Version of Crow’s Shadow Volumes,” *IEEE Computer Graphics and Applications*, September 1986.

[12] Tim Heidmann, “Real Shadows in Real Time,” *IRIS Universe*, (18) pp. 28-31, 1991.

[13] Tom McReynolds, David Blythe, and cohorts, “Creating Shadows,” *Programming with OpenGL: Advanced Rendering*, SIGGRAPH course notes, pp. 102-111, 1997.

[14] Mark J. Kilgard, “OpenGL-based Real-time Shadows,” web page, http://reality.sgi.com/opengl/tips/rts/

[15] Lance Williams, “Casting Curved Shadows on Curved Surfaces,” *SIGGRAPH ’78 Proceedings*, 12(3), pp. 270-274, 1978. Reprinted in *Tutorial: Computer Graphics: Image Synthesis, Computer Society Press*, 1988,

[16] William Reeves, David Salesin, and Robert Cook, “Rendering Antialiased Shadows with Depth Maps,” *SIGGRAPH Proceedings ’87*, 21(4), pp. 283-291, 1987.

[17] Tom McReynolds, David Blythe, and cohorts, “Shadow Maps,” *Programming with OpenGL: Advanced Rendering*, SIGGRAPH course notes, pp. 108-110, 1997.

[18] Michael Penna and Richard Patterson, “Reconstruction,” *Projective Geometry and Its Applications to Computer Graphics*, Prentice-Hall, 1986.

[19] Lynne Brotman and Norman Badler, “Generating Soft Shadows with a Depth Buffer Algorithm," *IEEE Computer Graphics and Applications*, October 1984.

[20] Paul Haeberli and Kurt Akeley, “The Accumulation Buffer: Hardware Support for High-quality Rendering,” *SIGGRAPH ’90 Proceedings*, pp. 309-318, August 1990.

[21] Mark J. Kilgard, “Rendering a Magic Halo with OpenGL,” web page, http://reality.sgi.com/opengl/tips/StenciledHaloEffect.html

[22] T.F. Wiegand, “Interactive Rendering of CSG Models,” Computer Graphics Forum, Vol. 15, No. 4, pp. 249-261, October 1996.

[23] Nigel Stewart, Geoff Leach, and Sabu John, “An Improved Z-Buffer CSG Rendering Algorithm,” Eurographics/SIGGRAPH Workshop on Graphics Hardware ’98, pp. 25-30, 25-30. http://www.eisa.net.au/~nigels/Research/egsggh98.pdf

[24] Nigel Stewart, Geoff Leach, and Sabu John, “A Single Z-Buffer CSG Rendering Algorithm for Convex Objects,” submitted to SIGGRAPH ’99, 1999. http://www.eisa.net.au/~nigels/Research/siggraph99.pdf

[25] James Bowman, “Visualize overdraw via stencil in OpenGL,” web page, http://reality.sgi.com/jamesb/article2/dinodraw.html

[26] Andrew Woo, Pierre Poulin, and Alain Fournier, “A Survey of Shadow Algorithms,” *IEEE Computer Graphics and Applications*, November 1990.

[27] Ronald Goldman, “Matrices and Transformation,” *Graphics Gems*, p. 474, 1990.