Volume Shadows Tutorial

Nuclear / the Lab

Introduction

As you probably know the most popular rendering technique, when speed is more important than quality (i.e. realtime rendering), is polygon rasterization. The problem with that technique is that it has nothing to do with the way that we actually perceive our environment in real life, so some things that we take as granted in nature (like shadows) are non-trivial to achieve. In contrast, in order to make a raytracer produce shadows is totally trivial and can be done by the very same processes that we use to render the picture. However, in polygon rasterization that is not the case, and polygon rasterization is what we are interested in when speed is important. So I am going to describe a popular algorithm of rendering shadows with polygon rasterization that is called shadow volumes (also known as stencil shadows).

Prerequisites

Rendering shadows is considered an advanced topic and thus requires some extend of knowledge and familiarity in computer graphics programming. So I will assume that you have good knowledge of linear algebra (coordinate systems, transformations and vector operations), that you have implemented a 3D visualization system that allows you to transform, light and draw polyhedral objects, and of course it goes without saying that you should be proficient with the programming language that you are going to use.

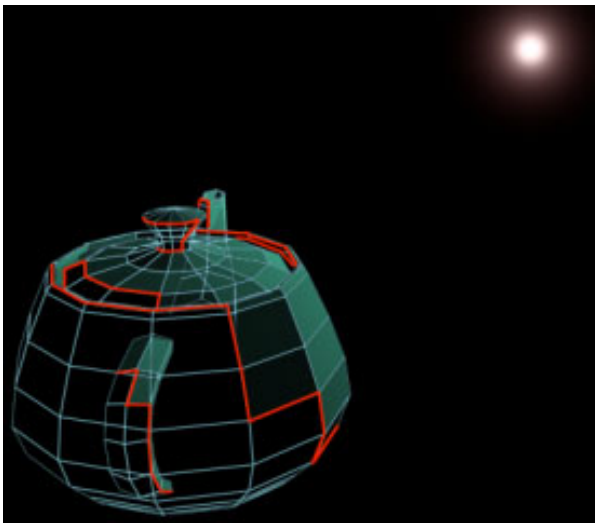
Conventions

The algorithm is really independent of graphics APIs and rendering systems, and I will present it as such, however there are some requirements from the rendering architecture:

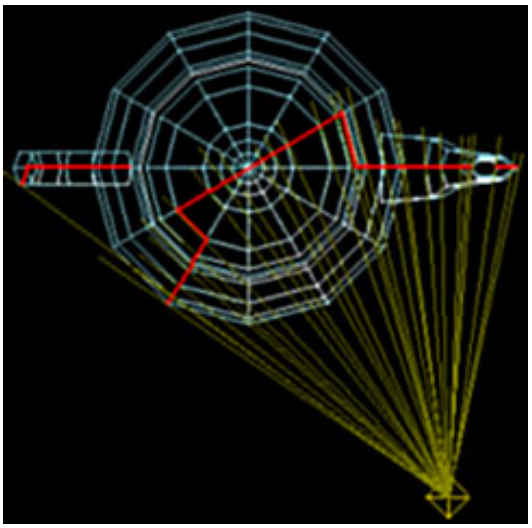
1. the rasterizer should be watertight, that means that if we draw two triangles with a shared edge (two shared vertices) there should be no pixels along the edge that are drawn twice (overlap) or not drawn at all (gaps) or the algorithm will break.
2. there must be a stencil buffer available. A stencil buffer is a buffer of integers as many as there are pixels in our framebuffer. When we draw a pixel we must be able to decide whether to draw it or not depending on the corresponding value of the stencil buffer for that pixel, and also we should be able to either specify that the stencil buffer be incremented, decremented or not affected by placing a pixel in the framebuffer. Such an interface is presented by both OpenGL and Direct3D and it is also very easy to implement in software.
3. A Depth Buffer must be available. Of course both Direct3D and OpenGL use a depth buffer (z-buffer usually) as the main Hidden Surface Removal method so this is not much of a requirement, and this kind of functionality can be implemented extremely easily in any software rendering system that can do any kind of interpolation inside the polygons it draws (gouraud, texture mapping etc.)

The Algorithm - Part 1: Shadow Volume Computation

The actual algorithm to render shadows has two distinct parts. The first part deals with the computations involved in creating what we call a shadow volume. Note that we need as many shadow volumes per object as there are lights in the scene that cast shadows, but I will describe the process as if there is only one light for simplicity, it is then trivial to add more, just keep a list of shadow volumes per object and calculate them all in turn.
The idea is to create a polygon mesh at the boundaries of the shadow that the object casts, to do that we need to find the contour edges of the object as seen from the light source and extrude them away from the light. But let's first define what we mean by contour edges. All edges must have exactly 2 adjacent polygons, a contour edge has one adjacent polygon facing towards the light and one adjacent polygon facing away from the light (see diagram 1 & 2)



diag.1: contour edges



diag.2: contour edges in 2d

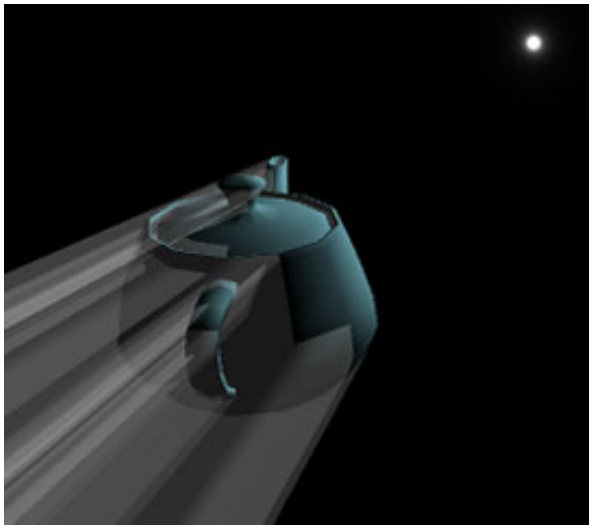
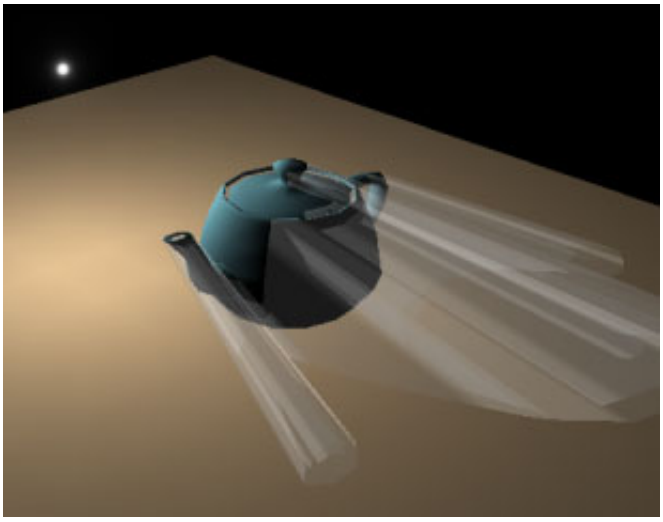
There are some subtle issues involved in finding the contour edges, we are going to use these edges for the shadow volume polygons, so we must take care to have their vertices defined in the correct order. Also we want to find them as quickly as possible. Here is an algorithm to do that in pseudocode.

```
// transform the light to the coordinate system of the object
LightPosition = LightPosition * Inverse(ObjectWorldMatrix);

for (every polygon) {
      IncidentLightDir = AveragePolyPosition - LightPosition;

      //if the polygon faces away from the light source....
      if (DotProduct(IncidentLightDir, PolygonNormal) >= 0.0) {
            for (every edge of the polygon) {
                  if (the edge is already in the contour edge list) {
                        // then it can't be a contour edge since it is
                        // referenced by two triangles that are facing
                        // away from the light
                        remove the existing edge from the contour list;
                  } else {
                        add the edge to the contour list;
                  }
            }
      }
}
```

Then when we have the list of contour edges, we must extrude them away from the light in order to create the boundary of the shadow volume. The extrusion is very simple, just create a quad (or two triangles if you use triangles as your polygons) for each of the edges in the list, the first two vertices of the quad being the two vertices of the edge and the other two the same vertices offset by their position * IncidentLightDir at each vertex (see pseudocode).

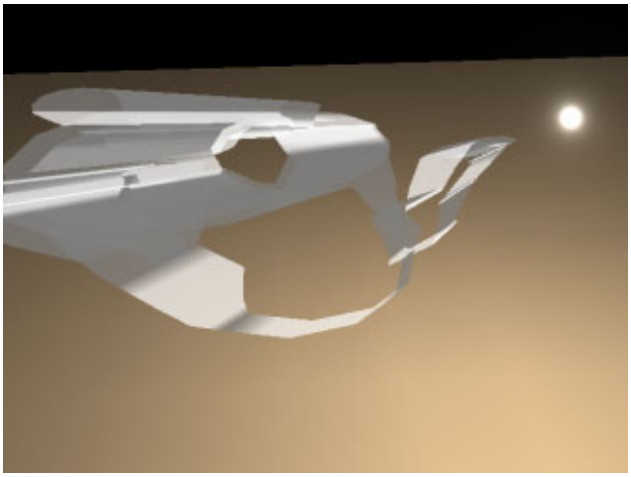
```
ExtrudeMagnitude = A_BIG_NUMBER;

for (every edge) {
    ShadowQuad[i].vertex[0] = edge[i].vertex[0];
    ShadowQuad[i].vertex[1] = edge[i].vertex[1];
    ShadowQuad[i].vertex[2] = edge[i].vertex[1] + ExtrudeMagnitude *
(edge[i].vertex[1] - LightPosition);
    ShadowQuad[i].vertex[3] = edge[i].vertex[0] + ExtrudeMagnitude *
(edge[i].vertex[0] - LightPosition);
}
```

Do not forget that for all these computations the light position must be transformed into the object's local coordinate system by the inverse of the object's local->world transformation. (if you are defining the lights in any other coordinate system than world coordinates substitute "local->world" above with "local->light_coordinate_system").

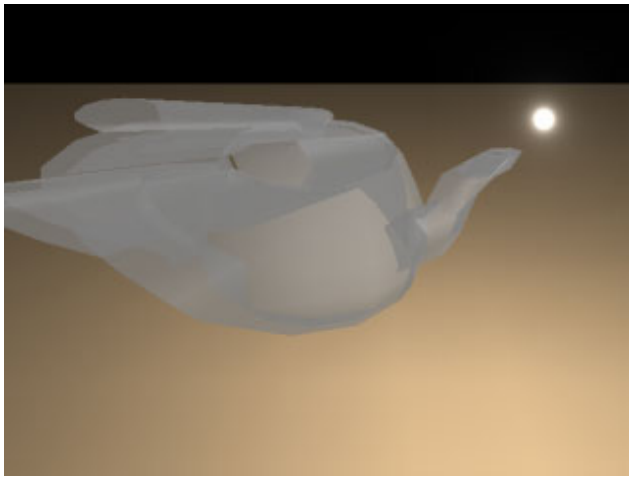



diag.3&4: extruded shadow volume

Another important issue is that the shadow volume is incomplete as it is now because it has a big hole where the object is actually located (see diagram 5), so we must cap that hole with polygons, that is easier than it sounds however as we can use the polygons of the object itself as a cap for the shadow volume (see diagram 6).



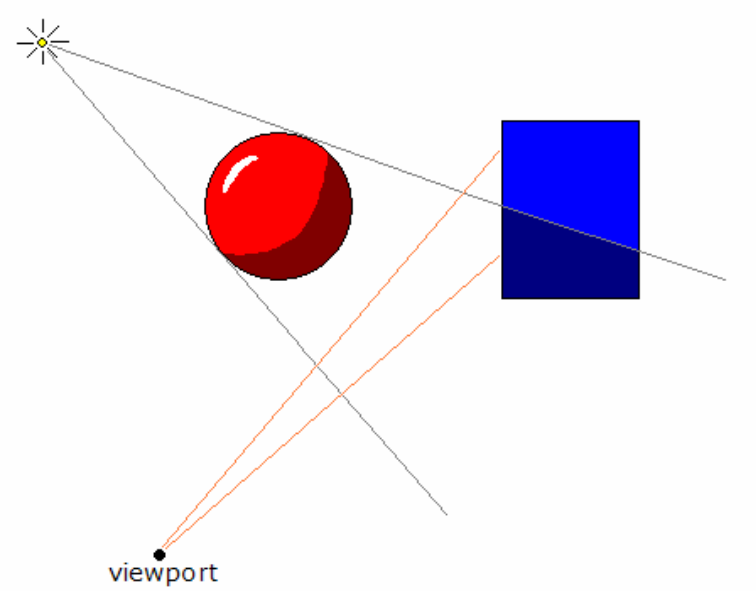
diag.5: uncapped shadow volume



diag.6: properly capped shadow volume

The Algorithm - Part 2: Rendering Shadows

Now that we have created the polygon meshes bounding our shadow volumes we must actually draw the shadows in the scene, or more precisely, draw our scene with shadows. The algorithm is simple once you understand it, we must be able to determine whether a pixel is in the space bounded by the shadow volumes or not. Let's imagine a ray originating from the pixel that we are interested in and into the scene, if assume there is only one shadow volume for simplicity, in order for that ray to hit a shadowed part of the scene it must pass once from the front faces of the shadow volume and not pass through the other side, while if it passes through both sides of the shadow volume and comes right through to the other side it is eventually going to hit an unshadowed part (see diagram 7).



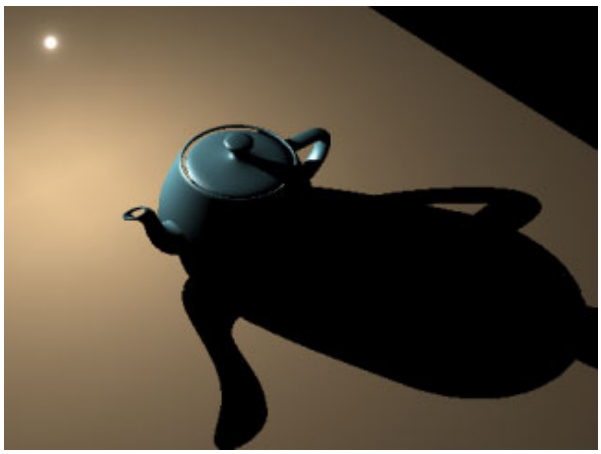
diag.7: crossing shadow volume boundaries

In generalizing the above for more than one shadow volumes, if the ray passes same number of front sides as back sides of shadow volumes then it hits an unshadowed spot. If the number of front faces that it passes through is greater than the number of back faces then it means that it hits a shadowed spot.
The way to keep track of how many front and back faces the ray hits is of course by using a counter that we increment when it passes through a front face and decrement when it passes through a back face. Then when the counter is zero it means that we are in unshadowed part, while when it is greater than zero we are in the shadows.

That is exactly where the stencil buffer comes in handy. As I said before the stencil buffer actually keeps a "counter" for every pixel on screen that we can increment and decrement when a pixel is written to the frame buffer, and then we can check against that "counter" to determine whether a pixel should be written or not. That is exactly the functionality that we need to implement the above algorithm. The procedure goes like this:

- clear the zbuffer and stencil buffer, also make sure zbuffering is enabled and stencil testing is disabled.
- Render the scene only with ambient light and any lights other than the shadow caster on. The important part here is that the zbuffer is updated so that a shadow volume polygon later will not be rendered if it is behind geometry.
- Disable writing to the zbuffer.
- Draw the front faces of the shadow volume, if they are actually drawn (pass depth test) increment the stencil buffer.
- Draw the back faces of the shadow volume, if they are actually drawn (pass depth test) decrement the stencil buffer.
- Enable stencil testing, clear zbuffer, enable writing to the zbuffer, enable all lights.
- Render scene where the stencil is zero (unshadowed).

That's it, think a bit about the whole procedure and it will make sense, we first "burn" the zbuffer with our scene, and in the color buffer we have an image of the scene that is not lit by the shadow caster (as if the whole scene is in shadow). We render the front faces of the shadow volume if they are visible, increasing the stencil at the pixels that we write. Then we render the back faces decreasing the stencil as we go. So far the effect is that if an object is inside the shadow volume, the stencil at the pixels corresponding to that object were increased by the front faces (as being inside the shadow implies being behind the front of the shadow boundary), but was not decreased by the back face pass later, since the back faces of the shadow volume are behind the object and not drawn. So now we have a stencil buffer that is 0 outside of the shadows, and greater than 0 inside, thus by rendering the scene once again fully lit only where it is not in shadow (stencil = 0), we have the final image in the framebuffer with our lit scene except where the shadows are, where the unlit first pass was not overwritten by the last pass.



Some additional issues

There is only one problem that has to be resolved in order of the shadows to be totally generic and robust. The problem is that when the viewpoint enters the shadow volume, the front faces of the shadow volume are not rendered since they are behind the viewpoint. So only the back faces are rendered, decrementing the stencil everywhere except inside the shadow volume. The effect of that is that the areas that should be shadowed are left unshadowed and the areas that should be in light are shadowed!
Alternatively, if we are making sure that the stencil does not go below zero (and wrap around actually as it is unsigned) at any point then there is no inversion, but all the shadows disappear (OpenGL does that by default, Direct3D needs D3DSTENCILOP_DECSAT to be specified).
To resolve that issue we can either clip the shadow volume to the view plane (thus making another cap if the camera is inside the volume) or test to see if the view point is inside the volume defined by the shadow polygons and if it is then at the beginning instead of clearing the stencil buffer to 0, clear it to 1.

Pros and Cons of the shadow volume algorithm

The good things about this algorithm are that it is totally generic and applicable everywhere, for as many lights and objects as we need, it produces self shadowing and precise shadow shapes in all kinds of geometry.
The bad things are that it is very CPU intensive, especially as polygon counts of the shadow casting objects get high, and it produces only "hard shadows".
A good trick to reduce shadow volume calculations is to have two versions of the shadow casting objects, one detailed that gets rendered, and a low-polygon-count one that is used to calculate the shadows. Also another good trick is to precalculate all static shadow volumes. that is, shadows generated by lights and objects that do not move.

That's about it, I consider shadows an important part of a computer generated image that can imbue an amount of realism that is missing without them. So I hope that I explained the algorithm good enough, and that I helped you bring some more realism to your computer graphics programs. For any questions, corrections, ideas or in fact if you just want to say hello and how you liked/disliked this tutorial, drop me a mail at nuclear@siggraph.org, I would be happy to receive any feedback. See you around.

John Tsiombikas
Nuclear / the Lab
nuclear@siggraph.org