Shadows in the graphics pipeline

Steve Marschner
Cornell University
CS 569 Spring 2008, 19 February

There are a number of visual cues that help let the viewer know about the 3D relationships between objects in the scene. The basic z-buffer pipeline gives you perspective and occlusion cues, and we've discussed lots of ways to compute shading to indicate surface shape. But one very important cue is shadows.

The visual system expects to see shadows where objects are near one another, and it uses shadows, among other things, to disambiguate different interpretations of a scene. For instance, in the slides you can see a classic optical illusion in which the balls can be made to float in the air or sit on the checkerboard just by moving the shadows. A second example shows how shadows help integrate a pasted-in object into a scene: the beach balls are ambiguous with no shadows; are they floating in the air or sitting on the ground? A simple contact shadow makes the balls stick to the ground—in fact, the presence or absence of shadows causes the upper ball to appear to change size. If there's no shadow holding it down it could be floating at the same distance as the other one, but with a shadow it must be farther away, and therefore larger.

Causes of shadows

Before I start talking about computing approximate shadows let's briefly discuss the right answer. As we know from writing simple ray tracers, a point is in shadow from a point light source if there is some object interrupting the line segment between the shading point and the source. With point light sources, shadows are an all-or-nothing affair: either the source is visible or it's occluded (in the absence of partially transparent objects, anyway). The first book photo in the slides is lit primarily by the sun, which is close to a point source, so it has a *hard shadow*.

For the second book photo I moved to a north window, where illumination comes from the whole sky, so the whole window acts as a light source. In this case, the book casts a *soft shadow*, in which the transition from illuminated to shadowed is gradual rather than abrupt. The reason for this is that with an area source visibility is no longer all-or-nothing; part of the source can be visible while the rest of it is occluded. If we think of an ant walking across the floor, as he enters the frame he can see the whole window, so he is fully illuminated; when he gets closer to the book, the book blocks part of the window and we say he's in the *penumbra*; when he is standing right behind the book he can't see the window at all and we say he's in the *umbra* of the shadow.

An important feature of soft shadows is *hardening on contact*. Look near where the book touches the floor; at that point the transition into the shadow is abrupt, because the edge of the book is so close by. Soft shadows always narrow to a hard edge as you approach a contact point, and the visual system uses this as a cue to know when objects are in contact.

The methods I'll discuss today are all for hard shadows, cast by point lights (or directional lights). But keep in mind that soft shadows are often desirable, and therefore artifacts that produce blurring of shadows may be just fine.

Fake shadows

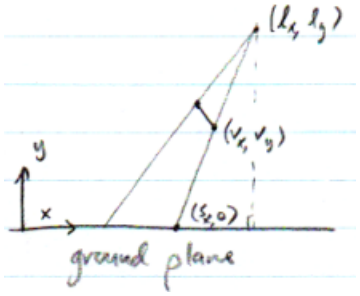
One method of rendering shadows bears mentioning before we discuss the "real" methods. In some applications the scene is constrained enough that you can identify *a priori* where the important shadows will be. Then you can just go ahead and draw them. For example, a car racing game that draws nothing but cars that are sitting on a flat road doesn't need a lot of fancy, general shadowing computations; it could just position a pre-drawn shadow texture underneath each car, scaled to match the car's dimensions. This is very easy and will do a fine job of anchoring the cars to the road.

Projection shadows

One special case that occurs frequently is an object (of arbitrary shape) casting a shadow on a plane (usually the ground). Shadows cast from a point light onto a planar receiver are particularly easy to deal with, and they introduce some of the ideas that we'll use for the general case.

The shadow that is cast on a plane is simply the projection of the volume occupied by the object onto the plane. The projection is simple to compute, so we can just draw it on the plane by rendering dark-colored polygons, and we're done.

Let's look at the projection of a single triangle. In 2D it looks like this:



and from similar triangles we can get:

$$\frac{s_x - l_x}{v_x - l_x} = \frac{l_y}{l_y - v_y}$$

From the same argument in the y-z plane:

$$\frac{s_z - l_z}{v_z - l_z} = \frac{l_y}{l_y - v_y}$$

We can formalize this using a matrix, with homogeneous coordinates doing the division for us:

$$s_x = \frac{v_x l_y - l_x v_y}{l_y - v_y}, \quad s_z = \frac{v_z l_y - l_z v_y}{l_y - v_y}$$

$$\begin{bmatrix} s_x \\ s_y \\ s_z \\ 1 \end{bmatrix} \propto \begin{bmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

This is great—we just compute this matrix up front (it only depends on the source and receiver geometry), and then whenever we draw an object we draw it a second time with this matrix applied to it, in black.

Will this just work? No—the depth values of these polygons are not in front of the ground; they are the same so we will likely see *z*-buffer precision artifacts. Think of some ways to fix this.

One nifty tool for this is to use the glPolygonOffset mechanism, which is intended for just these cases where you'd like to shift the polygons you're drawing just enough to make sure they are on the right side for the *z* compare. One could also draw the ground, then the shadows, then the objects, with no *z* compare for the shadows.

How could we do this with a texture? Set up a view that sees the surface head-on and render the triangles into that view, producing a texture that's black inside the shadow and white everywhere else. Then you can use that texture as an input to your shading calculations (which provides more convenience to do whatever you want in shadowed vs. unshadowed regions). It's also nice because this *shadow texture* keeps as long as the geometry stays fixed.

There are some problems, though. Only objects between the source and the receiver are supposed to cast shadows. What are some cases where things will be in shadow that should not be?

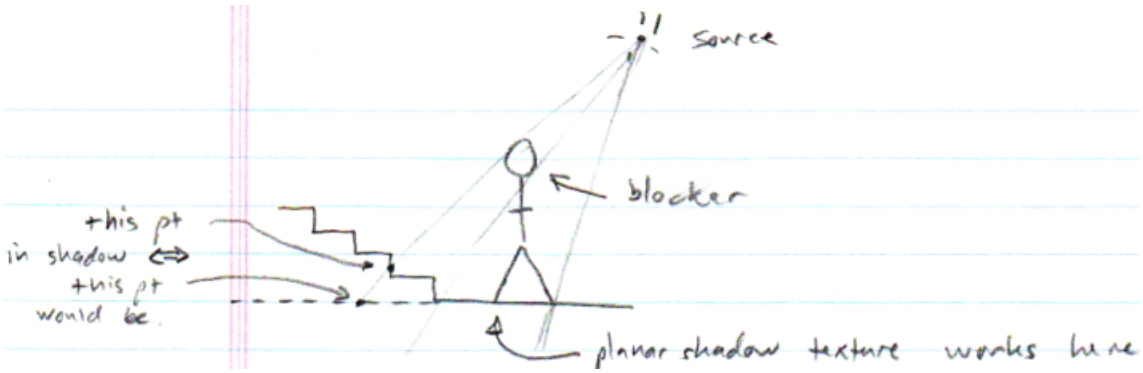
Think about what the transformation matrix looks like when we draw such a texture. We have a modeling transform followed by a projection onto the plane followed by the viewing projection that maps that plane to the texture image. This first projection is throwing away distance information, but just like in *z*-buffering we can produce a pseudo-depth that can tell us what is in front of or behind the source. By simply putting the light source height in the y component, in our ground-plane example, we can get a third coordinate that separates false shadows from real ones:

$$\begin{bmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & l_y \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{bmatrix} \rightsquigarrow s_y = \frac{l_y}{l_y - v_y}$$

Under this projection, *s_y* is greater than 1 when *v_y* is between 0 and *l_y*. For points above the light source it wraps around to negative numbers, and for points below the ground it takes on values less than 1. Thus if we use this projection, then set up an orthographic view with near at 1 and far at infinity, we will automatically clip out what we don't want and get a correct shadow.

Shadow textures

This shadow projection method works fine but it only works for planar receivers, and of course your receiver may not be planar. Suppose we want to cast shadows on terrain, or on the floor of a game level that includes stairs, etc. It turns out to be easy to do this as well, using a simple generalization of the shadow texture we computed for the plane.



This picture shows the relationship between shadows cast on a non-planar environment and shadows that *would be cast* on the ground plane in the absence of the environment. If we are shading a pixel on the stairs, how could we check whether it's in shadow?

The shadow texture for the floor is all we need—we just use the same projection we used to render the shadow texture in the first place, but this time we use it to compute texture coordinates instead of moving the vertices. Using a projective transformation to generate texture coordinates is called *projective textures*. A projective shadow texture can render shadows from any shape blocker on any shape receiver. (Note that game programmers sometimes call these "shadow maps" but I'll reserve that term for the next method.)

Note that this shadow texture is really just an silhouette image rendered from the point of view of the light source. The product of the shadow projection and the orthographic projection matrix is quite similar to a perspective projection for a camera at the location of the light source. The

Are we done? What part of the planar texture method breaks now? Why have I been calling the receiver an "environment" instead of a generic object?

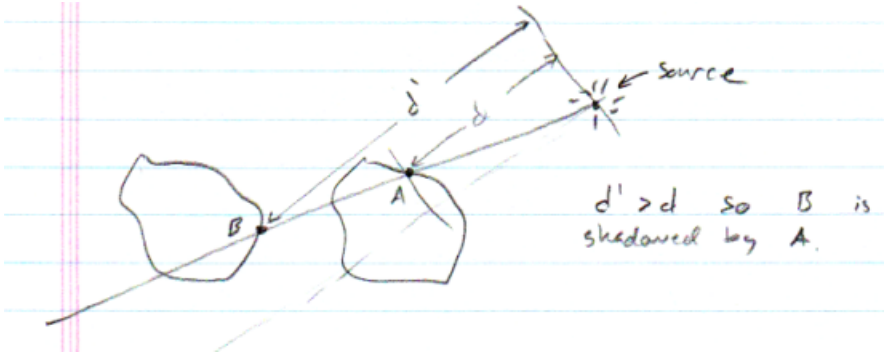
Shadow maps

The problem is that we have assumed we know what is the receiver and what is the blocker. If I try to use this method to render shadows for all objects in the scene, what will go wrong?

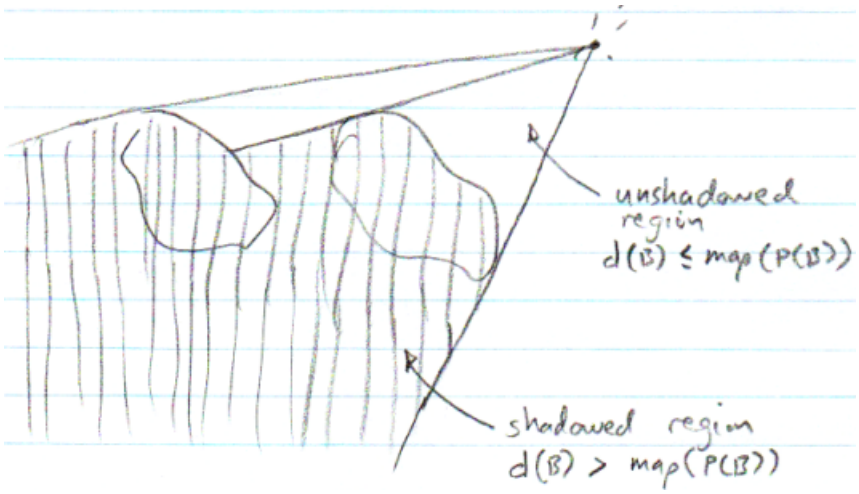
Among other things: objects between the light and the blocker will be shadowed; objects will shadow themselves; in fact, if I try to let all objects be blockers and receivers, everything will always be in shadow.

The projective texture coordinates serve to identify which objects are on the same ray from the light source. What we need is a way to sort out whether the object we're shading is closer to or farther from the source along that ray.

A shadow map does just this. It's a very simple idea: rather than storing a binary image of which rays are blocked, instead store the *distance at which the ray is blocked* in the texture map. Then, we do exactly the same lookup except, rather than just reading a binary "am I shadowed" answer from the map we instead read the distance at which the ray is blocked and see whether that is closer to the source than we are. If so, we are shadowed.



I think a good way to think about a shadow map is that it's just a clever numerical representation of the volume of space that is illuminated by the source:



For the geometers among you: the source illuminates a star-shaped region, which can be described by recording the distance to the boundary of the region along every direction from the center. Then you can test for inclusion in that region by looking up the distance value for your ray and comparing it to the distance of the point you're asking about.

One problem that immediately pops up is that less-than-or-equal comparison. For illuminated points on shadow-casting objects, it's equal, but of course it's stored with finite precision in the map. For shadow maps to work successfully you need to add a *shadow bias* that is greater than the numerical uncertainty in the depth values.

Other practical issues include choosing the size of the texture and the near and far distances. This needs to be done such that the "view frustum" of the shadow map camera encompasses all the receivers you want to look up shadows for.

Shadow volumes

Finally, there is one completely different way to compute shadows. I won't discuss it in much detail because we're not implementing it, but you should know of its existence. The idea is to explicitly construct the boundaries of the volumes that are shadowed by the primitives in your scene. For a triangle, this volume is a frustum of a trigonal pyramid. The idea is to draw these surfaces into a stencil buffer, which is a buffer that can keep track of an integer count and later be used to mask off part of the image, in such a way that a positive number will be left in that buffer exactly where there is a shadow. The shadow volume boundaries are drawn, after drawing the scene to establish the z buffer, with the z test enabled so that the stencil buffer is only updated at pixels where the shadow volume boundary is closer than the closest surface. In this way, The clever part is that front-facing shadow volume boundaries (where the ray is entering a shadow volume) are drawn with an increment of +1, and back-facing ones (where the ray is exiting a shadow volume) are drawn with an increment of –1, so that each stencil buffer pixel keeps a count of the number of shadow volumes boundaries that contain cross the visible point at that pixel (which is equal to the number of volumes the ray entered on the way from the viewpoint to the surface minus the number of volumes it exited). After all the volumes are drawn, the stencil buffer can be used to add shading only to the unshadowed pixels.