## 17.1 Reflections

Reflections are one of the most noticeable effects of interobject lighting. Getting it right can add a lot of realism to a scene for a moderate effort. It also provides very strong visual clues about the relative positioning of objects. Here, reflection is divided into two categories: highly specular "mirror-like" reflections and "radiosity-like" interobject lighting based on diffuse reflections.

Directly calculating the physics of reflection using algorithms such as ray tracing can be expensive. As the physics becomes more accurate, the computational overhead increases dramatically with scene complexity. The techniques described here help an application budget its resources, by attempting to capture the most significant reflection effects in ways that minimize overhead. They maintain good performance by approximating more expensive approaches, such as ray tracing, using less expensive methods.

### 17.1.1 Object vs. Image Techniques

Consider a reflection as a view of a "virtual" object. As shown in Figure 17.1, a scene is composed of reflected objects rendered "behind" their reflectors, the same objects drawn in their unreflected positions, and the reflectors themselves. Drawing a reflection becomes a two-step process: using objects in the scene to create virtual reflected versions and drawing the virtual objects clipped by their reflectors.

There are two ways to implement this concept: image-space methods using textures and object-space approaches that manipulate geometry. Texture methods create a texture image from a view of the reflected objects, and then apply it to a reflecting surface. An advantage of this approach, being image-based, is that it doesn't depend on the geometric representation of the objects being reflected. Object-space methods, by contrast, often must distort an object to model curved reflectors, and the realism of their reflections depends on the accuracy of the surface model. Texture methods have the most built-in OpenGL support. In addition to basic texture mapping, texture matrices, and

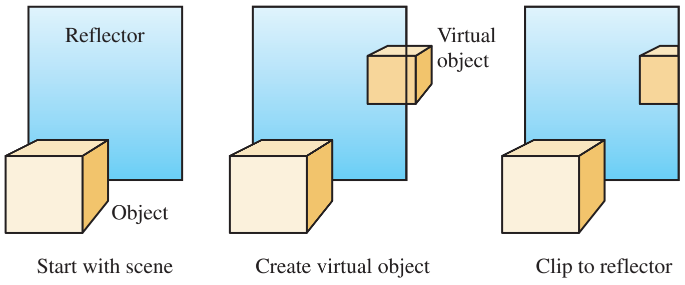

Figure 17.1 Mirror reflection of the scene.

texgen functionality, environment texturing support makes rendering the reflections from arbitrary surfaces relatively straightforward.

Object-space methods, in contrast, require much more work from the application. Reflected "virtual" objects are created by calculating a "virtual" vertex for every vertex in the original object, using the relationship between the object, reflecting surface, and viewer. Although more difficult to implement, this approach has some significant advantages. Being an object-space technique, its performance is insensitive to image resolution, and there are fewer sampling issues to consider. An object-space approach can also produce more accurate reflections. Environment mapping, used in most texturing approaches, is an approximation. It has the greatest accuracy showing reflected objects that are far from the reflector. Object-space techniques can more accurately model reflections of nearby objects. Whether these accuracy differences are significant, or even noticeable, depends on the details of the depicted scene and the requirements of the application.

Object-space, image-space, and some hybrid approaches are discussed in this chapter. The emphasis is on object-space techniques, however, since most image-space techniques can be directly implemented using OpenGL's texturing functionality. Much of that functionality is covered in Sections 5.4 and 17.3.

## Virtual Objects

Whether a reflection technique is classified as an object-space or image-space approach, and whether the reflector is planar or not, one thing is constant: a virtual object must be created, and it must be clipped against the reflector. Before analyzing various reflection techniques, the next two sections provide some general information about creating and clipping virtual objects.

**Clipping Virtual Objects** Proper reflection clipping involves two steps: clipping any reflected geometry that lies outside the edges of the reflected object (from the viewer's point of view) and clipping objects that extend both in front of and behind the reflector (or penetrate it) to the reflector's surface. These two types of clipping are shown in Figure 17.2. Clipping to a planar reflector is the most straightforward. Although the

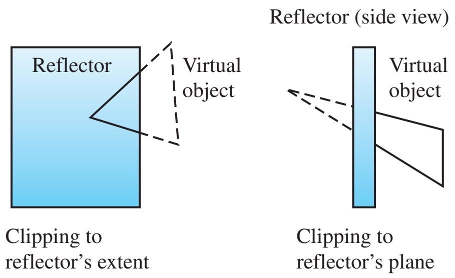

**Figure 17.2** Clipping virtual objects to reflectors.

application is different, standard clipping techniques can often be reused. For example, user-defined clip planes can be used to clip to the reflector's edges or surface when the reflection region is sufficiently regular or to reduce the area that needs to be clipped by other methods.

While clipping to a reflecting surface is trivial for planar reflectors, it can become quite challenging when the reflector has a complex shape, and a more powerful clipping technique may be called for. One approach, useful for some applications, is to handle reflection clipping through careful object modeling. Geometry is created that only contains the parts visible in the reflector. In this case, no clipping is necessary. While efficient, this approach can only be used in special circumstances, where the view position (but not necessarily the view direction), reflector, and reflected geometry maintain a static relationship.

There are also image-space approaches to clipping. Stencil buffering can be useful for clipping complex reflectors, since it can be used to clip per-pixel to an arbitrary reflection region. Rather than discarding pixels, a texture map of the reflected image can be constructed from the reflected geometry and applied to the reflecting object's surface. The reflector geometry itself then clips an image of the virtual object to the reflector's edges. An appropriate depth buffer function can also be used to remove reflecting geometry that extends behind the reflector. Note that including stencil, depth, and texture clipping techniques to object-space reflection creates hybrid object/image space approaches, and thus brings back pixel sampling and image resolution issues.

**Issues When Rendering Virtual Objects** Rendering a virtual object properly has a surprising number of difficulties to overcome. While transforming the vertices of the source object to create the virtual one is conceptually straightforward when the reflector is planar, a reflection across a nonplanar reflector can distort the geometry of the virtual object. In this case, the original tessellation of the object may no longer be sufficient to model it accurately. If the curvature of a surface increases, that region of the object may require retessellation into smaller triangles. A general solution to this problem is difficult to construct without resorting to higher-order surface representations.

Even after finding the proper reflected vertices for the virtual object, finding the connectivity between them to form polygons can be difficult. Connectivity between vertices can be complicated by the effects of clipping the virtual object against the reflector. Clipping can remove vertices and add new ones, and it can be tedious to handle all corner cases, especially when reflectors have complex or nonplanar shapes.

More issues can arise after creating the proper geometry for the virtual object. To start, note that reflecting an object to create a virtual one reverses the vertex ordering of an object's faces, so the proper face-culling state for reflected objects is the opposite of the original's. Since a virtual object is also in a different position compared to the source object, care must be taken to light the virtual objects properly. In general, the light sources for the reflected objects should be reflected too. The difference in lighting may not be noticeable under diffuse lighting, but changes in specular highlights can be quite obvious.

### 17.1.2 Planar Reflectors

Modeling reflections across planar or nearly planar surfaces is a common occurrence. Many synthetic objects are shiny and flat, and a number of natural surfaces, such as the surface of water and ice, can often be approximated using planar reflectors. In addition to being useful techniques in themselves, planar reflection methods are also important building blocks for creating techniques to handle reflections across nonplanar and nonuniform surfaces.

Consider a model of a room with a flat mirror on one wall. To reflect objects in this planar reflector, its orientation and position must be established. This can be done by computing the equation of the plane that contains the mirror. Mirror reflections, being specular, depend on the position of both the reflecting surface and the viewer. For planar reflectors, however, reflecting the geometry is a viewer-independent operation, since it depends only on the relative position of the geometry and the reflecting surface. To draw the reflected geometry, a transform must be computed that reflects geometry across the mirror's plane. This transform can be conceptualized as reflecting either the eye point or the objects across the plane. Either representation can be used; both produce identical results.

An arbitrary reflection transformation can be decomposed into a translation of the mirror plane to the origin, a rotation embedding the mirror into a major plane (for example the $x - y$ plane), a scale of $-1$ along the axis perpendicular to that plane (in this case the $z$ axis), the inverse of the rotation previously used, and a translation back to the mirror location.

Given a vertex $P$ on the planar reflector's surface and a vector $\mathbf{V}$ perpendicular to the plane, the reflection transformations sequence can be expressed as the following single $4 \times 4$ matrix $R$ (Goldman, 1990):

$$R = \begin{pmatrix} 1 - 2V_x^2 & -2V_xV_y & -2V_xV_z & 2(P \cdot V)V_x \\ -2V_xV_y & 1 - 2V_y^2 & -2V_yV_z & 2(P \cdot V)V_y \\ -2V_xV_z & -2V_yV_z & 1 - 2V_z^2 & 2(P \cdot V)V_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Applying this transformation to the original scene geometry produces a virtual scene on the opposite side of the reflector. The entire scene is duplicated, simulating a reflector of infinite extent. The following section goes into detail on how to render and clip virtual geometry against planar reflectors to produce the effect of a finite reflector.

#### Clipping Planar Reflections

Reflected geometry must be clipped to ensure it is only visible in the reflecting surface. To do this properly, the reflected geometry that appears beyond the boundaries of the reflector from the viewer's perspective must be clipped, as well as the reflected geometry that ends up in front of the reflector. The latter case is the easiest to handle. Since the reflector is planar, a single application-defined clipping plane can be made coplanar to the reflecting surface, oriented to clip out reflected geometry that ends up closer to the viewer.

If the reflector is polygonal, with few edges, it can be clipped with the remaining application clip planes. For each edge of the reflector, calculate the plane that is formed by that edge and the eye point. Configure this plane as a clip plane (without applying the reflection transformation). Be sure to save a clip plane for the reflector surface, as mentioned previously. Using clip planes for reflection clipping is the highest-performance approach for many OpenGL implementations. Even if the reflector has a complex shape, clip planes may be useful as a performance-enhancing technique, removing much of the reflected geometry before applying a more general technique such as stenciling.

In some circumstances, clipping can be done by the application. Some graphics support libraries support culling a geometry database to the current viewing frustum. Reflection clipping performance may be improved if the planar mirror reflector takes up only a small region of the screen: a reduced frustum that tightly bounds the screen-space projection of the reflector can be used when drawing the reflected scene, reducing the number of objects to be processed.

For reflectors with more complex edges, stencil masking is an excellent choice. There are a number of approaches available. One is to clear the stencil buffer, along with the rest of the framebuffer, and then render the reflector. Color and depth buffer updates are disabled, rendering is configured to update the stencil buffer to a specific value where a pixel would be written. Once this step is complete, the reflected geometry can be rendered, with the stencil buffer configured to reject updates on pixels that don't have the given stencil value set.

Another stenciling approach is to render the reflected geometry first, and then use the reflector to update the stencil buffer. Then the color and depth buffer can be cleared, using the stencil value to control pixel updates, as before. In this case, the stencil buffer controls what geometry is erased, rather than what is drawn. Although this method can't always be used (it doesn't work well if interreflections are required, for example), it may be the higher performance option for some implementations: drawing the entire scene with stencil testing enabled is likely to be slower than using stencil to control clearing the screen. The following outlines the second approach in more detail.

1. Clear the stencil and depth buffers.
2. Configure the stencil buffer such that 1 will be set at each pixel where polygons are rendered.
3. Disable drawing into the color buffers using `glColorMask`.
4. Draw the reflector using blending if desired.
5. Reconfigure the stencil test using `glStencilOp` and `glStencilFunc`.
6. Clear the color and depth buffer to the background color.
7. Disable the stencil test.
8. Draw the rest of the scene (everything but the reflector and reflected objects).

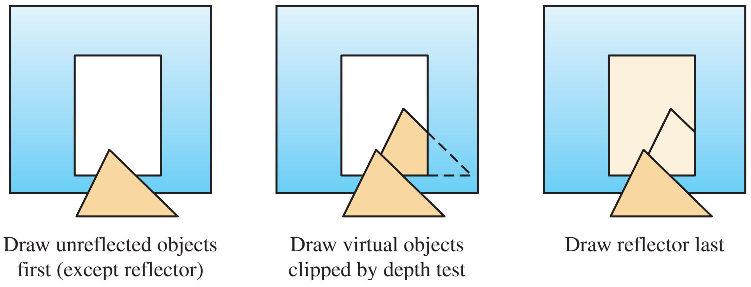
Draw unreflected objects first (except reflector)
Draw virtual objects clipped by depth test
Draw reflector last

**Figure 17.3** Masking reflections with depth buffering.

The previous example makes it clear that the order in which the reflected geometry, reflector, and unreflected geometry are drawn can create different performance trade-offs. An important element to consider when ordering geometry is the depth buffer. Proper object ordering can take advantage of depth buffering to clip some or all of the reflected geometry automatically. For example, a reflector surrounded by nonreflected geometry (such as a mirror hanging on a wall) will benefit from drawing the nonreflected geometry in the scene first, before drawing the reflected objects. The first rendering stage will initialize the depth buffer so that it can mask out reflected geometry that goes beyond the reflector's extent as it's drawn, as shown in Figure 17.3. Note that the figure shows how depth testing can clip against objects in front of the mirror as well as those surrounding it. The reflector itself should be rendered last when using this method; if it is, depth buffering will remove the *entire* reflection, since the reflected geometry will always be behind the reflector. Note that this technique will only clip the virtual object when there are unreflected objects surrounding it, such as a mirror hanging on a wall. If there are clear areas surrounding the reflector, other clipping techniques will be needed.

There is another case that can't be handled through object ordering and depth testing. Objects positioned so that all or part of their reflection is *in front* of the mirror (such as an object piercing the mirror surface) will not be automatically masked. This geometry can be eliminated with a clip plane embedded in the mirror plane. In cases where the geometry doesn't cross the mirror plane, it can be more efficient for the application to cull out the geometry that creates these reflections (i.e., geometry that appears behind the mirror from the viewer's perspective) before reflecting the scene.

Texture mapping can also be used to clip a reflected scene to a planar reflector. As with the previous examples, the scene geometry is transformed to create a reflected view. Next, the image of the reflected geometry is stored into a texture map (using `glCopyTexImage2D`, for example). The color and depth buffers are cleared. Finally, the entire scene is redrawn, unreflected, with the reflector geometry textured with the image of the reflected geometry. The process of texturing the reflector clips the image of the reflected scene to the reflector's boundaries. Note that any reflected geometry that ends up in front of the reflector still has to be clipped before the texture image is created.

The methods mentioned in the previous example, using culling or a clip plane, will work equally well here.

The difficult part of this technique is configuring OpenGL so that the reflected scene can be captured and then mapped properly onto the reflector's geometry. The problem can be restated in a different way. In order to preserve alignment, both the reflected and unreflected geometry are rendered from the same viewpoint. To get the proper results, the texture coordinates on the reflector only need to register the texture to the original captured view. This will happen if the $s$ and $t$ coordinates correlate to $x$ and $y$ window coordinates of the reflector.

Rather than computing the texture coordinates of the reflector directly, the mapping between pixels and texture coordinates can be established using `glTexGen` and the texture transform matrix. As the reflector is rendered, the correct texture coordinates are computed automatically at each vertex. Configuring texture coordinate generation to `GL_OBJECT_LINEAR`, and setting the $s$, $t$ and $r$ coordinates to match one to one with $x$, $y$, and $z$ in eye space, provides the proper input to the texture transform matrix. It can be loaded with a concatenation of the modelview and projection matrix used to "photograph" the scene. Since the modelview and projection transforms the map from object space to NDC space, a final scale-and-translate transform must be concatenated into the texture matrix to map $x$ and $y$ from $[-1, 1]$ to the $[0, 1]$ range of texture coordinates. Figure 17.4 illustrates this technique. There are three views. The left is the unreflected view with no mirror. The center shows a texture containing the reflected view, with a rectangle showing the portion that should be visible in the mirror. The rightmost view shows the unreflected scene with a mirror. The mirror is textured with the texture containing the reflected view. *Texgen* is used to apply the texture properly. The method of using *texgen* to match the transforms applied to vertex coordinates is described in more detail in Section 13.6.

The texture-mapping technique may be more efficient on some systems than stencil buffering, depending on their relative performance on the particular OpenGL implementation. The downside is that the technique ties up a texture unit. If rendering the reflector uses all available texture units, textured scenes will require the use of multiple passes.

Finally, separating the capture of the reflected scene and its application to the reflector makes it possible to render the image of the reflected scene at a lower resolution than the

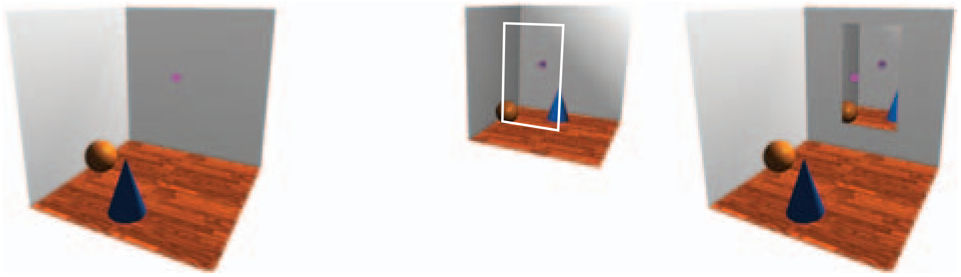

**Figure 17.4** Masking reflections using projective texture.

final one. Here, texture filtering blurs the texture when it is projected onto the reflector. Lowering resolution may be desirable to save texture memory, or to use as a special effect.

This texturing technique is not far from simply environment-mapping the reflector, using a environment texture containing its surroundings. This is quite easy to do with OpenGL, as described in Section 5.4. This simplicity is countered by some loss of realism if the reflected geometry is close to the reflector.

### 17.1.3 Curved Reflectors

The technique of creating reflections by transforming geometry can be extended to curved reflectors. Since there is no longer a single plane that accurately reflects an entire object to its mirror position, a reflection transform must be computed per-vertex. Computing a separate reflection at each vertex takes into account changes in the reflection plane across the curved reflector surface. To transform each vertex, the reflection point on the reflector must be found and the orientation of the reflection plane at that point must be computed.

Unlike planar reflections, which only depend on the relative positions of the geometry and the reflector, reflecting geometry across a curved reflector is viewpoint dependent. Reflecting a given vertex first involves finding the reflection ray that intersects it. The reflection ray has a starting point on the reflector's surface and the *reflection point*, and a direction computed from the normal at the surface and the viewer position. Since the reflector is curved, the surface normal varies across the surface. Both the reflection ray and surface normal are computed for a given reflection point on the reflector's surface, forming a triplet of values. The reflection information over the entire curved surface can be thought of as a set of these triplets. In the general case, each reflection ray on the surface can have a different direction.

Once the proper reflection ray for a given vertex is found, its associated surface position and normal can be used to reflect the vertex to its corresponding virtual object position. The transform is a reflection across the plane, which passes through the reflection point and is perpendicular to the normal at that location, as shown in Figure 17.5. Note that computing the reflection itself is not viewer dependent. The viewer position

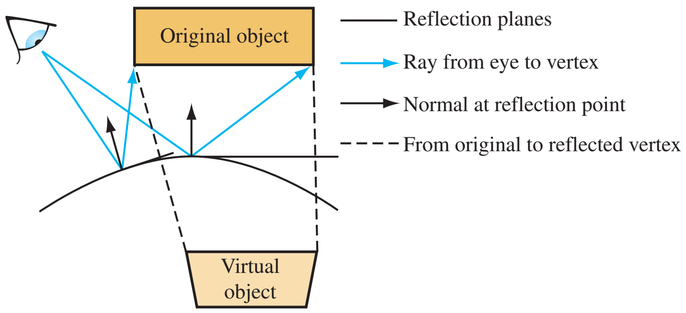

Figure 17.5 Normals and reflection vectors in curved reflectors.