Interactive Reflections on Curved Objects

Eyal Ofek Ari Rappoport

Institute of Computer Science, The Hebrew University

Abstract

Global view-dependent illumination phenomena, in particular reflections, greatly enhance the realism of computer-generated imagery. Current interactive rendering methods do not provide satisfactory support for reflections on curved objects.

In this paper we present a novel method for interactive computation of reflections on curved objects. We transform potentially reflected scene objects according to reflectors, to generate *virtual objects*. These are rendered by the graphics system as ordinary objects, creating a reflection image that is blended with the primary image. Virtual objects are created by tessellating scene objects and computing a virtual vertex for each resulting scene vertex. Virtual vertices are computed using a novel space subdivision, the *reflection subdivision*. For general polygonal mesh reflectors, we present an associated approximate acceleration scheme, the *explosion map*. For specific types of objects (e.g., linear extrusions of planar curves) the reflection subdivision can be reduced to a 2-D one that is utilized more accurately and efficiently.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

Keywords: ray tracing, interactive reflections, virtual objects method, reflection subdivision, explosion map.

1 Introduction

Interactive photo-realistic rendering is a major goal of computer graphics. Global view-dependent illumination phenomena greatly enhance image quality. An extremely important type of view-dependent phenomenon is reflection. Reflections on curved object are not supported well by current interactive rendering techniques. In this paper we address the problem of interactive rendering of reflections on curved objects.

Background. Current interactive graphics systems utilize hardware acceleration that directly supports hidden surfaces removal, simple local shading models and texture mapping. While the polygon throughput of these systems is impressive, the range of shading effects they provide hasn't changed much since their introduction. In particular, they lack support for global illumination phenomena in dynamic scenes.

Institute of Computer Science, The Hebrew University, Jerusalem 91904, Israel. http://www.cs.huji.ac.il/∼arir,∼eyalp arir,eyalp@cs.huji.ac.il

Global illumination phenomena greatly enhance the quality of synthetic imagery. They can be coarsely classified to view-independent and view-dependent phenomena. Among the former, diffuse illumination in static scenes [Sillion89] and shadows [Segal92] can be interactively rendered using current hardware. However, global *view-dependent* phenomena are crucial for providing life-like realism. When only view-independent effects are provided, the visual nature of the result can be dull and lifeless, even when the scene is dynamic.

An extremely important view-dependent illumination phenomenon is reflection. The dominant method for generating reflections is ray tracing [Whitted80, Glassner89]. In spite of extensive work on ray tracing acceleration schemes, [Jansen93] states that the only hope for interactive ray tracing lies in massively parallel computers, and even then satisfactory performance is not guaranteed.

Environment mapping [Blinn76, Greene86, Haeberli93, Voorhies94] generates at interactive rates reflections that are approximately correct when the reflected objects are relatively far from the reflector. However, when this condition is violated the results are of very poor accuracy.

It is well-known that reflections on planar surfaces can be generated by (1) mirroring the viewer along the reflecting plane, (2) creating a reflection image by rendering the scene from the new point of view, and (3) merging the main image with the visible portion of the reflector in the reflection image. Surprisingly, although this method can significantly accelerate ray tracing, it has been accurately documented only recently. The descriptions in [Foley90] (in which the method is called 'reflection mapping') and [McReynolds96] are correct only when the original viewer and all objects lie on the same side of the reflecting plane. A correct description is given in [Hall96]. [Diefenbach97] shows how to use variants of this method for interactive simulation of various general reflectance functions of planar objects. The concept of a reflected virtual world was also used in [Rushmeier86, Wallace87, Sillion89] for supporting specular reflections from planar objects in a radiosity context.

Contribution. In this paper we present a method for interactive rendering of reflections on *curved* objects, based on merging a primary image and a reflection image. The reflection image is generated by creating and rendering *virtual objects* corresponding to reflections of scene objects. Virtual objects are rendered like ordinary polygons, thus taking advantage of the features supported by the graphics system. They are created using a structure called the *reflection subdivision* and an associated approximate acceleration scheme, the *explosion map.*

The method presents a novel approach to the computation of reflections in computer graphics, and is unique in providing approximate reflections on curved objects at interactive rates. Moreover, the rendered scenes can be completely dynamic; no pre-processing is necessary. The method provides higher quality than environment mapping, because it allows reflected objects to be nearby the reflector and it supports equally well reflectors having a large curvature. For scenes in which reflected images of objects occupy more than a

few pixels and in which the depth complexity of the reflection image is not large, the method is much more efficient than ray tracing, because it efficiently exploits the spatial coherency of the reflection image. The price paid for the advantages of the method is that its performance is less efficient than that of environment mapping and the generated images are only polygonal approximations (as in most interactive systems). In addition, its accuracy depends upon the geometric nature of the reflector.

The paper is structured as follows. Section 2 gives an overview of the method. Sections 3, 4 and 5 deal with convex reflectors, discussing respectively the reflection subdivision, the explosion map, and special reflectors. Section 6 deals with non-convex reflectors. Results and an in-depth discussion are given in Sections 7 and 8.

2 Method Overview

In this section we give an overview of the virtual objects method. We present the general idea (2.1), image merging alternatives (2.2), a brief discussion on planar reflectors (2.3), and a high-level outline on non-planar reflectors (2.4).

2.1 General Idea

The virtual objects method is inspired by the following observation. Consider an image containing reflections. Two kinds of entities are visible: reflecting objects, or *reflectors*, and *reflected images* of reflected 3-D objects. When the reflector is a perfect planar one, the geometry of the reflected images is identical to images of the reflected objects from some other viewpoint. In fact, we cannot distinguish between 'real' objects and reflected images of objects. Interior designers utilize this phenomenon when covering walls with mirrors in order to make rooms seem larger. For non-planar reflectors, the appearance of reflected objects is a deformed version of their ordinary appearance. In general, there is no viewpoint from which they appear identical to their reflected images. The nature of the deformation depends upon the geometry of the reflector. Convex reflectors deform reflected objects to seem smaller, and concave reflectors produce reflected images that may seem larger than the reflected object or degenerate into strange chaotic images.

This observation inspires the following algorithm for generation of reflections (Figure 1): for every reflector and every object potentially reflected in it, compute a *3-D virtual object*, that, when rendered using ordinary 3-D rendering methods, will produce an image having a visual appearance similar to the object's reflected image. If depth relationships between the virtual objects are still correct, the rendered images of the virtual objects can be merged together using some hidden surfaces removal algorithm. The result can now be alpha blended with a reflector image containing view-independent lighting to produce the final image. The alpha blending coefficients are determined by the relative reflectivity of the reflector.

SceneRender (Scene S, View E):
(1) Render S without reflections into primary image I.
(2) For every visible reflector $R \in S$
(3) For every potentially reflected object $O \in S$
(4) $O' \leftarrow$ **VirtualObject** (R, O, E).
(5) Render O' into a reflection image I'.
(6) If multiple levels of reflections are desired
Call the algorithm recursively.
(7) Alpha blend I' and I, according to
the reflectivity of R.

Figure 1 The virtual objects method.

When virtual objects can be computed efficiently, the resulting method is very attractive, since reflected images are generated at the object, rather than the pixel, level. Most of this paper deals with step 4, efficient generation of virtual objects. Naturally, only visible reflectors are considered, and the scene can be stored in a data structure that supports culling of scene objects that cannot be reflected.

A comment about shading: for planar reflectors we can reflect the light sources as well as the scene objects and simply use the reflected ones. For non-planar reflectors, it is more accurate to compute shading values for vertices at the world coordinate system, and then use these values for the virtual vertices. On current architectures, this shading is most efficiently computed in software, and the hardware is used for rasterization and texturing.

2.2 Image Merging

The primary and reflection images can be merged in two ways. First, the reflection image can be used as a texture when rendering a reflector. Alternatively, the reflection image can be directly rendered on the screen (using a stencil bit-plane defining the screen image of the reflector.) The view-independent component of the reflector is now rendered, alpha blending it with the reflection image.

Texture mapping and stencil-guided image merging are standard features in interactive graphics systems, even current low-end ones. The choice of method depends on the actual graphics architecture available, especially on its memory organization. For more details, see [Ofek98, McReynolds96, Hall96].

2.3 Planar Reflectors

The method of [McReynolds96, Hall96, Diefenbach97] is a special case of the virtual objects method, when the reflectors are planar and when we consider the objects, rather than the viewpoint, as being mirrored. Note that in this case the method is essentially an image-space version of beam tracing [Heckbert84]. An attractive property of planar reflectors is that the location of a virtual point is a simple affine transformation, *mirroring*, of the real point. Moreover, this transformation does not depend on the viewer location, only on that of the reflector. In Figure 2(a), the location of the virtual image Q' of a scene point Q remains constant for two viewpoints E_1 and E_2. Hence, the same simple affine transformation can be used for all reflected polygons. Full details on how to generate the mirroring transformation are given in the above references.

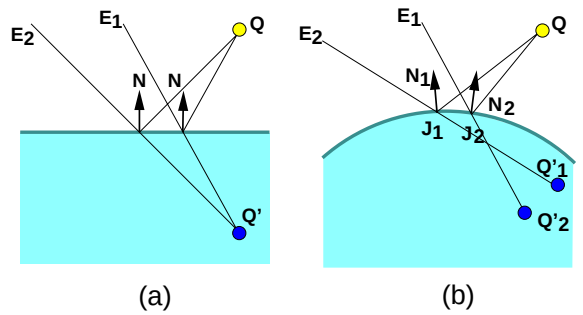


Figure 2 For planar reflectors, the virtual location of a point does not depend upon the viewpoint (a). This does not hold for curved reflectors (b).

Note that as presented so far, the method produces correct results only when the viewpoint and the reflected polygon are on the same side of the reflector. Consider a polygon lying on the other side of the reflector. After the mirroring transformation, it can erroneously obscure the reflector from the viewer, because they lie on the same

side of it. This problem can be overcome by not mirroring a polygon if all of its vertices lie behind the reflector. The test is done by plugging the vertex coordinates into the reflector plane equation and testing the sign of the result. However, this method does not solve the case when the polygon lies only partially behind the reflector. In many cases such polygons do not cause incorrect results because the virtual front part falls outside of the reflector stencil anyway. For planar reflectors, the problem can be solved very efficiently by defining the reflector plane as a front clipping plane.

2.4 Non-Planar Reflectors

Generation of virtual objects for non-planar reflectors is more difficult than for planar reflectors, because the main property of the planar case does not hold: the location of a virtual point is not a simple affine transformation independent of the viewer position (Figure 2(b)). In general, every reflected point is transformed differently.

Our approach is outlined in Figure 3. The reflected object is tessellated into polygons (step 1). The fineness of the tessellation depends upon the desired accuracy of the resulting reflection image. Tessellations are further discussed in Sections 7 and 8. In steps 2–5, virtual polygons are generated by computing virtual vertices for the tessellation vertices. The collection of all virtual polygons forms the desired virtual object rendered in step 5 of Figure 1. The main step is 4, computing a single virtual vertex; its description occupies much of the rest of the paper.

VirtualObject (Reflector R, Object O, View E):
(1) Tessellate O into polygons.
(2) For each polygon P
(3) For each vertex Q of P
(4) $Q' \leftarrow$ **VirtualVertex** (R, Q, E).
(5) Connect the Q's to form a virtual polygon P'.
(6) Connect the P's to form the virtual object O'.

Figure 3 Computing a virtual object O' for a potentially reflected object O on a non-planar reflector R.

Rendered polygons are consistent and possess no holes, because virtual objects are formed by connecting virtual vertices. Visibility relationships between virtual objects are preserved due to the usage of a hidden surfaces removal mechanism (in practice, a z-buffer) for them.

3 The Reflection Subdivision

In this section we start detailing our approach towards computing virtual vertices for curved reflectors. We assume here that the reflector is convex. Concave and other non-convex reflectors are discussed in Section 6. Our approach is based on approximating the reflector by a polygonal mesh. In many cases this is the format in which objects are given anyway; when they are given in a higher-level representation (e.g., a NURBS surface) they are tessellated. For simplicity, we assume that mesh polygons are triangles, but this is not necessary.

Intuition. Given a reflector R and an arbitrary scene point Q, we want to generate the corresponding virtual point Q' (consult Figure 2(b)). If we knew the point of reflection J and normal N on the boundary surface of R, we could easily compute Q' by mirroring Q along the tangent plane to R at J. In some cases, when we know the geometric nature of the reflector (e.g., a sphere), J can be computed by a direct formula. However, for a general convex polygonal mesh there is no direct formula.

We use an approximation. Every reflector triangle defines two space cells: a *reflected cell* and a *hidden cell*. Suppose that we can find the cell C, defined by triangle T, in which the scene point Q lies. A naive method would mirror Q across the plane containing T. However, this would clearly show the linear approximation of the reflector (imagine a reflecting sharply cut diamond!). Instead, we use the relative location of Q inside C to define a triplet of barycentric coefficients. These coefficients are used to interpolate the three tangent planes at the vertices of T, yielding a new tangent plane that is now used for mirroring Q.

In this section we study the space subdivision defined by the reflector and also explain why we need to compute virtual points for points that are not reflected. The full details of the computation are given in Sections 4 and 5.

The subdivision. Each vertex V_i of the tessellated reflector possesses a normal N_i. Reflector vertices are either *front-facing* or *back-facing*, according to whether their normals point towards or away from the viewer (a normal orthogonal to the line of sight is considered front-facing). Due to the convexity of the reflector, every front-facing vertex is visible by the viewer (when there are no other obscuring objects). Note that back-facing vertices might still be visible (this is a tessellation artifact). When all vertices of a mesh triangle are front-facing (back-facing), we refer to the triangle as being front-facing (back-facing). Otherwise we say that the triangle is a *profile* triangle.

For each front-facing vertex V_i we define two rays: (1) a *reflection ray* R_i, mirroring the ray from V_i to the viewer across the normal N_i, and (2) a *hidden ray* H_i, originating at V_i and extending to infinity in the opposite direction to that of the viewer. Figure 4 shows a 2-D version of the situation. In (a), reflection rays are shown in red and hidden rays in blue.

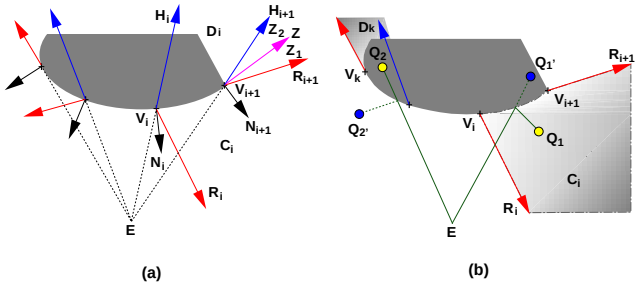


Figure 4 (a) The reflection subdivision in 2-D. C_i and D_i are the reflected and hidden cells defined by reflector vertices V_i, V_{i+1}. The ray Z bisects the unreflected region on the right into two parts Z_1, Z_2. (b) Computation of virtual vertices: the point Q_1 in the reflected cell C_i is transformed to Q_1' inside the hidden cell D_i; the point Q_2 in the hidden cell D_k is transformed to Q_2' outside the reflector in the reflected cell C_k.

Two reflection rays R_i, R_j corresponding to adjacent front-facing mesh vertices V_i, V_j define a ruled bi-linear parametric surface $s(V_i + tR_i) + (1 - s)(V_j + tR_j)$. Note that in general this surface is not planar, because the two rays are usually not co-planar. The two hidden rays H_i, H_j span an infinite truncated triangle containing the edge V_i, V_j.

Now consider the three vertices V_i, V_j, V_k of a front-facing mesh triangle V_{ijk}. The triangle induces two space regions: (1) A *reflected cell* C_{ijk} bounded by the three ruled surfaces corresponding to the triangle edges and by V_{ijk} itself (figure 10). (2) A *hidden cell* D_{ijk},

which is the infinite part of the truncated pyramid bounded by V_{ijk} and the triangles spanned by the hidden rays. We refer to the union of the reflected (hidden) cells as the reflected (hidden) region.

An important property of the reflected and hidden cells is that they do not intersect each other, since the reflector is convex. Therefore, we can define the *reflection subdivision* as the subdivision of space induced by these cells. Note, however, that these cells do not *cover* space; we call the part of space not covered by reflection or hidden cells the *unreflected region*. In Figure 4(a), the part of the unreflected region lying on the right side of the reflector is the union of Z_1, Z_2 (the reason for subdividing this region and the meaning of the ray Z are explained below). Points in the unreflected region can (in principle) be seen by the viewer, but cannot be reflected by the reflector. A point is potentially reflected by the reflector if and only if it lies in the reflected region. We say 'potentially' because its reflection may be obscured by the reflection of another point.

The unreflected and hidden regions. We compute virtual images for vertices of potentially reflected scene polygons (Figure 3, step 4). These virtual vertices are connected in order to generate virtual polygons, which are then rendered to create the reflection image (Figure 1, step 5). Scene polygons that lie completely in the hidden or unreflected regions can be discarded. However, *mixed polygons*, lying partially in these regions and partially in the reflected region, pose a problem. For such polygons, we would like to render the reflection of the part that lies in the reflected region. However, if we compute only one or two virtual vertices, we would not be able to connect these in order to generate virtual polygons. In some sense, the vertices lying in hidden or unreflected regions are representatives of a polygon area that we want to see reflected.

A naive way to deal with mixed polygons is to intersect them (exactly or approximately) with the region boundaries, thus forcing them to have a uniform classification. However, this is inefficient because the regions depend on the viewpoint. Another way is to subdivide them into smaller polygons, effectively doing an adaptive tessellation of scene objects. Subdivision is stopped when the 'lost' areas are deemed to be small enough.

A more efficient and elegant method is to define a virtual vertex for *every* polygon vertex, even for hidden and unreflected ones (e.g., vertex Q_2 in Figure 4(b)). These *doubly virtual* vertices are not real reflections; their sole purpose is to 'close' virtual polygons so that the graphics system could render them. In general, they lie outside the image of the reflector. Note that this actually is the approach taken in the planar reflector case. Hidden cells are easy to take care of, because there is a one-to-one correspondence between hidden and reflected cells. Moreover, it is possible to define a transformation that maps a reflected cell to exactly cover the corresponding hidden cell, and maps a hidden cell to exactly cover its corresponding reflected cell (see Section 4.2).

The unreflected region is more problematic. We would like to define a transformation for this region such that (1) the part of the region adjacent to a reflected cell will be transformed to be adjacent to its corresponding hidden cell (and vice versa), and (2) there is some continuity of the transformation between the unreflected and the reflected regions. To achieve such a transformation, we define for every contour edge of the reflector an auxiliary *bisecting surface* Z, which extends the edge into the unreflected region. Figure 4(a) shows a 2-D example. In 2-D we have a contour vertex and not a contour edge (it is simply the extreme vertex V_{i+1}), and the bisecting surface is simply a ray Z. Z is orthogonal to the normal N_{i+1} at V_{i+1} and extends V_{i+1} into the unreflected region, thus bisecting the region into two parts Z_1, Z_2. The desired transformation is simply a linear mirroring transformation that mirrors Z_1 into Z_2 and vice versa. In 3-D, the bisecting surface is non-linear, and we do not define it explicitly; it is defined implicitly by the transformation we use for computing virtual vertices (Section 4.2).

As in the planar case, doubly virtual vertices might cause their virtual polygon to obscure the reflector. The solution in the planar case, a front clipping plane, can be generalized to non-planar reflectors by utilizing a second z-buffer containing the reflector's geometry. Every pixel generated during rendering of the virtual polygons will be tested twice: once against the ordinary z-buffer, in order to produce correct depth relationships between all virtual polygons, and once against the reflector z-buffer, to ensure that pixels in front of the reflectors are discarded.

A second z-buffer is not easy to define efficiently on today's graphics architectures. Alternatives that are currently more practical are: (1) do not do anything, anticipating that the obscuring pixels will fall outside the screen mask of the reflector, (2) approximate the reflector using six clipping planes, an option available on standard architectures, and (3) tessellate the scene so that mixed polygons are very small. Surprisingly, the first approach works well in the vast majority of cases, due to the way objects are usually positioned relative to each other and the way they are viewed. The second option reduces the problem but does not guarantee the resulting quality. The third option also reduces the problem, but requires more computations since there are more virtual vertices to compute. Tessellations are discussed in Sections 7 and 8.

4 The Explosion Map Acceleration Method

In some cases it is very efficient to compute the reflection subdivision and search it to find the cell in which a point lies (Section 5). In the general 3-D case, a faster indexing scheme is preferable.

In this section we describe an approximation method, the *explosion map*, which is a data structure for accelerating the computation of virtual vertices. It is prepared for each reflector separately, and recomputed whenever the viewpoint or the reflector are moved. The map is an image whose pixel values hold IDs of reflector triangles, and which represents a spherical 2-D cross section of the subdivision. To compute a virtual image of a scene point, we compute explosion map coordinates for it, thus yielding the ID of a specific triangle. The virtual image is computed using that triangle.

The explosion map is somewhat similar to a circular environment map [Haeberli93] in that it is an image in which a circle corresponds to the reflection directions (Figure 5(b)). However, it is unlike an environment map in that the latter contains renderings of other scene objects, while the explosion map contains only the reflector (Figure 5(a)). We next detail the computation (4.1) and utilization (4.2) of the map.

4.1 Computing an Explosion Map

An explosion map is a function of the tessellated reflector, the viewpoint, a 3-D sphere, and a desired resolution. The sphere should be centered at a point that is an intuitive 'center' of the reflector (as in environment mapping), and its radius should be large enough so that it bounds the reflector (actually, a sphere is not essential; we need any convex geometric object that approximates the reflector's shape). The map resolution should be large enough so that there are substantially more map pixels than reflector triangles. In practice, a resolution of 200^2 is sufficient when the reflector has been tessellated into several hundred triangles. The depth resolution of the map should have enough bits to hold unique IDs for all reflector vertices plus one more bit (needed to distinguish between ordinary triangles and extension polygons, defined below).

The basic operation in computing the map, **MapCoords**, involves

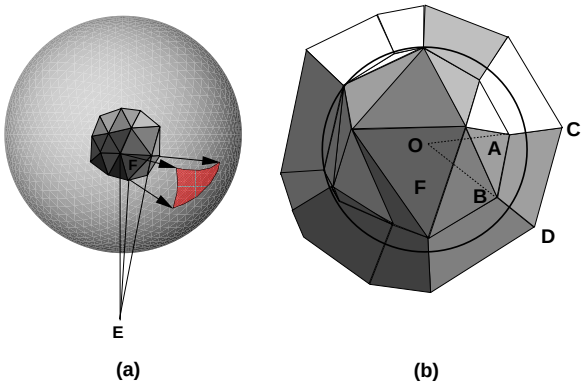


Figure 5 Explosion map: (a) reflection rays and intersection points on a bounding sphere; (b) the resulting map. *C* and *D* are extension vertices of *A* and *B*.

deriving the map coordinates $T = (t_x, t_y)$ corresponding to a normalized direction vector $N = (x, y, z)$ going from the center of the sphere to an arbitrary direction. If the resolution of the map is r^2, N is mapped to $T = (\frac{sx}{(2(z+1))^{1/2}} + s/2, \frac{sy}{(2(z+1))^{1/2}} + s/2)$, where s is a number a little smaller than r. This mapping is similar to that used for generating a circular environment map from a map rendered on the faces of a box [Haeberli93]. The pixels to which directions are mapped all fall inside a circle of radius $s/2$. The circle represents all possible reflection directions.

The map itself is computed as follows (Figure 6). For every front-facing reflector vertex, we compute map coordinates by intersecting its reflection ray with the sphere and calling **MapCoords** with the direction from the sphere's center to the intersection point (step 3). Back-facing vertices are denoted as such (step 4) to facilitate fast identification of profile triangles in step 6. For each front-facing reflector triangle (recall that a triangle is called front-facing if all its vertices are front-facing, and is called profile if only some of its vertices are front-facing), the corresponding triangle defined by the map coordinates is filled on the map, using its unique ID as color (step 5). Polygon fill can be done by the graphics hardware. For profile triangles, the normals of their back-facing vertices are projected in the direction of the viewer such that they are orthogonal to the line from the viewer through the vertex (step 7). The triangles thus become front-facing, and are now filled on the map as done for triangles that were front-facing originally (step 8).

So far, the interior of a map circle of radius $s/2$ has been partially filled, but not completely. This is due to the existence of the unreflected region and the fact that the filled map triangles are linear. As we explained in Section 3, we want the directions into the unreflected region to be filled on the map as well so that we could use it to compute doubly virtual vertices. To ensure that all directions are filled on the map, in **ExtendMap** the map is extended to cover the circle as follows. For each profile triangle V_{ijk} having two back-facing vertices (say V_i, V_j), we define an *extension polygon* E_{ij} in map coordinates and fill it with the ID of V_{ijk}. The vertices of E_{ij} are T_i, T_j, and extensions of each of these vertices in the direction away from the circle's center (in Figure 5(b), the extensions of vertices A, B are C, D). The extensions should be long enough so that the circle is completely covered. In practice, it is enough that the length of the segment from the center to each extended vertex is $0.6s$. Extension polygons effectively comprise an implicit representation of the bisecting surfaces Z explained in Section 3. Other methods for representing the unreflected region on the map are discussed in [Ofek98].

ExplosionMap (Reflector *R*, View *E*, Center *C*, Distance *d*, Resolution *r*):
(1) Let *S* be a sphere centered at *C* having radius *d*.
(2) Let *M* be an image of size $r \times r$.
(3) For each reflector vertex V_i
  If V_i is front-facing
    Let R_i be the reflection ray of V_i.
    Let I_i be the intersection of R_i with *S*.
    Let J_i be the normalized direction from *C* to I_i.
    $T_i \leftarrow$ **MapCoords** (J_i, *r*).
  Else
(4) Denote V_i as back-facing.
(5) For each reflector triangle V_{ijk}
  If V_{ijk} is front-facing
    Fill the triangle T_i, T_j, T_k on *M*,
    using the ID of V_{ijk} as the color.
(6) Else if V_{ijk} is a profile triangle
(7) Fix its back-facing normals.
(8) Compute and fill T_i, T_j, T_k as before.
(9) **ExtendMap** (*R*, *M*).

Figure 6 Computing an explosion map.

4.2 Computing Virtual Vertices

The explosion map circle represents a mapping of all possible reflection directions. We use it to directly generate the final virtual vertex Q' corresponding to a potentially reflected scene vertex Q. For each reflector we compute two explosion maps: a *near map* and a *far map*. The near map is computed using a sphere that bounds the object but does not intersect any other object, and the far map is computed using a sphere that bounds all the scene. It is important to understand that although the topologies of the two maps are quite similar (because cells do not intersect each other), their geometries are different; reflection rays, which determine the geometry of map vertices, evolve non-linearly.

In addition to the explosion maps, we store a *hidden map* and an auxiliary z-buffer of the reflector. The hidden map is simply an item buffer of the visible mesh triangles. In other words, it is a discrete map in which a visible mesh triangle is mapped to a 2-D triangle filled by the ID of the mesh triangle. The map resolution can be smaller than that of the frame buffer (say, 200^2).

The basic operation needed is **MapToVirtualVertex**, whose arguments are a map *M*, a 3-D point *Q* and a corresponding map point *I*. Assume that the ID in $M(I)$ is that of an ordinary mesh triangle *V* (not an extension polygon) having 2-D vertices A, B, C (these are the T_i's computed in step 3 of Figure 6). The output is the virtual point Q'. The operation is implemented in three steps: (1) compute barycentric coordinates s, t of *I* relative to *V* by solving the two linear equations in two variables $(1-(s+t))A+sB+tC = I$; (2) use s, t as weights in a weighted average of the 3-D vertices and normals of *V* that yields a plane of reflection *U*; and (3) mirror *Q* across *U* to produce Q'. Note that negative barycentric coordinates are perfectly acceptable. The computation can be performed in integers or floating point, to reduce aliasing artifacts resulting from the discrete nature of the map. Extension polygons are handled similarly, using four bilinear coordinates instead of three. This treatment of extension polygons effectively implements the non-linear mirroring transformation of the unreflected region motivated in Section 3.

Computation of virtual vertices for a scene vertex *Q* is shown in Figure 7. We first determine if *Q* is hidden (steps 1, 2), by testing it in screen coordinates against the reflector's z-buffer. If it is, *Q*'s virtual image is computed by the hidden map (step 3). Note that an obvious optimization here is to do this only for hidden vertices that belong to mixed polygons, since we don't need virtual images for polygons that are hidden completely.

VirtualVertex (Reflector R, Point Q, View E):
(1) Let I be the screen coordinates of Q (using E).
(2) If Q is hidden by a mesh triangle V
(3) Return **MapToVirtualVertex** (*HiddenMap*, Q, I).
(4) Let c be the direction from the center of R to Q.
(5) $T \leftarrow$ **MapCoords** (c, r).
(6) $Q'_n \leftarrow$ **MapToVirtualVertex** (*NearMap*, Q, T).
$Q'_f \leftarrow$ **MapToVirtualVertex** (*FarMap*, Q, T).
(7) Let d_n, d_f be the relative distances of Q from the near and far spheres.
Return $\frac{Q'_n/d_n + Q'_f/d_f}{1/d_n + 1/d_f}$.

Figure 7 Computing virtual vertices using the explosion and hidden maps.

When Q is not hidden we use the explosion maps. The normalized direction from the center of the reflector to Q is used to obtain map coordinates, in the same way used for creating the maps (steps 4, 5). Note that the map coordinates T are the same for both maps, but the triangle IDs found at T are different. In general, none of these triangles corresponds to the correct reflection cell in which Q is located, because we approximated the correct ray of reflection of Q by a ray from the center of the reflector (when higher accuracy is desired, we can use an improved approximation or locally search the correct cell [Ofek98].) Each of these triangles defines an auxiliary virtual vertex (step 6), and a weighted average of those is taken to obtain the final virtual vertex (step 7). There may be other ways to choose the weights than the obvious one shown. Figure 13 shows near and far explosion maps, in which polygon IDs are encoded by colors for visualization purposes.

5 Improved Efficiency for Linear Extrusions

For some common reflectors, it is possible to compute virtual vertices more efficiently than the explosion map, by directly utilizing the reflection subdivision to find the cell in which a scene point lies. Among these reflector are linear extrusions of planar curves (e.g., cylinders) and cones. For spheres, there is an efficient method that does not use the reflection subdivision at all. In general, if an implicit equation defining the reflector is available, the reflection point can be computed as in [Hanrahan92] (although this method is slow). Below we detail the case of an extruded reflector. The direct computation for cones and spheres is simple and given in [Ofek98].

Consider a 2-D reflection subdivision, as shown for example in Figure 4. We can optimize the step of identifying the cell in which a point lies by organizing the reflection cells in a hierarchy. Define $C_{i,j}$ to be the region bounded by reflection rays R_i, R_j and the line segment (V_i, V_j). Note that $C_{i,j}$ contains every cell $C_{k,r}, i \le k < r \le j$. Classifying a point with respect to a cell $C_{i,j}$ amounts to a few 'line side' tests, implemented by plugging the point into the line's equation and testing the sign of the result. If we find that the point is not contained in $C_{i,j}$, we know that it is outside all contained cells $C_{k,r}$. A binary search can thus be performed on the hierarchy. Note that there are no actual computations involved in generating the hierarchy, since it is implicitly represented by the numbering of the reflector vertices. A similar hierarchy can be defined for the hidden cells as well. Membership in the two (at most) unreflected cells can be tested easily. Consequently, the cell in which a point is located can be found using a small number ($O(\log n)$ where n is the reflector tessellation resolution) of 'line side' tests.

Suppose that the reflector is a linear extrusion of a convex 2-D planar curve. We can reduce the computation of a virtual vertex to 2-D by (1) projecting the viewer and all scene points onto the plane, (2) performing the 2-D computation, obtaining a line of reflection L_Q for each scene vertex, (3) extruding L_Q to 3-D to form a plane of reflection T, and (4) computing a final virtual vertex by mirroring the original vertex across T. The screen in Figure 17 is a linear extrusion of a convex planar curve.

6 Non-Convex Reflectors

Concave reflectors. The computations we perform for concave reflectors are identical to those for convex ones, but it is interesting to note that concave reflectors produce significantly more complicated visual results. In Figure 8 we see a viewer E in front of a concave reflector and three reflection rays. The reflection of an object located in region A (left) looks like an enlarged, deformed version of the object. The reflection of an object located in region B (middle) looks like an enlarged, deformed, upside-down version of the object. The reflection of objects located in region C (right) is utterly chaotic. This chaotic nature is inherent in the physics of reflections and is not an artifact of computations or approximations.

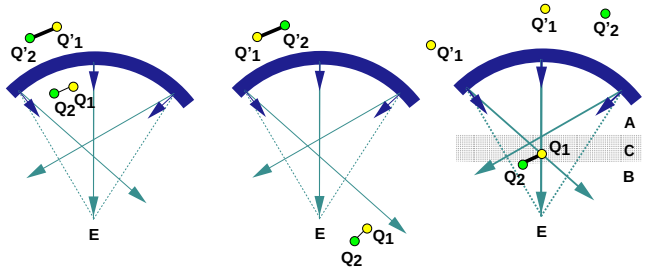


Figure 8 Behavior of reflections on concave reflectors.

Reflections of objects lying in regions A and B can be computed exactly as for convex reflectors, because in these regions the reflection subdivision is well-defined (since the reflection cells are disjoint). Reflections of objects lying in region C or intersecting that region are unpredictable and chaotic anyway, so almost any policy for computing virtual vertices will be satisfactory. In particular, we can simply use the value computed by the explosion map, thereby treating concave reflectors exactly as convex ones.

Figure 11 shows a concave reflector. On the right, we see the reflector, the reflection rays, a reflected planar object, and the computed virtual object, all these from a point of view different from the viewer's. On the left we see the final image from the viewer's point of view. The two explosion maps and the hidden map are shown at the bottom right. Note that reflected objects must be very close to this reflector to cross from region B to regions A or C.

Reflectors of mixed convexity. Reflectors that are neither convex nor concave should be decomposed into convex and concave parts. For many objects this can be done fully automatically [Spanguolo92]. Some polygonal surfaces contain saddles, resulting in a decomposition that is too fine. In such cases it is advised that users decompose the object manually. Note that the actual requirement is not of pure convexity of concavity, but rather that the reflection cells would not self-intersect in areas where reflected objects lie. Devising automatic algorithms that take this into consideration when decomposing the object is an interesting topic for future work. When manual decomposition is used, reflectors cannot dynamically change their shape in an arbitrary way, but the scene can still be dynamic. Figure 9 shows a reflector with a convex part (red) and a concave part (green). Note the seamless transition of the reflection image between the convex and concave parts.

7 Results

We have implemented our algorithms using OpenGL on SGIs running Irix and on PCs running Windows '95 and NT. Figure 12 shows a cylinder reflector modeled as a linear extrusion of a circle. Figure 13 demonstrates the effect of varying reflector tessellation resolution. The bottom part shows the near and far explosion maps. We see that using 128 triangles the reflection image already has an approximately correct geometric form, and that using 2048 rather than 512 triangles barely makes a difference. Figure 14 shows the effect of varying the tessellation of the reflected object (using 512 reflector triangles). A tessellation of 7x7 is sufficient. A lower resolution would suffice for objects farther away from the reflector,

Figure 16 shows a scene with four reflecting spheres, a table, and a window, rendered by our method (top) and by Rayshade, a well-known raytracer (bottom). A checkerboard texture was used in order to emphasize the reflections. The geometric shapes of the reflections in the two images are visually very similar. The texture in the bottom image is sharper because we use the graphics hardware for texture mapping.

On an SGI O_2, the top image required **less than a second**, and the bottom one required **50 seconds**. For Rayshade, we turned off shadows rays, highlights, and anti-aliasing, and we used a single sample per pixel and a manually tuned uniform grid as an acceleration scheme. Image resolution is 512^2.

Figure 15 shows the same scene, from a slightly different viewpoint and using real textures. The shadows are pre-computed textures. Figure 17 shows a reflecting TV modeled as an extrusion of a convex planar curve. Figure 18 shows recursive reflections on a planar mirror. Figure 19 shows a mask composed of several convex and concave pieces. Note the correct reflections of the red and green spheres on both 'cheeks' of the mask and on the nose. Figure 20 shows several reflecting polyhedra and a reflecting sphere. All of these scenes (except Figure 19) are displayed in real-time on an SGI Infinite Reality. We haven't tried the scene of Figure 19 on such a machine; on an SGI O_2, Figure 19 requires about a second with our method, and 1.5 minutes using Rayshade.

8 Discussion

The virtual objects method is the first method capable of accurately approximating reflections on curved objects at interactive rates. In this paper we presented the basic method for a single level of reflection and its implementation for general polygonal meshes and for linear extrusions. Clearly, the method possesses both advantages and disadvantages. We discuss these below, both in isolation and in the context of other methods.

Quality. In general, the quality produced by the method is satisfactory, especially for interactive use. The explosion map gives good results even for planar or nearly planar reflectors. Like any approximation method, ours might produce visible artifacts. The most noticeable ones occur when objects are not tessellated finely enough, in which case their reflections look too much like their real-world images and are not deformed according to the geometry of the reflector. In addition, reflections might be slightly translated inaccurately because we do not compute the exact explosion map cells to which vertices are mapped.

Other visible artifacts can be seen near the boundaries of the reflector, when the transformation used to create doubly virtual vertices is not a good approximation to the correct reflection. In this case the seam between convex and concave regions might be visible. Even in this case, reflections are self-consistent and do not exhibit holes. When the reflector shape on the explosion map is far from convex, our heuristic for representing the unreflected region (extension polygons) might yield visible artifacts. Obviously, doubly virtual vertices might still hide the reflector when not using a second z-buffer. However, as we noted earlier, this usually does not happen because their screen images tend to fall outside the screen image of the reflector.

An attractive property of the method that has not been mentioned so far is that it supports interactive rendering of *refractions*, by using refraction rays instead of reflection rays. There are some additional differences, detailed in [Ofek98].

Tessellation strategies. As shown in Section 7, some tessellation of reflected objects is usually essential for providing sufficient accuracy. The finer the tessellation, the more accurate the reflections. At the same time, increasing the tessellation has an adverse impact on performance. These considerations are identical to those employed in interactive rendering of curved objects in general. There are two standard approaches: (1) usage of uniform tessellations, pre-computed such that quality is satisfactory, and (2) usage of hierarchical tessellations (levels of detail, etc).

Both approaches can be taken in our case as well. When the distances from a reflector to reflected objects and viewer remain approximately constant, we can pre-compute a uniform tessellation. The tessellation resolution of the reflected object should be chosen such that its virtual polygons cover several dozen pixels. Otherwise, hierarchical tessellations can be used. These can exhibit the same artifacts as when they are used for ordinary objects, e.g. discontinuities during animation. Note that the reflector tessellation resolution can be lower than that used when rendering its view-independent image. Using hierarchical tessellations is a topic for future work.

Performance. In the worst-case, all scene points can indeed be reflected on every convex part and every concave part of every reflector. Denote by r the number of visible reflectors and by $n(n')$ the number of vertices in the original (tessellated) scene. The time complexity of the method is $O(r \times n')$, which is thus worst-case optimal for a given degree of tessellation. For a single reflector, the step of computing the explosion and hidden maps is linear in the size of the reflector and is roughly equivalent to rendering the reflector three times at low resolution. The step of computing a virtual vertex for a scene vertex requires a relatively small *constant* number of operations. Moreover, the operations performed are highly regular, and are probably not too difficult to parallelize or implement in hardware. The cost of rendering virtual polygons is similar to rendering the whole scene. If deforming reflectors are desired, they should be subdivided into convex and concave parts on each frame, which costs time linear in their size. Naturally, as the depth complexity of the reflection image increases, the time complexity of our method diverges from the optimal.

The scenes shown in this paper run interactively (1-30 frames per second) on an SGI O_2 workstation. This performance was achieved *without any optimization*; in particular, no method for culling objects that cannot be reflected has been used. On today's systems, without further optimizations the number of reflected objects cannot be much larger than shown while still guaranteeing interactive performance.

Comparison to other methods. We can compare our method to environment mapping or ray tracing, which are currently the only techniques capable of computing reflections on curved objects. Both visual accuracy and efficiency should be considered.

Environment mapping is relatively accurate only when reflected ob-

jects are relatively far from the reflector and when the curvature of the reflector is not large. When the scene is static, time complexity is linear in the size of the reflector, because the map can be precomputed. This is in general much faster than our method. When the scene is dynamic, the map must be recomputed on each frame for each reflector. This also holds when only the viewer changes, unless the special hardware of [Voorhies94] is used. Complexity is $r \times n$, which is closer to our method but still more efficient. To what degree depends on the amount of tessellation. However, environment mapping simply does not provide realistic accuracy. Seeing reflections of objects that are nearby as if they are very far creates an uneasy feeling and definitely cannot be qualified as realistic.

Ray tracing obviously produces higher quality images than our method and supports a wider range of illumination phenomena. Regarding efficiency, the relevant characteristics of our method are: (1) it operates at the object level rather than the pixel level (we have an object and we want to know where it is reflected, rather than having the point of reflection and seeking an object), (2) it transforms the problem into one that standard graphics systems can handle, (3) it transforms the computation into a local one involving a single reflector-reflected pair, instead of the global ray tracing computation ('find the nearest object'); global visibility relationships are automatically handled by the z-buffer, and (4) it uses both the CPU and the graphics system, dividing (but not necessarily balancing) the load between them. When these properties are significant, our method is more efficient than ray tracing. Ray tracing can be expected to perform better when (1) reflected objects do not cover many pixels, (2) there are many curved reflectors, or (3) the depth complexity of the reflected images is large. It may or may not be faster when there is no graphics hardware. Note that our method scales much better than ray tracing to larger image resolutions, while ray tracing scales better with scene depth complexity.

It is very difficult to predict the point from which ray tracing is more efficient. On the relatively simple scenes shown in this paper, the method is at least an order of magnitude more efficient than Rayshade, a well-known available ray tracer, even when it uses a manually tuned acceleration scheme.

Future work. Both efficiency and quality issues should be further investigated. Efficiency issues include: acceleration using global scene organization techniques, hierarchical tessellations, possible hardware implementation, acceleration using time coherence, and usage of the method to accelerate other illumination methods. Quality issues include refining the initial approximation given by the explosion map, improved methods for filling the unreflected region on the map, using the method for rendering refractions, automatic decomposition of reflectors of mixed convexity, quantifying the degree of error introduced by our approximations, and additional levels of recursive reflections.

Conclusion. We feel that correct reflections from small objects are not very important. Such reflections, reflections on complex mixed convexity objects, and reflections of distant objects can be convincingly emulated using environment mapping. High quality reflections are therefore needed for relatively large objects with relatively uniform convexity (or concavity). A typical scene does not contain too many curved objects like these. As a result, although the time complexity of the method is theoretically quadratic in the number of reflectors, in practice its complexity is linear in the size of the scene (it can be sub-linear if scene databases are used for culling objects). Applicability will increase with increases in processing power and graphics hardware. Even today, there are many applications in which the number of objects in the scene is less important than the rendering quality. In these cases, our method is at least an order of magnitude faster than ray tracing and provides higher visual quality than environment mapping.

Our experience is that interacting with scenes containing reflections is immensely more enjoyable than with scenes without reflections. Reflections bring dull and lifeless scenes to life.

Acknowledgements. We thank Dani Lischinski for commenting on a draft of this paper and for fruitful discussions. We also thank Amichai Nitsan for his involvement in part of the implementation. Lastly, we warmly thank Leo Krieger for his continuous support.

References

[Blinn76] Blinn, J., Newell, M., Texture and reflection in computer generated images. *Comm. ACM*, 19:542–546, 1976.

[Diefenbach97] Diefenbach, P.J., Badler, N.I., Multi-pass pipeline rendering: realism for dynamic environments. Proceedings, *1997 Symposium on Interactive 3D Graphics*, ACM Press, 1997.

[Foley90] Foley, J.D., Van Dam A., Feiner, S.K., Hughes, J.F., Computer Graphics: Principles and Practice, 2nd ed., Addison-Wesley, 1990.

[Glassner89] Glassner, A. (ed), An Introduction to Ray Tracing. Academic Press, 1989.

[Greene86] Greene, N., Environment mapping and other applications of world projections. *IEEE CG&A*, 6(11), Nov. 1986.

[Haeberli93] Haeberli, P., Segal, M., Texture mapping as a fundamental drawing primitive. Proceedings, *Fourth Eurographics Workshop on Rendering*, Cohen, Puech, Sillion (eds), 1993, pp. 259–266.

[Hall96] Hall, T., Tutorial on planar mirrors in OpenGL, posted to comp.graphics.api.opengl, Aug. 1996.

[Hanrahan92] Hanrahan, P., Mitchell, D., Illumination from curved reflectors. Proceedings, Siggraph '92, ACM Press, pp. 283–291.

[Heckbert84] Heckbert, P.S., Hanrahan, P., Beam tracing polygonal objects. *Computer Graphics*, 18:119–127, 1984 (Siggraph '84).

[Jansen93] Jansen, F.W., Realism in real-time? Proceedings, *Fourth Eurographics Workshop on Rendering*, Cohen, Puech, Sillion (eds), 1993.

[McReynolds96] McReynolds, T., Blythe, D., Programming with OpenGL: Advanced Rendering, course #23, Siggraph '96.

[Ofek98] Ofek, E., Modeling and Rendering 3-D Objects. Ph.D. thesis, Institute of Computer Science, The Hebrew University, 1998.

[Rushmeier86] Rushmeier, H.E., Extending the radiosity method to transmitting and specularly reflecting surfaces. Masters's thesis, Cornell University, 1986.

[Segal92] Segal, M., Korobkin, C., van Widenfelt, R., Foran, J., Haeberli, P., Fast shadows and lighting effects using texture mapping. *Computer Graphics*, 26:249–252, 1992 (Siggraph '92).

[Sillion89] Sillion, F., Puech, C., A general two-pass method integrating specular and diffuse reflection. *Computer Graphics*, 23(3):335–344 (Siggraph '89).

[Spanguolo92] Spanguolo, M., Polyhedral surface decomposition based on curvature analysis. In: *Modern Geometric Computing for Visualization*, T.L. Kunii and Y. Shinagawa (Eds.), Springer-Verlag, 1992.

[Voorhies94] Voorhies, D., Foran, J., Reflection vector shading hardware. Proceedings, Siggraph '94, ACM Press, pp. 163–166.

[Wallace87] Wallace, J.R., Cohen, M.F., Greenberg, D.P, A two-pass solution to the rendering equation: a synthesis of ray tracing and radiosity methods. *Computer Graphics*, 21:311–320, 1987 (Siggraph '87).

[Whitted80] Whitted, T., An improved illumination model for shaded display. *Comm. of the ACM*, 23(6):343–349, 1980.

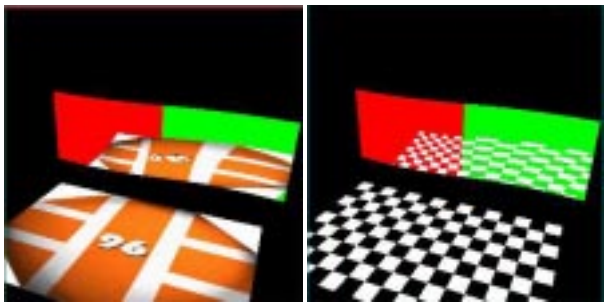


Fig. 9: Mixed convexity reflector, with seamless reflections.

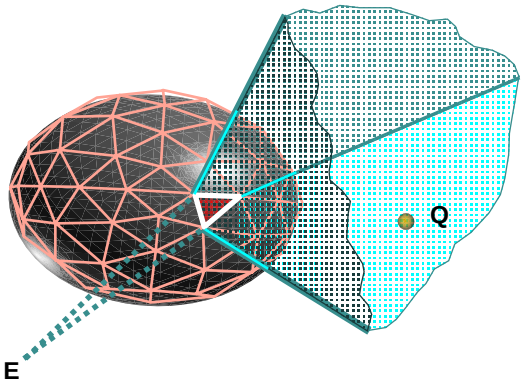


Fig. 10: A 3−D reflected cell.

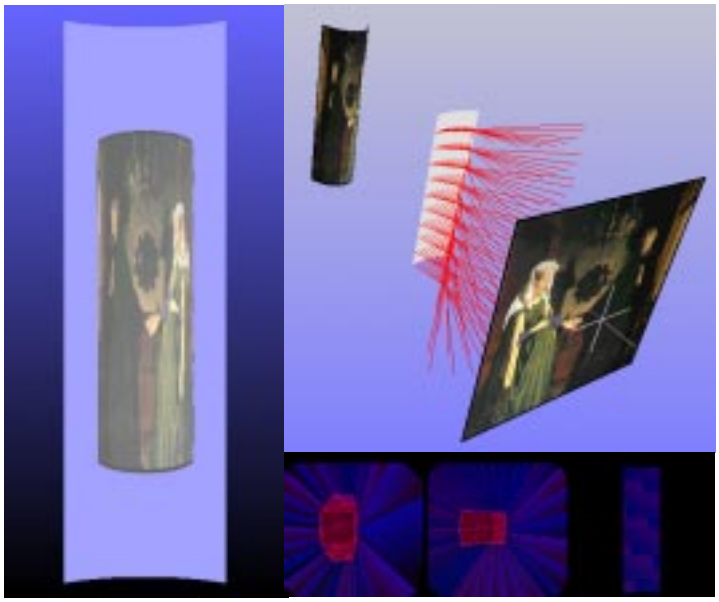


Fig. 11: Virtual object and reflection rays.

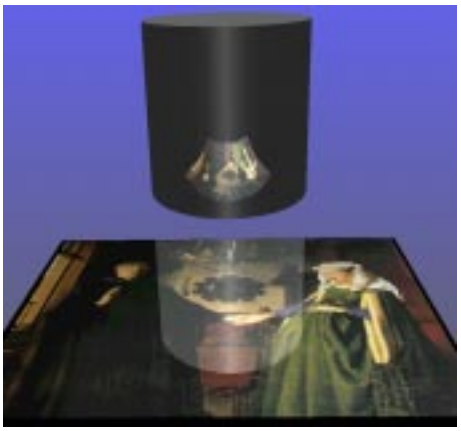


Fig. 12: Linear extrusion.

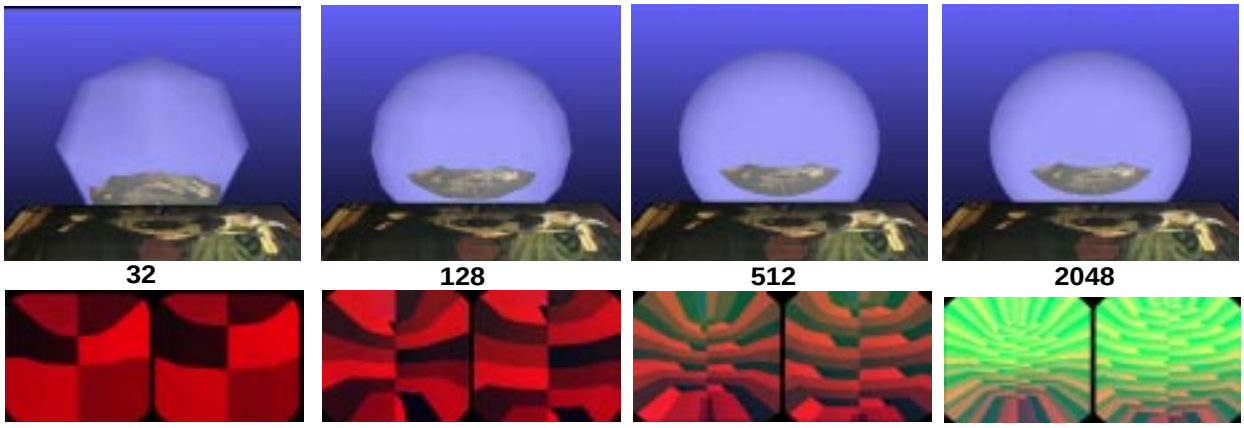


Fig. 13: Varying the tessellation resolution of the reflector.

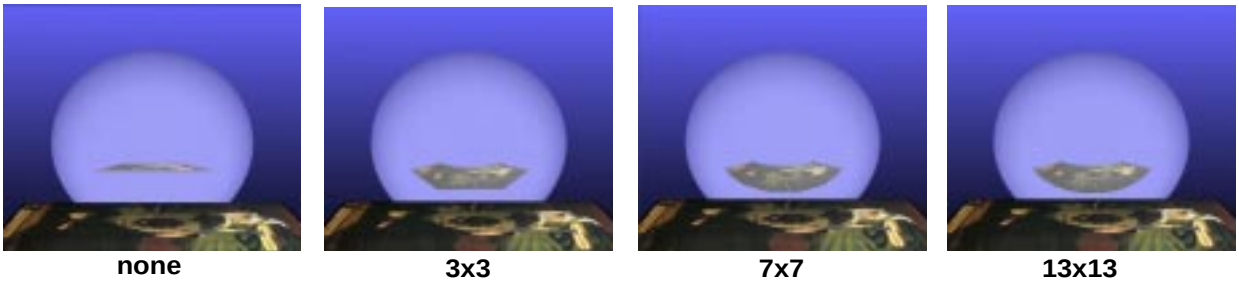


Fig. 14: Varying the tessellation resolution of the reflected object.

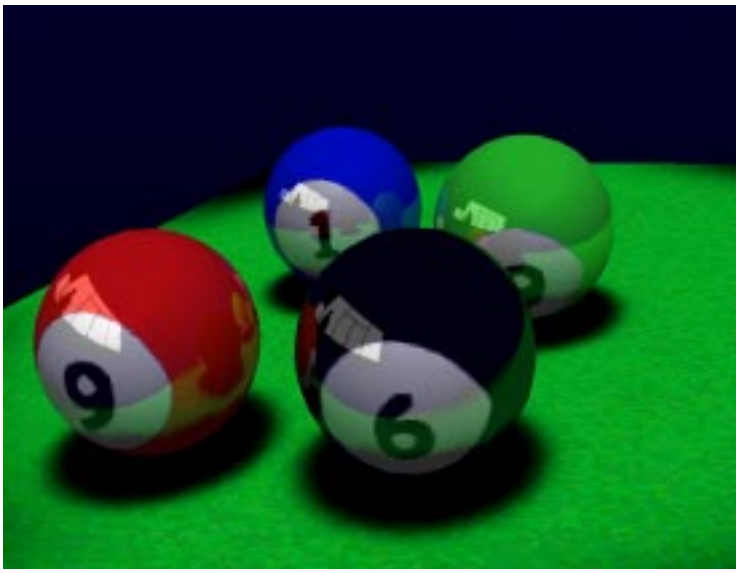


Fig. 15: Four reflecting spheres.

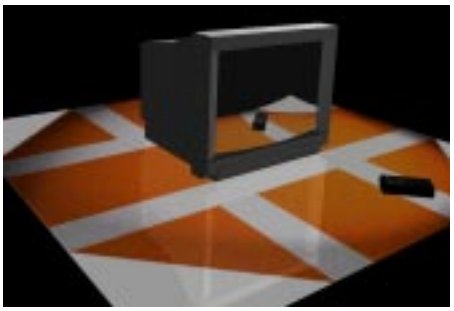
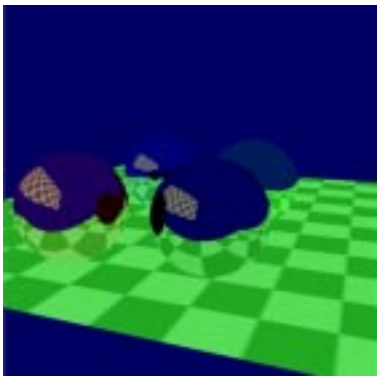


Fig. 17: TV.

Fig. 19: Reflector containing several convex and concave pieces.



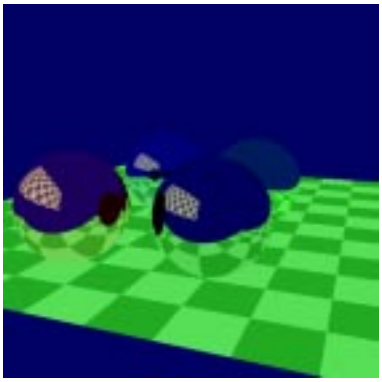


Fig. 16: Top: our method. Bottom: Rayshade.

Fig. 18: Recursive reflections.

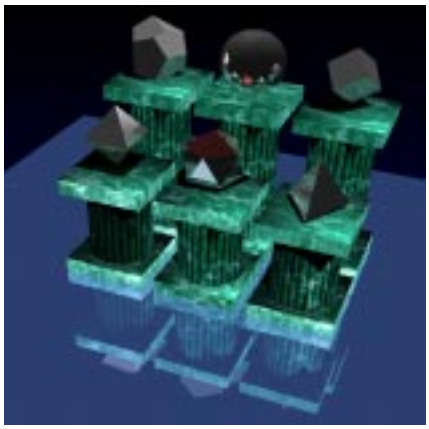


Fig. 20: Polyhedra and sphere.