Accurate Interactive Specular Reflections on Curved Objects

Pau Estalella[1], Ignacio Martin[1], George Drettakis[2], Dani Tost[3], Olivier Devillers[4], Frederic Cazals[4]

University of Girona[1] Girona, Spain

REVES/INRIA[2] Sophia-Antipolis, France

UPC[3] Barcelona, Spain

GEOMETRICA/INRIA[4] Sophia-Antipolis, France

Abstract

We present a new method to compute interactive reflections on curved objects. The approach creates virtual reflected objects which are blended into the scene. We use a property of the reflection geometry which allows us to efficiently and accurately find the point of reflection for every reflected vertex, using only reflector geometry and normal information. This reflector information is stored in a pair of appropriate cubemaps, thus making it available during rendering. The implementation presented achieves interactive rates on reasonably-sized scenes. In addition,we introduce an interpolation method to control the accuracy of our solution depending on the required frame rate.

1 Introduction

Real-time reflections from curved objects are an important visual cue for synthetic images. We are used to seeing the reflections of the rest of the environment, in any real-life setting containing shiny or mirror-like reflectors. In addition, reflections have an important aesthetic value, which designers of virtual environments would like to exploit, for example in VR applications for design and training, or computer games. Depending on the application, the *accuracy* and *speed* of computation of these reflections are very significant; either or both of these factors severely restrict the use of reflections on curved objects in most real-time applications.

Despite advances in graphics hardware and corresponding algorithms, it is currently not possible to render accurate real-time reflections for curved objects. Ray-tracing, and in particular PC-cluster-based solutions [15] have shown impressive results: it is fair to say however, that these solutions still cannot produce real-time reflections on a single workstation for high-resolution images. Another solution to real-time reflections on curved objects is environment mapping, and in particular the recomputation of the maps at every frame for every reflector (e.g., [11]), to accommodate moving objects. This appears to be the solution of choice in recent computer games: however, it is costly (4-6 rendering passes of the entire environment per reflector), and can be very inaccurate for reflected objects close to the reflector, or in regions of the reflector with high-curvature.

The idea of using multiple passes to render reflections using graphics hardware was proposed in [5], for planar reflectors. The idea is to compute the geometry of the reflected “virtual objects” and render these in a second pass. An extension to curved objects was proposed by [13]; however, this approach does not provide a guarantee of accuracy and implies a non-negligeable run-time cost per reflector.

In this paper, we present a novel solution to computing real-time reflections on curved objects which exploits information locally available at render time.

Our main contributions are: (i) The computation of real-time reflections with controllable accuracy for a large class of curved objects (convex or concave objects), (ii) a demonstration of geometric properties of the geometry of reflection, allowing the definition of an error function and an associated search algorithm, which allows us to find reflection points rapidly (iii) the introduction and use of position/normal reflector cube maps, resulting in a local search algorithm without access to the geometry of the scene, (iv) an approximation method that performs the search of the reflected positions of the vertices of a bounding grid of the reflectors in order to quickly compute the reflected vertices of the mesh by interpolation.

We next discuss previous work and then present some basic concepts. After describing our algorithm and presenting the results, we conclude with a discussion of the advantages and shortcomings of the proposed approach.

2 Previous Work

The most widely-used approach for rendering specular reflections is ray-tracing [16, 6]. Despite recent efforts, real-time ray-tracing (e.g., [15]) is still not available on simple, stand-alone workstations. Therefore its use in real-time animation and in video-games is still limited.

For planar surfaces, an alternative is the Multipass Pipeline Rendering (MPR) proposed in [5]. It consists of recursively rendering the scene using the hardware-provided pipeline, adding additional levels of reflections at each pass. The authors define a *virtual viewpoint* for each mirror surface computed by reflecting the viewer using the beam-tracing specular transformations [9]. The mirror images must be recomputed if the reflectors or the reflected surfaces move.

In complex environments such as outdoor scenes with trees and plants, the projection step for each mirror surface can be avoided by using precomputed radiance maps (e.g., [1]). Radiance maps are reflected images of the scene from different viewpoints, computed in a pre-process, and used at runtime applying warping techniques. However, this improvement is of little use in dynamic environments since radiance maps must be recomputed if the objects of the scene move.

Many specular surfaces such as car bodywork, pans and panoramic glass walls, are curved and therefore not suitable for the MPR strategy. Environment Maps (EM) [2] provide simple approximations of reflections on curved objects. They assume that the environment is infinitely distant from the reflector and, therefore, that the reflections on the object surface are similar to those computed from a central point of the object. An omnidirectional image of the environment is thus computed by projecting through the center of the object onto a cube- or sphere-map [7]. Parameterized Environment Maps (PEM) [8] improve traditional EM by adding self reflections. A PEM is a sequence of EM recorded over a set of viewpoints. The EM are computed with ray-tracing and segmented into several layers that separate different shading terms and local and distant parts of the environment. PEMs improve the realism of the reflections but increase the cost of the pre-processing stage dedicated to their computations, which is view-independent and thus must be re-computed if the reflector or the reflected surfaces move.

In [3], environment maps are precomputed for different viewpoints and warped at render time; an approach which also warps but incrementally updates pre-computed EM for dynamic scenes was presented in [10]. Modern graphics hardware allows render-to-texture capabilities to be used to recompute EM at each frame for each reflector, with real-time performance [11]; the approach however becomes expensive for scenes with a large number of reflectors. The last three methods can treat dynamic scenes, but are still inaccurate for reflections of close-by objects.

An analytic approach for the computation of virtual vertices on curved reflectors based on the path perturbation theory, is described in [4]. It is based on a pre-processing step in which sparse sets of rays are traced in the scene using standard ray-tracing. At run-time, the actual reflection points are computed by updating the reflection points of nearby pre-computed rays. This method operates on implicit surfaces, and has an relatively expensive preprocessing step, which needs to be recomputed when reflectors move.

In [13] a solution is proposed for reflections on curved reflectors, extending the idea of MPR. The virtual object is calculated by computing the virtual vertices of all its polygons and topologically connecting them. The approach is based on a tessellation of reflectors into triangles, and the computation of a data structure called the *explosion map* to accelerate the computation of virtual vertices. The explosion map is re-computed for each reflector and viewpoint. Using this structure, it is possible to quickly determine if the reflection of a point of the reflected surface is visible for the given viewpoint, and if so, to compute an approximate reflection plane tangent to the reflector in order to mirror the point about it. This approach is the most closely related previous work, and we will compare it to our solution in Section 8.

3 Overview of Computation for Virtual Reflected Objects

Our method uses the concept of virtual object proposed in [5]. During rendering, for each reflector, and each reflected mesh, we create a virtual mesh that we render giving to the observer the illusion of viewing the scene reflected on the curved reflector.

Consider a curved reflector ρ and a point V which will be reflected by ρ. We want to find the *reflection point* R on ρ so that we can compute the virtual vertex V' and render the reflection. We base the search of R on the following property.

Let P be a point on the reflector ρ. The normal at point P is defined as N_P. The observer position is O. The normalized *bisector vector* B_P is the bisector of the angle $\widehat{OPV}$. We also define the *bisector direction vector* B'_P to be the vector defined from the end of unit vector N_P to the end of unit vector B_P.

The *reflection point* R is the point on the reflector surface with normal N_R, such that the vector B_R coincides with vector N_R. No other point on the reflector's surface has this property. All these quantities are shown in Fig. 1.

Therefore, to compute the virtual vertex V', we perform a search to find the position of the *reflection point* R. We start with a point P on the surface, and we use an error function which indicates how far P is from the reflection point R. We define the error function as the angle (Fig. 1) between the bisector B_P and the normal N_R.

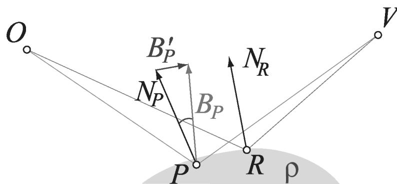

Figure 1: The basic geometry used: the observer O, a point P on the reflector, a vertex V which will be reflected on the reflector ρ, the normal N_P and the bisector B_P. The point R is the reflection point, at which $B_R = N_R$, by definition.

We want to perform this search using information which is available at render time, and with minimal overhead. Given O and V, and a starting point P on the reflector, we will use a pair of cube maps to search for R. These maps, which we call *reflector cube-maps*, encode positions on the reflector surface and their corresponding normals. We will search for R within these maps.

In the next section, we explain how we create the reflector cube maps and give details of the error function and its properties. We then present the search algorithm to find the reflection point given these maps. We next describe an approximation method based on interpolation that reduces the number of searches to perform, followed by the treatment of special-cases. We conclude with results and a discussion of our approach.

4 Finding the reflection points

4.1 Reflector Cube-Maps

The cubes maps are an encoding of points on the reflectors (position map) and normals and will be used during the search of the reflection point at rendering time. The creation of the reflector cube maps is a pure pre-processing step. It is completely independent of the view and the rest of the scene, and thus can be computed once and stored with each object during modelling.

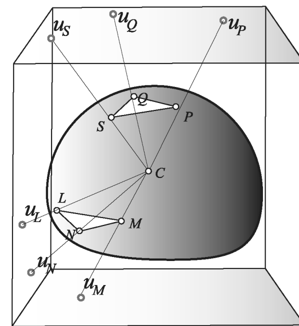

Figure 2: Projection of reflector's triangles against the reflector cube map faces.

To create the reflector position and normal cube maps, we project a vertex P of the reflector by intersecting the ray defined by the center C of the reflector and P with the appropriate cube-map face (see Fig. 2). From now on, we use u_P to designate the 2D cube map texture point corresponding to P.

To draw the position map we assign each vertex's 3D position to a vertex color. To draw the normal map we assign the vertex's normal to a vertex color. This way the triangles drawn are filled with linearly interpolated 3D positions and normals, which should ideally match the reflector's surface.

An important feature of our approach is the fact that this map can be created with a highly tessellated

version of the reflector, which can be different from that used for rendering. The accuracy of the reflections will thus be higher, without additional cost at rendering time.

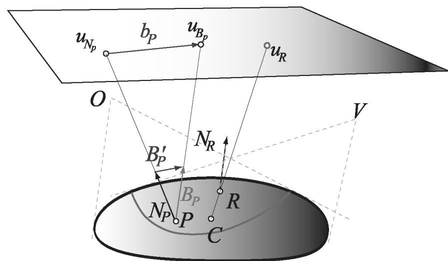

Figure 3: Definition of the projected bisector direction b_p.

4.2 The error function

Recall that the error function must be determined using the local properties available during rendering, i.e., the positions of the reflector and the normal. For any point P on the reflector's surface, we can compute the angle between B_P and N_P. In practice, we compute the dot product $B_P \cdot N_P$ of the normalized vectors, and use the resulting value between minus one and one as the error value. The error function is thus defined as follows:

$$E(P) = (1 - B_P \cdot N_P)/2 \tag{1}$$

The error function has a zero at the reflection point and a value of one when B_P and N_P point in opposite directions.

We call *region of interest*, the region bounded by the bitangents (red dashed lines in Fig. 3) corresponding to O and V on the reflector; This is the region in the interior of the green line in the figure. Points outside this region cannot reflect V. We will use the following theorem in what follows.

In the region of interest and for a convex reflector the error function E has a unique minimum at R, by definition of the reflection point. The projection of this vector on the cube map is the *projected bisector direction* b_P. Vector b_P can also be defined by the endpoints of the intersections of the rays defined by N_P and B_P with the cube-map face. We use the projected bisector in the search for the minimum later. These quantities are illustrated in Fig. 3.

For a given point P in the region of interest, if we follow the bisector direction vector B'_P, by less than the length PR, to a new point Q, we approach the reflection point R. This is proved in a separate document annexed to the submission.

Thus, to find the reflection point R, we use an algorithm which follows the vector B'_P. If we knew PR, we could guarantee that we advance towards to minimum of the error function; since this information is not available, we will approximate it, and demonstrate that the method works well in practice.

The vectors B'_P define a vector field over the surface of the object, and b_p defines the vector field $V(s,t)$ in the cube-maps. For a given configuration of a reflector, an observer O and a viewer V, we can visualize vector field $V(s,t)$ in a region of the cube map close to the projected reflection point u_R (see Fig. 4(left)). We can see that, as expected, the vector flow lines lead to u_R.

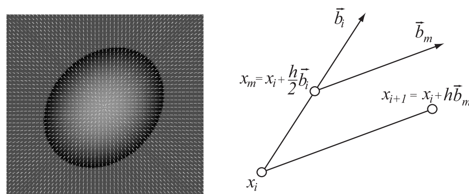

Figure 4: Left: Visualisation of the vector field. Right: Illustration of search algorithm.

4.3 Search algorithm

The algorithm operates on the reflector cube maps, with the terminating condition of the error function (Eq. 1) being below a given threshold ϵ.

At a given point u_P, we obtain N_P directly from the surface normals texture, and B_P is computed as the bisector of the angle $\widehat{OPV}$, with O and V the observer and the vertex being reflected, and P the position extracted from the surface positions texture (see Fig. 1). We thus obtain the projection bisector direction b_P.

As illustrated in Fig. 4(left), the vector field $V(s,t)$ has a "sink"; following the vectors of the field to reach this sink is a well-known numerical problem. This sink, where $V(s,t)$ is 0, is precisely the reflection point. Finding the minimum, and thus the reflection point, reduces to solving the equation

$V(s,t) = 0$. We use the midpoint method (second order Runge-Kutta) with adaptive step size to solve this equation. Our approach requires just two texture evaluations per step, while giving the desired accuracy.

The method works as follows: We compute the projected bisector b_i at the current point, the projected bisector b_m at half the step distance in the direction of b_i, and move a full step length in the direction of b_m (see Fig. 4(right)). We adjust the step length depending on the difference of b_i and b_m. If the vectors are very different, we reduce the stepsize. If they are very similar, we increase it.

5 Interpolation-Based Optimization

Computing the virtual meshes by directly applying the search algorithm to all the reflected mesh vertices has two drawbacks. First, the computational cost is linear with the number of mesh vertices of the reflector. It does not depend on the actual size of the reflection areas in the image. Thus, when the reflection is small in the current view, which is quite often with curved reflectors, most of the computation is simply wasted. A second drawback is that although the search algorithm is precise, since the number of iterations is fixed, the solutions have variance, which may produce visible artifacts such as flickering during animation.

In order to adjust the rendering cost to the real needs of precision and to remove artifacts, we have developed the following optimization. For each mesh M, in a pre-process, we compute a regular isothetic bounding grid $G(M)$ of $N_x \times N_y \times N_z$ cells. We store only the set of cells of the grid that contain at least one vertex of the mesh.

In a pre-process, for each vertex of the mesh, we find the grid cell of $G(M)$ to which the vertex belongs and we compute the vertex local coordinates (u, v, w) inside the cell. During rendering, for each reflector ρ and each mesh M, we create a virtual grid $G_\rho(M)$ by computing the virtual vertices of all the grid vertices using the search algorithm described above. Then, we compute the coordinates of the virtual position of all the mesh vertices by interpolation using the local coordinates (u, v, w) of each vertex in its corresponding virtual cell.

In the current implementation, the number of cells per grid axis N_x, N_y and N_z is chosen such that the total number of cells is a constant user-defined fraction of the number of vertices of the mesh, and such that the cells are as close to cubes as possible in order to provide a smooth interpolation.

6 Special Cases

The basic algorithm described above gives the general principle of our approach. A case that requires special treatment is that of triangles which are partially hidden by the reflector: we explain how to deal with this case. Our approach currently has some requirements on the geometry of the reflectors; we discuss these briefly.

6.1 Hidden vertices

Given an observer and a convex reflector, we can partition the space into three different regions, following the naming convention in [13] (see Figure 5(left)). The first region is called the reflected region, corresponding to the subset of space seen by the observer, reflected by the reflector. The second region is the hidden region, the subset of the space occluded from the observer by the reflector. In the case of a closed reflecting surface, the union of these two regions is the entire space. For open reflecting surfaces, the union of these two disjoint regions leaves out the so called unreflected region.

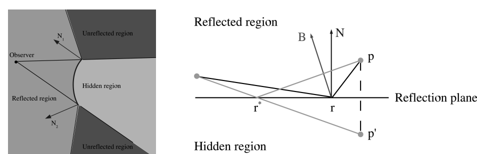

Figure 5: The reflected (green), hidden (cyan) and unreflected (red) regions. Their respective extents are related to observer position and reflector shape (blue.) Computation of bisector B during the inverse search requires computing the reflection point p of p' w.r.t. the current normal N and reflector surface point r.

When computing the virtual vertices for reflections, scene triangles that lie completely in the hidden or unreflected regions can be discarded without further work. Triangles that lie partially in these regions and partially in the reflected region must be processed, and we must find reflected positions for the vertices outside the reflected region.

For any given point P in the reflected region we have one and only one point P' in the hidden region that corresponds to the virtual specular reflection point. Symmetrically, for every point P' in the hidden region, we have one and only one point P in the reflected region. For a given point P' in the hidden region, we wish to find the reflection point using the same algorithm. To do this, we first compute the reflection point p of p' to compute the bisector B, as can be seen in Figure 5(right). This operation is performed at every step of the iterative search for the reflection point. The rest of the search proceeds as before, allowing the algorithm to find the reflection point R.

6.2 Geometry Preprocessing

We currently have some restrictions on the input. For open reflectors, we require an accompanying closed object in the model so we can fill the reflector cube maps. Since open objects are often modeled as trimmed closed objects, this is not a major restriction. In the long run, an automatic preprocessing algorithm could perform this step. We also require objects to be segmented into concave or convex pieces. Again, this is is currently done manually as a preprocess.

7 Results

We present results of our approach for a scene representing a kitchen. Clearly, since this paper is about real-time reflections, the results are best seen in the accompanying video. The video shows viewer navigations in the kitchen varying the number of grid vertices. It show be noted that the frame rates are slightly lower than the real ones, because of the frame capture and composition software.

The two left-most image of Fig. 6 show images computed using our approach performing the search for the full set of vertices and approximating the search by interpolation, using a grid with a number of vertices of 10% of the number of mesh vertices. An environment map image of the same image is also shown along with a ray-traced image. The ray-traced image has been computed using NuGraph and the lighting conditions are not exactly the same than those of our software. However, the reflections in the ray-traced image can be taken as a reference solution. From Figure 6, it is clear that environment maps, even when recomputed at each frame, produce reflections which are very inaccurate for close-by objects. The image of the scene computed applying the search for the full set of vertices shows artifacts caused by the fixed resolution of the iterative search. The image computed by interpolation and using the search for the grid vertices removes these artifacts and gives a smooth, visually pleasant reflection. Our method produces results which are accurate to the precision of the subdivision of the objects in the scene. The "Kitchen" scene of Fig. 6 with the approximation strategy runs at an average of 10 fps and at 1.3 fps if the search is performed for all vertices. Varying the number of selected vertices in Figure Fig. 7 the rates are 16.3 fps for 4.1% exact vertices, 13.6 fps for 5.5%, 10.1 fps for 9.6% and 8.3 fps for 14.3%, respectively. We have observed that above 10% the quality difference is not noticeable. Finally, Figure Fig. 8 shows the same view computed with no grid interpolation, and using different number of iterations for the vertex search.

All results are reported on a Pentium Xeon 3.2Ghz with an Asus A400 GeForce 6800 graphics card running Linux RedHat 9.

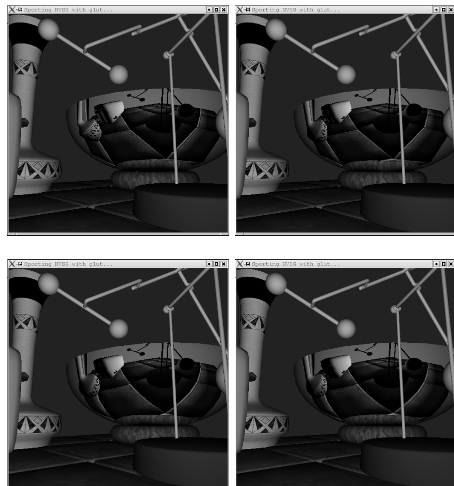

Figure 7: The Kitchen scene. Same view with different densities for the grids. The number of grid vertices is a fixed percentage of the number of mesh vertices. Top left uses 4.1%, top right uses 5.5%, bottom left uses 9.6%, and bottom right uses 14.3%

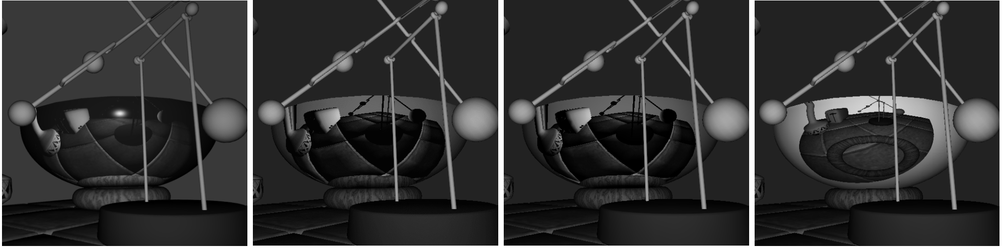

Figure 6: The Kitchen scene. From left to right. The reference ray-traced image, the scene rendered using our approach searching for all the vertices, the scene rendered using our approach, searching for only 10% of the vertices and approximating the others and the scene using dynamic environment maps (i.e., recomputed at each frame). Our approach for full search as well as with the approximation is clearly much closer to the ray-traced solution than DEM.

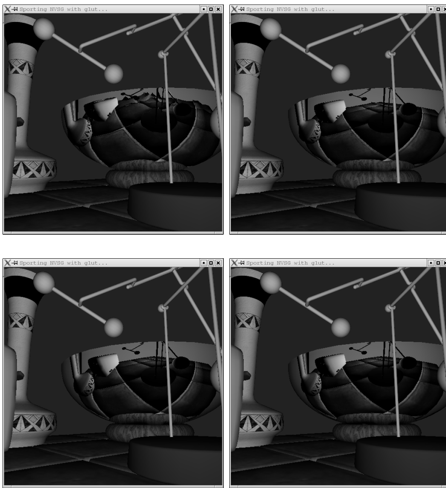

Figure 8: The Kitchen scene. Same view with different number of iterations used for the local search. No grid used, all vertices are reflected. Top left uses 3 iterations, top right uses 5 iterations, bottom left uses 7 iterations, and bottom right uses 9 iterations. Using more iterations gives no noticeable improvements.

8 Discussion and Conclusions

The results show that our approach provides very accurate reflections on curved objects, with real-time performance.

Moreover, the approximation mechanism speeds up rendering allowing users to tune the accuracy of the reflections. As a consequence, our approach is sufficiently fast and produces accurate reflections.

In terms of existing alternatives, the accuracy of environment maps, even if they are computed at every frame, is simply insufficient for objects close to the reflector. Although we do not have a full comparison with the method presented in [13], the authors point out that their method will often find an incorrect triangle in an explosion map, and thus the reflection point found will be incorrect. The method could be improved with an additional search [12] to find a better approximation to the reflection point. However, an efficient implementation, which would be implementable in hardware, would require a structure similar to our reflector cube maps, since the explosion maps do not contain normal information, and the geometry stored in the explosion map is reflected. In terms of computation cost, we believe that the computation of the explosion map and our search for reflection points have comparable cost, but with higher accuracy.

In terms of limitations, our approach cannot handle torus-like geometries for the reflectors; however no other real-time alternative exists. Our current implementation requires a certain amount of preprocessing for reflectors (see Section 6.2). Currently, edges of large polygons are not correctly reflected, since we only reflect the endpoints. Also, the cost of the additional rendering pass per reflector is currently significant, since every vertex has to be reflected on every reflector. However, we believe that these problems can be addressed, and although they currently limit the use of our approach, they are not major hurdles.

Using a level-of-detail (LOD) grid hierarchy could greatly reduce the cost of the reflection computation. The advantage is that the size of the grid determines the quality of the reflection. The size

of the grid should be selected proportionally to the area of the reflection in image space for each view. Other accelerations are possible, for example only doing the computation for visible reflectors, or using other culling techniques.

Another improvement concerns the requirements for geometry processing the input (see Section 6.2). This step is currently done manually; automating this processing is an important avenue of research.

The current implementation does not treat multiple reflections, but a straightforward extension of our approach could handle this. We believe however that it would first be necessary to implement the LOD improvement to make such an approach feasible. Computation of refraction would require the study of the analytic properties of the equivalent error functions, but is most probably possible.

Finally, a hardware-based version of the proposed approach is currently being implemented for both the search of the exact vertices and the interpolation of the approximated vertices.

References

[1] Rui Bastos, Kenneth Hoff, William Wynn, and Anselmo Lastra. Increased photorealism for interactive architectural walkthroughs. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 183–190. ACM Press, 1999.

[2] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547, October 1976.

[3] Brian Cabral, Marc Olano, and Philip Nemec. Reflection space image based rendering. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 165–170. ACM Press, 1999.

[4] M. Chen and J. Arvo. Perturbation methods for interactive specular reflections. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):253–264, July/ September 2000.

[5] Paul J. Diefenbach and Norman I. Badler. Multi-pass pipeline rendering: realism for dynamic environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 59–ff. ACM Press, 1997.

[6] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1989.

[7] Ned Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11), November 1986. revised from Graphics Interface '86 version.

[8] Ziyad S. Hakura, John M. Snyder, and Jerome E. Lengyel. Parameterized environment maps. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 203–208. ACM Press, 2001.

[9] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 119–127, July 1984.

[10] Alexandre Meyer and Céline Loscos. Real-time reflection on moving vehicles in urban environments. In *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 32–40. ACM Press, 2003.

[11] Kasper Høy Nielsen and Niels Jørgen Christensen. Real-time recursive specular reflections on planar and curved surfaces using graphics hardware. In *Journal of WSCG*, volume 10, pages 91–98, Plzen , Czech Republic, February 2002. University of West Bohemia.

[12] Eyal Ofek. *Modeling and Rendering 3-D Objects*. PhD thesis, Institute of Computer Science, The Hebrew University, 1998.

[13] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 333–342. ACM Press, 1998.

[14] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge (UK) and New York, 2nd edition, 1992.

[15] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques*, pages 277–288, 2001.

[16] Turner Whitted. An improved illumination model for shaded display. *CACM, 1980*, 23(6):343–349, 1980.