EUROGRAPHICS 2006 / E. Gröller and L. Szirmay-Kalos
(Guest Editors)

Volume 25 (2006), Number 3

Accurate Specular Reflections in Real-Time

David Roger and Nicolas Holzschuch

ARTIS–GRAVIR† IMAG INRIA

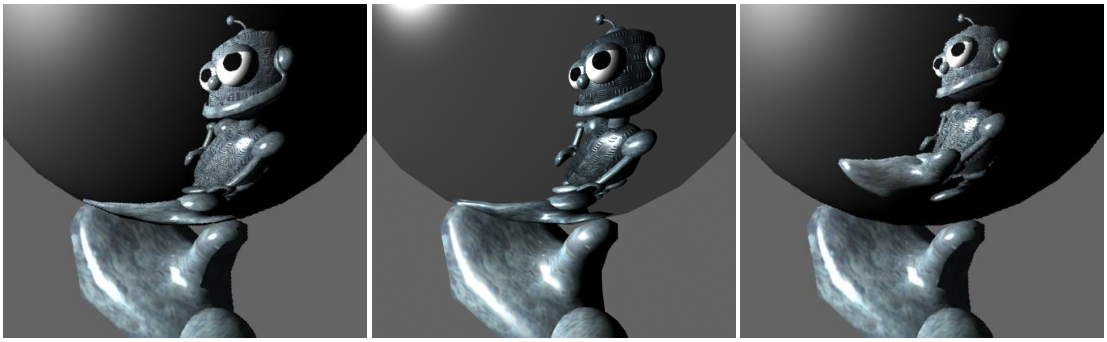


Figure 1: Left: Specular reflections computed with our algorithm. Middle: ray-traced reference. Right: Environment map reflection.

Abstract

Specular reflections provide many important visual cues in our daily environment. They inform us of the shape of objects, of the material they are made of, of their relative positions, etc. Specular reflections on curved objects are usually approximated using environment maps. In this paper, we present a new algorithm for real-time computation of specular reflections on curved objects, based on an exact computation for the reflection of each scene vertex. Our method exhibits all the required parallax effects and can handle arbitrary proximity between the reflector and the reflected objects.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

1. Introduction

Reflections on specular objects are important in our perception of a synthetic 3D scene. They convey important information about the specular reflector itself, conveying its shape and its fabric. They can also give information about the relative spatial positions of objects or the distance between the reflector and the reflected object. Finally, they give information about objects that are not directly visible (see Figure 1).

Real-time computation of specular reflections is usually done using environment mapping. While these techniques perform quite well in a wide variety of cases, they have their shortcomings. They perform best if the reflected object is at a large distance from the reflector, but as the reflected object moves closer to the specular reflector, reflection errors become more visible. The worst case for environment mapping techniques is when the reflector is in contact with the object being reflected, as in Figure 1. Environment mapping technique also suffer from the parallax problem: from all the points on the specular reflector, we are seeing the same side of the reflected objects, even if the specular reflector is large enough to see the different sides of an object.

In this paper, we present a new method for computing specular reflections. Our method is *vertex based:* we com-

† GRAVIR is UMR 5527 GRAVIR, a joint research laboratory of CNRS, INRIA, INPG and UJF.

pute the accurate reflected position of each vertex in the scene, then interpolate between these positions. The advantage of our method is that it is computing the reflection of the object depending on the position on the reflector. We are therefore exhibiting all parallax effects, and we can handle proximity and even contact between the reflector and the reflected objects.

However, our method also has obvious limitations: as it is vertex-based and uses the graphics hardware for linear interpolation between the projections of the vertices, artifacts can appear if the model is not finely tessellated enough. These artifacts can be overcome using either adaptive tessellation or curvilinear interpolation. If the model is finely tesselated, these artifacts are not visible. Our algorithm provides solutions for situations where no convincing solutions existed before.

Our paper is organized as follows: in the next section, we review previous work on real-time computation of specular reflections. Then, in section 3, we present our algorithm for computing vertex-based specular reflections on curved surfaces. In section 4, we present experiments on various scenes and comparisons with existing methods. Finally, in section 5, we conclude and present future directions for research.

2. Previous Works

Ray-tracing has historically been used to compute reflections on specular objects. Despite several advances using either highly parallel computers [WSB01, WBWS01, WSS05] or GPUs [CHH02, PBMH02], ray-tracing is not, currently, available for real-time computations on a standard workstation.

Planar specular reflectors are easy to model, at the cost of a second rendering pass, with a camera placed in the mirror position of the viewpoint [McR96]. Curved reflectors are more complex; the easiest method uses environment mapping [BN76].

Environment mapping computes an image of the scene and maps it on the reflector as if it was located at an infinite distance. The reflection only depends on the direction of the incoming vector from the viewpoint, and can be easily computed in real-time on graphics hardware. Obviously, environment mapping suffers from parallax issues, since the reflection depends on a single image computed from a single point of view. There is also the question of accuracy: since all objects are assumed to be at an infinite distance, their reflection is not necessarily accurate, and the difference becomes larger as the object gets closer to the reflector.

There has been much research to improve the original environment mapping algorithm. To remove the parallax issues, Martin and Popescu [MP04] interpolate between several environment maps. Yu *et al.* [YYM05] used an *environment light-field*, containing all the information of a light field, but organized like an environment map. Both methods remove parallax issues, at the cost of a longer precomputation time. The specular reflector is also restricted, and can only be moved inside the area where the light field or the environment maps were computed. If it is moved outside of this area, the environment light field must be recomputed, a costly step.

Other research have dealt with distance-based reflection. The simplest method is to replace the infinite-radius sphere associated with the environment map by a finite-radius sphere [Bjo04]; the reflection changes with the position of the reflector in the environment, but parallax effects can not be modeled.

More accurate methods use the Z-buffer to compute a distance map along with the environment map. For each pixel of the environment map, they know both its color and the distance to the center of the reflected object. Patow [Pat95] and Kalos *et al.* [SKALP05] used this information to select the proper pixel inside the environment map. Their reflections change depending on the distance between the reflector and the reflected object. Kalos *et al.* [SKALP05] use the GPU for a fast computation of the reflected pixel, and achieve real-time rendering for moderately complex scenes. Still, image based methods are inherently limited to the information included in the original image.

For planar reflectors, the easiest way to compute the reflection is vertex-based, using an alternative camera to compute the image of the scene as reflected by the planar reflector. For curved reflectors, there is no simple rule to tell the position of the reflection of the objects. Even for a finite-radius sphere, the simplest specular reflector, the position of the reflection depends on a 4th-order polynomial.

Mitchell and Hanrahan [MH92] used the equation of the underlying surface to compute the characteristic points in the caustic created by a curved reflector. Ofek [Ofe98] and Ofek and Rappoport [OR98] computed the *explosion map* to find intersected triangles ID based on the reflected vector. Chen and Arvo [CA00b, CA00a] used ray-tracing to compute the reflection of some vertices, then applied perturbation to these reflections to compute the reflection of neighboring vertices.

Estalella *et al.* [EMD*05] computed the reflection of scene vertices on curved specular objects by an iterative method. At each iteration, the position of the reflection of the vertex is modified, using the angles between the normal, the vertex and the viewpoint, in the direction where these angles will follow Descartes' law. They did a fixed number of iterations, and have implemented the method only on the CPU. In a subsequent work, developed concurrently with ours, Estalella *et al.* [EMDT06] extended this work to the GPU, searching the position of the reflection of the vertex in image space.

Our method is comparable to that of Estalella *et al.* [EMD*05, EMDT06], but we use a different refinement

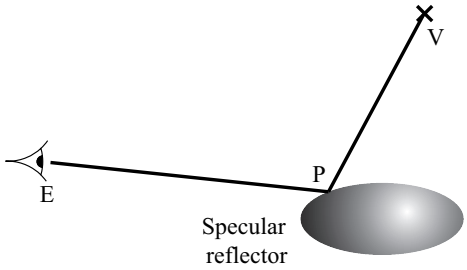


Figure 2: *Finding the reflection of a given vertex*

criterion, keeping geometric bounds on the reflected position for robustness. We use these geometric bounds for adaptive refinement, stopping the iteration as soon as we reach sub-pixel accuracy. In our experience, these two elements are of great importance: in all the scenes we used, we encountered robustness-related issues, especially for reflections at grazing angle. We also noticed that the number of iterations required to reach convergence varies greatly with the position of the reflection.

3. Algorithm

3.1. Principles

Our algorithm is vertex-based: we compute the reflected position of all the scene vertices, then let the graphics hardware interpolate between these vertices and solve visibility issues with a Z-buffer. Our algorithm therefore inserts itself as a replacement for the usual projection of the vertices. Knowing the position of the viewpoint, E, for each vertex V, we find the point P on the specular reflector that corresponds to the position of V (see Figure 2).

The difficult part in this algorithm is computing P as a function of V and E. Except in the most basic case of planar specular reflectors, there is no simple relationship between P, V and E. Even for a sphere, the explicit position of P depends on a polynomial of the fourth order; finding the roots of this polynomial is feasible, but takes actually longer than the iterative method we use.

According to Fermat's principle, light travels along paths of extremal length, so P must correspond to an extremum of the optical path length $\ell = EP + PV$. We are searching for extrema of ℓ, or equivalently, for zeros of its first order derivative, the gradient $\nabla\ell$.

This is an optimization problem, with a function of two parameters (the surface of the specular reflector is a 2D manifold). Usually, optimization problems are solved with *line search* methods, such as the gradient descent or the conjugate gradient methods. These method progress iteratively from an initial guess. At each step, they know the *direction* in which they should progress, but not necessarily the *distance* along this direction. Knowing this distance accurately requires knowledge about the second derivatives of the function.

Our application is inherently graphical: we are displaying the result of our computations on the screen, and changing parameters — the viewpoint, the reflected scene, the reflector — dynamically. One of the most important points for such graphical applications is temporal coherency: the reflection of one point must not change suddenly between frames. We therefore need *spatial* information about the accuracy of the computations: if we have not yet computed the position of one point with sub-pixel accuracy, we run the risk of seeing temporal discontinuities at the next frame. We also observed in our experiences that the number of iterations required for convergence varies greatly with the configuration of the vertex. Spatial information about convergence help in adapting the number of iterations to the current case.

Line search methods typically use residuals to check the numerical accuracy of the computations, but they do not provide information about the spatial accuracy. At each step, we know the distance traveled from the previous step, but this information is only *linear*. Since the reflector is a 2-dimension surface, it can happen that the algorithm has closed in on the result along one dimension, but is still far from it on the other dimension.

The *secant method* searches for roots of one function f by replacing it with a linear interpolation between samples, picking the root of the linear interpolation and iterating. While the secant method does not guarantee that the root remains bracketed, it provides a good information about the accuracy achieved so far, and converges faster than the simpler bisection method. Newton's method converges faster than the secant method, but requires computing the derivative of f.

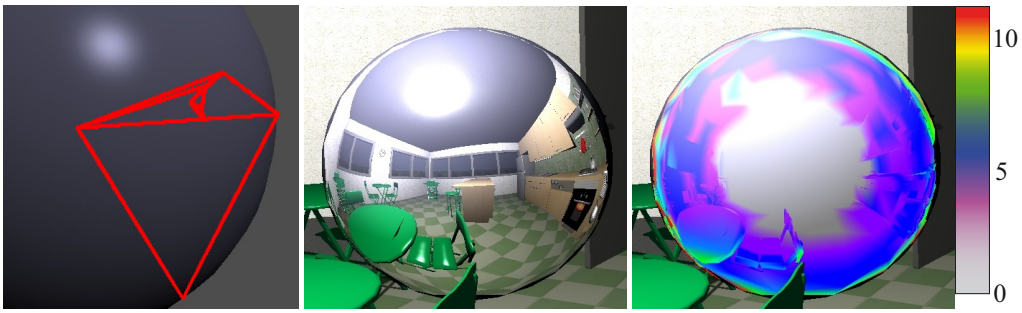
Since we are looking for zeros of $\nabla\ell$, we apply to it a variant of the secant method. At each step, we maintain a triangle of sample points where we compute $\nabla\ell$ and linearly interpolate between these gradients. At each step, the triangle of sample points gives us approximate geometric bounds on the projection of the vertex.

3.2. Algorithm for specular vertex reflection

Our algorithm for computing the reflection of a 3D scene in a specular reflector uses the following steps:

1. render the scene into the framebuffer, with direct lighting and shadowing;
2. for all vertices of the scene, find their reflection on the specular reflector;
3. interpolate between these vertices, computing lighting and doing hidden surface removal.

For each vertex, finding the position of its reflection is done iteratively, using a variant of the secant method on the gradient of the optical path length: at each step, we maintain a triangle of sample points, and we:



(a) Example of successive triangles generated by our algorithm (b) Example image rendered with our algorithm (c) Number of iterations required for convergence

Figure 3: *Convergence of our iterative system.*

- compute the gradient of the optical path length for each sample point (see section 3.3.2),
- linearly interpolate between these gradients,
- find the resulting gradient with the smallest norm (see section 3.3.3).
- discard the original sample point with the largest gradient, replace it by the new sample point and iterate (see Figure 3(a)).

At each step, the projected area of the triangle gives us an indication of the accuracy of our computations. We stop the computation if this area falls below a certain threshold.

Our method converges quickly in most cases, in 5 to 10 iterations in moderately complex cases but can require up to 20 iterations for certain difficult points, such as vertices whose reflection is close to the boundary of the reflector (see Figure 3(c)).

The method is robust enough to converge even if the initial set of sample points is poorly chosen. However, it converges faster if the sample points are close to the actual solution. Section 3.3.4 describes our strategy for picking the initial sample points.

Once we have computed the reflection of each vertex, we project it on the screen and let the graphics hardware does linear interpolation between the vertices. We exploit the fact that we know the spatial position of the point being reflected to compute direction-dependent lighting (see Section 3.3.5).

Hidden surface removal requires special handling, as we have several possible sources of occlusion: the scene and the reflector may be hiding each other, parts of the reflector may be hiding themselves, and parts of the reflected scene are hiding other parts of the reflected scene. Section 3.3.6 describes our solution to these combined occlusion issues.

The entire algorithm was implemented on the GPU, using programmable capabilities for vertex and fragment processing. Hardware implementation issues are described in section 3.3.7.

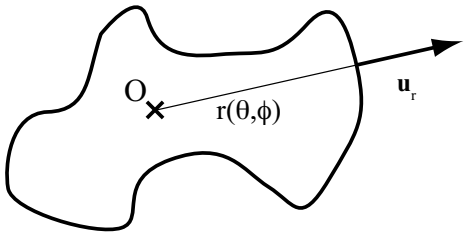


Figure 4: *To reduce dimensionality, we assume that the reflector is star-shaped.*

3.3. Details of the algorithm

3.3.1. Specular reflector parameterization

In order to provide interesting reflections, it is better if our reflector is actually smooth. We also assume that it is parameterizable. Finally, to reduce the dimensionality of the problem, we assume that the reflector is star-shaped: there is a point O that is directly connected to all the points on the surface of the reflector (see Figure 4).

This reduces the equation of the specular reflector to a scalar function, r. Using spherical coordinates, for example, all point $P(\theta, \phi)$ on the receiver can be expressed as:

$$P(\theta,\phi) = O + r(\theta,\phi)\boldsymbol{u}_r \quad \text{with} \quad \boldsymbol{u}_r = \begin{pmatrix} \sin\theta\cos\phi \\ \sin\theta\sin\phi \\ \cos\theta \end{pmatrix}$$

For our algorithm, we will also need the variations of the surface of the reflector: we also compute the derivatives of the function r.

In a preliminary step, r and its partial derivatives are computed and stored in a texture. Although our algorithm works with any kind of reflector, the star-shaped hypothesis allows us to retrieve all the required information about the specular reflector at any given point with a single texture read. This

will be useful for implementing our algorithm efficiently on the GPU.

Using spherical coordinates introduces singularities in the parameterization, at the poles. To avoid numerical issues in our computations, we do not use r or its partial derivatives directly, but we only use 3-dimensional vectors such as P or ∇r. All computations and interpolations are done in 3D space, never in parameter space.

3.3.2. Optical path derivatives

Assuming we have a sample point on the surface of the reflector, we can compute the length ℓ of the optical path length from the viewpoint E to the vertex V through P (see Figure 2):

$$\ell = EP + PV$$

The gradient of the optical path length depends on the derivative of point P on the reflector surface:

$$\begin{aligned} \nabla\ell &= \nabla(EP) + \nabla(PV) \\ \nabla\ell &= \mathrm{d}(P)\left(\frac{\overrightarrow{EP}}{EP} + \frac{\overrightarrow{PV}}{PV}\right) \end{aligned}$$

Here $\mathrm{d}(P)$ is the derivative of point P, a linear form operating on a vector. With our parameterization of P on a star-shaped reflector, $\mathrm{d}(P)$ is also reduced in dimension, and we can express $\nabla\ell$ as a function of ∇r:

$$\nabla\ell = (\nabla r \cdot \boldsymbol{e})\boldsymbol{u}_r + (\boldsymbol{u}_\theta \cdot e)\boldsymbol{u}_\theta + (\boldsymbol{u}_\phi \cdot e)\boldsymbol{u}_\phi \qquad (1)$$

with:

$$\boldsymbol{e} = \frac{\overrightarrow{EP}}{EP} + \frac{\overrightarrow{PV}}{PV}$$

$$\boldsymbol{u}_\theta = \begin{pmatrix} \cos\theta\cos\phi \\ \cos\theta\sin\phi \\ -\sin\theta \end{pmatrix} \qquad \boldsymbol{u}_\phi = \begin{pmatrix} -\sin\phi \\ \cos\phi \\ 0 \end{pmatrix}$$

∇r can be expressed as a function of the partial derivatives of r, but it is not actually necessary in our case. We are storing information about r and its derivatives in a texture, which will be accessed by the GPU. As a single texture read gives access to 4 channels, we store r and its gradient ∇r, saving computations.

3.3.3. Finding a better estimate for vertex reflection

At each step, we have a triangle of sample points (A, B, C). For all points D, expressed in barycentric coordinates with respect to (A, B, C):

$$D = \alpha A + \beta B + (1 - \alpha - \beta)C$$

we compute an approximation $\tilde{\nabla} d_D$ the gradient of ℓ using linear approximation:

$$\begin{aligned} \tilde{\nabla} d_D &= \alpha\nabla d_A + \beta\nabla d_B + (1 - \alpha - \beta)\nabla d_C \\ &= \alpha\boldsymbol{a} + \beta\boldsymbol{b} + \boldsymbol{c} \end{aligned}$$

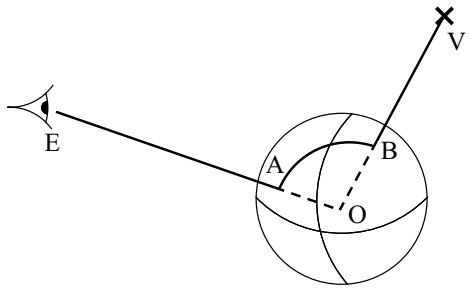


Figure 5: *On a sphere, the reflection lies on the arc* (A, B)

Ideally, we would like to select (α, β) such that $\tilde{\nabla} d_D = 0$. However, this is not always possible, unless the vectors $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ are linearly dependent. So we pick (α, β) so that $\|\tilde{\nabla} d_D\|$ is minimum: we derivate $\|\tilde{\nabla} d_D\|^2$ with respect to α and β, and find (α, β) such that both derivatives are null. This is equivalent to solving the linear system:

$$\begin{cases} \alpha\boldsymbol{a}^2 + \beta(\boldsymbol{a} \cdot \boldsymbol{b}) + (\boldsymbol{a} \cdot \boldsymbol{c}) &= 0 \\ \alpha(\boldsymbol{a} \cdot \boldsymbol{b}) + \beta\boldsymbol{b}^2 + (\boldsymbol{b} \cdot \boldsymbol{c}) &= 0 \end{cases}$$

whose determinant is:

$$\delta = \boldsymbol{a}^2\boldsymbol{b}^2 - (\boldsymbol{a} \cdot \boldsymbol{b})^2$$

The (α, β) parameters give us a new point D. We discard the point in (A, B, C) with the largest gradient and replace it with point D, then iterate.

In some circumstances, the determinant δ of the system can be null or very small, making the system ill-conditioned. When it happens, we backtrack in time, replacing one of the points $\{A, B, C\}$ by the most recently discarded point. Of course, we cannot replace the most recently added point, or the system would enter an infinite loop.

3.3.4. Initialization

Our method is efficient and converges even if arbitrary sample points are used as a starting triangle. However, the convergence is faster if the starting triangle is small and close to the result. It is not necessary for our initial guess to actually enclose the result, since our algorithm is able to extrapolate outside the triangle if necessary.

For a spherical reflector, the reflection of a vertex V is in the plane defined by V, the eye E and the center of the sphere O. Ofek [Ofe98] shows that the reflected vertex is bound on the arc of circle $[AB]$ where A (resp. B) is the projection of V (resp. E) on the reflector (see Figure 5).

For non-spherical reflectors, this property does not hold. We nevertheless use A and B as as two of our initial points. The third point C is chosen so that ABC is an equilateral triangle.

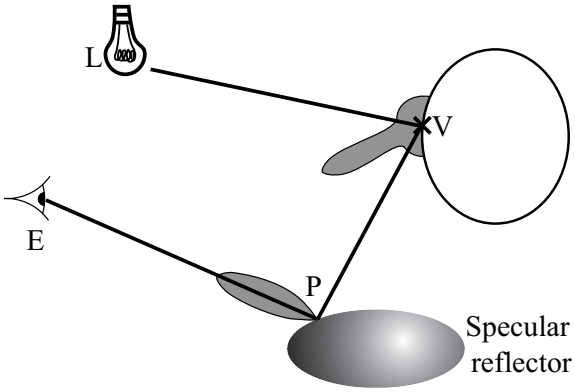


Figure 6: *Computing the illumination of the reflected scene: illumination at the reflected point is computed using its BRDF, with $\overrightarrow{VL}$ and $\overrightarrow{VP}$ as incoming and outgoing directions; it is then multiplied by the BRDF on the reflector, with $\overrightarrow{PV}$ and $\overrightarrow{PE}$ as incoming and outgoing directions.*

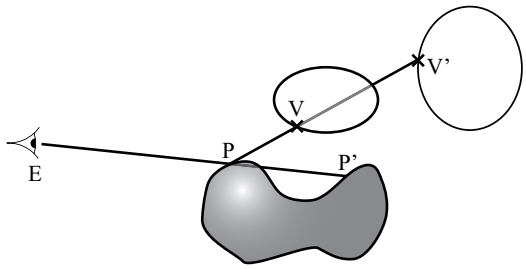


Figure 7: *For a ray originating from the eye, we have to resolve visibility issues both between P and P′, on the reflector, and between V and V′, on the reflected ray.*

3.3.5. Direction-dependent lighting on the reflected scene

When we display a fragment of the reflected scene, we know its spatial position V and the approximate spatial position of its reflection P. We use this information to compute directionally-dependent lighting:

- compute illumination at point V, using its BRDF, with the light source L as the incoming direction and the reflected point P as the outgoing direction (see Figure 6).
- multiply this by the BRDF of the specular reflector at point P, using the reflected point V as the incoming direction, and the viewpoint E as the outgoing direction.

This simple rule allows us to have directional lighting on the reflected scene. The lighting on the reflected scene is thus not necessarily the same as the lighting on the original scene.

3.3.6. Multiple Hidden-Surface Removal

Hidden surface removal requires special handling, as we have several possible sources of occlusion (see Figure 7): the scene and the reflector may be occluding each other, and we also have to conduct hidden-surface removal on the reflected scene. The ideal solution would be to use several depth buffers, or a multi-channel depth-buffer. As these are not available, we have designed a workaround.

For each vertex V, when we compute its projection P, we store in the depth buffer the distance between P and V. This way, the Z-buffer of the graphics card naturally removes fragments of the reflected scene that are hidden by other objects.

To solve the other occlusion issues, we use the following strategy:

- pre-render the frontmost back-facing polygons of the reflector into a depth texture; clear the Z-buffer and framebuffer.
- render the scene, with lighting and shadowing; clear the stencil-buffer.
- render the reflector, with hidden surface removal. For pixels that are touched by the reflector, set the stencil buffer to 1.
- clear the depth buffer and render the reflected scene using our algorithm. The fragments generated are discarded if the stencil buffer is not equal to 1 (using the classical stencil test) and if they are further away than the backfaces of the reflector (using the depth texture computed at the first step).
- (optional) enable blending and render the reflector, computing its illumination.

Our strategy correctly handles occlusions between the reflector and the scene (using the stencil test), as well as self occlusion of the reflector, using the depth texture. Note that we have to use frontmost back-facing polygons: using the frontmost front-facing polygons would falsely remove all the reflected scene for locally convex reflectors, since we are linearly interpolating between reflected points that are on the surface of the reflector.

3.3.7. GPU implementation

We have implemented our algorithm on the GPU for better efficiency. To compute the reflected position of one vertex, we need access to the equation and derivatives of the specular reflector. Since we stored these in a texture to handle arbitrary specular reflectors, this limits us to two possible implementation strategies:

- place our algorithm in vertex shader, using graphics hardware with vertex texture fetch (NVidia GeForce 6 and above).
- place our algorithm in a fragment shader and render the reflected positions of the vertices in a Vertex Buffer Object. In a subsequent pass, render this VBO. This requires hardware with render-to-vertex-buffer capability, which was not available to us at the time of writing.

We have used the first strategy, but found that it suffers from several limitations: there are less vertex processing units than fragment processing units on GPUs, so we are not taking full advantage of its parallel engine; a texture fetch in a vertex processor has a large latency; vertex processors can not currently read from cube maps or rectangular textures, forcing us to use square textures.

As pointed out by [EMD*05], it makes sense to use a cube map to store the information about the specular reflector, since reflector information is queried based on a direction vector $\boldsymbol{d}$, as cube maps are. In the current implementation, we have to convert the vector $\boldsymbol{d}$ into spherical coordinates (θ, ϕ), a costly step.

An implementation of our algorithm using the second strategy is likely to have much better rendering times, as well as a simpler code.

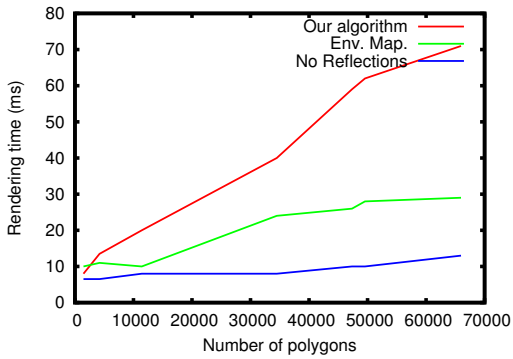


Figure 10: *Observed rendering time (in ms) for rendering scenes, with no specular reflections, with environment-mapping specular reflections and with our algorithm.*

4. Experiments and Comparisons

4.1. Comparison with other reflection methods

The strongest point of our algorithm is its ability to produce reflections with great accuracy. Figure 1 and Figure 8 show, for comparison, pictures generated with our algorithm, ray-traced pictures for reference, and pictures generated with environment mapping. Our method handles all the reflection issues, including contacts between the reflector and the reflected object. Differences between our method and the environment mapping method especially appear for objects that are close to the reflector, such as the hand in Figure 1 and the handle of the kettle in Figure 8. Notice how the reflection of the handle of the kettle appears to be flying in the reflection of the room in Figure 8(c).

For objects that are close to the reflector, our algorithm exhibits all the required parallax effects. One of the problems with environment mapping techniques is when objects are visible from some parts of the reflector but not from its center. In figure 9, our algorithm properly renders the back of the chair.

Another strong point of our algorithm is its robustness and temporal stability. As shown in the accompanying video, reflections computed by our algorithm exhibit great temporal stability, without temporal aliasing. This property is essential for practical applications, such as video games.

4.2. Rendering speed

As we have seen in Figure 3(c), the number of iterations required for convergence depends greatly on the position of the reflection. Reflections close to the center of the reflector converge quickly, in less than 5 iterations, while reflections of objects located close to the silhouette of the reflector take longer to reach convergence.

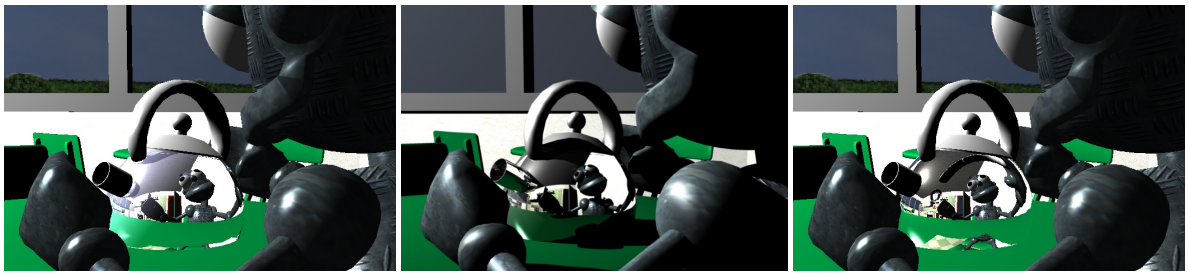
As a consequence, the rendering time depends on the respective position of the object and the reflector. We observe the worse timing results if the scene being reflected completely surrounds the reflector. In that case, many objects are reflected on the silhouette, dragging the rendering process. These scenes are also more interesting to render, which is why we used them nevertheless in all our timings results.

Figure 10 shows the rendering times for scenes of various sizes, surrounding the specular reflector. For comparison, we plotted the rendering time for the scene, without specular reflections, with specular reflections simulated by environment mapping and with specular reflections computed by our algorithm. The extra cost introduced by our algorithm is always larger than that of environment maps, but it remains within the same order of magnitude. We observe satisfying performances for scenes up to 40,000 polygons, and we also observe that rendering times depend linearly on the number of vertices (all timings in this section were measured on a 2-processor Pentium IV, running at 3 GHz, with a NVidia GeForce 7800 graphics card). We measure rendering times in ms by taking the reciprocal of the observed framerate, multiplied by 1000.

4.3. Robustness and early exit

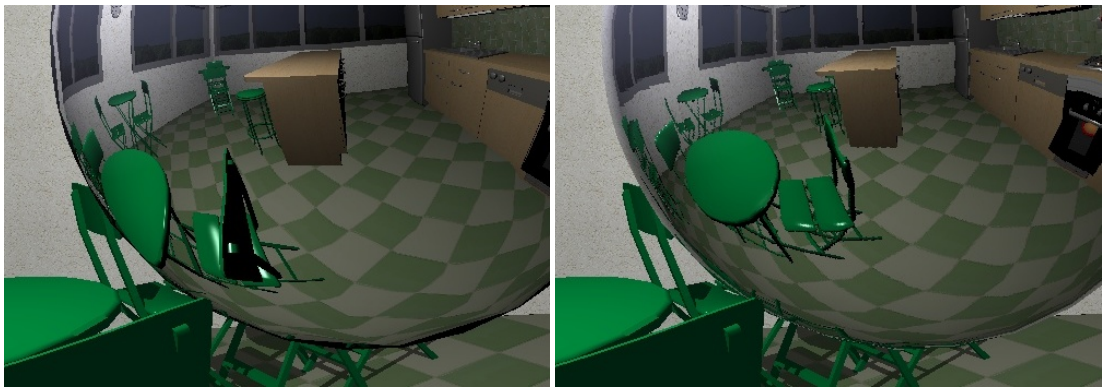
As our method is based on a triangle of sample points and uses the gradient of the optical path at each sample point, it has several advantages:

- we make large steps if we are far from the solution, and smaller steps as we approach the solution (see Figure 3(a)). This ensures faster convergence, even with poor initial conditions.
- the method is remarkably robust, and converges even for difficult cases, such as vertices reflected at grazing angles; in that case, it takes longer to reach a converged solution, but it reaches one, sometimes after more than 10 iterations (see Figure 3(c)).



(a) Our method (b) Ray traced reference (c) Environment mapping

Figure 8: *Comparison of our results (left) with ray-tracing (center, for reference) and environment-mapping (right). The difference are especially visible for objects that are close to the kettle, such as its handle and the right hand of the character.*



(a) Our algorithm (b) Environment mapping

Figure 9: *Our algorithm is able to display objects that are not visible from the center of the reflector. Notice here how the back of the chair is properly rendered.*

We note that for simple cases, our method reaches convergence very quickly (less than 5 iterations), while for difficult cases it requires more computations. As we are doing our computations on the vertex processing units, the fact that different vertices require different computation times is not a big issue. In our tests, we found that using the early exit greatly improved the speed of the computations compared to using a fixed number of iterations.

Spatial consistency could become a larger issue if we moved the computations to the fragment processing unit, but we observe (see Figure 3(c)) that all the vertices from one object have similar complexities; all these vertices should take roughly the same computation time, ensuring that early exit also works well in this situation.

4.4. Concave reflectors

Concave reflectors are a special case. As noted by [Ofe98, OR98], concave reflectors divide space into three zones. Objects that are in the first zone, close to the reflector, are reflected only once and upside-up. Objects that are in the

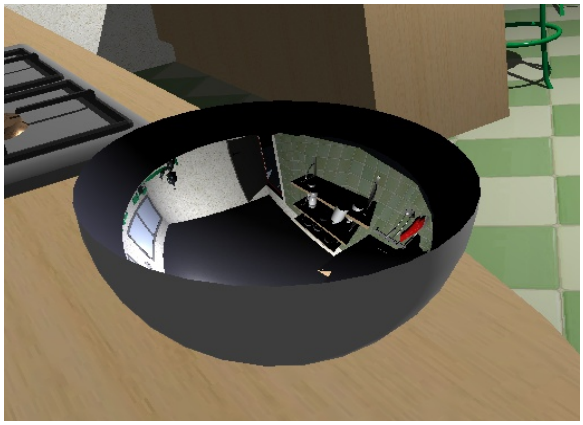


Figure 11: *Example of a reflection with a concave reflector. As our algorithm only captures the first reflection of the scene in the bowl, the top of the bowl looks empty.*

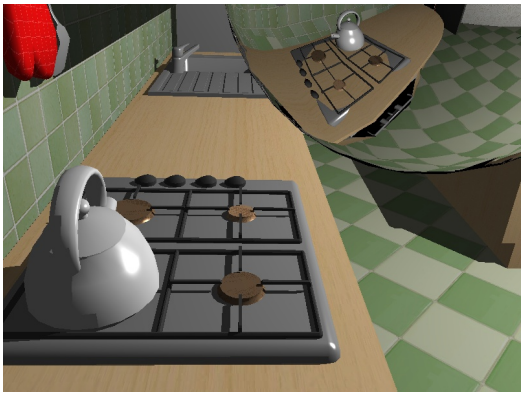


Figure 13: *Example of Z-fighting when a small object is layered on top of a larger object.*

third zone, far from the reflector, are reflected only once, and upside-down (as in Figure 11). Objects that are in the second zone, between the other two, can have several reflections, sometimes an infinite number, and their reflection is numerically unstable.

As with [Ofe98, OR98], our algorithm properly handles objects that are either completely in the first or the third zone, but not objects that cross or are in the second zone.

In our experiments, another problem appeared: concave objects are highly likely to cause secondary reflections (reflections with several bounces inside the specular reflector). As our algorithm only captures the first reflection of the scene by the specular reflector, the place where these secondary reflections should be looks empty.

4.5. Tesselation issues

One of the biggest drawback of our algorithm is that we are only computing the exact reflection position at the vertices, and we let the graphics hardware interpolate between the reflected positions. Currently, the graphics hardware is only able to interpolate linearly. This has several consequences. The first one is that the interpolated objects are located *behind* the front face of the reflector if the reflector is locally convex. Thus, the front face of the reflector would hide the reflection. We had to ensure that the front face of the reflector was not present in the Z-buffer to avoid this problem. The second one is that for objects that are not finely tesselated, we see interpolation artifacts. These artifacts can either be discontinuities between neighboring faces with different levels of tessellation, or a reflection that looks straight, as in Figure 12(a). The third consequence appears for thin objects layered on top of another, larger object (see Figure 13). Because we are linearly interpolating Z-values as well as position, the back object may pop in front of the other object, partially occluding it.

The solution to these issues would be to use curvilinear interpolation, or adaptive tessellation. In the meantime, we apply our algorithm to well-tessellated scenes (see Figure 12(b)). Note that curvilinear interpolation of depth values would be easier with current graphics hardware than curvilinear interpolation in pixel space.

5. Conclusion and Future Works

We have presented an algorithm for computing reflections on curved specular surfaces, using vertex-based computations. Our algorithm produces realistic specular reflections in real-time, showing all the required parallax effects. Our algorithm is iterative, with an adaptive number of iterations, and has a geometry-based criterion for deciding convergence.

In its current form, our algorithm uses linear interpolation between the projections of the vertices, resulting in artifacts for scenes that are not finely tessellated. Solutions to this problem are either adaptive tessellation or curvilinear interpolation techniques.

The strongest point of our algorithm is that it can handle arbitrary geometry on the reflector and the reflected object, including contact between the two surfaces. It is for this situation — close proximity between the reflected object and the reflector — that current environment-map methods do not provide convincing results. We think that our algorithm would be best used as a complement to existing methods, handling the reflection of close objects, while environment-map based methods would be used for the reflection of further objects and the background.

As our algorithm provides a method to compute the reflected ray passing by two endpoints, it can be used for other computations, such as caustics and refraction computations.

References

[Bjo04] Bjorke K.: Finite-radius sphere environment mapping. In *GPU Gems*. Addison-Wesley, 2004.

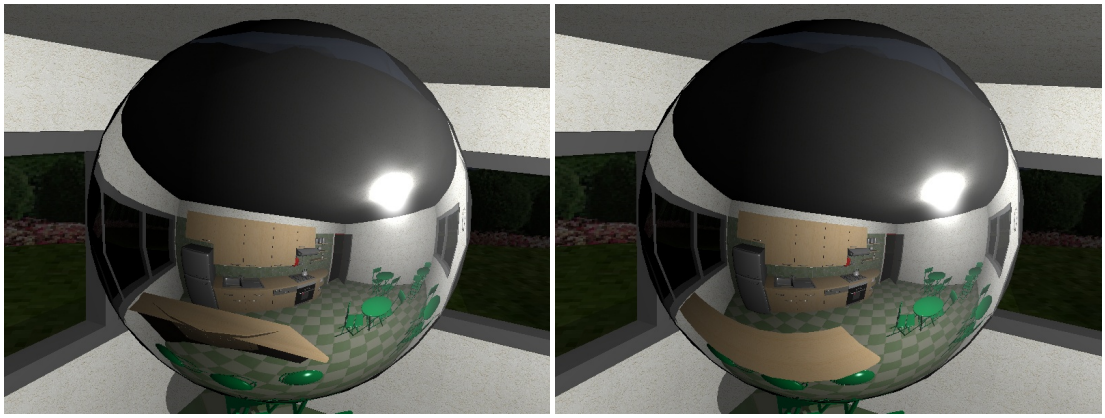
[BN76] Blinn J. F., Newell M. E.: Texture and reflection in computer generated images. *Communications of the ACM 19*, 10 (1976), 542–547.

[CA00a] Chen M., Arvo J.: Perturbation methods for interactive specular reflections. *IEEE Transactions on Visualization and Computer Graphics 6*, 3 (2000), 253–264.

[CA00b] Chen M., Arvo J.: Theory and application of specular path perturbation. *ACM Transactions on Graphics 19*, 4 (2000), 246–278.

[CHH02] Carr N. A., Hall J. D., Hart J. C.: The ray engine. In *Graphics Hardware 2002* (2002).

[EMD*05] Estalella P., Martin I., Drettakis G., Tost D., Devillers O., Cazals F.: Accurate interactive specular reflections on curved objects. In *Proceedings of VMV 2005* (Nov. 2005).



(a) The bar is not tessellated, and its reflection is not curved — as it should be.

(b) The problem disappears if we tessellate the bar.

Figure 12: *The scene has to be well tesselated, or artifacts appear because we cannot render curved triangles.*

[EMDT06] Estalella P., Martin I., Drettakis G., Tost D.: A gpu-driven algorithm for accurate interactive reflections on curved objects. In *Rendering Techniques 2006 (Proc. EG Symposium on Rendering)* (June 2006).

[McR96] McReynolds T.: Programming with OpenGL: Advanced rendering. Siggraph'96 Course, 1996.

[MH92] Mitchell D., Hanrahan P.: Illumination from curved reflectors. *Computer Graphics (Proc. of SIGGRAPH '92) 26*, 2 (1992), 283–291.

[MP04] Martin A., Popescu V.: *Reflection Morphing*. Tech. Rep. CSD TR#04-015, Purdue University, 2004.

[Ofe98] Ofek E.: *Modeling and Rendering 3-D Objects*. PhD thesis, Institute of Computer Science, The Hebrew University, 1998.

[OR98] Ofek E., Rappoport A.: Interactive reflections on curved objects. In *Proc. of SIGGRAPH '98* (1998), pp. 333–342.

[Pat95] Patow G. A.: Accurate reflections through a Z-buffered environment map. In *Proceedings of Sociedad Chilena de Ciencias de la Computacin* (1995).

[PBMH02] Purcell T. J., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (Proc. of Siggraph 2002) 21*, 3 (July 2002), 703–712.

[SKALP05] Szirmay-Kalos L., Aszdi B., Laznyi I., Premecz M.: Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum(Proceedings of Eurographics '05) 24*, 3 (2005).

[WBWS01] Wald I., Benthin C., Wagner M., Slusallek P.: Interactive rendering with coherent ray tracing. *Computer Graphics Forum (Proc. of EUROGRAPHICS 2001) 20*, 3 (2001).

[WSB01] Wald I., Slusallek P., Benthin C.: Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001 (Proc. 12th EUROGRAPHICS Workshop on Rendering)* (2001), pp. 277–288.

[WSS05] Woop S., Schmittler J., Slusallek P.: Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (Proc. of Siggraph 2005) 24*, 3 (2005), 434–444.

[YYM05] Yu J., Yang J., McMillan L.: Real-time reflection mapping with parallax. In *Proc. I3D 2005* (2005), pp. 133–138.