Eurographics Symposium on Rendering (2006)
Tomas Akenine-Möller and Wolfgang Heidrich (Editors)

# A GPU-driven Algorithm for Accurate Interactive Reflections on Curved Objects

Pau Estalella[1] and Ignacio Martin[1] and George Drettakis[2] and Dani Tost[3]

[1]Universitat de Girona, Spain
[2]REVES/INRIA Sophia-Antipolis, France
[3]Universitat Politecnica de Catalunya, Barcelona, Spain

**Abstract**

*We present a GPU-driven method for the fast computation of specular reflections on curved objects. For every reflector of the scene, our method computes a virtual object for every other object reflected in it. This virtual reflected object is then rendered and blended with the scene. For each vertex of each virtual object, a reflection point is found on the reflector's surface. This point is used to find the reflected virtual vertex, enabling the reflected virtual scene to be rendered. Our method renders the 3D points and normals of the reflector into textures, and uses a local search in a fragment program on the GPU to find the reflection points. By reorganizing the data and the computation in this manner, and correctly treating special cases, we make excellent use of the parallelism and stream-processing power of the GPU. In our results we show that, with our method, we can display high-quality reflections of nearby objects interactively.*

## 1. Introduction

Realistic effects such as reflections and refractions are very important for many applications, including computer games and virtual reality. However, in these applications, interactivity is essential, incurring a trade-off between realism and speed. Ray-tracing is the approach traditionally used to compute such effects in computer graphics. Whilst recent approaches using PC-clusters can achieve interactivity to some extent [WSBW01], this approach is still not feasible as a standard solution for consumer PCs, especially in a game-like context, where scenes contain moving objects. In addition, ray-tracing is a completely different rendering architecture, which would require a significant shift for existing GPU-based workflows. Clearly, an algorithm which provides interactive, high quality specular reflections from curved reflectors within a standard GPU-based pipeline, would be very useful in such contexts.

Previous work on interactive reflections on planar surfaces has been based on rendering the *reflected scene*, in an additional pass [DB97]. Extensions have been proposed for curved objects (e.g., [OR98, EMD*05]), where a reflecting triangle or a reflection point for each vertex are found on the reflector, allowing the reflected scene to be rendered. Our work continues this direction of research, in particular to search for reflection points [EMD*05] for each scene vertex reflected.

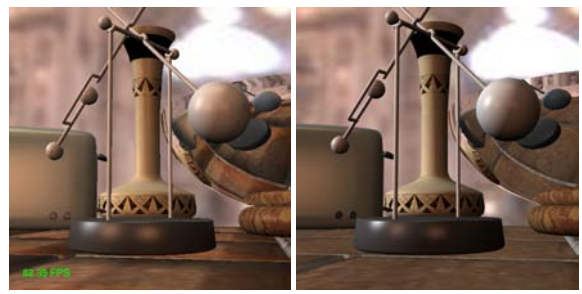

**Figure 1:** *Left, a frame of an interactive sequence using our method, running at 82fps. Right, a ray-traced image (5s/frame)*

Our key observation is that, with appropriate reorganisation of the data and the computation, this search can be a simple and local operation. In addition, it involves computations such as dot products, which can be done in parallel for all the vertices to be reflected. All these properties make this an ideal candidate for GPU processing. Our solution encodes appropriate information of positions and normals in texture memory in a first pass, then performs the search for reflection points in a fragment program. The reflected vertices are

rendered to texture, and a final pass renders the reflections in the reflectors. Our approach is a classic case of reorganising computation to be stream-processor friendly. Thanks to the data and computation reorganisation, the resulting speed on the GPU allows high-quality reflections to be integrated into a standard rendering pipeline.

Although the idea of searching for reflections points is based on the approach of [EMD*05], we present a new search algorithm which is GPU-friendly, and runs in a fragment program. Our new approach results in an *order of magnitude speedup* compared to [EMD*05]; in addition it is much more robust and stable thus totally avoiding flickering during camera motion or animation. Such speedup e.g., from 8fps (using [EMD*05]) to 80fps (Fig. 1(a)) can make the difference of going from "just about interactive" to real-time.

## 2. Previous and related work

For brevity, we restrict this section to a rapid review of a subset of the literature on reflections. Environment Maps (EM) [BN76] are frequently used to achieve specular reflection effects on curved objects. They assume that the environment is infinitely distant from the reflector, making reflections from the centre of the object a sufficient approximation [Gre86]. Some improvements have been proposed, using warping [CON99], pre-generated EMs for multiple interreflections [Kil99], or an extension to self-reflections using parameterized EMs [HSL01]. Dynamic environments can be treated combining precomputation and warping [ML03], and they can also be used to compute recursive specular reflections [NC02]. The above methods do not provide satisfactory results when the viewer is close to the reflector.

Ray-tracing using the GPU is another technique used to compute specular reflections at interactive frame rates [Pur04]. These techniques rely on acceleration structures that are computed in a pre-process step, and thus are not suitable for fully dynamic environments. In [SKALP05] an approximate ray tracing method is presented that achieves real-time performance. Although this method can handle complex reflectors, it only performs well for simple reflected scenes containing large polygons because in this case the visibility function is simple enough to be stored with distance impostors.

Multipass rendering was first used for planar mirrors [DB97], by projecting a virtual object through the plane for each reflection and blending this into the scene. Precomputed radiance maps [BHWL99] can be used to avoid the projection step. For curved reflectors, an interesting solution was proposed in [OR98]. For each vertex to be reflected, an efficient data structure (the *explosion map*) accelerates the search for a triangle used to perform the reflection. Although this is efficient, there can be problems with accuracy in some cases. An analytic approach has also been developed [CA00], using a preprocessing step based on path-perturbation theory. When objects move, the preprocessing step needs to be recomputed.

Recently, in [EMD*05] the reflected (or *virtual*) vertices $V'$ are found by first localizing the reflection point $R$ on each reflector. Geometry and normal information is stored in a pair of cubemaps, thus making it available during rendering. Their implementation achieves interactive rates on medium-sized scenes. Our method is based on the same general principle, i.e., searching for reflection points to reflect scene vertices. In all of the above methods however, there is no direct way to truly exploit the power of modern graphics GPUs.

In a method developed concurrently with ours [RHS06], a similar approach has been presented. The foundations and scope of both methods are similar; the main difference is the search method and the GPU mapping. The results achieved are comparable to ours in speed. However our approach will handle changing reflectors more efficiently, and the view-dependent nature of the textures used for the search avoids potential resolution problems when the viewpoint is close to the reflector.

## 3. Overview

At each frame and for every reflector, all the reflected objects are computed, rendered and blended with the rendered reflectors, using the stencil buffer, in order to compose the final image. The following pseudo-code summarizes this process.

```
foreach frame
    computeReflectedScenes()
    drawNonReflectors()
    drawReflectorsWithStencil()
    drawReflectedScenes()
endfor
```

The reflected scenes contain the reflected objects, which we call *virtual objects* from now on. Virtual objects are composed of *virtual vertices* topologically connected as in the original objects. A virtual vertex $V'$ is computed by specular reflection of the corresponding vertex $V$ onto a point of reflection $R$ on the reflector's surface. As illustrated in Figure 2, the point of reflection $R$ is such that the bisector vector $B_R$ of the angle formed by the observer $O$, $R$ and $V$ is the same as the surface normal vector $N$ at $R$. This can be identified when $N \cdot B$ is equal to 1. It has been proven that, on convex and closed reflector surfaces, the point of reflection is unique [EMD*05]. However, concave and mixed curvature objects can also be handled on some situations. The efficiency of the approach thus depends on the speed of the search for the reflection points.

Using the current viewpoint, we render the 3D points and normals of the reflector into texture memory on the GPU, then search for the pixel in these textures for which $N \cdot B$ is

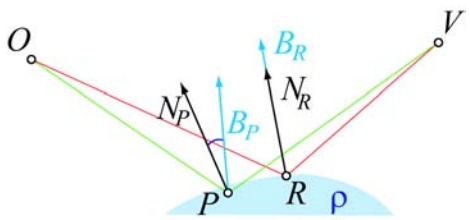


**Figure 2:** *The basic geometry used: the observer O, a point P on the reflector, a vertex V which will be reflected on the reflector ρ, the normal $N_P$ and the bisector $B_P$. The point R is the reflection point, at which $B_R = N_R$, by definition (figure adapted from [EMD*05]).*

one. This search is performed in a fragment shader, greatly enhancing the speed and the parallelism of the computation. However, care has to be taken for a number of specific issues: hidden vertices which need to be reflected to maintain the topology, and partially visible reflectors. We present our solutions to these issues, and discuss the use of temporal coherence to accelerate our method.

## 4. Reflections on curved reflectors

Our approach operates entirely on the GPU and does not require any pre-computation. At each frame we render the reflector, using the same camera used for the observer, into two floating-point textures, one storing the 3D coordinates of the reflector's points and the other the normal vectors at these points. From now on, we will call these textures $T_P$ and $T_N$ for clarity.

The method's inputs are these previously rendered textures, the vertex coordinates, and the search starting points. Vertex coordinates and starting points are stored in vertex arrays and fed to the fragment program as texture coordinates.

For each vertex of the scene, we search its reflection point in the reflector's textures $T_P$ and $T_N$. This search is performed by a fragment program on the GPU. Each vertex to reflect is rendered as a *GL_POINT* primitive that generates a single fragment, and then the fragment program performs the search and computes the reflected vertex.

The method's resulting reflected vertices are stored one after another, row after row in a frame buffer. When the whole scene has been reflected the results are copied to a vertex array. We use this vertex array to render the reflected scene on the reflector.

For the initial frame, we start the search at the central texel of the reflector's bounding box projection. We iteratively examine a 3x3 neighbourhood in the texture. A crosshair pattern is used (see Fig. 3(left)) to determine the first four samples we test. For each sample the fragment program computes the normalized bisector vector given the coordinates of the reflection point in $T_P$, its normal vector coordinates in $T_N$, the vertex coordinates, and the observer's position stored as a global parameter.

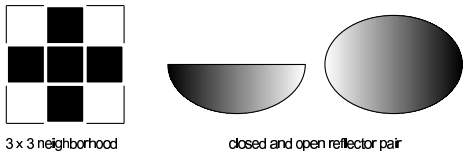


**Figure 3:** *(Left) The crosshair pattern used to sample the textures for the reflection point search. (Right) An example of an open/closed reflector pair.*

Then, the fragment program computes the dot product between the computed bisector vector and the normal vector from $T_N$. The point of reflection that we are looking for has a dot product of one. In practice, the search stops if the central texel of the 3x3 neighbourhood has the maximum scalar product value. Otherwise, the process continues iteratively by shifting the neighbourhood to center it at the sample with the maximum scalar product. The search is implemented as a non-fixed length loop.

The following pseudo-code summarizes this process:

```
computeReflectedScenes()
    foreach reflector Ri
        renderAndStore3DandNormalTextures()
        setUpRenderTargets()
        setUpCg()
        sendVerticesToGPU()
        copyResultToReflectedVertexArray()
    endfor
```

## 5. Handling special cases

As described in detail in [OR98,EMD*05], a specular reflector and an observer position give rise to the three regions, shown in Fig. 4(left). Using the terminology of [OR98], these are: Region A, the "reflected region", in front of the reflector, region C the "hidden region" from which no object can be reflected, and region B the "unreflected region", which occurs for "open" reflectors, i.e., objects which are not modeled as solids, which can occur in a typical production workflow.

### 5.1. Open reflectors and hidden vertices

If the search exits from the limits of the reflector, we know that original vertex is in region *B* or *C*. If it is in region *B* we have an open reflector. We assume that these objects have been modeled as a open/closed reflector pair (see Fig. 3(right) for an example). By open reflector we mean the original reflector geometry, while by closed reflector, we

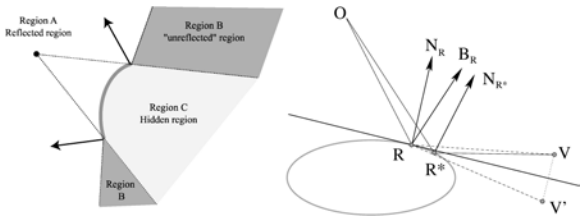


**Figure 4:** *(Left) The reflected (A), hidden (C) and unreflected (B) regions. Their respective extents are related to observer position and reflector shape. (Right) Computation of the bisector $B_R$ during the inverse search requires computing the reflected vertex $V$ of $V'$ with respect to the current normal $N_R$ at reflector surface point $R$. The real reflection point is $R*$, where $N_{R*} = B_{R*}$.*

mean a watertight object containing the open reflector. All the search computations are performed with the closed reflector, and the original model is used to render the stencil buffer. In practice this means that in our implementation we do not have $B$ regions. We discuss this limitation in Sect. 7.

If the vertex lies in region $C$, we have a hidden vertex. We also have to compute a reflected vertex since we must reflect all the vertices. Moreover, in this case, the reflected vertex has to be outside of region $C$. We use the same approach as that described in [EMD*05], by inverting the roles of the original and reflected vertices. In particular, we perform an inverse search (Figure 4 (right)) in which the original vertex $V$ becomes the reflected vertex, and the new original vertex is $V'$, where:

$$V' = R + 2N((V-R)\cdot N) - (V-R) \qquad (1)$$

At the end of the search we have found a vertex $V'$ that is directly reflected into $V$. Note that the "reflection plane" in Fig. 4(b) is defined by the current position and normal of $R$, which are updated at every evaluation.

To identify this case in practice, we use the following observation: If a vertex $V$ is in $A$, we will find a unique reflection point $R$ with normal $N$, and $V$ will be in the positive half-space of the plane defined by $R$ and $N$. If we detect that this condition does not apply, we can assume that $V$ is in Region C, and we can apply the inverse search. In practice, we have observed that it is sufficient to check the plane half-space condition at the end of the standard search, and simply use the plane defined by $R$ and $N$ for the inverse reflection, if required.

### 5.2. Partially visible reflectors

Reflectors are often partially visible in the current frame. In this case the search can stop at the border of the texture without reaching the maximum. This means that all the vertices that should be reflected in the hidden part of the reflector will have a reflection point on the border of the texture. This causes small, but noticeable artifacts.

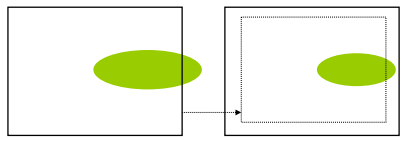

**Figure 5:** *The modified frustum has a larger field of view and captures more of the reflector.*

We adopt a simple solution, that is to render the 3D positions and normals using a slightly wider angle camera (Figure 5). Using a wider angle camera means that textures $T_P$ and $T_N$ will contain a greater portion of the reflector. In the video we show the effect of increasing the size of the frustum.

### 5.3. Frame-to-frame coherence

We improve the efficiency of the search, by taking advantage of frame-to-frame coherence. At each frame, and for each vertex, we use the pixel corresponding to the reflection point for the previous frame as the starting pixel of the search. To do so, for each vertex, we store the pixel coordinates of these starting points in a texture $T_{SP}$. This simple strategy considerably reduces the length of the searches.


**Figure 6:** *Two images of the ants scene with 8 reflectors. The left image shows a general view of the scene, while on the right we show a close up of the ant eyes showing the distorted scene. This scene runs at 12 fps and contains 30,153 vertices. In the left image the reflected images are very small, but we are computing a full reflection of the scene.*

## 6. Results

We implemented our approach on an nVidia GeForce 7800 with 256 MB memory using Cg for the fragment shaders under Linux. All timings are for this configuration. The scene loader subdivides geometry which is too large (typically too long and/or thin), to achieve better quality reflections.

In what follows, and in the accompanying video, we show

our method running on three scenes. The first is the "kitchen scene" (the same scene as in [EMD*05], see Fig. 8), contains 14,979 vertices after subdivision (11,379 in the original scene). The scene contains one large metal bowl as a reflector. The viewpoint moves towards the bowl, and thus at the end of the animation the bowl covers a large part of the screen. The scene runs at about 87 fps on average.

The second is the "UFO scene", which contains 81,977 vertices after subdivisions (63,884 for the original scene), and one big reflector (see Fig. 7). Here the UFO lands in the central square of a city, and we can see the reflections of the building and trucks driving through the square. The scene runs at about 12 fps on average.

The third scene is the "ant scene", which contains 30,153 vertices after subdivision (14,859 for the original scene), and eight reflectors corresponding to the eyes of four ants (see Fig. 6). This scene has the largest increase in vertices after tessellation due to the very long and thin triangles in the cans, required to obtain smoothly curved reflections in the eyes of the ant. This scene shows that our method can easily handle multiple reflectors. The scene runs at about 12 fps on average.

Finally, Fig. 9 shows four images of the kitchen scene reflected on a non-convex reflector. Fig. 9(a) and (b) show a concave reflector, and (c)-(d) show a section of a torus. These images have been computed using the same fragment program as for the convex reflectors.

Table 1 shows the length of the iterations over the animation sequences. For each frame of the animation we compute the average length of the iterations, and at the end we compute the average over the entire sequence.

| | Kitchen | UFO | Ant |
|---|---|---|---|
| Max frame average | 14.9 | 71.2 | 7.5 |
| Total average | 4.7 | 4.2 | 3.3 |

**Table 1:** *Max frame average is the average maximum number of iterations for the search at each frame. Total average is the search length average for the entire sequence.*

From the above comparisons, and the video, we can see that our method results in high quality reflections for relatively complex scenes, especially when we are close to the reflector.

## 7. Discussion, future work and conclusions

Our approach is conceptually simple, and organizes the data and the computation in a way which makes excellent use of GPU parallel and stream processing. This allows us to achieve the high frame rates and high quality reflections shown in the results.

Although the basic idea for reflection point search is similar to [EMD*05], the new GPU search algorithm presented provides two main advantages over the original method. First, the speedup achieved is one order of magnitude. Second, the new search algorithm provides much more stable results than the original idea. This is because our technique performs the search in image space and finds the best pixel for the reflected point. Flickering is completely avoided while in the previous case a high number of iterations would be needed resulting in a corresponding drop in frame rate.

The choice of the wider frustum parameter value to avoid artifacts is a somewhat "ad-hoc". As shown in the video however, this choice does not have a major impact on quality. Also, the need to model both closed and open reflectors is a constraint on content creation. It is often likely that the number of curved reflectors viewed at close range will be limited, thus alleviating this problem. For other objects, environment maps can be used. The need to incur additional subdivision for higher quality reflections could be overcome with the possibility to generate additional geometry on the GPU, which will probably become a feature in next generation systems.

To extend our method to multiple reflections, a multipass approach should be developed. This should not be difficult since we only need to adapt our search method to a slightly different space, the reflected image. This also means that the search space and the next-bounce reflected images get smaller and smaller in image space.

We have shown that our method can also handle, with some limitations, concave and even mixed curvature objects. The limitation is that vertices to be reflected must have a single reflection point, which is shared by all methods based on the virtual scene paradigm [OR98, EMD*05]. We are currently investigating ways to efficiently identify the geometry of these configurations and correctly render objects that have more than one reflection onto a single reflector.

One improvement that could greatly speed up our algorithm is the use of levels of detail (LODs) and mipmapping. One form of LOD we could use is the interpolation scheme presented in [EMD*05] – this should be directly applicable to our approach. Also, since convex reflections make the virtual scenes appear smaller in image space, the use of geometric LODs and mipmapping would enhance the speed and the quality of the reflections. Such optimizations will make implementing multiple reflections practical.

Recently, a GPU-driven method for computing approximate image-space refractions of nearby objects have been proposed, restricted to two interfaces [Wym05]. We believe that our approach can be easily extended to refractions; it will be interesting to see the relative quality achieved by the two approaches.

In conclusion, we have presented a simple, but efficient method for computing reflections on curved reflectors. Our technique runs interactively on modern GPUs for interesting, dynamic models. We believe that it is thus useful for

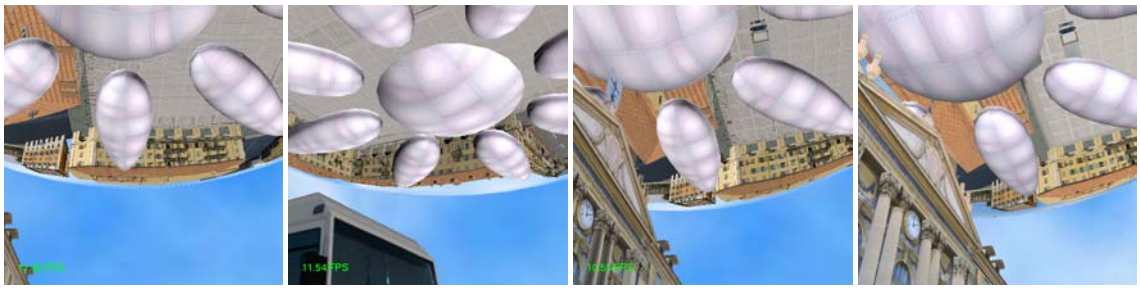

**Figure 7:** *(Left) Three frames of the UFO scene (12 fps; 81,997 vertices). (Far right) Ray-traced image for comparison.*

interactive applications which use a traditional GPU-based rendering pipeline, such as games or VR.

## 8. Acknowledgements

We thank Autodesk-Alias for their generous donation of the Maya software, used in these examples, and Alexandre Olivier-Mangon who created the kitchen and UFO environments and the corresponding animations. Also to Carles Bosch who helped in the video production. The work has been funded by projects TIC2001-2226-C02-02 and TIN2004-07672-C03-01 from the Spanish government.

## References

[BHWL99] BASTOS R., HOFF K., WYNN W., LASTRA A.: Increased photorealism for interactive architectural walkthroughs. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics* (1999), ACM Press, pp. 183–190.

[BN76] BLINN J. F., NEWELL M. E.: Texture and reflection in computer generated images. *Comm. of the ACM 19*, 10 (Oct. 1976).

[CA00] CHEN M., ARVO J.: Perturbation methods for interactive specular reflections. *IEEE Transactions on Visualization and Computer Graphics 6*, 3 (July/Sept. 2000), 253–264.

[CON99] CABRAL B., OLANO M., NEMEC P.: Reflection space image based rendering. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (1999), ACM Press, pp. 165–170.

[DB97] DIEFENBACH P. J., BADLER N. I.: Multi-pass pipeline rendering: realism for dynamic environments. In *Proc. Symposium on Interactive 3D graphics'97* (1997).

[EMD*05] ESTALELLA P., MARTIN I., DRETTAKIS G., TOST D., DEVILLIERS O., CAZALS F.: Accurate interactive specular reflections on curved objects. In *Proc. of VMV 2005* (2005).

[Gre86] GREENE N.: Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications 6*, 11 (Nov. 1986).

[HSL01] HAKURA Z. S., SNYDER J. M., LENGYEL J. E.: Parameterized environment maps. In *Proc. of the symposium on Interactive 3D graphics'01* (2001).

[Kil99] KILGARD M.: *Perfect Reflections and Specular Lighting Effects With Cube Environment Mapping*. Tech. rep., 1999.

[ML03] MEYER A., LOSCOS C.: Real-time reflection on moving vehicles in urban environments. In *Proceedings of VRST '03* (2003), ACM Press, pp. 32–40.

[NC02] NIELSEN K. H., CHRISTENSEN N. J.: Real-time recursive specular reflections on planar and curved surfaces using graphics hardware. In *Journal of WSCG* (Plzen , Czech Republic, February 2002), vol. 10, University of West Bohemia, pp. 91–98.

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

[Pur04] PURCELL T. J.: *Ray Tracing on a Stream Processor*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, March 2004.

[RHS06] ROGER D., HOLZSCHUCH N., SILLION F.: Accurate specular reflections in real-time. *Computer Graphics Forum (Proceedings of Eurographics 2006) (To Appear)* (2006).

[SKALP05] SZIRMAY-KALOS L., ASZÓDI B., LAZÁNYI I., PREMECZ M.: Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum (Proceedings of Eurographics 2005) 24*, 3 (September 2005).

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive distributed ray tracing of highly complex models. In *Rendering Techniques* (2001), pp. 277–288.

[Wym05] WYMAN C.: An approximate image-space approach for interactive refraction. In *ACM SIGGRAPH 2005* (2005).