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 inter-reflections [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

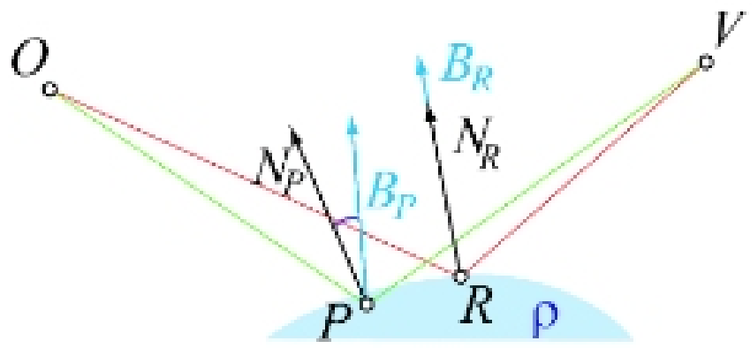

Figure 2: *The basic geometry used: the observer O, a point P on the reflector, a vertex V which will be reflected on the reflector ρ, the normal N_P and the bisector B_P. The point R is the reflection point, at which $B_R = N_R$, by definition (figure adapted from [EMD*05]).*

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.

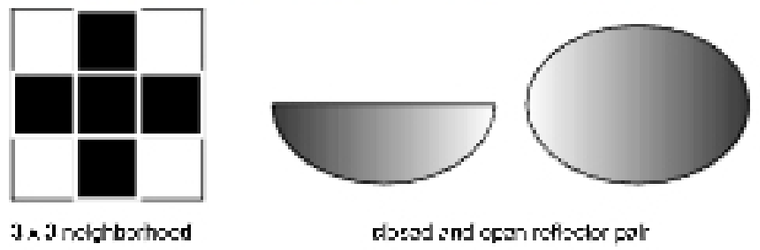

Figure 3: *(Left) The crosshair pattern used to sample the textures for the reflection point search. (Right) An example of an open/closed reflector pair.*

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