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

Real-time Multi-perspective Rendering on Graphics Hardware

Xianyou Hou[1,2,3] Li-Yi Wei[1] Heung-Yeung Shum[1] Baining Guo[1]

[1]Microsoft Research Asia [2]Institute of Computing Technology, CAS [3]GUCAS

Abstract

Multi-perspective rendering has a variety of applications; examples include lens refraction, curved mirror reflection, caustics, as well depiction and visualization. However, multi-perspective rendering is not yet practical on polygonal graphics hardware, which so far has utilized mostly single-perspective (pin-hole or orthographic) projections.

In this paper, we present a methodology for real-time multi-perspective rendering on polygonal graphics hardware. Our approach approximates a general multi-perspective projection surface (such as a curved mirror and lens) via a piecewise-linear triangle mesh, upon which each triangle is a simple multi-perspective camera, parameterized by three rays at triangle vertices. We derive analytic formula showing that each triangle projection can be implemented as a pair of vertex and fragment programs on programmable graphics hardware. We demonstrate real-time performance of a variety of applications enabled by our technique, including reflection, refraction, caustics, and visualization.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture: Graphics Processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

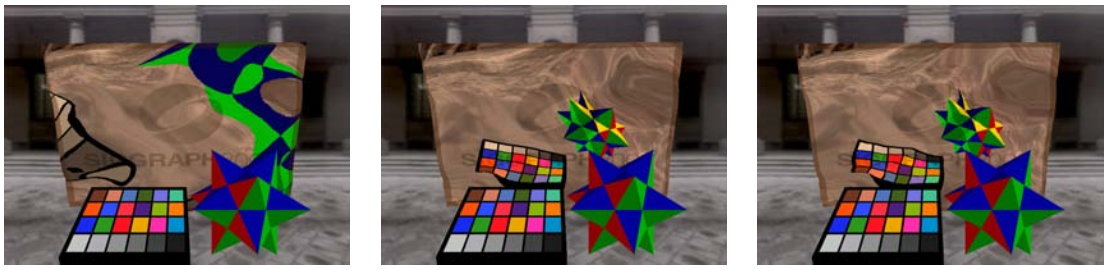
Keywords: multi-perspective rendering, GPU techniques, graphics hardware, reflection, refraction, caustics, visualization

1. Introduction

A single-perspective image contains light rays passing through a single viewpoint. A multi-perspective image, in contrast, collects light rays across a multiple (and possibly infinite) number of viewpoints. For example, in viewing reflections off a curved mirror, the set of rays converging at a pin-hole camera is single-perspective, whereas the set of rays reflected off the curved mirror usually do not intersect at a single point and could only be modeled by a multi-perspective projection. Even though the majority of real-time rendering applications are based on single-perspective projection (e.g. pin-hole cameras), multi-perspective projection describes a variety of common phenomena, such as curved reflections [OR98, HSL01, YM05], refractions [GS04, Wym05], non-pinhole cameras [KMH95, SGN01], or caustics [WS03, PDC*03, Wym06]. In addition, multi-perspective images have important applications in depiction and visualization [RB98, AZM00, SK03].

So far, the most natural and common rendering method for multi-perspective projection is ray tracing. For example, reflection can be simulated by bouncing rays off curved mirror surfaces, and refraction can be achieved by bending rays through curved lens. Unfortunately, mapping ray tracing to graphics hardware is tricky, as it requires random accessibility with a database of scene triangles, which does not fit well with a SIMD feed-forward graphics pipeline [PBMH02, CHH02]. This problem is further exacerbated with dynamic scenes [CHCH06], which, unfortunately, is common for interactive applications such as gaming.

A variety of graphics hardware techniques have been proposed to simulate multi-perspective rendering without ray tracing. However, these techniques are often heuristics and applicable to only individual phenomena; to our knowledge there is no general framework for multi-perspective rendering on graphics hardware. For example, even though techniques exist for supporting multiple center-of-projections in VR applications [KKYK01, SSP04], they are limited to a



Environment map | Ray tracing | Our technique (63 fps)

Figure 1: Curved reflection of nearby objects is a classic example of multi-perspective rendering. Our multi-perspective framework renders more accurate reflection than a traditional environment map, which is single-perspective. In this example, we utilize 104 camera triangles to approximate the mirror. Each camera triangle is $64^2/2$ in size and the final rendering size is 800×600. All reported frame-rates are measured on a NVIDIA Geforce-7800-GT chip.

few discrete single-perspective viewpoints and therefore unsuitable for continuously varying projections like general curved reflections or refractions. These phenomena can be simulated by representing scene objects as image sprites [SKALP05, Wym05], but it remains unclear how to robustly resolve the dis-occlusion problem, and how to perform visibility sorting of these multiple depth sprites efficiently via hardware z-buffering.

So why is multi-perspective rendering difficult on polygonal graphics hardware? For a general multi-perspective model, each pixel can have a unique camera projection matrix. This means that the proper vertex transformation is unknown until the vertex is projected to a known pixel location; a classic chicken-and-egg dilemma.

We resolve this difficulty by grouping and rendering coherent bundles of rays together, rather than by tracing individual rays. Our main idea is inspired by beam tracing [HH84], in that a curved reflector/refractor is tessellated into a coarse polygonal representation followed by rendering rays bouncing off the same polygon together as a coherent beam. However, beam tracing cannot be directly implemented on graphics hardware, as rays belonging to the same beam might not intersect at the same eye point, which is a necessary condition for a pin-hole camera modeling. In our technique, we model each beam of rays as a multi-perspective camera triangle, and render these rays by passing down geometry as in a traditional feed-forward graphics pipeline. Specifically, we render each multi-perspective camera triangle in a separate rendering pass, where the multi-perspective projection is achieved via a pair of customized vertex and pixel programs. The final multi-perspective rendering is then accomplished by stitching together result images rendered by individual camera triangles.

A noteworthy feature of our technique is that the entire rendering process is fully compatible with a feed-forward pipeline, and no ray tracing scene database is used in particular. As a result, we can render dynamic scenes naturally without the need for any pre-processing, such as building an acceleration data structure or converting scene objects into image sprites. The disadvantage of our technique is that it is brute force, since we have to render each scene triangle in each multi-perspective camera triangle in the worse case. However, we will present acceleration techniques to ameliorate this performance problem. In addition, we believe our technique would scale well with future generation graphics chips due to its affinity with a feed-forward graphics pipeline.

1.1. Overview of Our Methodology

Given a general curved projection surface of a multi-perspective camera (such as a curved reflector), we first approximate it with a piecewise-linear triangle mesh. At each mesh vertex we compute a ray direction, based on the original camera model. Under this representation, each mesh triangle can be considered a simple multi-perspective camera, where the ray of each camera pixel is computed from the three vertex ray. Due to the shared vertex directions, we guarantee at least C^0 continuity of rays across adjacent triangles.

We render individual multi-perspective cameras via the ordinary feed-forward polygonal graphics pipeline, and then map this piecewise-linear rendering result back onto the original multi-perspective surface via standard texture mapping. Specifically, we collect texture maps corresponding to individual camera triangles into a joint texture atlas. Even though this approach is only approximate, it provides visually satisfactory results and eliminates the problem of per-pixel projection matrix.

We demonstrate that correct multi-perspective projection of each camera triangle can be computed via a pair of vertex and fragment programs on programmable graphics hardware [LKM01]. Since a straight line in space can be projected into a curve in a multi-perspective camera, for correct ras-

terization of a scene triangle, we customize both vertex and fragment programs. In our vertex program, we compute the vertices of the bounding triangle for the projected triangle, defining the proper region for rasterization. In our fragment program, we render the projected triangle out of the bounding triangle by calculating the 3D intersection point between the rasterized triangle and the ray emanating from the current pixel, and from this intersection we interpolate the pixel attributes (depth, color, and texture coordinates) for correct multi-perspective effects.

1.2. Our Contribution

The key contribution of our approach is the idea of casting this difficult multi-perspective rendering problem into an algorithm suitable for feed-forward graphics pipeline. To our knowledge, this approach has not been published before. We provide implementations for rendering multi-perspective images via special shader programs on graphics hardware. We demonstrate a variety of applications enabled by our technology, including reflection, refraction, caustics, and visualization.

2. Our Approach

As described in [Gla00], any continuous multi-perspective projection can be parameterized by two curved surfaces; one for lens and the other for imaging. This parameterization is general enough to model a variety of multi-perspective phenomenon such as real camera lens and curved reflectors, most of which have continuous variations in the projection directions. This continuous requirement also avoids the difficulty of depth compositing multiple single-perspective images [AZM00]. Because of these advantages, we adopt this continuous representation for multi-perspective projection. Our goal now is to approximate this projection on graphics hardware.

One naive method is to pre-compute the projection points on the imaging surface for a dense sample of 3D locations, and store the result within a texture map. This technique is general enough to handle many situations (as long as each 3D point has a unique projection), but suffers from the usual sampling and texture storage problems.

In our approach, we analytically compute the projection without sampling or extra texture storage. Given a multi-perspective projection surface, we approximate it via a piecewise-linear triangle mesh, as shown in Figure 2. At each vertex of the piecewise-linear mesh, we specify a ray based on the property of the original input. (Note that this is equivalent to the representation in [Gla00].) For example, in simulating a multi-perspective camera, the ray directions are determined by bending the eye rays through the camera; while in rendering reflection, we specify normals as the ray directions and compute the true reflection directions by reflecting the eye position around the interpolated normals at each pixel.

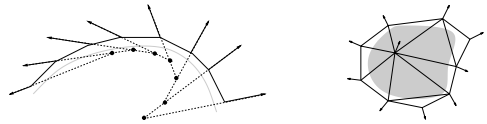


Figure 2: Illustration for piecewise-linear approximation (shown in black) of a curve projection surface (shown in gray). In 2D, each triangle (line segment) is a pin-hole camera with center-of-projection at the intersection of the two vertex directions. However, in 3D, each triangle is a multi-perspective camera because the three vertex rays might not intersect at the same point.

For GPU rendering, we treat each triangle on the simplified mesh as a simple multi-perspective camera. To render the multi-perspective projection of the entire mesh, we first render the sub-images at each camera triangle and store the results into a joint texture map. From this joint texture map, we render the original projection surface in a subsequent rendering pass via standard texture mapping. The correspondence between the original surface and the joint texture map is computed in a pre-process and depends on the particular application at hand.

The projection of each triangle camera is determined by the projections directions at the three triangle vertices. In general, this may not be a pinhole or orthographic camera since the three rays might not intersect at the same point. Our goal is to find a parameterization for the rays at each camera pixel so that (1) the parameterization interpolates exactly at the three camera vertex directions and (2) the parameterization is C^0 continuous across the edges shared by adjacent camera triangles. Even though alternative techniques exist for parameterizing non-pinhole cameras [YM05, MPS05], neither preserves projection continuity across adjacent cameras in general situations. GLC guarantees C^0 continuity only if the adjacent cameras are co-planar or if the cameras are infinitely small [Yu05], while [MPS05] utilizes radial distortions for extending object silhouettes so there is no C^0 continuity even if two adjacent cameras are co-planar.

To satisfy these continuity requirements, we propose the use of barycentric interpolation, as illustrated in Figure 3. Given a camera triangle $\triangle v_1 v_2 v_3$ with normalized rays $\bar{d_1}$, $\bar{d_2}$, and $\bar{d_3}$, we define the ray $\bar{d}$ at an arbitrary pixel $\bar{p}$ via standard barycentric coordinates:

$$\begin{aligned} \bar{d} &= w_1\bar{d_1} + w_2\bar{d_2} + w_3\bar{d_3} \\ w_i &= \frac{area(\triangle p v_j v_k)}{area(\triangle v_1 v_2 v_3)}_{i,j,k=1,2,3, i\neq j\neq k} \end{aligned} \quad (1)$$

Obviously, barycentric interpolation would return exact values of $\bar{d_1}$, $\bar{d_2}$, and $\bar{d_3}$ at the camera triangle vertices, and the computation is C^0 continuous across adjacent triangles, so both requirements listed above are satisfied.

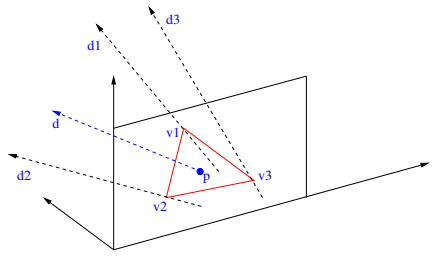


Figure 3: Multi-perspective camera parameterization. The camera triangle is parameterized by three rays at the triangle vertices. Given an arbitrary pixel $\bar{p}$, its ray $\bar{d}$ can be found via barycentric interpolation from $\bar{d}_1$, $\bar{d}_2$, and $\bar{d}_3$.

Given this parameterization, we can compute the projection $\bar{p}$ of a space point $\bar{q}$ or vice versa (Figure 4). The computation of $\bar{q}$ from $\bar{p}$ can be efficiently performed by intersecting the ray $\bar{d}$ from $\bar{p}$ with scene triangles. The computation of $\bar{p}$ from $\bar{q}$ involves solving a quartic equation, as detailed in Apppendix A. Even though quartic equations can be solved algebraically, they are still too complex for efficient GPU implementation; fortunately, we do not need to explicitly perform this operation, since we only need to compute $\bar{q}$ from $\bar{p}$.

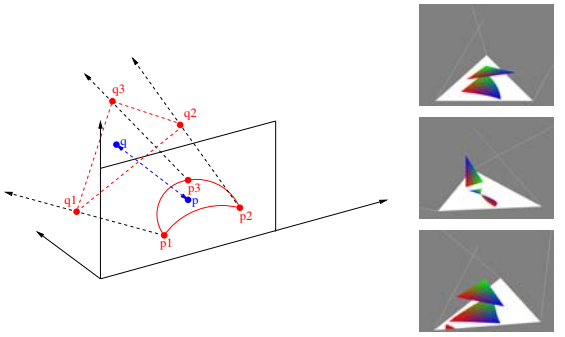


Figure 4: Projection of a scene triangle $\triangle q_1q_2q_3$. Left: In general situations, the projection $\triangle p_1p_2p_3$ of $\triangle q_1q_2q_3$ is curvilinear under our multi-perspective camera. Right: Depending on the ray directions at the camera triangle (shown in white) vertices, the projected triangle (shown in color) can have various shapes.

Figure 4 demonstrates the projection $\triangle p_1p_2p_3$ from a scene triangle $\triangle q_1q_2q_3$, following Equation 1. In general, $\triangle p_1p_2p_3$ can have curved edges due to our multi-perspective projection (Figure 4 top right); two curved edges may intersect each other (Figure 4 middle right); and a space point might have multiple projections (Figure 4 bottom right). Obviously, these effects cannot be achieved in a fixed-function graphics pipeline because a fixed-function rasterization can only produce straight projected lines from straight space lines.

To resolve this issue, we utilize programmable features of current GPUs, and customize both vertex and fragment programs to render the curvilinear multi-perspective projection effects. Due to the independent processing of triangles in a feed-forward graphics pipeline, we only discuss how to properly render a scene triangle $\triangle q_1q_2q_3$; the same operation would be applied for every other triangles. In our vertex program, we estimate the bounding triangle $\triangle b_1b_2b_3$ for the (unknown) projected triangle $\triangle p_1p_2p_3$, defining the proper region for rasterization. In our fragment program, we render $\triangle p_1p_2p_3$ out of $\triangle b_1b_2b_3$ by calculating the 3D intersection point $\bar{q}$ between scene triangle $\triangle q_1q_2q_3$ and the ray $\bar{d}$ emanating from the current pixel $\bar{p}$, and from this intersection we interpolate the pixel attributes (depth, color, and texture coordinates) for correct multi-perspective effects. The details are described below.

2.1. Vertex Program: Compute Bounding Triangle

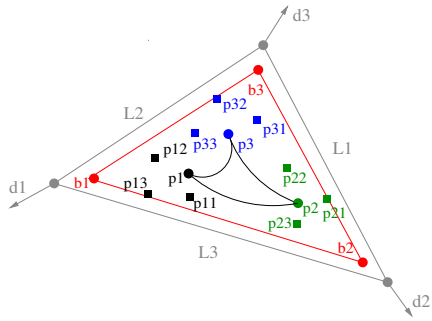


Figure 5: The bounding triangle $\triangle b_1b_2b_3$ for a projected triangle $\triangle p_1p_2p_3$. Line L_i is orthogonal to camera vertex direction $\bar{d}_i$, and p_{ij} indicates the projection of scene triangle vertex $\bar{q}_i$ with respect to $\bar{d}_j$.

In our vertex program, we compute, for each space vertex of the rasterized scene triangle $\triangle q_1q_2q_3$, the corresponding vertex on the bounding triangle $\triangle b_1b_2b_3$ so that it wholly contains the projection $\triangle p_1p_2p_3$ (Figure 5). Even though a variety of techniques exist for calculating bounding triangles, we have to choose an algorithm that is suitable for efficient vertex program implementation; a complex algorithm that yields tight bounding region might be less favorable due to its complexity. In addition, the bounding algorithm must be robust enough to handle a variety of situations as depicted in Figure 4. To satisfy these goals, we propose the following bounding triangle computation.

For each scene triangle vertex $\{q_i\}_{i=1,2,3}$, we compute corresponding projections $\{p_{ij}\}_{i,j=1,2,3}$ where p_{ij} is the projection of q_i in the direction of d_j (see Figure 5 for illustration). Obviously, $\triangle p_1p_2p_3$ is wholly contained within the convex hull formed by $\{p_{ij}\}$ since the projection $\bar{d}$ of each $\bar{p}$ inside $\triangle p_1p_2p_3$ is computed from barycentric interpolation of $\{d_i\}_{i=1,2,3}$ (Equation 1). So now if we could find a bounding triangle $\triangle b_1b_2b_3$ that contains all the nine points $\{p_{ij}\}$, then we can guarantee that $\triangle b_1b_2b_3$ contains $\triangle p_1p_2p_3$.

Since we do not know the exact shape of $\triangle p_1p_2p_3$ (it can have odd shapes as shown in Figure 4 and even the computation of $\{p_i\}_{i=1,2,3}$ involves solving quartic equation), we simply enforce the edges of $\triangle b_1b_2b_3$ to be parallel to the lines $\{L_i\}_{i=1,2,3}$ where each L_i is orthogonal to $\bar{d}_i$ and lies on the camera plane. (In singularity situations of $\{\bar{d}_i\}$ such as one of them being orthogonal to the camera plane or two of them as parallel, we simply pick $\{L_i\}$ to be the edges of the camera triangle.) To compute edge $\overline{b_ib_j}$, all we need to do is to find the point among $\{p_{ij}\}_{i,j=1,2,3}$ that is outmost from $L_{k,k\neq i\neq j}$ (in terms of the signed edge equation). For example, in Figure 5, p_{13} determines $\overline{b_1b_2}$ because p_{13} is closet to L_3 among $\{p_{ij}\}_{i,j=1,2,3}$. Even though this algorithm does not provide the most tight bounding, it works reasonably well; when $\{d_i\}$ is coherent and $\triangle q_1q_2q_3$ is not far away from the camera plane, the bounding region is usually pretty tight.

The above computation for $\triangle b_1b_2b_3$ can be efficiently implemented in our vertex program. As a pre-process, we compute the line equation for each $\{L_k\}_{k=1,2,3}$ and store the results as program constants. Inside our vertex program, we compute b_i from $\bar{q}_i$ as the intersection of the two lines $\{\overline{b_ib_j}\}_{j=1,2,3,j\neq i}$. Each $\overline{b_ib_j}$ can be determined by figuring out which $\{p_{ik}\}_{i=1,2,3}$ evaluates maximum value with the line equation $L_{k,k\neq i\neq j}$. In total, this requires only six line equation evaluations and a single line-line intersection computation inside our vertex program.

We pass down this bounding triangle $\triangle b_1b_2b_3$ to the rest of the graphics pipeline for rasterization and fragment computation. For each computed vertices $[\bar{b}_i]_{i=1,2,3}$, we attach information of all three of the original vertex $[\bar{q}_i]_{i=1,2,3}$ (via unused texture coordinate slots) so that the fragment program can perform correct interpolation and projection.

2.2. Fragment Program: Space Point for Pixel

In our fragment program, we determine, for each pixel $\bar{p}$, the corresponding 3D point $\bar{q}$ on $\triangle q_1q_2q_3$ so that $\bar{q}$ projects onto $\bar{p}$. This $\bar{q}$ can be efficiently computed via ray-triangle intersection between the ray $\bar{d}$ at $\bar{p}$ and the plane containing $\triangle q_1q_2q_3$ [CHH02]. From the barycentric coordinates of $\bar{q}$ on $\triangle q_1q_2q_3$, we interpolate depth, color, and texture coordinates from the corresponding values stored on $[\bar{q}_i]_{i=1,2,3}$. If the intersection point $\bar{q}$ is outside $\triangle q_1q_2q_3$, we simply throw away the pixel (via fragment kill) without further computation since it is not part of the projected triangle $\triangle p_1p_2p_3$. In some sense, we use our fragment program to carve out the curved boundary of $\triangle p_1p_2p_3$. Note that this effect cannot be achieved via [LB05] because it operates in texture space and therefore cannot handle triangle boundaries.

2.3. Scene Construction

Here, we describe further details on how we construct the camera triangles and the joint texture maps. From a curved reflector/refractor surface, we perform mesh simplification via progressive meshes [Hop96], and select a simplified mesh with proper number of triangles as our multi-perspective camera mesh to meet the target quality/performance goals. We then establish the correspondence between the original surface and the coarse camera mesh via normal shooting as described in [SGG*00]; this essentially determines the texture coordinates parameterizing the original surface over the coarse camera mesh. For each camera mesh triangle, we allocate a uniform-size right angle triangle in the joint texture atlas (similar to [SGG*00]). Because of this, the texture packing is trivial without wasted texture space as in a general atlas.

Our current tessellation and packing process is static; we leave dynamic parameterization as a potential future work.

2.4. Limitations and Discussion

The major disadvantages of our approach are (1) the rendering every triangle in each camera, and (2) the overestimation of bounding triangle size. Further disadvantages include our inability of taking advantage of early z-cull since our fragment program computes per-pixel depth values based on ray-triangle intersection. Below, we discuss our strategies to address these performance issues.

Object Culling One possible acceleration to reduce our vertex workload is to pre-compute bounding boxes for scene objects, and for each camera, only to render those objects which fall within the camera viewing frustum. This can be achieved either in object space on CPU, or via the occlusion query feature on NVIDIA graphics cards. (However, care must be taken with occlusion query because it may incur pipeline flush depending on the particular graphics card.)

Bounding Triangle Culling Even though in theory every triangle needs to be rasterized by every camera, in reality, it often happens that each camera only sees a subset of the triangles, even for objects surviving the culling. In our approach, we can further cull away triangles whose bounding region is outside the camera as follows. We associate a clipping region with each camera triangle and use the hardware viewport clipping to clip away bounding triangles that are totally outside the clipping region. Note that even though our multi-perspective camera could in theory have non-planar side-polygons on the clipping frustum, our clipping can be simply performed by considering the bounding triangle on the camera plane, since our vertex program always transforms vertices onto the camera plane and we compute the true 3D position by ray-triangle intersection in the fragment program. If we were to perform culling in the 3D viewing frustum, the problem would be much more complicated; fortunately, we do not have to.

Camera Tessellation Level To achieve optimal performance, we have to balance the workload between vertex and fragment stages. This can be achieved by proper tessellation of our multi-perspective projection surface to yield the optimal camera triangle sizes; the more and smaller the camera triangles, the more vertex work (because every scene vertex needs to be transformed in every camera triangle without other acceleration) but less fragment work (because the overdraw ratio between bounding triangles and projected triangles are smaller due to increased coherence between the rays at camera triangle vertices). The ideal level of subdivision depends on the specific scene and graphics hardware, and can be determined empirically. (See Table 1 and Figure 6 for an example.) Note that since we do not perform any context switch (i.e. pipeline flush) between camera triangles, there is no stall penalty associated with using multiple camera triangles.

We have performed a performance profiling of our algorithm with a particular scene as demonstrated in Figure 7.

We discuss further disadvantages of our method when compared with other techniques in specific application domains (Section 3).

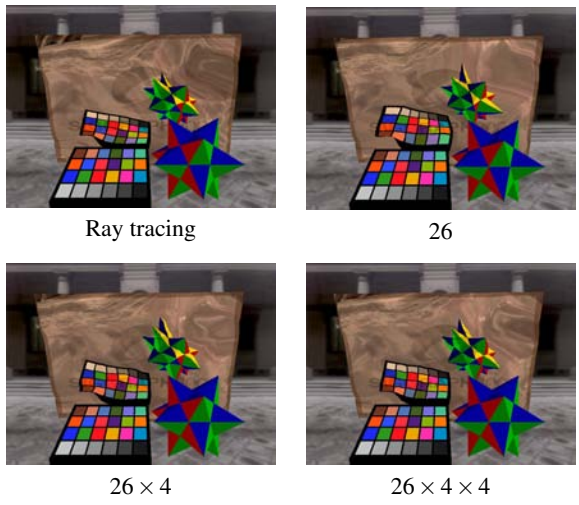


Figure 6: Quality comparison with different amount of camera tessellation. The reference ray tracing image is shown on top left, with the rest of the images show our results with different number of tessellation camera triangles. As shown, the quality improves with increasing number of tessellations, but beyond a certain limit the payoff levels off.

3. Applications

Our core algorithm described above enables a variety of real-time applications on graphics hardware. Below, we present a subset of the potential applications and describe their implementation details. Even though techniques exist for modeling each of the following individual applications, these techniques are often heuristics and there is a lack of general framework for multi-perspective rendering on graphics hardware. For each application, we compare our technique with previous work, discussing relative pros and cons.

We summarize our application scene statistics in Table 1 for easy reference.

3.1. Curved Reflection of Nearby Geometry

Reflection off curved mirrors is a classic example of multi-perspective projection [YM05]. However, it remains a daunting task to render curved reflection of nearby geometry for dynamic scenes in real-time. Previous work either assumes distant objects as in the classical environment maps [BN76], planar reflectors [DB97], static scenes [YYM05], or relies on heavy pre-computation [HSL01]. [SKALP05] renders near reflection by approximating scene objects as depth imposters; because these imposters are rendered from a single perspective, disocclusion artifacts may appear if the imposters are reused from a different viewpoint [MPS05]. [OR98] is one of the few methods that satisfy the above requirements, but since its image quality depends on fine tessellation of scene geometry, this computation requirement may prevent real-time performance.

Our technique supports curved local reflection by tessellating the reflector surface into piecewise-linear camera triangles, as illustrated in Figure 2. We pre-compute the correspondence between the original reflector and its piece-wise linear approximation. During rendering, the individual camera triangles are first rendered as described in Section 2 with following minor difference; instead of assigning projection rays at vertices and perform barycentric interpolation for pixels, we instead assign surface normals at vertices, perform interpolation in rasterizer, and from interpolated normals and eye point we compute the reflection rays per fragment. This would yield a more accurate reflection computation than direct interpolation as pointed out by [YM05]. We then texture map the original reflector by the rendered camera triangles via the pre-computed correspondence.

Figure 1 compares our technique with environment mapping and ray tracing. As shown, our technique provides more accurate rendering of near-object reflections than environment maps while operating in real-time.

Compared to [OR98], the disadvantage of our approach is that we require a separate rendering pass for each camera while [OR98] needs only one pass. The advantage of our approach is that we do not need to subdivide the scene geometry. In addition, since we do not need to build virtual vertices as in [OR98], our technique naturally handles convex, concave, and mixed-convexity reflectors (as demonstrated in Figure 1).

Figure 7 illustrates a more complex scene with a shiny teapot reflecting many dynamic objects. Due to the mas-

Scene	# scene △	# camera △	# bounding △ fragments	# rendered fragments	overdraw ratio	camera texture resolution	frame rate (fps)
reflection	108	26	2002513	62860	32	$128^2/2$	51
	108	26×4	920156	62734	15	$64^2/2$	63
	108	26×4^2	783566	62823	12	$32^2/2$	5
refraction	252	12	2475722	106349	23	$128^2/2$	46
caustics	2	400	945962	475852	2	$64^2/2$	35
visualization	6841	400	213248	213248	1	$64^2/2$	46

Table 1: Demo scene statistics. In the reflection demo, we demonstrate the performance effect of three different camera tessellation levels, with quality comparison shown in Figure 6. See our discussion in Section 2.4. Overdraw ratio = $\frac{\text{\# bounding } \triangle \text{ fragments}}{\text{\# rendered fragments}}$.

# scene △	# camera △	no acc (fps)	acc (fps)
800	100	4.5	20.1
1200	100	3.6	14.6
1600	100	2.1	9.2
400	256	1.5	6
800	256	0.5	2.1
1200	256	0.35	1.8
400	512	0.8	2.5
800	512	0.3	1.0

Figure 7: Performance profiling of our algorithm. In this scene, we reflect multiple independent and dynamic objects off a shiny teapot. By controlling the number of camera and scene triangles we can easily profile the performance of our algorithm (reported in fps), with and without our acceleration discussed in Section 2.4. Notice in general the performance does not scale linearly with respect to the # of camera △ due to load balance between vertex and fragment stages; see our discussion on "Camera Tessellation Level" in Section 2.4.

sive number of independent objects, this scene is difficult to render efficiently and robustly via sample-based techniques [SKALP05, Wym05]. Our technique can easily handle this due to its feed-forward geometry natural.

3.2. Curved Refraction of Nearby Geometry

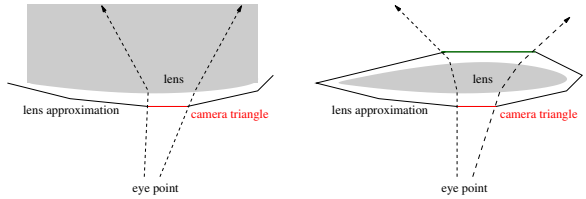
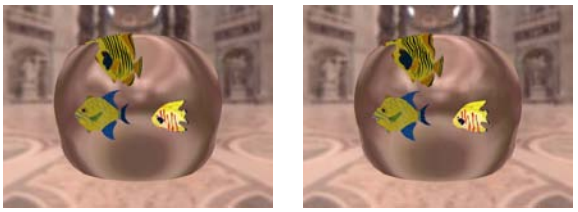


Figure 8: Curved refraction by multi-perspective projection. (a) Our technique can be applied directly for single refraction. (b) In multiple refraction, the multiple-bending of vertex ray directions is beyond the basic parameterization of our multi-perspective camera.

Refraction through a single curved interface is another classic example of multi-perspective projection. There exists a variety of techniques for simulating refraction on GPUs. [GS04, DB97] simulates multiple refractions of planar interfaces. [Sou05] renders refractions through mostly planar surfaces with small bumps such as water waves by perturbing texture coordinates. These techniques all achieve real-time performance, but could not be applied for curved refractors. [Wym05] renders curved refractors by storing as depth images the backface of the refractor as well nearby scene objects. The technique can handle two-sided refractions, but is approximate in nature and suffers from sampling and aliasing artifacts since it is image-based.

Our technique can be naturally applied for rendering single curved refraction by approximating the refraction interface via a piece-wise linear camera surface, as shown in Figure 8 (a). In fact, our algorithm for rendering single refractions is quite similar to our reflection algorithm; the major difference lies in the computation of rays at camera triangle vertices. Figure 9 demonstrates curved refractions rendered by our technique.

A disadvantage of our approach is that our multi-perspective camera cannot directly handle multiple refractions. As illustrated in Figure 8 (b), multiple refractions



Ray tracing Our technique (46 fps)

Figure 9: Near-object refraction through curved interface. We utilize 12 camera triangles to approximate the refractor. Each camera triangle is $128^2/2$ in size and the final rendering size is 800×600.

would bend the rays so that the camera vertex projections are no longer straight lines, and this is beyond the basic parameterization of multi-perspective cameras. We leave it as a future work to extend our technique for multiple refractions. Compared to [Wym05], the advantage of our approach is that we allow hardware depth sorting of all refracted geometry without conversion to any image-based representation. As a consequence we project the scene geometry with greater accuracy and we do not suffer from geometry sampling or aliasing artifacts.

3.3. Caustics

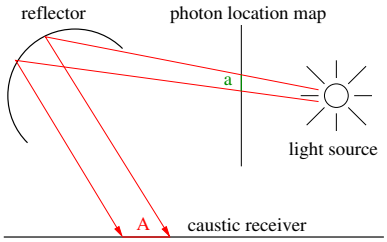
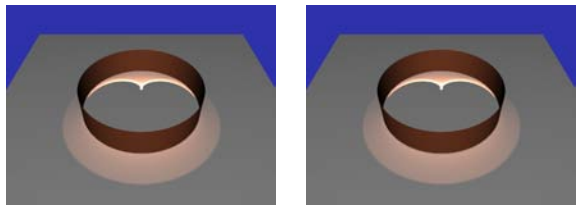


Figure 10: Rendering caustics via our technique. In the first pass, we render a multi-perspective image from the light source's point of view, and store the result in the photon-location map. In the second pass, we render a normal image from the eye's point of view, splatting pixels from the photon-location map for adding caustics.

Caustics occur when light is focused by curved reflection or refraction. Our technique can be directly extended for rendering caustics via a standard two-pass process as demonstrated in previous work [Wym06, PDC*03, WS03]. As illustrated in Figure 10, in the first pass, we render multi-perspective reflection or refraction of scene geometry into the light source's point of view, via our techniques as described earlier. The result is stored in a photon location map, recording the information about the surface point reached by the light, such as position and depth. In the second pass, we approximate caustics intensity as the relative solid angle (seen from light source's point of view) of the photon-location map triangle and caustic triangle (i.e. $\frac{angle(\triangle a)}{angle(\triangle A)}$ in



Ray tracing Our technique (35 fps)

Figure 11: Rendering caustics by our technique. We utilize 400 camera triangles to approximate the reflector. Each camera triangle is $64^2/2$ in size. The photon-location map size is 512×512 and the final rendering size is 640×480. The frame-rate is lower than other applications because more than 50% of the frame time is spent on photon splatting.

Figure 10) as described in [Wym06]. We render the final image from the eye's point of view and add caustics by splatting pixels from the photon-location map. Figure 11 demonstrates caustics effects rendered by our technique.

3.4. Multi-perspective Visualization

In addition to physical phenomenon such as reflection, refraction, or caustics, multi-perspective rendering could also be applied for visualization. As demonstrated in previous work [AZM00, SK03, YM04], multi-perspective rendering could unveil more information on a single image than traditional single-perspective rendering. However, most previous methods rely on image capturing or offline ray tracing for multi-perspective rendering, and this greatly diminishes its applicability in real-time applications.

Our technique can be directly applied for real-time multi-perspective visualization by constructing a (potentially dynamically varying) multi-perspective camera, approximated by our piecewise-linear surface. An example rendering effect achieved by our technique is demonstrated in Figure 12.

4. Concluding Remarks

Since its inception, programmable graphics hardware has been applied to a variety of applications beyond what was originally intended for real-time polygon rendering. So far, the majority of these extensions focus on the pixel-shading effects; the geometry transformation remains relatively unexplored except for a few applications such as vertex skinning or displacement.

In this paper, we have demonstrated that the vertex and fragment programs can be customized for multi-perspective rendering, allowing real-time applications to break from the barrier of the traditional pin-hole camera model. Even though the speed of our implementation might not be as fast as say, OpenRT [DWBS03] on CPUs, we believe our

Original Our visualization (46 fps)

Figure 12: Multi-perspective visualization by our technique. In this example, we utilize 400 camera triangles surrounding a walker.

approach will favor better with the performance improvement in future feed-forward graphics hardware. Furthermore, OpenRT and other ray tracing techniques are often limited to static scenes (or with limited animation) whereas our technique naturally handles dynamic scenes with unrestricted motions. We hope the publication of our work can inspire future research for real-time applications of multi-perspective rendering, as well a deeper exploration of the potential of programmable graphics hardware.

One major limitation of our current implementation is the over-estimation of bounding triangle region; as shown in Table 1, the ratio of rasterized bounding triangle fragments to the rendered fragments can be high, depending on the particular scene characteristics. Even though we perform fragment kill for all fragments that do not lie inside the triangles, there is still significant computation wasted. This problem can be addressed by either finding a better bounding triangle estimation algorithm on our current barycentric parameterization, or by devising a fundamentally new multi-perspective camera parameterization that allows for tighter bounding triangle estimation than our current barycentric technique. Specifically, even though [YM05] does not provide the necessary C^0 continuity we require, we have found their GLC parameterization much more elegant for mathematical analysis, and in particular for computing a tighter bounding triangle. We envision extending GLC for C^0 or higher continuity across non-planar tiling would be both theoretically interesting and practically important.

Another potential future work is to extend our technique for multi-bounce reflections and refractions via proper multi-pass rendering. Finally, since most current multi-perspective visualizations are based on single static images [RB98, SK03], an interesting future direction is to ask what kind of new experience could be achieved if the user could freely navigate within a virtual environment in real-time under a multi-perspective camera?

Acknowledgements

We would like to thank Kurt Akeley for pointing out the relevant VR literature, Xin Tong for commenting on an early draft of the paper, Yanyun Chen for suggestion on disambiguating the reflection image, and Dwight Daniels for proofreading. We would also like to thank anonymous reviewers for their comments. The light probe environment maps are courtesy of Paul Debevec.

Appendix A: Projection Computation

Figure 3 illustrates a triangle under general linear projection. For triangle vertices $\bar{v_1}$, $\bar{v_2}$, and $\bar{v_3}$, projection directions are specified as unit vectors in $\bar{d_1}$, $\bar{d_2}$, and $\bar{d_3}$. Now given an arbitrary point $\bar{q}$ in space, our goal is to find its projection $\bar{p}$ on the $\triangle v_1 v_2 v_3$ plane, so that $\bar{q} - \bar{p}$ is interpolated from $d_1 d_2 d_3$ via the same barycentric coordinates of p.

Specifically, we need to solve the following system of equations

$$\begin{aligned} \bar{p} &= w_1\bar{v_1} + w_2\bar{v_2} + w_3\bar{v_3} \\ \bar{q} - \bar{p} &= c\left(w_1\bar{d_1} + w_2\bar{d_2} + w_3\bar{d_3}\right) \\ 1 &= w_1 + w_2 + w_3 \end{aligned} \tag{2}$$

where the unknowns include $\bar{p}$, the barycentric coordinates w_1, w_2, w_3 and the scaling constant c.

Adding the first two lines of the above equation, we obtain

$$\begin{aligned} \bar{q} &= w_1\left(\bar{v_1} + c\bar{d_1}\right) + w_2\left(\bar{v_2} + c\bar{d_2}\right) \\ &+ (1 - w_1 - w_2)\left(\bar{v_3} + c\bar{d_3}\right) \end{aligned} \tag{3}$$

Since we have three unknowns w_1, w_2, c and three equations ($\bar{q}$ is a three-vector), Equation 3 can be solved under general conditions. Specifically, the equation can be reduced to a quartic polynomial of w_1 and solved exactly [HE95].

$$\sum_{i=0}^{4} c_i w_1^i = 1 \tag{4}$$

Since Equation 4 has four roots, we find the true solution by eliminating non-real roots and those solutions that are outside the camera triangle. But in general, it is possible for a space point $\bar{q}$ to project onto multiple positions; an example is shown in Figure 4.

References

[AZM00] AGRAWALA M., ZORIN D., MUNZNER T.: Artistic multiprojection rendering. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000* (2000), pp. 125–136.

[BN76] BLINN J. F., NEWELL M. E.: Texture and reflection in computer generated images. *Commun. ACM 19*, 10 (1976), 542–547.

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast gpu ray tracing of dynamic meshes using geometry images. In *To appear in the Proceedings of Graphics Interface 2006* (2006).

[CHH02] CARR N. A., HALL J. D., HART J. C.: The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), pp. 37–46.

[DB97] DIEFENBACH P. J., BADLER N. I.: Multi-pass pipeline rendering: realism for dynamic environments. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics* (1997), pp. 59–ff.

[DWBS03] DIETRICH A., WALD I., BENTHIN C., SLUSALLEK P.: The OpenRT Application Programming Interface - Towards a Common API for Interactive Ray Tracing. In *Proceedings of the 2003 OpenSG Symposium* (2003). Available at http://www.openrt.de.

[Gla00] GLASSNER A. S.: *Cubism and Cameras: Free-form Optics for Computer Graphics*. Tech. Rep. MSR-TR-2000-05, January 2000.

[GS04] GUY S., SOLER C.: Graphics gems revisited: fast and physically-based rendering of gemstones. *ACM Trans. Graph. 23*, 3 (2004), 231–238.

[HE95] HERBISON-EVANS D.: Solving quartics and cubics for graphics. In *Graphics Gems V* (1995), pp. 3–15.

[HH84] HECKBERT P. S., HANRAHAN P.: Beam tracing polygonal objects. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), pp. 119–127.

[Hop96] HOPPE H.: Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques* (1996), pp. 99–108.

[HSL01] HAKURA Z. S., SNYDER J. M., LENGYEL J. E.: Parameterized environment maps. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics* (2001), pp. 203–208.

[KKYK01] KITAMURA Y., KONISHI T., YAMAMOTO S., KISHINO F.: Interactive stereoscopic display for three or more users. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), pp. 231–240.

[KMH95] KOLB C., MITCHELL D., HANRAHAN P.: A realistic camera model for computer graphics. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (1995), pp. 317–324.

[LB05] LOOP C., BLINN J.: Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph. 24*, 3 (2005), 1000–1009.

[LKM01] LINDHOLM E., KLIGARD M. J., MORETON H.: A user-programmable vertex engine. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), pp. 149–158.

[MPS05] MEI C., POPESCU V., SACKS E.: The occlusion camera. *Computer Graphics Forum 24*, 3 (2005), 335–342.

[OR98] OFEK E., RAPPOPORT A.: Interactive reflections on curved objects. In *Proceedings of SIGGRAPH 98* (July 1998), Computer Graphics Proceedings, Annual Conference Series, pp. 333–342.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), pp. 703–712.

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), pp. 41–50.

[RB98] RADEMACHER P., BISHOP G.: Multiple-center-of-projection images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1998), ACM Press, pp. 199–206.

[SGG*00] SANDER P. V., GU X., GORTLER S. J., HOPPE H., SNYDER J.: Silhouette clipping. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques* (2000), pp. 327–334.

[SGN01] SWAMINATHAN R., GROSSBERG M. D., NAYAR S. K.: *Non-Single Viewpoint Catadioptric Cameras: Geometry and Analysis*. Tech. Rep. cucs-004-01, Columbia University, 2001.

[SK03] SEITZ S. M., KIM J.: Multiperspective imaging. *IEEE Computer Graphics and Applications 23*, 6 (November/December 2003), 16–19.

[SKALP05] SZIRMAY-KALOS L., ASZODI B., LAZANYI I., PREMECZ M.: Approximate ray-tracing on the gpu with distance impostors. In *Eurographics 2005* (2005).

[Sou05] SOUSA T.: Generic refraction simulation. In *GPU Gems II* (2005), pp. 295–305.

[SSP04] SIMON A., SMITH R. C., PAWLICKI R. R.: Omnistereo for panoramic virtual environment display systems. In *VR* (2004), pp. 67–74.

[WS03] WAND M., STRASSER W.: Real-time caustics. *Computer Graphics Forum 22*, 3 (2003), 611–620.

[Wym05] WYMAN C.: Interactive image-space refraction of nearby geometry. In *Proceedings of GRAPHITE 2005* (2005).

[Wym06] WYMAN C.: Interactive image-space techniques for approximating caustics. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (2006).

[YM04] YU J., MCMILLAN L.: A framework for multiperspective rendering. In *Rendering Techniques* (2004), pp. 61–68.

[YM05] YU J., MCMILLAN L.: Modelling reflections via multiperspective imaging. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1* (2005), pp. 117–124.

[Yu05] YU J.: Personal communication, 2005.

[YYM05] YU J., YANG J., MCMILLAN L.: Real-time reflection mapping with parallax. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games* (2005), pp. 133–138.