Interactive Image-Space Refraction of Nearby Geometry

Chris Wyman*
University of Iowa

Abstract

Interactive applications often strive for realism, but framerate constraints usually limit realistic effects to those that run efficiently in graphics hardware. One effect largely ignored in interactive applications is refraction. We build upon a simple, image-space approach to refraction [Wyman 2005] that easily runs on modern graphics cards. This image-space approach requires two passes on a GPU, and allows refraction of distant environments through two interfaces. Our work explores extensions allowing the refraction of nearby opaque objects, at the cost of one additional pass to render nearby geometry to texture and a more complex fragment shader for computing refracted color. Like all image-based algorithms, aliasing can occur in certain circumstances, especially when a few texels are magnified to cover a sizable portion of screen space. However, our plausible refractions should suffice for many applications.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: interactive rendering, refraction, hardware

1 Introduction

While photorealism is beneficial in many scenarios, for many applications interactivity must take precedence over realism. Thus many researchers have approximated realistic effects to optimize for speed. Increasing computational power through parallelism [Wald et al. 2002] allows interactive realism with standard ray and path tracing techniques, yet the high cost for such parallelism often prohibits its use in mainstream applications. Recently, researchers have developed hardware-accelerated or hardware-assisted approximations for shadows [Assarsson and Akenine-Möller 2003; Chan and Durand 2003; Kautz et al. 2004; Wyman and Hansen 2003], caustics [Wand and Straßer 2003; Purcell et al. 2003], global illumination [Ng et al. 2004; Sloan et al. 2002], reflection [Ofek and Rappoport 1999], and basic refraction [Guy and Soler 2004; Schmidt 2003; Ts'o and Barsky 1987].

This paper expands upon recent work on refraction [Wyman 2005], which eliminated the restriction limiting refraction to a single interface in interactive applications. This proves important, since few situations arise in scenes where refraction occurs at *only* one surface. Generally people look *through* dielectric objects (e.g., a magnifying glass), rather than *into* them (e.g., a lake). However, this image-space technique is limited to dielectric objects inside distant environments. This paper explores a number of approaches to view opaque, nearby objects refracted through such dielectric geometry (see Figure 1).

Figure 1: *This 250,000 polygon glass dragon can refract nearby geometry at interactive rates, including complex objects such as a second copy of the dragon.*

Various ways to approximate refraction through a single surface have been proposed. One technique utilizes the programmable features of GPUs to compute refracted or pseudo-refracted directions through visible geometry. Using the refracted direction, a color is determined by either indexing into a distant environment map [Lindholm et al. 2001] or a perturbed texture describing nearby geometry [Oliveira 2000; Sousa 2005]. Schmidt [2003] used a geometric technique, similar to the reflection work of Ofek and Rappoport [1999]. Ofek's method exhibits problems with reflections off concave objects, but Schmidt's approach shows similar problems for refractions through either concave or convex objects. Accurately connecting the refracted virtual vertices generally proves difficult, frequently leading to unrealistic results.

A few researchers have considered multi-sided refraction. Guy and Soler [2004] interactively rendered simple convex gemstones. Using plane-based refraction, they compute refracted vertices (similar to Diefenbach and Badler [1997]) and update the resulting facet tree each frame using GPU-based shaders. Such per-facet planar refractions may not scale well to more complex objects, and are limited to objects with a single normal per facet.

Kay and Greenberg [1979] introduced a "thickness" parameter to account for two-sided refractive objects. Their approach works for objects of uniform thickness, such as a glass window pane, but geometry often varies considerably in thickness. Diefenbach and Badler [1997] used multiple passes to render planar reflections, refractions, and shadows interactively on graphics hardware. Ohbuchi [2003] suggested a vertex tracing preprocess to approximate interactive refractions on a prototype multimedia processor.

Recent work [Wyman 2005] uses a two-pass image-space approach to render approximate, two-interface refractions interactively on modern graphics hardware. Unfortunately that technique only handles refractions of distance environments. This work looks to eliminate that restriction.

Heidrich et al. [1999] represented geometry as a light field

*E-mail: cwyman@cs.uiowa.edu

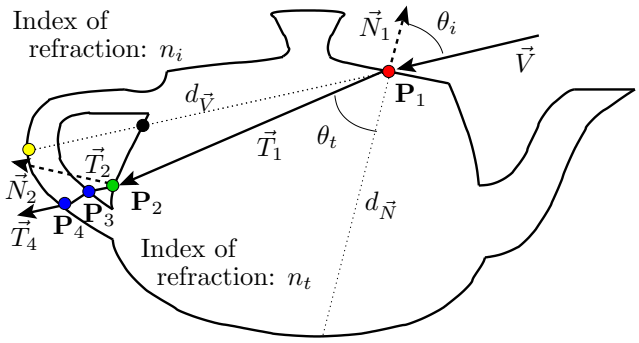

Figure 2: *Vector $\vec{V}$ hits the surface at $\mathbf{P}_1$ and refracts in direction $\vec{T}_1$ based upon the incident angle θ_i with the normal $\vec{N}_1$. Physically accurate computations lead to further refractions at $\mathbf{P}_2$, $\mathbf{P}_3$, and $\mathbf{P}_4$. Our method only refracts twice, approximating the location of $\mathbf{P}_2$ using distances $d_{\vec{N}}$ and $d_{\vec{V}}$.*

and examined compression schemes to fit such dense samplings in memory for interactive rendering. Still, the high memory overhead makes light fields unattractive for many applications. Hakura and Snyder [2001] accelerated ray tracing by utilizing the GPU as a secondary processor. Unfortunately their technique does not produce real-time results, even using both the CPU and GPU solely for rendering.

2 Basic Refraction

Given basic information about the geometry at hitpoint $\mathbf{P}_1$ (see Figure 2), refracted ray behavior following Snell's Law can be computed:

$$n_i \sin\theta_i = n_t \sin\theta_t,$$

where n_i and n_t are the indices of refraction for the incident and transmission media. θ_i describes the angle between the incident vector $\vec{V}$ and the surface normal $\vec{N}_1$, and θ_t gives the angle between the transmitted vector $\vec{T}_1$ and $-\vec{N}_1$.

Computing refractions through complex objects is trivial with a ray tracer, as rays are independently intersected with the geometry, and refracted rays recursively apply Snell's Law at each additional intersection. Unfortunately, in a GPU's stream processing paradigm performing independent recursive intersections for different pixels proves expensive. Consider the example in Figure 2. Rasterization determines $\vec{V}$, $\mathbf{P}_1$, and $\vec{N}_1$, and a simple fragment shader can compute $\vec{T}_1$. However, precisely determining point $\mathbf{P}_2$ on a GPU requires resorting to accelerated ray-based approaches [Purcell et al. 2003]. Since ray tracing techniques for GPUs are still relatively slow, multiple-bounce, ray traced refractions for complex polygonal objects are not interactive.

3 Review of Image-Space Refraction

Instead of using the GPU for ray tracing, an image-space approach (e.g., [Wyman 2005]) approximates the refraction with values easily computable via rasterization.

Consider the information necessary for refraction (see Figure 2). At each point $\mathbf{P}_j$ an incident direction $\vec{I}_j$, a normal $\vec{N}_j$, and the indices of refraction n_i and n_t are required to apply Snell's Law. $\vec{I}_j$ is always known; at depth $j = 1$, $\vec{I}_j$ is the viewing vector $\vec{V}$. At higher depths, $\vec{I}_j$ is the refraction direction $\vec{T}_{j-1}$ from the previous interface. The trick on the GPU is quickly finding subsequent normals ($\vec{N}_2$, $\vec{N}_3$, etc.) Obviously a normal $\vec{N}_j$ varies depending on the corresponding surface point $\vec{P}_j$, so a GPU-based refraction approximation could first approximate the locations of the secondary refractions ($\tilde{\mathbf{P}}_j$, $j > 1$), then use these locations to determine the normals.

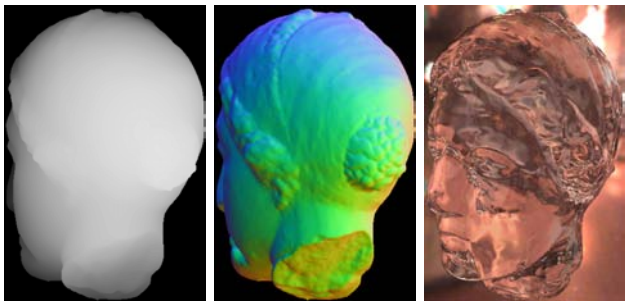

Figure 3: *Pass results for basic image-space refraction. (Left) The distance to back faces, (center) normals at back faces, and (right) the final result.*

Wyman [2005] showed how to compute a single secondary refraction at $\mathbf{P}_2$ by approximating the distance $\|\mathbf{P}_2 - \mathbf{P}_1\|$ via interpolation between two easily computable distances $d_{\vec{V}}$ and $d_{\vec{N}}$. This interpolated distance $\tilde{d}$ is then used to approximate $\mathbf{P}_2$ as:

$$\tilde{\mathbf{P}}_2 = \mathbf{P}_1 + \tilde{d}\vec{T}_1 \approx \mathbf{P}_1 + \|\mathbf{P}_2 - \mathbf{P}_1\|\vec{T}_1 = \mathbf{P}_2.$$

We use the same interpolation for $\tilde{d}$ in this paper, where:

$$\tilde{d} = \frac{\theta_t}{\theta_i} d_{\vec{V}} + \left(1 - \frac{\theta_t}{\theta_i}\right) d_{\vec{N}}.$$

After approximating $\tilde{\mathbf{P}}_2$, a normal can be found by projecting to screen space and indexing into a texture containing the normals at backfacing surfaces.

The basic image-space approach can thus be performed in three basic steps (see Figure 3):

Precompute Precompute $d_{\vec{N}}$ at each vertex.
First pass Render normals and depths of backfacing surfaces to a buffer.
Final pass Draw front faces:
- A. Compute $\vec{T}_1$ based on $\vec{V}$, $\mathbf{P}_1$, and $\vec{N}_1$.
- B. Compute $d_{\vec{V}}$ between front & back faces.
- C. Interpolate between $d_{\vec{V}}$ and precomputed $d_{\vec{N}}$.
- D. Compute $\tilde{\mathbf{P}}_2 = \mathbf{P}_1 + \tilde{d}\vec{T}_1$.
- E. Project $\tilde{\mathbf{P}}_2$ to buffer from 1^{st} pass to find $\vec{N}_2$.
- F. Refract and return color from environment.

The two major restrictions of this approach are that objects can only refract infinite environment maps, not nearby objects, and that refractions are only allowed at two interfaces. The next section describes new work that extends this technique to refract nearby objects.

4 Refraction of Nearby Geometry

Consider a refractor with nearby geometry, as in Figure 4. Using the image-space technique from above, we have an approximate exitant location $\tilde{\mathbf{P}}_2$ and an approximate exitant

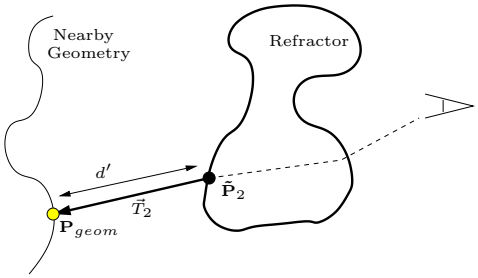

Figure 4: *After using previous work to compute* $\tilde{\mathbf{P}}_2$ *and* $\vec{T}_2$*, finding* $\mathbf{P}_{geom}$*, where* $\vec{T}_2$ *ray hits nearby geometry, simply requires approximating* d'*.*

refraction vector $\vec{T}_2$. As with computing $\tilde{\mathbf{P}}_2$, the trick to locating $\mathbf{P}_{geom}$ on the nearby geometry is approximating the distance d', where $d' = \|\mathbf{P}_{geom} - \tilde{\mathbf{P}}_2\|$.

Once again, ray tracing easily solves this problem – simply intersect $\vec{T}_2$ with all other geometry in the scene. Unfortunately, such intersections have significant cost in complex, dynamic scenes. Our goal was to find a simple image-based way to approximate this intersection. The rest of this section describes three techniques we explored for intersecting nearby geometry to find a refracted color.

4.1 The Kay-Greenberg Technique

Kay and Greenberg [1979] introduced a refraction technique for rasterization-based graphics systems. Their method assumes relatively thin surfaces with a constant, predefined thickness and assumes rays travel parallel to the viewing direction when not inside the refractive media.

While the image-space refraction eliminates the first restriction, a simple approach for handling nearby geometry keeps the second assumption. In practice, this works as follows: before the final refraction pass from Section 3, draw all the non-refractive nearby geometry, as seen from the eye, to a texture. After computing $\tilde{\mathbf{P}}_2$ in the final pass, project it into screen-space using the standard OpenGL projection matrix, and use the coordinates to index into this texture. If the texture indicates that no geometry is visible, use $\vec{T}_2$ to index into the background environment map.

While this technique is simple to implement and requires little additional overhead, it gives relatively unsatisfactory results, as seen in Figure 5. In fact, objects rarely look more than slightly refractive, as only geometry directly behind the refractor can be visible in the refraction.

4.2 The Ray-Plane Intersection Technique

While ray tracing is too costly to interactively intersect the refracted scene geometry, we can rely on ray tracing for inspiration. For instance, if scene geometry consists of a few planar surfaces, we can use exact ray-plane intersection on a per-pixel level. For instance, in Figure 5, the non-refractive geometry consists of only two planes. Explicitly intersecting the two planes containing the color chart and the wood texture only requires 15 extra pixel shader instructions. As shown, this method compares favorably to a ray traced solution. Furthermore, a few additional planes can easily be added using a pixel shader to loop over a list of planes.

In practice, the ray-plane method works as follows: before the final refraction pass, draw all the nearby geometry to a texture (the NG texture). On the CPU, determine the

Figure 6: *(Left) Distortion occurs when using the ray-plane approach. (Center) Some rays completely miss the dragon with the iterative method introduced by Ohbuchi [Ohbuchi 2003]. (Right) Our technique seeds the iterative method with a better location, significantly reducing the problem.*

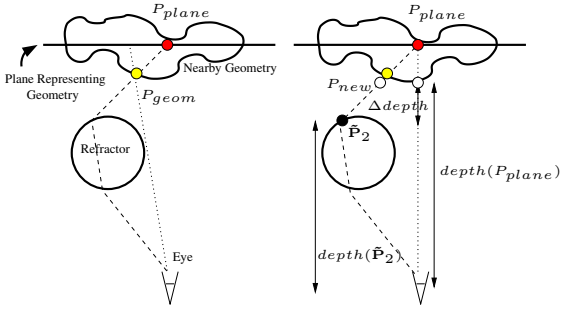

Figure 7: *(Left) In the ray-plane technique, rays leaving the refractor intersect the approximation plane at* $\mathbf{P}_{plane}$ *instead of the geometry at* $\mathbf{P}_{geom}$*. Using* $\mathbf{P}_{plane}$ *instead of* $\mathbf{P}_{geom}$ *to index into the NG texture may generate incorrect colors. (Right) Instead of using* $\mathbf{P}_{plane}$ *to find a color, the NG texture's z-buffer value can be used to compute* $\Delta depth$*, which is used to iteratively generate a new intersection* $\mathbf{P}_{new}$*.*

equations for a small number of planes describing the scene. After computing $\vec{T}_2$ in the final render pass, intersect the ray $\tilde{\mathbf{P}}_2 + t\vec{T}_2$ with each of the precomputed planes. Project the nearest intersection point into the NG texture and return the color stored (or the background).

While at first this technique seems limited to planar objects, complex geometry lying relatively far from the refractor can reasonably be approximated using a plane. This may seem unintuitive, but the plane intersection is simply used to determine an index into the NG texture. The geometry rendered into that texture can be arbitrary, as seen in Figure 6. However, representing complex geometry with a plane causes in distortion due to inaccurate distances to the refractor.

4.3 The Iterative Lookup Technique

When using ray-plane intersection to approximate complex geometry, difficulty finding the correct intersection location on the plane causes this distortion. Rays often intersect the object ($\mathbf{P}_{geom}$) either in front or behind the representative plane ($\mathbf{P}_{plane}$). In such cases, the texel accessed by projecting $\mathbf{P}_{plane}$ can be wrong (see Figure 7).

Instead of using the ray's intersection with the plane to approximate the color, $\mathbf{P}_{plane}$ can instead be used to iteratively approximate the ray-object intersection $\mathbf{P}_{new}$. Instead of directly using the color in the NG texture, as in Section 4.2, a z-buffer associated with the texture can ap-

Figure 5: *Compare image-based refraction of nearby geometry using (left) the Kay-Greenberg technique and (center) the ray-plane intersection technique with (right) a ray traced image.*

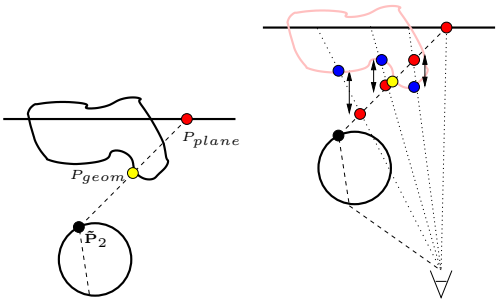

Figure 8: *(Left) Using the iterative technique, $\mathbf{P}_{plane}$ may incorrectly indicate some refraction rays completely miss nearby geometry. (Right) Indexing into the NG texture at multiple locations along the refracted ray and iterating from the best point significantly reduces the problem. The "best" sample is where red and blue points are closest.*

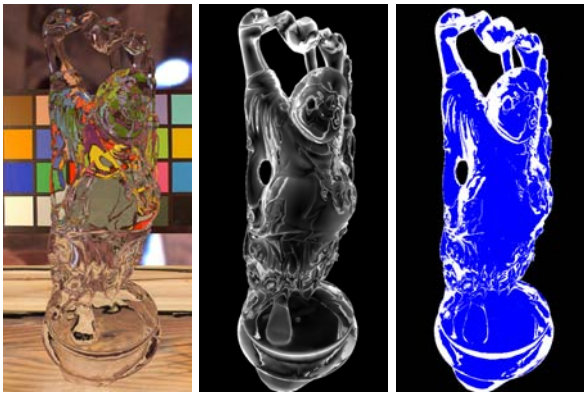

Figure 9: *(Left) Geometry seen through Buddha. (Center) Feature size guarantees for visibility of nearby geometry. White pixels guarantee all features greater than 10 pixels. (Right) 71% of pixels, in blue, guarantee visibility for refracted geometry features at least 3 pixels large.*

proximate the distance from $\tilde{\mathbf{P}}_2$ to a better approximation of the intersection, $\mathbf{P}_{new}$. This process can be repeated, using $\mathbf{P}_{new}$ to compute a further approximation $\mathbf{P}'_{new}$, etc. Typically fewer than five iterations are necessary. This basic idea was suggested by Ohbuchi [2003].

The problem with the iterative technique, as proposed by Ohbuchi and described above, is that the initial intersection $\mathbf{P}_{plane}$ may falsely indicate that $\vec{T}_2$ misses all nearby geometry and instead hits the background! In that case, the solution gives erroneous results (see Figures 6 and 8) characterized by refracted geometry partially disappearing. Only when $\mathbf{P}_{plane}$ and background geometry project to the same NG texel can the refraction see nearby geometry. This is similar to the limitation of the Kay-Greenberg approach, where $\tilde{\mathbf{P}}_2$ and the background geometry need to project to the same texel.

To reduce the problem, we propose indexing into the NG texture's z-buffer multiple times and using the point with the closest z-value as input to the iterative technique. This is equivalent to intersecting the refracted ray $\vec{T}_2$ with multiple planes perpendicular to the viewing direction, projecting each intersection into the NG texture, and choosing the texel whose depth best corresponds to the actual distance (see Figure 8, where the best point minimizes the distance between red and blue points). Specifically, if d_{far} is the distance to the far plane, we compute $\tilde{\mathbf{P}}_2 + \alpha_i \vec{T}_2$ for a number of values $0 \leq \alpha_i \leq d_{far}$, and project the results into the NG texture's z-buffer.

The number of values α_i used to find a good seed for the iterative approach determines the feature size of nearby geometry guaranteed to be captured with our technique. In our experiments, seven intersections were required to insure all features larger than ten pixels were visible at 512^2 resolutions (as in the accompanying video). For 71% of pixels, seven intersections capture all features larger than three pixels, as seen in Figure 9. The screen-space distance $\vec{T}_2$ travels between $\tilde{\mathbf{P}}_2$ and the far plane (divided by the number of intersections α_i) determines the guaranteed feature size. Obviously, when $\vec{T}_2$ varies significantly from the view direction this distance (and hence the error) increases. However, the areas of greatest error roughly correspond to areas of total internal reflection, where inherently noisy refractions will mask artifacts.

The refraction algorithm from Section 3 is then updated to handle nearby geometry as follows:

Precompute Precompute $d_{\vec{N}}$ at each vertex.
First pass Render normals and depths at backfacing surfaces to a buffer (Fig. 3).
2nd pass Draw nearby geometry to texture (Fig. 10).
Final pass Draw front faces:
- A-F. As before, through computation of $\tilde{\mathbf{P}}_2$ and $\vec{T}_2$.
- G. Compute $\tilde{\mathbf{P}}_2 + \alpha_i \vec{T}_2$ for various α_i.
- H. Project intersections into NG texture z-buffer.
- I. Use best intersection to find $\mathbf{P}_{new}$, then iterate.
- J. Project iterated location to find color.

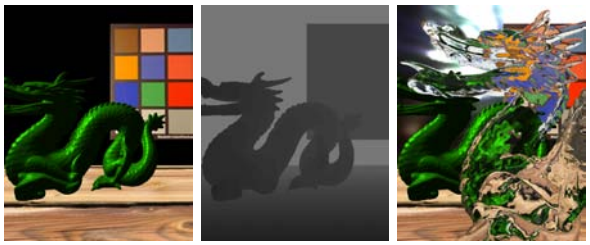

Figure 10: *(Left) The texture of nearby geometry. (Center) The NG texture's z-buffer. (Right) The final result.*

	2-Sided Refraction at 1024^2	With Nearby Geometry at 1024^2	With Nearby Geometry at 512^2
Buddha	53.3 fps		
Buddha & Dragon		10.7 fps	15.6 fps
Buddha w/2 planes		20.3 fps	45.7 fps
Dragon	20.7 fps		
Dragon w/2 planes		8.5 fps	17.8 fps
Sphere	164.4 fps		
Sphere & Dragon		16.9 fps	21.3 fps
Sphere w/1 plane		61.0 fps	212.8 fps
Venus	37.9 fps		
Venus & Dragon		9.4 fps	12.6 fps

Table 1: *Framerate comparisons for scenes of varying complexity using our iterative approach.*

5 Implementation and Results

We implemented our work in OpenGL on an AGP 8x nVidia GeForce 6800 with 128 MB memory, using Cg to generate standard ARB vertex and fragment shaders. The algorithm requires three passes. The first pass renders the distance to back facing polygons and stores their normals, as in Figure 3. The second pass renders nearby geometry to a texture, as in Figure 10. The final pass renders front facing polygons and computes refractions with a fragment shader. Our final pass vertex shader compiles to 27 ARB assembler instructions and the fragment shader requires 226 instructions. This compares to 23 and 102 instructions for image-space refraction without nearby geometry.

Running on a 3.0 GHz Pentium 4 with 2 GB of memory, we get the running times shown in Table 1. Our code is unoptimized, yet we achieve good speeds for even complex models. At least for complex geometry, the bottleneck is not our refraction shader, as rendering complex objects in the background significantly reduces the framerate. The Buddha, Venus, and Dragon models we use have 50k, 100k, and 250k polygons (respectively). Note that some timings are for scenes in the accompanying video but not depicted here.

Unless otherwise specified, all refractors have an index of refraction of 1.2, except for the spheres, which have an index of 1.5. Figure 1 shows our iterative approach on the complex dragon model, with both simple and complex background geometry. The left image in Figure 1 shows the same scene as in Figure 5. Our results remain comparable with ray traced results as the index of refraction changes, as seen in Figure 11. While the errors of our technique increase with higher indices of refraction, the largest occur in noisy-appearing regions of the refraction, particularly areas of total internal reflection. In such regions, viewers may not notice even sizable discrepancies from real refractions.

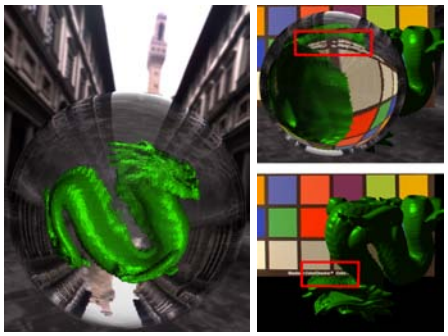

Figure 13: *Errors arising in image-space refraction. (Left) Aliasing from a combination of $d_{\vec{V}}$ discretization and a low resolution NG texture. (Top) The artifact visible in the highlighted region occurs because the NG texture (bottom) does not store information about changing visibility, particularly in the highlighted region where some refraction rays should pass between the objects.*

Figure 14: *Errors arising in image-space refraction. (Left) Incorrect handling of total internal reflection. (Right) Aliasing problems become more noticeable as $\vec{T}_2$ diverges further from the view vector. In the sphere, aliasing becomes visible near the silhouettes.*

6 Limitations

The four major limitations of image-space refraction are: only two refractive interfaces, incorrect handling of total internal reflection, aliasing, and occasional incorrect visibility for nearby geometry due to a single NG texture. The first two limitations are inherited from previous work and could perhaps improve with future work.

Aliasing arises in numerous steps during refraction, as virtually all data are stored inside discretized textures. For instance, even though the ray-plane method computes an exact ray-plane intersection in the pixel shader, the results differ slightly from ray traced results due to aliasing during computation of $\tilde{\mathbf{P}}_2$ and $\vec{T}_2$. Figure 12 shows where these errors occur.

Aliasing also occurs when indexing into the NG texture to determine which nearby objects are visible. The iterative process can multiply the problem, particularly in areas where a small object is magnified to a significant size, as in Figure 13. However even in less magnified areas, aliasing becomes more noticeable as $\vec{T}_2$ varies further from the viewing vector. This is evident near the silhouettes of the sphere in Figure 14.

Another problem that occurs, particularly when multiple

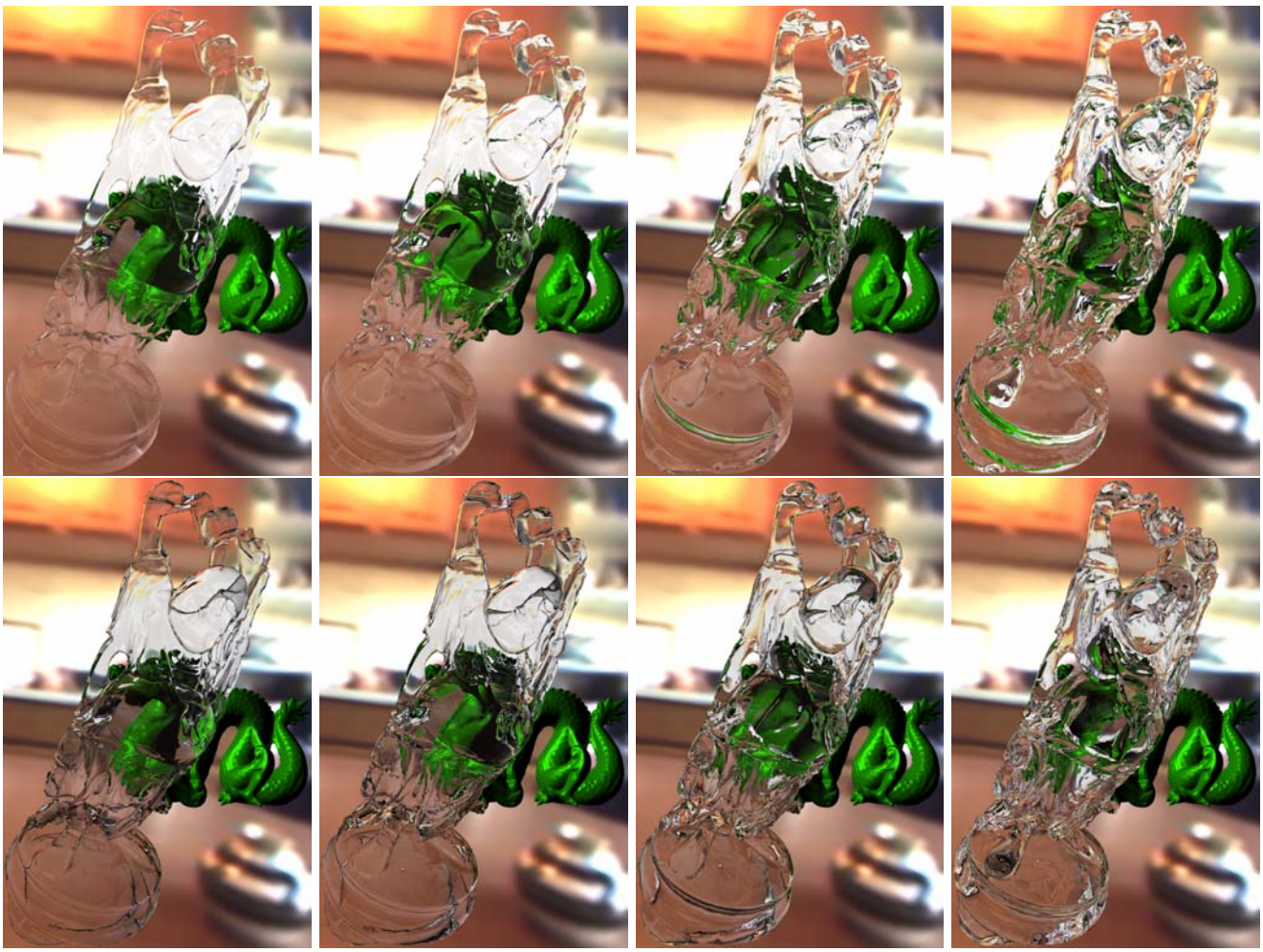

Figure 11: *A 250k polygon dragon behind a 50k polygon glass Buddha. The index of refraction increases from left: 1.05, 1.1, 1.2, and 1.4. Compare our iterative method (top) to ray tracing (bottom).*

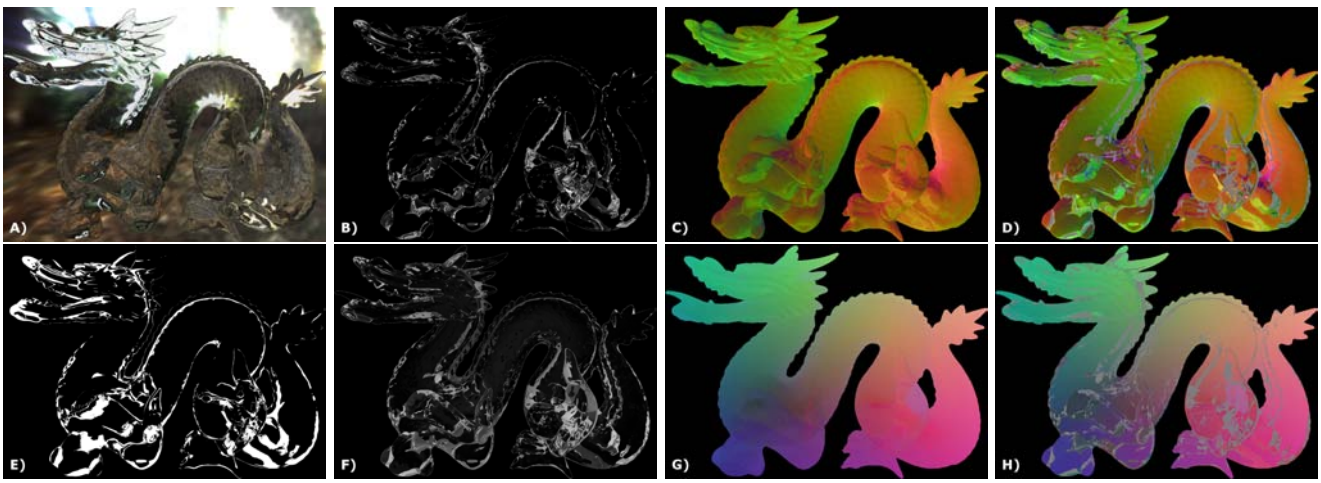

Figure 12: *Aliasing errors from (a) the original image-space refraction technique. Refracted rays $\vec{T}_2$ computed in (c) image-space versus rays computed by (d) ray tracing. The visualized angular differences (b) show largest errors in or near regions of total internal reflection (e). Compare the exitant positions $\tilde{\mathbf{P}}_2$ computed in (g) image-space versus exact positions from (h) ray tracing. The difference image (f) is scaled such that white indicates a distance error $\frac{1}{2}$ the width of the dragon. Most regions without total internal reflection have errors of a few pixels or less.*

objects approach the refractor, arises from the geometry texture lacking complete visibility information (see Figure 13). This points out an implicit assumption in our technique: that visibility between refracted geometry does not change over the surface of the refractor.

7 Conclusions and Future Work

This paper examined extensions to a simple, image-space refraction approach that allow refraction of nearby geometry in addition to distant environments. These techniques run interactively on current GPUs, even for quite complex models, and results of the iterative technique compare favorably with ray traced results.

The problems of our method include that refractions are limited to two interfaces. However, for many applications two interfaces are sufficient to generate plausible results. Aliasing can poses problems on a number of levels. Most noticeably, aliasing occurs because nearby geometry is stored in a texture; a refractor can magnify the texels to cover many pixels. Finally, since nearby geometry is stored in a single texture, visibility changes over the refractor's surface are always not correctly handled.

Recent work [Donnelly 2005] has examined ways of implementing per-pixel displacement mapping in hardware. Intersecting refracted rays with nearby geometry is quite similar to per-pixel displacement mapping, only with *dynamic* displacement maps. Future work allowing for dynamic changing displacement maps could potentially increase the speed and accuracy of image-space refraction.

Finally, caustics arise from light focused off reflective or refractive objects. A number of interactive caustic techniques are based on previous work for one-interface refraction. We plan on examining future work extending our technique to generate interactive image-space caustics through two or more interfaces.

References

Assarsson, U., and Akenine-Möller, T. 2003. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics 22*, 3 (July), 511–520.

Chan, E., and Durand, F. 2003. Rendering fake soft shadows with smoothies. In *Proceedings of the Eurographics Symposium on Rendering*, 208–218.

Diefenbach, P., and Badler, N. 1997. Multi-pass pipeline rendering: Realism for dynamic environments. In *Proceedings of the Symposium on Interactive 3D Graphics*, 59–70.

Donnelly, W. 2005. *GPU Gems 2*. Addison-Wesley, March, ch. Per-Pixel Displacement Mapping with Distance Functions, 123–136.

Guy, S., and Soler, C. 2004. Graphics gems revisited: Fast and physically-based rendering of gemstones. *ACM Transactions on Graphics 23*, 3, 231–238.

Hakura, Z. S., and Snyder, J. M. 2001. Realistic reflections and refractions on graphics hardware with hybrid rendering and layered environment maps. In *Proceedings of the Eurographics Rendering Workshop*, 289–300.

Heidrich, W., Lensch, H., Cohen, M. F., and Seidel, H.-P. 1999. Light field techniques for reflections and refractions. In *Proceedings of the Eurographics Rendering Workshop*, 187–196.

Kautz, J., Lehtinen, J., and Aila, T. 2004. Hemispherical rasterization for self-shadowing of dynamic objects. In *Proceedings of the Eurographics Symposium on Rendering*, 179–184.

Kay, D. S., and Greenberg, D. 1979. Transparency for computer synthesized images. In *Proceedings of SIGGRAPH*, 158–164.

Lindholm, E., Kligard, M. J., and Moreton, H. 2001. A user-programmable vertex engine. In *Proceedings of SIGGRAPH*, 149–158.

Ng, R., Ramamoorthi, R., and Hanrahan, P. 2004. Triplet product wavelet integrals for all-frequency relighting. *ACM Transactions on Graphics 23*, 3, 477–487.

Ofek, E., and Rappoport, A. 1999. Interactive reflections on curved objects. In *Proceedings of SIGGRAPH*, 333–342.

Ohbuchi, E. 2003. A real-time refraction renderer for volume objects using a polygon-rendering scheme. In *Proceedings of Computer Graphics International*, 190–195.

Oliveira, G., 2000. Refractive texture mapping, part two. Gamasutra, November. http://www.gamasutra/features/20001117/oliveira_01.htm.

Purcell, T., Donner, C., Cammarano, M., Jensen, H. W., and Hanrahan, P. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the SIGGRAPH/Eurographics Conference on Graphics Hardware*, 41–50.

Schmidt, C. M. 2003. *Simulating Refraction Using Geometric Transforms*. Master's thesis, Computer Science Department, University of Utah.

Sloan, P.-P., Kautz, J., and Snyder, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics 21*, 3, 527–536.

Sousa, T. 2005. *GPU Gems 2*. Addison-Wesley, March, ch. Generic Refraction Simulation, 295–305.

Ts'o, P. Y., and Barsky, B. A. 1987. Modeling and rendering waves: wave-tracing using beta-splines and reflective and refractive texture mapping. *ACM Trans. Graph. 6*, 3, 191–214.

Wald, I., Kollig, T., Benthin, C., Keller, A., and Slusallek, P. 2002. Interactive global illumination using fast ray tracing. In *Proceedings of the Eurographics Rendering Workshop*, 15–24.

Wand, M., and Strasser, W. 2003. Real-time caustics. *Computer Graphics Forum 22*, 3, 611–620.

Wyman, C., and Hansen, C. 2003. Penumbra maps: Approximate soft shadows in real-time. In *Proceedings of the Eurographics Symposium on Rendering*, 202–207.

Wyman, C. 2005. An approximate image-space approach for interactive refraction. *ACM Transactions on Graphics 24*, 3 (July), 1050–1053.