An Approximate Image-Space Approach for Interactive Refraction

Chris Wyman*
University of Iowa

Abstract

Many interactive applications strive for realistic renderings, but framerate constraints usually limit realism to effects that run efficiently in graphics hardware. One effect largely ignored in such applications is refraction. We introduce a simple, image-space approach to refractions that easily runs on modern graphics cards. Our method requires two passes on a GPU, and allows refraction of a distant environment through two interfaces, compared to current interactive techniques that are restricted to a single interface. Like all image-based algorithms, aliasing can occur in certain circumstances, but the plausible refractions generated with our approach should suffice for many applications.

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

Keywords: interactive rendering, refraction, hardware

1 Introduction

In many applications, interactivity must take precedence over realism. This has motivated many researchers to approximate realistic effects in order to optimize for speed. Increasing the available computational power through parallelism [Wald et al. 2002] can allow interactive realism using standard ray and path tracing techniques, though high cost often prohibits such parallelism in mainstream applications. Hence, researchers have developed hardware-accelerated 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 addresses one of the major limitations of most GPU-based interactive refraction algorithms: only allowing refraction through a single surface. Few situations arise in real scenes where refraction occurs at *only* one surface. Often people look *through* dielectric objects (e.g., a pair of glasses), so handling multiple interfaces proves important.

Researchers have proposed various ways to approximate refraction through a single surface. One approach uses the programmable features of GPUs to compute refracted or pseudo-refracted directions through visible front facing polygons. The refracted color is then determined by either indexing into a distant environment map [Lindholm et al. 2001] or a perturbed texture describing nearby geometry [Oliveira 2000]. Schmidt [2003] used a geometric technique, similar to Ofek and Rappoport's [1999] approach to reflection. Ofek's method exhibits problems when reflecting off concave objects, but Schmidt's approach exhibits similar problems even when refracting through convex objects. In such cases, accurately connecting the refracted virtual vertices to render proves difficult.

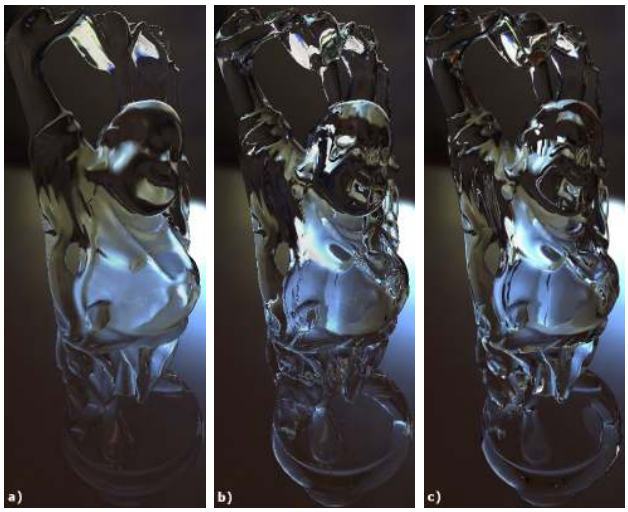

Figure 1: *Previous interactive techniques (a) only refract through one surface [Oliveira 2000]. Our approach (b) refracts through two surfaces at 53 frames per second, and compares favorably with ray tracing (c).*

A few researchers have considered multi-sided refraction. Guy and Soler [2004] interactively rendered simple convex gemstones. Since they analytically compute refracted vertices and update the resulting facet tree every frame, their approach may not scale well to more complex objects. Kay and Greenburg [1979] introduced a "thickness" parameter to account for two-sided refractive objects; this works for objects of uniform thickness, such as a window pane, but many objects vary considerably in thickness. Diefenbach and Badler [1997] used a multipass method to render planar refractions interactively on graphics hardware. Ohbuchi [2003] suggested a vertex tracing preprocess to approximate interactive refractions on a prototype multimedia processor. Heidrich et al. [Heidrich et al. 1999] used a light field representation to trade higher memory consumption for interactivity.

While not the focus of this paper, Hakura et al. [2001] accelerated ray tracing by leveraging GPU resources. Unfortunately, this approach does not produce real-time results, even utilizing both the CPU and GPU solely for rendering.

2 Basic Refraction

Computing refractions through a single interface is relatively simple given basic information about the surface at the hitpoint $\mathbf{P}_1$ (see Figure 2). The behavior of refracted rays

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

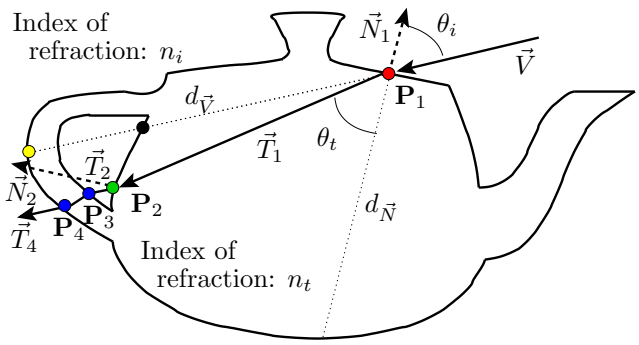

Figure 2: *Vector $\vec{V}$ hits the surface at $\mathbf{P}_1$ and refracts in direction $\vec{T}_1$ based upon the incident angle θ_i with the normal $\vec{N}_1$. Physically accurate computations lead to further refractions at $\mathbf{P}_2$, $\mathbf{P}_3$, and $\mathbf{P}_4$. Our method only refracts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

follows Snell's Law, given by:

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

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 the negated surface normal.

When ray tracing, refracting through complex objects is trivial, as refracted rays are independently intersected with the geometry, with subsequent recursive applications of Snell's Law. Unfortunately, in the GPU's stream processing paradigm performing independent operations 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 $\vec{T}_1$. Unfortunately, exactly locating point $\mathbf{P}_2$ is not possible on the GPU without resorting to accelerated ray-based approaches [Purcell et al. 2003]. Since GPU ray tracing techniques are relatively slow, multiple-bounce refractions for complex polygonal objects are not interactive.

3 Image-Space Refraction

Instead of using the GPU for ray tracing, we propose to approximate the information necessary to refract through two interfaces with values easily computable via rasterization. Consider the information known after rasterization. For each pixel, we can easily find:

- the incident direction $\vec{V}$,
- the hitpoint $\mathbf{P}_1$, and
- the surface normal $\vec{N}_1$ at $\mathbf{P}_1$.

Using this information, the transmitted direction $\vec{T}_1$ is easily computable via Snell's Law, e.g., in Lindholm et al. [2001].

Consider the information needed to find the doubly refracted ray $\vec{T}_2$. To compute $\vec{T}_2$, only $\vec{T}_1$, the point $\mathbf{P}_2$, and the normal $\vec{N}_2$ are necessary. Since finding $\vec{T}_1$ is straightforward, our major contribution is a simple method for approximating $\mathbf{P}_2$ and $\vec{N}_2$. Once again, we use an approximate point $\tilde{\mathbf{P}}_2$ and normal $\vec{N}_2$ since accurately determining them requires per-pixel ray tracing.

After finding $\vec{T}_2$, we assume we can index into an infinite environment map to find the refracted color. Future work may show ways to refract nearby geometry.

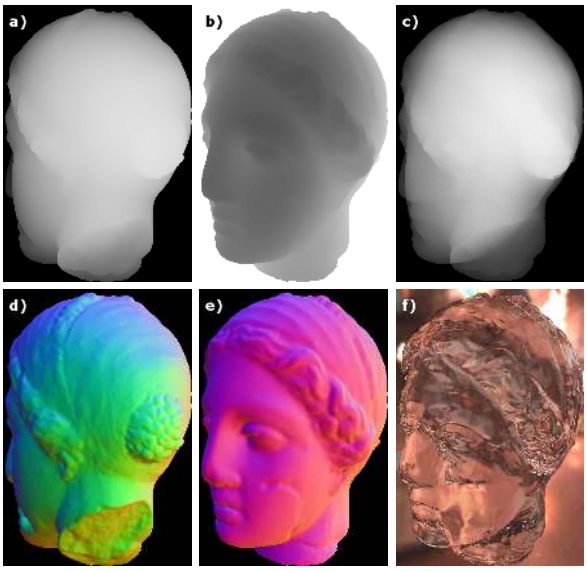

Figure 3: *Distance to back faces (a), to front faces (b), and between front and back faces (c). Normals at back faces (d) and front faces (e). The final result (f).*

3.1 Approximating the Point $\mathbf{P}_2$

While too expensive, ray tracing does provide valuable insight into how to approximate the second refraction location. Consider the parameterization of a ray $\mathbf{P}_{origin} + t\,\vec{V}_{direction}$. In our case, we can write this as: $\mathbf{P}_2 = \mathbf{P}_1 + d\,\vec{T}_1$, where d is the distance $\|\mathbf{P}_2 - \mathbf{P}_1\|$. Knowing $\mathbf{P}_1$ and $\vec{T}_1$, approximating location $\tilde{\mathbf{P}}_2$ simply requires finding an approximate distance $\tilde{d}$, such that:

$$\tilde{\mathbf{P}}_2 = \mathbf{P}_1 + \tilde{d}\,\vec{T}_1 \approx \mathbf{P}_1 + d\,\vec{T}_1$$

The easiest approximation $\tilde{d}$ is the non-refracted distance $d_{\vec{V}}$ between front and back facing geometry. This can easily be computed by rendering the refractive geometry with the depth test reversed (i.e., `GL_GREATER` instead of `GL_LESS`), storing the z-buffer in a texture (Figure 3a), rerendering normally (Figure 3b), and computing the distance using the z values from the two z-buffers (Figure 3c). This simple approximation works best for convex geometry with relatively low surface curvature and a low index of refraction.

Since refracted rays bend inward (for $n_t > n_i$) toward the inverted normal, as n_t becomes very large $\vec{T}_1$ approaches $-\vec{N}_1$. This suggests interpolating between distances $d_{\vec{V}}$ and $d_{\vec{N}}$ (see Figure 2), based on θ_i and θ_t, for a more accurate approximation $\tilde{d}$. We take this approach in our results, precompute $d_{\vec{N}}$ for every vertex, and interpolate using:

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

A precomputed sampling of d could give even better accuracy if stored in a compact, easily accessible manner. We tried storing the model as a 64^2 geometry image [Praun and Hoppe 2003] and sampling d in 64^2 directions for each texel in the geometry image. This gave a 4096^2 texture containing sampled d values. Unfortunately, interpolating over this representation resulted in noticeably discretized d values, leading to worse results than the method described above.

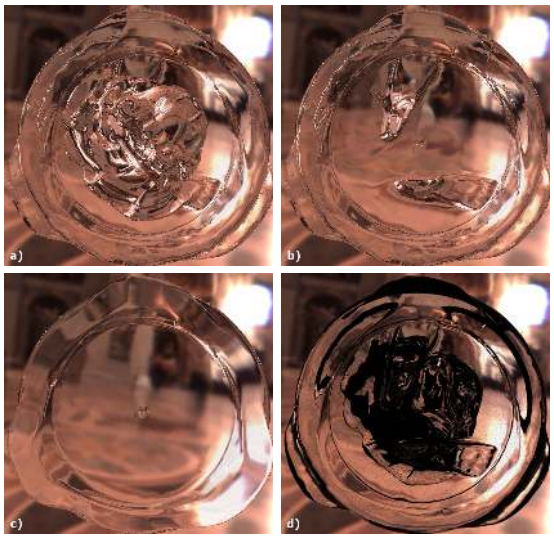

Figure 4: *The Buddha's base using (a) the furthest surface or (b) the* 2^{nd} *surface for our secondary refraction, (c) a single refractive interface, and (d) ray tracing with ray depth 6.*

3.2 Determining the Normal $\vec{N}_2$

After approximating $\tilde{\mathbf{P}}_2$, the last requirement to compute $\vec{T}_2$ is the normal $\vec{N}_2$. This information is stored with the mesh at each vertex, but determining *which* polygon $\vec{T}_1$ intersects is costly unless we rely on an image-space approach.

Consider the case of convex objects, where we can rasterize all the geometry using two passes: once culling front faces and once culling back faces. If we render the first pass to texture using the surface normal as the color (as in Figure 3d), then during the second pass we can project our approximate exitant point $\tilde{\mathbf{P}}_2$ into texture space and index into the texture to find the normal.

Extending this approach beyond convex objects proves problematic, as multiple surfaces can project to the same z-buffer texel. Our approach may extend via more textures for additional refractions, but it is unclear how to approximate further intersection locations (e.g., $\mathbf{P}_3$, $\mathbf{P}_4$ in Figure 2). Furthermore, two refractive interfaces give plausible results for many complex objects.

Without extending our approach for additional interfaces, two ways exist to handle concave objects: compute the secondary refraction point $\tilde{\mathbf{P}}_2$ based on the distance to either the furthest surface from the eye or the second surface (i.e., first backfacing surface) from the eye (the yellow or black point in Figure 2). The only difference lies in which surface is rendered to texture in the first pass and later used for computing $d_{\vec{V}}$ and $\vec{N}_2$. Figures 4 and 5 show the difference. When rendering objects with high depth complexity, using the furthest polygon shows geometry users may expect to see, whereas using the second polygon gives smoother results while ignoring more distant polygons. For nearly convex objects, this difference is barely noticeable.

3.3 Problem Cases

This approach has three problems: there is no built-in way to deal with total internal reflections, $\tilde{\mathbf{P}}_2$ may not project onto a backfacing polygon, or $\tilde{\mathbf{P}}_2$ may project onto the *wrong* polygon due to concavity.

Fragment shaders could handle total internal reflection (TIR) using a loop to bounce between front and back faces. However, this significantly slows the code in addition to multiplying errors inherent in an image-space approach. Our implementation disallows angles θ_i greater than the critical angle θ_{crit}, and clamps θ_i to θ_{crit}. Effectively, such rays "refract" tangent to the exitant surface. TIR regions tend to be noisy with only a single sample; for complex objects, such as the Buddha in Figure 1, raytracers require 16 or 64 samples per pixel in these regions for noise free results. While not physically accurate, our approach to TIR gives plausible, relatively smooth results with one sample per pixel.

Figure 5: *Two views of a horse. References are ray traced with 64 samples per pixel, the right insets compare our method with secondary refractions at either back faces or second surfaces and ray tracing with one sample per pixel.*

Consider the case where $\tilde{d}$ is too large. This causes $\tilde{\mathbf{P}}_2$ to fall outside the refractive object's silhouette, in a black texel in Figure 3d. Conceptually, this means $\vec{T}_1$ intersects the object's side. This problem occurs infrequently, generally for concave objects with indices of refraction above 1.5. To avoid artifacts in our results, we set the exitant normal perpendicular to the camera's lookat vector $\vec{L}_{at}$. In particular, we project $\vec{T}_1$ onto the plane perpendicular to $\vec{L}_{at}$ (i.e., $\vec{N}_2 \equiv \vec{T}_1 - (\vec{L}_{at} \cdot \vec{T}_1)\vec{L}_{at}$). Improving $\tilde{d}$ reduces this problem.

The final problem occurs for concave refractors, near regions where the object's depth complexity rises above two. As shown in Figure 5, the approximate point $\tilde{\mathbf{P}}_2$ from a view ray that initially hits the horse's body may project onto the leg even though $\mathbf{P}_2$ actually lies on the far side of the body. This can occur despite the choice of surface for the secondary refraction, and it is an artifact of our approach that becomes objectionable mainly for highly concave objects.

The supplementary DVD contains an additional figure examining these errors on a per-pixel basis.

4 Results

We implemented this approach in OpenGL on an AGP 8x nVidia GeForce 6800 with 128 MB memory, using Cg vertex and fragment shaders (available on the DVD as supplementary material). The algorithm requires two passes. The first pass renders the distance to back facing polygons and stores their normals, as in Figure 3a, d. The second pass renders front facing polygons and computes refractions with a fragment shader.

Assuming *BackfaceZBuf* and *BackfaceNormals* are the textures computed in the first pass, pseudocode for this fragment shader follows:

for all fragments F (given $\mathbf{P}_1$, $\vec{V}$, and $\vec{N}_1$), **do**
  $\vec{T}_1$ = *Refract(* $\vec{V}$, $\vec{N}_1$ *)*
  $d_{\vec{V}}$ = *DistanceFrontFaceToBackFace(F, BackfaceZBuf)*

	Polygon Count	2-Sided Refraction	1-Sided Refraction	Ray Traced Scene
Buddha	50,000	53.3 fps	136.8 fps	121.9 s
Cow	5,804	122.9 fps	295.7 fps	31.6 s
F-16	4,592	149.2 fps	395.9 fps	26.8 s
Sphere	1,600	164.3 fps	323.5 fps	6.1 s
Teapot	6,320	115.7 fps	306.9 fps	49.3 s
Venus	100,000	37.9 fps	112.2 fps	147.2 s

Table 1: *Comparison of framerates and ray tracing time for models of varying complexity.*

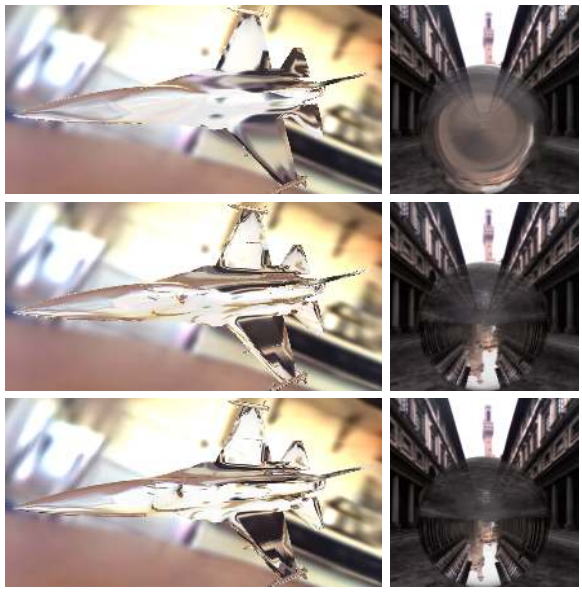

Figure 6: *Refraction through only one interface (top), using our technique (center), and ray traced (bottom). The indices of refraction are 1.2 for the jet and 1.5 for the ball.*

$d_{\vec{N}} = DistanceAlongNormal(\ \mathbf{P}_1\ )$
$\tilde{d} = WeightDistance(\ -\vec{N}_1 \cdot \vec{T}_1,\ \vec{V} \cdot \vec{T}_1,\ d_{\vec{V}},\ d_{\vec{N}}\ )$
$\tilde{\mathbf{P}}_2 = \mathbf{P}_1 + \tilde{d}\,\vec{T}_1$
$tex_{far} = ProjectToScreenSpace(\ \tilde{\mathbf{P}}_2\ )$
$\vec{N}_2 \approx TextureLookup(\ tex_{far},\ BackfaceNormals\ )$
$\vec{T}_2 \approx Refract(\ \vec{T}_1,\ \vec{N}_2\ )$
return $IndexEnvironmentMap(\ \vec{T}_2\ )$

Running on a 3.0 GHz Pentium 4 with 2 GB of memory, we get the running times shown in Table 1 for 1024^2 images. Our code for all three techniques is unoptimized, yet we achieve high speeds even for complex models. Note that some timings are for scenes in the accompanying video but not depicted here.

Most of our results use an index of refraction of 1.2, with the exception of the sphere in Figure 6 and the teapot in the video, which both have an index of 1.5. Complex geometry typically leads to chaotic and noisy-appearing refractions for moderate to high indices of refraction, even in antialiased ray traced images. In such cases a single sample per pixel, either ray traced or using our approach, leads to objectionable flickering noise during animation. We found supersampling in OpenGL helps alleviate this problem, though more than the 4 samples per pixel we tried are probably required.

5 Conclusions

This paper presented a simple, image-space approach for generating plausible interactive refractions through two surfaces. This method runs interactively on current GPUs, even for fairly complex models. The biggest limitation, which we plan to address in the future, restricts refraction to infinite environments maps, disallowing refraction of one object through another. Furthermore, we wish to better handle concave objects. Another avenue for future work examines applying interactive refractions to quickly generate caustics.

Acknowledgments: Thanks to Paul Debevec for the light probes and to Jim Cremer and the anonymous reviewers for their constructive feedback. This work was partially supported by a University of Iowa Old Gold Fellowship.

References

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

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*, 171–178.

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. http://www.gamasutra.com/features/20001117/oliveira_01.htm.

Praun, E., and Hoppe, H. 2003. Spherical parameterization and remeshing. *ACM Transactions on Graphics 22*, 3, 340–349.

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.

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 Transactions on Graphics 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.