6.3

Interactive Refractions and Caustics Using Image-Space Techniques

Chris Wyman, **University of Iowa**

Introduction

Realism plays an important role in virtually all computer graphics applications, but owing to computation constraints, most applications limit realism to that achievable with only a few milliseconds of computing time. Commonly, applications limit material properties and complex illumination, often allowing only diffuse or Phong surfaces under rather simplistic local lighting models, sometimes augmented by simple mirrors. Yet other specular effects such as refraction and caustics play important roles in many real scenes that one may want to simulate.

One of the major problems with such complex effects is their global nature. In the case of caustics, the reflective or refractive focusing of light, specular surfaces virtually anywhere in the environment potentially play an important role. Absent this complex focusing of light, even the simpler problem of rendering the distortion caused by refraction involves recursive ray-object intersection queries that are difficult to evaluate efficiently using GPUs.

Fortunately, most interactive applications aim to achieve only plausible realism instead of photo-realism. This allows the use of approximate techniques for effects such as soft shadows [Hasenfratz03], ambient occlusion [Bunnell05], indirect illumination [Dachsbacher05], et cetera. This article describes an approximate technique for rendering plausible refractions at interactive rates, using an image-space technique. After introducing this approach, we show how this method may be applied to render caustics at interactive rates. Figure 6.3.1 shows examples combining these methods.

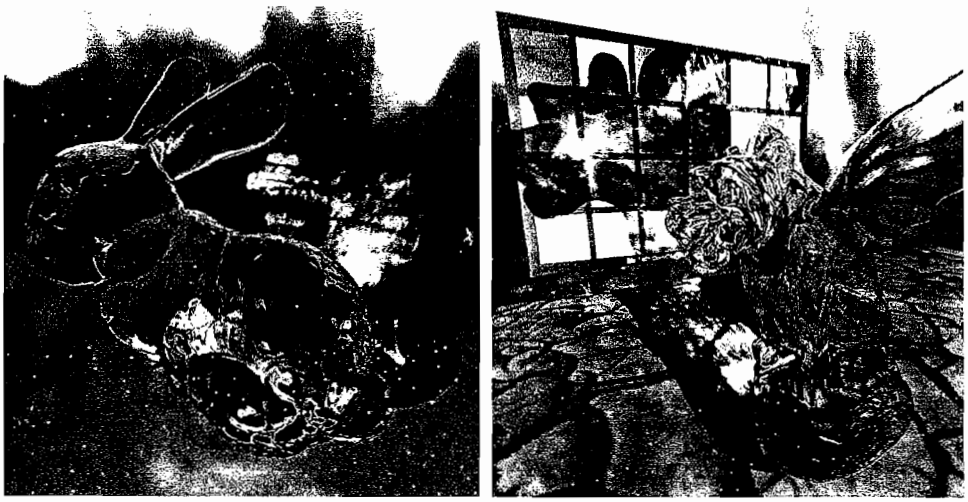

FIGURE 6.3.1 Interactive renderings achievable using image-space refraction and caustics.

Approximate, Image-Space Refraction

Approximate image-space refraction [Wyman05] aims to solve one of the major problems of previous GPU-based refraction techniques (e.g., [Lindhom01] [Oliveira00] [Schmidt03] [Sousa05]), namely the limitation to refract through only a single interface. While these existing approaches work well when rendering pools of water, most other refractive geometry involves light that passes through at least two interfaces. The primary difficulty for rendering, of course, is determining the location of additional refractions; Snell's law can easily be applied repeatedly inside a pixel shader. Ray tracing allows arbitrarily complex refractions, but most applications cannot afford GPU-based ray tracing.

The observation underlying our approach is quite simple. Standard rasterization finds the location of the initial intersection with scene geometry, and shaders can compute a refraction direction. Once the refracted direction is known, the problem of finding the second refraction reduces to a question of distance: how far along the refracted ray the second surface is located.

Unfortunately, without knowing the refractor geometry inside the pixel shader, we cannot determine this distance exactly. Only very simple refractors can currently be stored explicitly for efficient traversal inside shaders. When explicit representation proves infeasible, rasterization techniques often fall back on image-space approximations. Our approach represents the refractor geometry using an image and approximates the location of the second refraction using distances easily computed on the GPU. Snell's law can be applied a second time, and the twice-refracted ray can be

intersected with the background, using a variety of techniques. While it may be possible to extend this work to more than two refractive interfaces, we assume only two refractions occur along any given ray. Despite this limitation, our approach produces plausible results that compare well with ray-traced images.

Refraction Algorithm

Given our assumption that two refractions occur along the path of the ray, the basic steps in computing a pixel's color are as follows:

1. Find the initial intersection point $\mathbf{p}_1$ with the geometry.
2. Find the surface normal $\mathbf{N}_1$ at point $\mathbf{p}_1$.
3. Refract according to Snell's law to compute the transmitted vector $\mathbf{T}_1$.
4. Intersect the ray $\mathbf{p}_1 + d_1\mathbf{T}_1$ with the refractor to find the second intersection $\mathbf{p}_2$.
5. Find the surface normal $\mathbf{N}_2$ at point $\mathbf{p}_2$.
6. Refract according to Snell's law to compute the transmitted vector $\mathbf{T}_2$.
7. Intersect the ray $\mathbf{p}_2 + d_2\mathbf{T}_2$ with the background geometry at $\mathbf{p}_3$.
8. Compute shading at point $\mathbf{p}_3$.

Steps 1 and 2 are performed by the rasterizer, assuming normal information is passed into OpenGL or DirectX, and step 3 is straightforward to implement using pixel shaders [Oliveira00]. Steps 4 and 5 are discussed in the following sections, after which the application of Snell's law in step 6 can again be performed in a pixel shader. Step 7 is a difficult problem currently under active research, but a number of existing techniques give good results in certain circumstances. Depending on the complexity, the shading at the final point $\mathbf{p}_3$ can be computed in advance and stored in a texture, determined in the refraction shader, or performed in a deferred shading pass. The first six steps are depicted graphically in Figure 6.3.2.

Approximating Distance d_1

Given that both $\mathbf{p}_1$ and $\mathbf{T}_1$ are straightforward to compute, the difficulty in allowing rays to refract twice is identifying the value of d_1 when the transmitted ray again intersects the refractor. As ray tracing currently lacks sufficient speed, we propose approximating this distance with values easily computed or identified with a GPU.

One easily computed value is d_v, the distance between front and back surfaces of the refractor along the viewing direction (see Figure 6.3.3). Depth-peeling techniques can easily determine this value. Clearly, this approximation of d_1 is exact when the incident and transmitted indices of refraction (n_i and n_t) are equal. However, as n_i and n_t diverge, this approximation becomes more inaccurate.

Assuming that the refractor is more dense than its environment (i.e., $n_t > n_i$), the refracted ray $\mathbf{T}_1$ bends towards the negative normal $-\mathbf{N}_1$. In other words, as n_t increases toward infinity, $\mathbf{T}_1$ approaches $-\mathbf{N}_1$. Thus, for very large indices of refraction, the distance d_N shown in Figure 6.3.3 approximates the distance d_1 reasonably

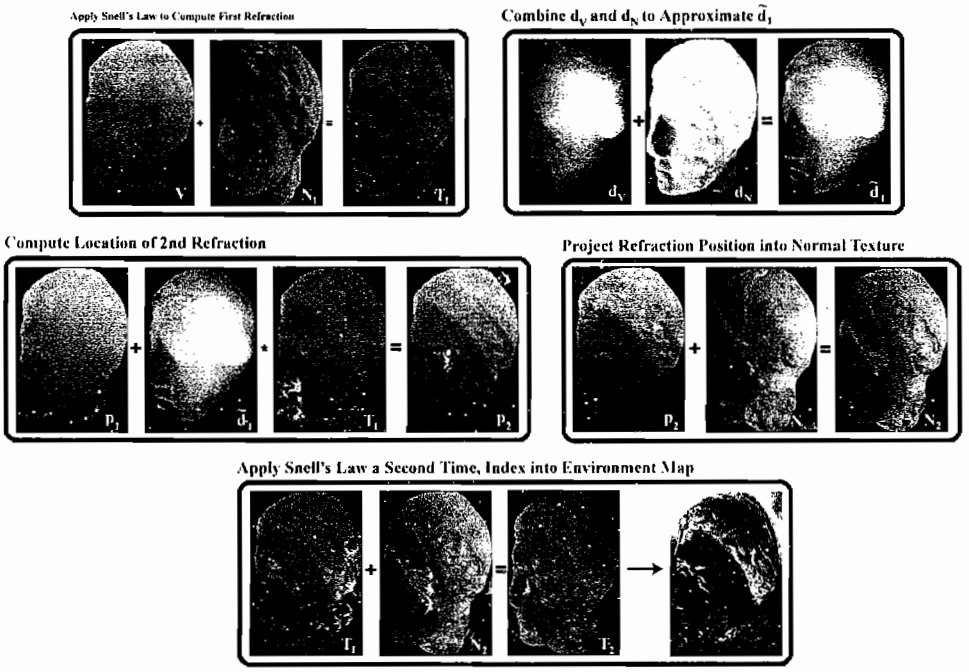

FIGURE 6.3.2 A graphical overview of rendering image-space refraction.

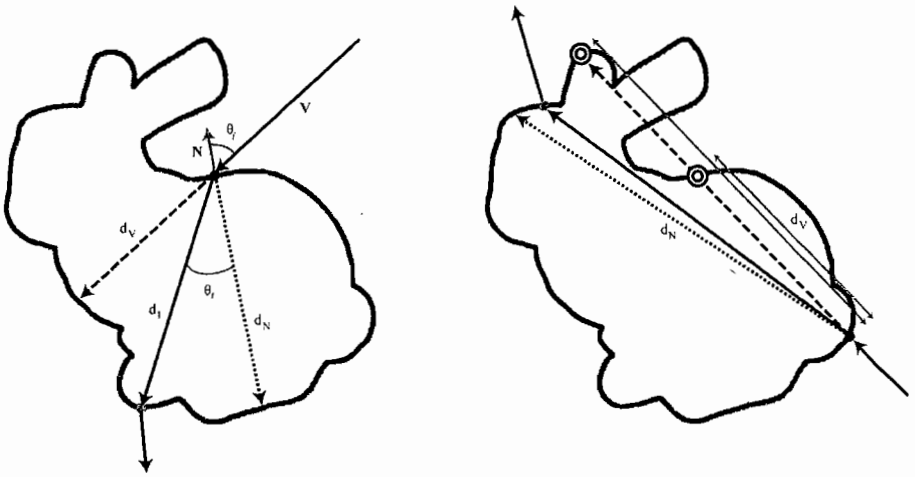

FIGURE 6.3.3 The two distance approximations d_v and d_N and interpolating between them based on θ_i and θ_t. In the case of concave refractors, we set d_v to either the distance to the furthest surface or the distance to the second surface.

well. Again, the approximation d_N is only accurate in those circumstances and loses validity as n_t decreases.

Because these two values give good approximations in the extreme cases, we propose interpolating between them based on the angles between the vectors $\mathbf{V}$, $\mathbf{N}_1$, and $\mathbf{T}_1$. Given the angles shown in Figure 6.3.3, a straightforward linear interpolation gives the approximate distance, $\tilde{d}_1$:

$$\tilde{d}_1 = \frac{\theta_t}{\theta_i} d_V + \left(1 - \frac{\theta_t}{\theta_i}\right) d_N. \tag{6.3.1}$$

When combined with the scheme for identifying the surface normal $\mathbf{N}_2$ discussed below, the approximation $\tilde{d}_1$ gives quite plausible results. Note, however, that this distance is not a physically based approximation, as the structure of the secondary refracting surface is not considered.

Approximating Surface Normal $\mathbf{N}_2$

Given the distance approximation $\tilde{d}_1$, point $\mathbf{p}_2$ can be trivially computed as $\mathbf{p}_1 + \tilde{d}_1 \mathbf{T}_1$. Without knowing the triangle containing $\mathbf{p}_2$, however, exactly determining $\mathbf{N}_2$ proves difficult since normals are associated with vertices in OpenGL and DirectX. Furthermore, owing to inaccuracies in the computation of $\tilde{d}_1$, $\mathbf{p}_2$ is unlikely to lie exactly on the refractor's surface.

Fortunately, a cheap multipass approach neatly sidesteps the issue. By first rendering a buffer storing surface normals at potential back-facing refractions, a simple texture lookup allows determination of $\mathbf{N}_2$. This can be implemented (typically concurrently with the depth peeling required for computing d_v) as follows:

1. As a first pass, render back-facing surface normals to texture map (see Figure 6.3.2).
2. After computing $\tilde{d}_1$, apply the projection matrix to $\mathbf{p}_2$ to determine texture coordinates.
3. Index into the texture map to find the normal $\mathbf{N}_2$.

Note that computing $\mathbf{N}_2$ this way masks some of the physical inaccuracies of the distance approximation. Slight deviations in $\tilde{d}_1$ still project to the same texel. Furthermore, the texture map captures some of the discontinuities not captured by the $\tilde{d}_1$ approximation.

Fixing Problem Cases

As with all image-space approximations, the algorithm has a few problems and limitations. The first two are straightforward. We have not addressed how to deal with total internal reflection or with nonconvex refractors. The third problem is a bit trickier: what happens if, owing to an inaccurate $\tilde{d}_1$, the texel we index into for $\mathbf{N}_2$ contains

no valid normal. Intuitively, this happens when $\tilde{d}_1$ is too large and we index into a black texel in Figure 6.3.2.

Because graphics applications have survived for years using only a single refraction, simply ignoring total internal reflection seems viable. Unfortunately, ignoring the issue leads to the image in Figure 6.3.4, where regions of total internal reflection have been rendered in black. Until someone develops extensions that handle recursive reflections and refractions, we suggest using one of three ad hoc methods to fill these regions with plausible colors:

- Degenerate to previous methods and use only a single refraction. This is equivalent to defining $\mathbf{T}_2 = \mathbf{T}_1$.
- "Refract" so that $\mathbf{T}_2$ leaves the surface perpendicular to the surface normal. Effectively, this clamps the incident angle at $\mathbf{p}_2$ to the critical angle.
- Compute a reflection vector (instead of a refraction vector) at $\mathbf{p}_2$ and ignore any additional interactions with the refractor.

Figure 6.3.4 compares these approaches. We recommend using the second or third approach, depending on application requirements, as the first behaves poorly in dynamic scenes.

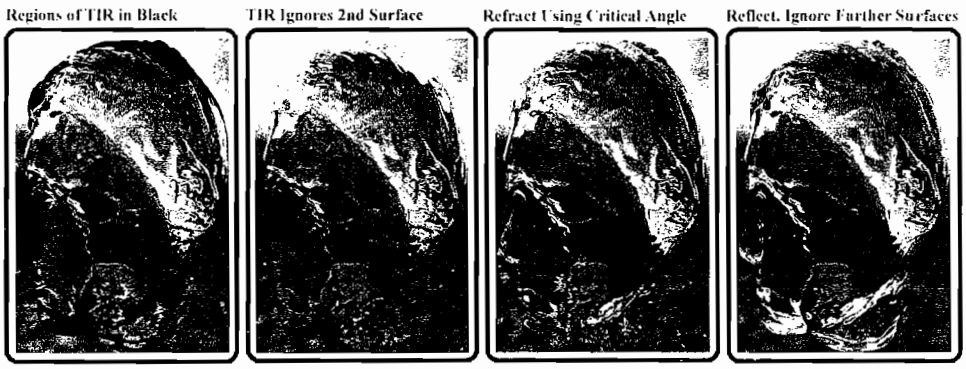

FIGURE 6.3.4 Regions of total internal reflection (TIR) can be handled in three ways: ignoring the second surface and continuing in direction $\mathbf{T}_1$, clamping to the critical angle and refracting, or reflecting and ignoring further ray-surface interactions.

The second problem occurs with nonconvex refractors. For these objects, the depth complexity is sometimes greater than 2, and it is unclear which surface to use as the second refractive interface (see Figure 6.3.3). In our experiments we used either the surface farthest from the eye or the surface closest to the front interface. Both approaches give plausible results but can lead to objectionable artifacts for refractors with long, narrow protrusions. Subjectively, we prefer using the surface farthest from the eye as the second refractive interface.

Finally, when the approximation $\tilde{d}_1$ overestimates the true distance, $\mathbf{p}_1$ may not project onto a texel containing a valid surface normal. Without using a better approximation or an iterative process to correct the overestimation, little information is available to fix the problem. Fortunately, this situation occurs relatively rarely for most objects, and many occurrences fall in regions of total internal reflection, where refractions naturally appear chaotic. Intuitively, these rays have exited the side of the refractor, where the surface normal is perpendicular to the viewing direction. Thus, instead of using a normal from the texture map, we construct a surface normal $\mathbf{N}_1$ perpendicular to $\mathbf{V}$ and refract as usual. We define $\mathbf{N}_2$ as follows:

$$\mathbf{N}_2 \equiv \mathbf{T}_1 - (\mathbf{V} \cdot \mathbf{T}_1)\mathbf{V}. \tag{6.3.2}$$

Putting it Together

Given the previous discussion, implementing approximate two-sided refraction is straightforward. The method requires two passes. The first pass renders the back refractor surface, with surface normals stored in the color channel. Both the color buffer and the depth buffer are stored in textures for use as input to the second pass.

The second pass renders the refractor using a fragment shader similar to the following snippet of Cg code. Note that Fresnel reflection and solutions to the three problems discussed above can easily be added, though we have removed them for brevity.

```
// Inputs:
//   Matrix: projectionMatrix
//   Texture: backfaceDistTex, backfaceNormalTex, environmentMapTex
//   Computed in rasterization: P_1, N_1, V, d_N, screenSpaceCoord
void main( ... )
{
  // Find the refraction direction T_1
  T_1 = refraction( V, N_1, index_i, index_t );

  // Compute d_V, the weights for d_V and d_N, and d_tilde, and P_2
  distToBackFace = tex2D( backfaceDistTex, screenSpaceCoord.xy );
  distToFrontFace = screenSpaceCoord.z;
  d_V = unproject( distToBackFace ) - unproject( distToFrontFace );
  weight_dV = Compute θt Over θi ( );
  d_tilde = weight_dV * d_V + (1 - weight_dV) * d_N;

// Compute our approximate location for the 2nd reflection
  P_2 = T_1 * d_tilde + P_1;

  // Project P_2, scale and bias to get a valid texture coord
  normalTexCoords = ScaleBias( mul( projectionMatrix, P_2 ) );
  N_2 = tex2D( backfaceNormalTex, normalTexCoords.xy );

  // Refract at the second surface, index into environment map
  T_2 = refraction( T_1, -N_2, index_t, index_i );
  outputColor = texCUBE( environmentMapTex, T_2 );
}
```

Examples and Discussion

Our images for this chapter were all captured during interactive sessions of our demo application, either running at a 512^2 or 1024^2 image resolution. Timings even for high polygon count models (like the 100,000 triangle Venus head) remain above 50 frames per second on a GeForce 6800 when the geometry covers the entire screen. Please refer to the folder for this article on the CD-ROM. Simpler models, like those shown in Figure 6.3.5, currently render at speeds of up to 300 frames per second.

ON THE CD

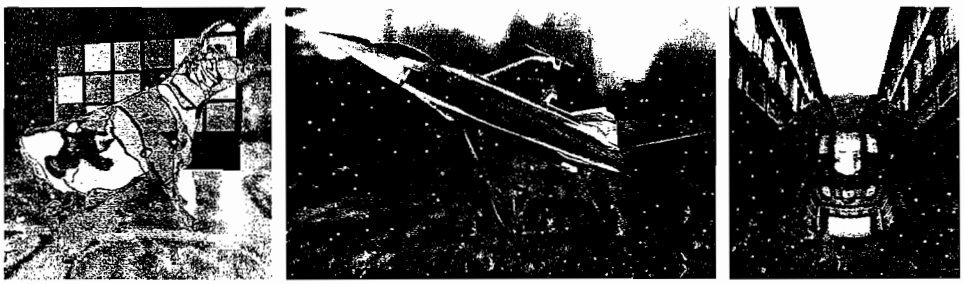

FIGURE 6.3.5 Further examples of interactive image-space refraction.

While allowing refractive objects to be suspected inside an infinite environment map is academically interesting, it isn't particularly useful. However, there are a number of existing techniques for intersecting $\mathbf{T}_2$ with other scene geometry, allowing for much more complex refractions, as shown in Figure 6.3.1. Intersections with a small number of planar surfaces can be done explicitly with ray-plane intersections in the fragment shader. Alternatively, a number of image-space search techniques for intersecting more complex geometry have been proposed, including techniques based on linear search [Wyman05b], binary search [Policarpo05], or secant root finding [Szirmay-Kalos05].

A couple of additional implementation notes:

- Refraction with high indices of refraction (>1.2) will require supersampling to improve temporal coherence in regions of total internal reflection.
- We precomputed d_N via ray tracing and stored it in the surface normal's w-component. Depending on model complexity, we found that using $\tilde{d}_1 = d_V$ or $d_N = 0$ often gives acceptable results when precomputation is impossible.

Approximate, Image-Space Caustics

Dynamic global illumination has long remained the realm of offline renderers that can afford to spend minutes or hours per frame. A variety of recent techniques have suggested that caustics, the focusing of light from a specular object, can be interactively rendered. However, these techniques limit light to a single bounce, just as in traditional interactive reflection and refraction schemes.

The key idea motivating this section is that interactive reflection and refraction techniques are vital to quickly rendering high-fidelity caustics. The rest of this chapter investigates an image-space approach for rendering caustics [Wyman06], similar to [Szirmay-Kalos05], that can easily be applied with any interactive reflection or refraction scheme, though we show results using the refraction method presented above (as in Figure 6.3.1).

Again, the observation underlying our approach is quite simple. Caustics are the focusing of light off a specular surface. This means light travels multiple paths in space to reach a focal region. Note that this is a visibility problem—we want to determine how many times a point is visible from the light. A common technique that determines visibility from the light is shadow mapping, but it only determines a binary visibility value. If we could augment shadow mapping to instead count the light rays reaching a given point, we could render both shadows and caustics.

Caustics Algorithm

Fortunately, render-to-texture and render-to-vertex-array allow a relatively straightforward implementation of such an algorithm, which borrows from the two-pass approach of photon mapping [Jensen01]. The idea is to render the scene once from the light, storing where photons land, then render a second time gathering nearby photons to determine a final color.

Given an interactive technique for rendering reflections or refractions, the difficult part is efficiently storing and gathering the photons. Photon map implementations typically use balanced kd-trees to hold photons, but building such trees is slow ($O[N \log N]$ at best). Instead, we propose adding a third render pass, which renders photons into a *caustic map* that counts photons reaching each point visible in the corresponding shadow map. The basic algorithm follows and is depicted graphically in Figure 6.3.6:

1. Render scene from the light, storing locations of photon hits (instead of pixel colors) in a vertex array. We call this the *photon buffer*.
2. Render the photons (as points or splats) into a light-space caustic map, using alpha-blending to count the number of photons affecting each texel.
3. Render the scene from the eye, applying shadowing and adding in the contribution from caustic photons. The caustic map is projected onto the scene using projective texturing.

Rendering From the Light

The goal of the first pass is to determine the photons' final locations. The leftmost bubble in Figure 6.3.6 shows a glass gargoyle rendered from the light's point-of-view using the refractive technique described above, as well as two buffers used in the second and third steps: a depth map and the photon buffer.

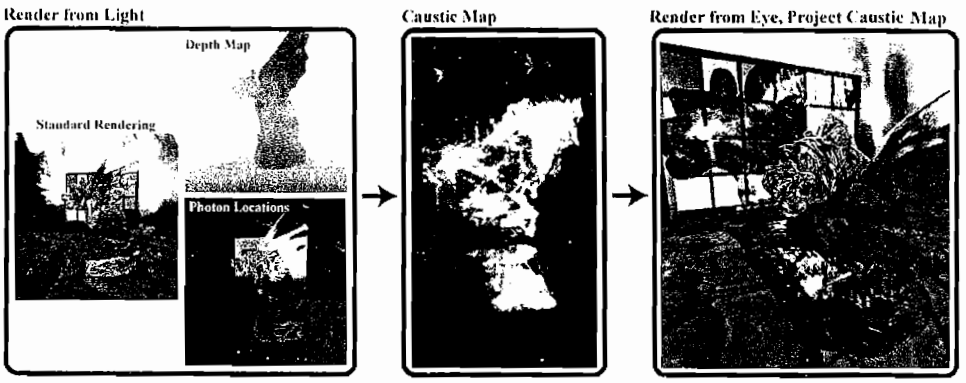

FIGURE 6.3.6 A graphical overview of rendering image-space caustics.

The depth map is a standard z-buffer, used for shadow mapping. The photon buffer stores the per-pixel positions, $\mathbf{p}_3$, computed during refraction. In Figure 6.3.6, (x,y,z) values are mapped to (r,g,b). So, to render from the light:

1. From the light's position, render the scene using reflection and/or refraction.
2. Instead of storing a final color in each pixel, output the location, $\mathbf{p}_3$, where the photon hits the opaque background geometry.

For scenes with pixels that both reflect and refract (i.e., have a Fresnel effect), two photon buffers are necessary (one for the reflected photons and one for the refracted photons).

To achieve more complex effects, additional data may be required. For example, another buffer might store photon attenuation (to simulate colored glass) or each photon's incident direction at the hit point (to simulate caustics on nondiffuse surfaces). These buffers can be generated in a single render pass using multiple render targets.

Rendering the Caustic Map

Using the photon buffer as input, the second pass gathers individual photons into the caustic map. Ideally, the caustic map augments the light's depth map; the depth map determines if photons *directly* hit a surface, and the caustic map determines if photons *indirectly* hit the surface. The idea is to count how many photons indirectly hit each point.

Given a buffer storing photon locations (as shown in Figure 6.3.6),

1. Treat each photon as a point primitive.
2. Ignore photons that are not reflected or refracted (i.e., that directly hit a surface).
3. Render the remaining points into the caustic map, with additive alpha-blending enabled to determine a total per-pixel count.

To reduce noise in the caustic map, each photon should be treated as a splat, with energy distributed over multiple texels. We used a splat with Gaussian weights. Note that there is a tradeoff between noise and caustic crispness. If very sharp caustics are desired, a large number of relatively small splats must be used to create a high-resolution caustic map. For very blurry caustics, a small number of splats in a coarse caustic map are sufficient. Most of our examples use 512^2 photons with 7^2 Gaussian splats.

To preserve energy, the weights of the Gaussians must sum to 1 when the resolutions of the photon buffer and caustic map are the same. Otherwise, the weights should sum to $R = pixelsInCausticMap/totalPhotons$. Technically, each photon subtends a different solid angle from the light and should thus be individually weighted, but we found that this effect can be ignored in practice.

Applying the Caustic Map

Once the caustic map has been computed, rendering the scene is straightforward:

1. Render the scene normally, using your favorite technique for shadows, reflections, and refractions.
2. When rendering diffuse surfaces, add the caustic contribution (modulated by the material properties) to the result of each pixel.

Putting it Together

In our implementation, we only update the caustic map after changes to the light or scene geometry. This allows rendering caustics at about the cost of an additional texture lookup in scenes where geometry moves infrequently. Ultimately, rendering caustics relies on an interactive reflection or refraction. Obviously, both the first and last passes use this reflection or refraction routine, so the approach must be quick enough to run twice per frame while maintaining interactivity.

Examples and Discussion

We typically use 512^2 or 1024^2 resolution for the photon buffer, the caustic map, and the final screen resolution. In fully dynamic scenes like those shown in Figure 6.3.1, we get speeds of around 40 frames per second (on a GeForce 6800) at 512^2, using refractors of roughly 50,000 polygons. At that resolution, the bottleneck in our approach is the vertex processor, as each pixel in the photon map is treated as a point primitive. By reducing the photon buffer resolution, this bottleneck can be moved to the pixel processor, as using fewer points requires larger splats to reduce noise. Figures 6.3.7 and 6.3.8 show the quality difference when using different resolution photon buffers and caustic maps. Note that our implementation couples the photon buffer and shadow map sizes.

FIGURE 6.3.7 A glass F-16 rendered using 64^2, 256^2, and 1024^2 photons in a 512^2 caustic map. Speeds were 80, 50, and 8 frames per second, respectively, on a GeForce 6800.

FIGURE 6.3.8 The F-16 with 64^2, 256^2, and 1024^2 photons in a 128^2 caustic map. Speeds are the same as in Figure 6.3.7.

We found that using a photon buffer with twice the resolution of the caustic map generally gives good-quality results, even during animation. Adding more photons does not help, as shown in Figure 6.3.8. Using fewer photons leads to noise and issues with frame-to-frame photon coherence.

Using 128^2 photons for computing a 64^2 caustic map gives good, but very blurry, results. As shown in Figure 6.3.7, using 1024^2 photons for a 512^2 caustic map gives very precise and sharp caustics, at the cost of a significantly lowered frame rate. Varying these values allows each application to find a unique cost-quality compromise. Note that our accompanying demo allows you to vary these parameters interactively to quickly compare settings.

Conclusion

This chapter has presented two image-space techniques, one for interactive refraction through complex objects and one for interactive caustics rendering. These are both

multipass techniques that rely on the use of images to store intermediate values. While both approaches exhibit typical image-space aliasing problems, they do not rely on expensive ray casting to determine intersections and therefore provide a viable middle ground in the quality-performance spectrum.

References

[Bunnell05] Bunnell, M. "Dynamic Ambient Occlusion and Indirect Lighting," *GPU Gems 2*, edited by Matt Pharr. Addison-Wesley, 2005: 223–233.

[Dachsbacher05] Dachsbacher, C. and M. Stamminger. "Reflective Shadow Maps." *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games.* 2005: pp. 203–208.

[Hasenfratz03] Hasenfratz, J.-M., M. Lapierre, N. Holzschuch, and F. X. Sillion. "A Survey of Real-time Soft Shadows Algorithms." *Computer Graphics Forum,* 22(4), (2003): pp. 753–774.

[Jensen01] Jensen, H. *Realistic Image Synthesis Using Photon Mapping.* AK Peters, 2001.

[Lindhom01] Lindholm, E., M. J. Kilgard, and H. Moreton. "A User-Programmable Vertex Engine." *Proceedings of SIGGRAPH 2001.* 2001: pp. 149–158.

[Oliveira00] Oliveira, G. "Refractive texture mapping, part two." Available online at *http://www.gamasutra.com/features/20001117/oliveira_01.htm*, Nov. 17, 2000.

[Policarpo05] Policarpo, F., M. Oliveira, and J. Comba. "Real-time Relief Mapping on Arbitrary Polygonal Surfaces." *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pp. 155–162.

[Schmidt03] Schmidt, C. M. "Simulating Refraction Using Geometric Transforms." Master's thesis, Computer Science Department, University of Utah, 2003.

[Sousa05] Sousa, T. "Generic Refraction Simulation." *GPU Gems 2*, edited by Matt Pharr. Addison Wesley, 2005: pp. 295–305.

[Szirmay-Kalos05] Szirmay-Kalos, L., A. Aszodi, I. Lazanyi, and M. Premecz. "Approximate Ray-Tracing on the GPU with Distance Impostors." *Computer Graphics Forum, 24*(3), (2005): pp. 695–704.

[Wyman05] Wyman, C. "An Approximate Image-Space Approach for Interactive Refraction." *ACM Transactions on Graphics 24*(3), (2005): pp. 1050–1053.

[Wyman05b] Wyman, C. "Interactive Image-Space Refraction of Nearby Geometry." *Proceedings of GRAPHITE 2005*, pp. 205–211.

[Wyman06] Wyman, C. and S. Davis. "Interactive Image-Space Techniques for Approximating Caustics." *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pp. 153–160.