Interactive Rendering of Caustics using Interpolated Warped Volumes

Manfred Ernst
University of Erlangen

Tomas Akenine-Möller
Lund University

Henrik Wann Jensen
UC San Diego

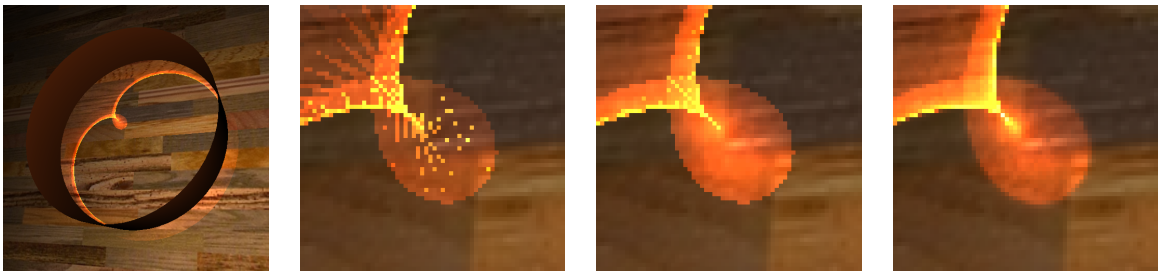

Figure 1: A metallic ring on a wooden floor creates the classic cardioid-shaped caustics, rendered at 512×512 *pixels with* $2 \times 64 \times 8$ *triangles on the ring. From left to right: a) the full scene rendered using our new algorithm, b) closeup of algorithm using constant intensity across caustic triangles (62 fps from the view in the leftmost image), c) closeup of our algorithm using interpolation (47 fps without any precomputation from the view in the leftmost image), d) closeup of the reference image computed using photon mapping with one million photons (130 seconds).*

Abstract

In this paper we present an improved technique for interactive rendering of caustics using programmable graphics hardware. Previous real-time methods have used simple prisms for the caustic volumes and a constant intensity approximation at the receiver. Our approach uses interpolated caustic volumes to render smooth high-quality caustics. We have derived a simple formula for evaluating the density of wave-fronts along a caustic ray, and we have developed a precise method for rendering caustic volumes bounded by bilinear patches. The new optimizations are well suited for programmable graphics hardware and our results demonstrate interactive rendering of caustics from refracting and reflecting surfaces as well as volume caustics. In contrast to previous work, our method renders high quality caustics generated by specular surfaces with much fewer polygons.

Key words: Caustics, caustic volumes, volume caustics, real-time rendering, graphics hardware.

1 Introduction

Caustics are beautiful and complex patterns of light generated by specular to diffuse light transport. Examples include the shimmering light at the bottom of a swimming pool and light focused through a glass of wine onto a table. There are several techniques for rendering caustics in high-quality offline rendering systems, but only a limited set of techniques for real-time rendering of caustics are available. This means that caustics are rarely included in applications such as games.

In this paper we present a new technique for interactive rendering of smoothly interpolated caustics using programmable graphics hardware. Our method is based on caustic volumes [19, 13] as shown in Figure 2. We render these caustic volumes using programmable graphics hardware similar to approaches for real-time shadow rendering that uses the shadow volumes algorithm [5]. Our contributions are summarized below:

- Fast and simple interpolation of the light intensity over each caustic triangle to avoid blocky appearance.
- The use of warped caustic volumes as shown in Figure 2, which is significantly more accurate than the prisms used in previous work.
- Fast analytic computation for volume caustics in homogeneous participating media.
- An algorithm formulated for the standard programmable feed-forward rasterization architecture with many optimizations that makes for high performance.

These key properties are illustrated in Figure 1.

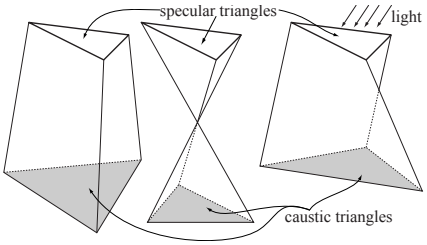

Figure 2: Three different cases of caustic volumes. From left to right: caustic volumes that are diverging, converging, and warped, where the sides clearly are not planar.

2 Previous Work

Rendering of caustics has been an active area of research in graphics for roughly 20 years. Early work concentrated on rendering complex caustic patterns, while recent work has focused on faster rendering.

Work based on ray tracing was started in 1986, when Arvo [1] presented backwards ray tracing as the first practical technique for rendering caustics. His method traced rays "backwards" from the light sources into the scene storing information about the caustics in texture maps. Collins [4] introduced an improved backwards ray tracing technique, which tracks the wavefront [12] of the caustic rays. This method gives higher accuracy, but is limited to planar diffuse receivers. Jensen [10] uses photon tracing (similar concept as backwards ray tracing) and stores the caustics in a caustics photon map — the photon map can render caustics on arbitrary geometry with non-diffuse materials. The photon mapping algorithm has been extended to handle participating media and effects such as volume caustics [11]. Recently, Guenther et al. [6] use a cluster of 18 dual Athlon PCs to render caustics at 10–20 frames per second with photon mapping. Wyman et al. [20] use a shared-memory machine with 32 CPUs together with precomputation in order to obtain real-time rendering of caustics. Due to the precomputation phase, a limited set of scene configurations can be rendered in real time. Purcell et al. [14] use graphics hardware to render images with caustics using the photon map algorithm. They use the graphics hardware to implement a breadth-first stochastic ray tracer. A scene similar to our ring scene takes about 8 seconds to render.

Heckbert and Hanrahan [7] introduced the concept of beam tracing, and this concept was used by Watt [19] in a two-pass algorithm capable of rendering caustics. In the first pass, a beam of light is created for each water surface polygon, and for each receiver it hits, a pointer is stored to the water surface polygon. The projected polygon of the beam where it hits the receiver is called a *caustic polygon*. In the second rendering pass, the intensity of the caustic polygon is proportional to the area of the water surface polygon divided by the area of the caustic polygon.

Shinya et al. [15] present pencil tracing, where a pencil is a set of rays in the vicinity of a given axial ray. This work can be seen as an extension of beam tracing [7] in the sense that it handles refractions more accurately, and also provides error tolerance analysis in an elegant manner. However, it is not targeted for use with graphics hardware.

Nishita and Nakamae present a sophisticated shading model for rendering the optical effects within water [13]. In contrast to Watt [19], they also take into account the volumetric effects, and can thus render volumetric caustics. For each scanline, their rendering process finds the intersected volumes, and accumulates their results into an accumulation buffer. Homogeneous participating media can be handled. Iwasaki et al. [8] present an algorithm that implements Nishita and Nakamae's techniques using graphics hardware. Since they use a constant intensity across caustic triangles, they have to use large water surface meshes (512×512 vertices) to avoid blocky appearance of the caustics, which makes performance suffer. Also, they do not handle warped caustic volumes, as shown to the right in Figure 2. However, they can account for a homogeneous participating media, which makes for beautiful images of volumetric caustics.

Iwasaki et al. [9] present a technique, based on Nishita and Nakamae's algorithm, that can render reflective and refractive caustics using graphics hardware. Their algorithm is a volume rendering technique, where receiver objects are sliced by several planes, and caustics rendered on each of these. The rendering quality increases with the number of slicing planes. This algorithm cannot handle participating media and does not use warped caustic volumes. Furthermore, no interpolation over caustic triangles is used. Frame rates of 2.5–10 images per second are reported for water meshes of 64×64 to 128×128 vertices.

The methods that use caustic polygons and assume uniform intensity across each caustic polygon all require a finely tessellated specular surface, and have problems when the curvature of the specular surface is high. This issue was addressed by Brière and Poulin [2] by a superior approach that tracks the wavefront of each ray of the volume and uses barycentric interpolation over each caustic triangle. This avoids the blocky appearance of the techniques presented above. Their algorithm is targeted for ray tracing, and rendering times of minutes up to many hours are reported. In our work, a simpler and faster interpolation scheme over each caustic triangle is used.

Stam [16] renders approximate textures of underwater caustics on a plane using wave theory. Trendall and Stew-

art [17] show that general calculations can be performed using the graphics hardware of 2000. Their test application is that of refractive caustics on the bottom plane of a pool of water. This work is important in the sense that it considered the graphics hardware as a general tool for different calculations. Neither of these two techniques generalize to arbitrary receivers. Wand and Strasser [18] take a completely different approach by placing sample points on the specular surfaces, and then treating each sample point as a camera that projects an image of the incoming light onto diffuse receivers. They take into account local curvature to decrease antialiasing, and can use high dynamic range images as lights. The number of passes of this algorithm is directly proportional to the number of sample points on the specular surfaces, and in the presented images, undersampling problems are visible.

Neither of the algorithms above is capable of rendering caustics onto arbitrary receivers at interactive frame rates using only a single PC with a graphics card without a blocky appearance or noise in the caustics. In the following, we present an algorithm based on caustic volumes that addresses these issues.

3 Caustics Algorithm

The geometry in the scene is separated into *generators* and *receivers*. Generators are objects with a specular component in their BRDF, while receivers always have a diffuse component. Glossy effects are neglected for caustic computations. An object may be both a generator and a receiver.

Rendering caustics using a caustic volume approach can be carried out according to a simple algorithm:

```
foreach visible point P on a receiver
  foreach caustic volume V
    if P is inside V
      Compute & accumulate caustic intensity
```

Various hierarchical data structures have been proposed to reduce the number of point-in-volume tests for CPU based rendering. However, GPUs perform best when executing highly optimized brute-force algorithms, and we use a highly optimized inner loop rather than a hierarchical method to obtain real-time performance.

Simplifying assumptions about the geometry of the caustic volumes are often made to reduce the computational cost. Artifacts and the inability to render self-intersecting volumes are the price for this speedup. Our method computes exact point-in-volume tests efficiently, and it is not limited to GPU implementations.

Without loss of generality, we will outline the algorithm for a scene with one refractive generator (e.g. a water surface) and one light source in order to simplify the description. The implementation works with an arbitrary number of reflective/refractive generators and multiple lights. All geometry is stored as indexed triangle sets. Caustic volumes are computed by the CPU for every frame. Therefore, we can handle fully dynamic scenes. At each vertex $\mathbf{v}^i$ of the generator, the vector from the light source to $\mathbf{v}^i$ is refracted at the surface using the vertex normal $\mathbf{n}^i$. Refracted vectors $\mathbf{r}^i$ are stored in an array, similar to vertices and normals. Each triangle Δ^{jkl} consisting of vertices $\mathbf{v}^j$, $\mathbf{v}^k$ and $\mathbf{v}^l$ generates a caustic volume bounded by the rays $\mathbf{v}^j + t\mathbf{r}^j$, $\mathbf{v}^k + t\mathbf{r}^k$ and $\mathbf{v}^l + t\mathbf{r}^l$, where $t \geq 0$.

Our algorithm works like this: In a first pass, the world space positions of the receivers are rendered to a texture using a simple fragment program. Because of this, receiver geometry can be arbitrary. It need only be possible to render it with OpenGL/DirectX. In the second pass, the crucial operation is to draw a bounding volume for each caustic volume. Using the positions from the texture, point-in-volume tests are computed for every visited pixel by a fragment shader. For points inside the volume, caustic intensity is computed and accumulated in the frame buffer. The accumulation is achieved by simple additive blending. Intensity calculation and point-in-volume tests need some more detailed explanation.

The following section lists the limitations of our algorithm, and then follows the details of our warped caustic volumes. In Section 3.3, we describe our method for testing whether a point, $\mathbf{p}$, is inside such a volume. For points inside a volume, we compute its intensity as described in Section 3.4 as well as volumetric effects in the presence of participating media (Section 3.5).

3.1 Limitations

The limitations of our algorithm are summarized in the following list:

- Only a single specular bounce
- No shadowing for caustics
- Generators must be triangular meshes
- It must be possible to render receivers into the Z-buffer
- Reflected/Refracted vectors are linearly interpolated over specular triangles

3.2 Warped Caustic Volumes

Previous algorithms for rendering caustics with caustic volumes using graphics hardware have all assumed that the volumes are prisms, that is, with planar side surfaces. However, due to different normals at the vertices of a

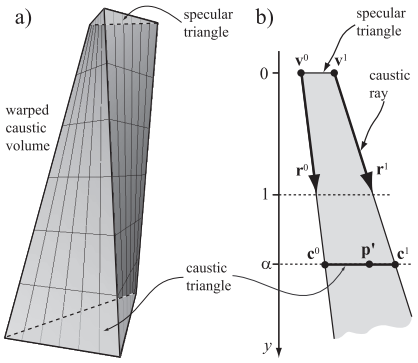

Figure 3: a) An example of a warped caustic volume. b) The two-dimensional coordinate system in which our caustic volumes are represented.

specular triangle, this is in general not true. An example of this is shown to the left in Figure 3.

The side surfaces of a caustic volume are defined by the three vertices, $\mathbf{v}^i$, $i \in [0,1,2]$, of the specular triangle, and the reflected/refracted vectors, $\mathbf{r}^i$, associated with each vertex. Assuming a finite caustic, the exact same side surfaces could each be defined by four possibly non-planar points; two from the specular triangle, and two from the bottom of the caustic volume. Such a surface is the simplest of bilinear patches. Our assumption here is that the reflected/refracted vectors when traveling from one vertex, $\mathbf{v}^i$, to another, $\mathbf{v}^j$, can be created by linearly interpolating $\mathbf{r}^i$ into $\mathbf{r}^j$. In terms of the generated side surfaces, this is an approximation in the case of refraction, but it is much more precise than the use of planar side surfaces.

For a point, $\mathbf{p}$, inside a caustic volume, we need to find the area of the caustic triangle of $\mathbf{p}$. The choice of the plane used to find the caustic triangle is arbitrary as long as it passes through $\mathbf{p}$. Intersecting a plane with a bilinear patch results, in general, in a quadratic curve. Thus the "caustic triangle" would have curved edges. However, by choosing the normal of that plane to the same as the normal of the specular triangle, i.e., the two triangles are parallel, we are guaranteed that the caustic triangle will have straight edges. This is the approach we take for our computations.

To simplify the shader optimizations presented in Section 4, we change the coordinate system so that the y-axis coincides with the normal of the specular triangle's plane, and so that the y-coordinates of the $\mathbf{v}^i$ are all set to zero. Furthermore, the directions $\mathbf{r}^i$ are scaled so that the y-components are set to one, i.e., $r^i_y = 1$. This is illustrated to the right in Figure 3.

In Figure 4, we show the differences between using triangulated volumes and using warped volumes.

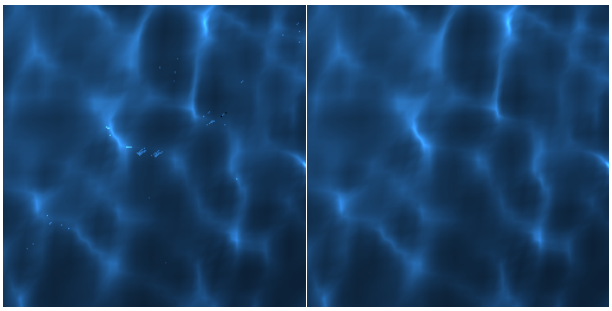

Figure 4: Caustics from a section of an ocean surface with 64×64 points rendered at 256×256 pixels. The left image was rendered with triangulated prisms as bounding volumes. Gaps between adjacent prisms result in clearly visible artifacts. The right image was rendered with tightly fitting bounding volumes and our exact point in volume test.

3.3 Point-in-Volume Test

The point-in-volume test is the performance hot-spot of the algorithm, since this is executed for every pixel covered by the bounding volume of each caustic volume. Thus, thorough optimization is crucial for real-time performance. The basic idea to test if a point $\mathbf{p}$ in world space is inside of a volume V is simple. First, $\mathbf{p}$ is transformed into the local coordinate system of V, yielding $\mathbf{p}'$. All subsequent computations are carried out in the local coordinate system. The vertices $\mathbf{c}^0$, $\mathbf{c}^1$ and $\mathbf{c}^2$ of the caustic triangle Δ_c are computed by intersecting the caustic rays with a virtual plane with normal $\mathbf{n} = (0,1,0)$ containing $\mathbf{p}'$ (see Figure 3 to the right). Any point-in-triangle test can now be used to check if $\mathbf{p}'$ is inside Δ_c. We apply a test that computes three values β_i proportional to the barycentric coordinates, because they are needed for linear interpolation later on. The actual barycentric coordinates could be computed from β_i by division by the area of the Δ_c. Below, we show how the unnormalized β_i can be computed, and then we describe some optimizations.

$$
\begin{aligned}
\alpha &= p'_y \\
\mathbf{c}^i &= \mathbf{v}^i + \alpha \mathbf{r}^i, \quad i \in [0,1,2] \\
\mathbf{e}^i &= \mathbf{p}' - \mathbf{c}^i, \quad i \in [0,1,2] \\
\beta_0 &= |\mathbf{e}^1 \times \mathbf{e}^2| \\
\beta_1 &= |\mathbf{e}^2 \times \mathbf{e}^0| \\
\beta_2 &= |\mathbf{e}^0 \times \mathbf{e}^1|
\end{aligned}
$$

The point $\mathbf{p}'$ is inside V if all β_i have the same sign.

A significant performance gain can be achieved, when the point-in-triangle test is done in two dimensions, using a projection of the involved points onto a plane. The ratios of the barycentric coordinates are invariant under

this transformation. We choose to project along the local y-axis, since that will give numerically stable results.

Due to the code above, all $\mathbf{c}^i$ and $\mathbf{p}'$ lie in the same plane, $y = \alpha$, which means that the y-component of all $\mathbf{e}^i$ will be zero. Thus, the computations of the β^is simplify to:

$$\begin{aligned}\beta_0 &= |\mathbf{e}^1 \times \mathbf{e}^2| = p'_x(c^2_z - c^1_z) + p'_z(c^1_x - c^2_x) + c^1_z c^2_x - c^1_x c^2_z \\ \beta_1 &= |\mathbf{e}^2 \times \mathbf{e}^0| = p'_x(c^0_z - c^2_z) + p'_z(c^2_x - c^0_x) + c^2_z c^0_x - c^2_x c^0_z \\ \beta_2 &= |\mathbf{e}^0 \times \mathbf{e}^1| = p'_x(c^1_z - c^0_z) + p'_z(c^0_x - c^1_x) + c^0_z c^1_x - c^0_x c^1_z\end{aligned}$$

To test whether a point is inside a caustic volume, a fragment shader computes the β's as shown above, and tests whether all β's have the same sign. If this is the case, the point is inside. The comparison is further optimized in Section 4.2.

3.4 Caustics Interpolation

In order to avoid the blocky appearance of caustics, we interpolate the intensities inside the caustic triangles. Doing linear interpolation requires the computation of intensity values per vertex, as opposed to computing a single value for an entire caustic triangle. For this purpose, we considered a wavefront based method [2], but decided to use a simpler and more efficient function to calculate the intensity at a given depth, α, inside a caustic volume. It is suited for both an optimized hardware implementation and linear interpolation. The area of the caustic triangle $A(\Delta_c)$ is defined as [1]:

$$A(\Delta_c) = |(\mathbf{c}^1 - \mathbf{c}^0) \times (\mathbf{c}^2 - \mathbf{c}^0)|$$

To simplify notation, we introduce a set of new vectors:

$$\mathbf{k}^i = \mathbf{v}^i - \mathbf{v}^0, \quad \mathbf{l}^i = \mathbf{r}^i - \mathbf{r}^0, \quad i \in [1, 2]$$

Note that $k^i_y = l^i_y = 0$. We can rewrite the area function as a function of the depth, α, along the y-axis in the coordinate system from Figure 3 as:

$$A(\alpha) = |(\mathbf{k}^1 + \alpha\mathbf{l}^1) \times (\mathbf{k}^2 + \alpha\mathbf{l}^2)| = a + \alpha b + \alpha^2 c$$

where the *triangle area coefficients* a, b and c are defined as:

$$\begin{aligned} a &= k^1_z k^2_x - k^1_x k^2_z, \\ b &= k^1_z l^2_x - k^1_x l^2_z + l^1_z k^2_x - l^1_x k^2_z, \\ c &= l^1_z l^2_x - l^1_x l^2_z. \end{aligned}$$

The a, b and c can be precomputed and sent to the fragment program as texture coordinates. For our hardware implementation, we rewrite the function as a dot product:

$$A(\alpha) = (1, \alpha^2, \alpha) \cdot (a, c, b)$$

[1] Actually, this is twice the area, but we always compute ratios of triangle areas, and so these terms cancel.

If all caustic volumes are described in the same coordinate system, interpolation could be done as follows. For every vertex $\mathbf{v}^i$ of a generator, the triangle fan around this vertex is used to compute an area function for the vertex. The n triangles around $\mathbf{v}^i$ are denoted Δ_j, $j \in [1, 2, \ldots, n]$. The area coefficients for a vertex $\mathbf{v}^i$ could be computed by just summing up the triangle area coefficients, a_j, b_j, c_j of the n triangles around $\mathbf{v}^i$:

$$\overline{a}_i = \sum_{j=1}^{n} a_j, \quad \overline{b}_i = \sum_{j=1}^{n} b_j, \quad \overline{c}_i = \sum_{j=1}^{n} c_j,$$

Note that $A(\Delta_j) = A_j(0) = a_j$, so $\overline{a}_i$ is the sum of the areas of the triangle fans around vertex $\mathbf{v}^i$. A per vertex caustic intensity I_i for $\mathbf{v}^i$ at depth α is computed using the area $\overline{a}_i$ of the triangle fan around $\mathbf{v}^i$ and the area $\overline{A}_i(\alpha)$ of the caustic fan:

$$\begin{aligned} \overline{A}_i(\alpha) &= \overline{a}_i + \alpha\overline{b}_i + \alpha^2\overline{c}_i, \\ I_i &= \overline{a}_i / \overline{A}_i(\alpha). \end{aligned}$$

where $\overline{a}_i$, $\overline{b}_i$, and $\overline{c}_i$ are computed for each vertex by the CPU. An interpolated intensity I at the point $\mathbf{p}'$ in a caustic volume, defined by the vertices $\mathbf{v}^0$, $\mathbf{v}^1$ and $\mathbf{v}^2$, is computed using the non-normalized barycentric coordinates β_i, $i \in [0, 1, 2]$:

$$I = \frac{\beta_0\overline{a}_0 + \beta_1\overline{a}_1 + \beta_2\overline{a}_2}{\beta_0\overline{A}_0(\alpha) + \beta_1\overline{A}_1(\alpha) + \beta_2\overline{A}_2(\alpha)}$$

The problem with this approach is that caustic volumes, in general, need not share a common coordinate system. Fortunately, we can enforce this condition locally, by flattening the fan around vertex $\mathbf{v}^i$ for the computation of the coefficients. For this purpose, the local y-axis is chosen to be the vertex normal $\mathbf{n}^i$. All vertices in the fan are projected along their reflected/refracted vectors into a plane with normal $\mathbf{n}^i$ containing $\mathbf{v}^i$. This transformation is depicted in Figure 5.

The transformation is still problematic because the point $\mathbf{p}$ would have to be transformed into three different coordinate systems to compute the three vertex intensities. For a real-time system, this is a costly operation. However, it can be avoided when area coefficients are computed for every vertex of every triangle and not for every point in the mesh. The fan is then flattened to a plane with the geometric normal of the specular triangle. In this way, the coordinate system for the fan is the same as the coordinate system of the caustic volume. The y-coordinate of $\mathbf{p}'$ can be used for both the point-in-volume test and the intensity calculation. Exactly how I is computed using the GPU is described in Section 4.

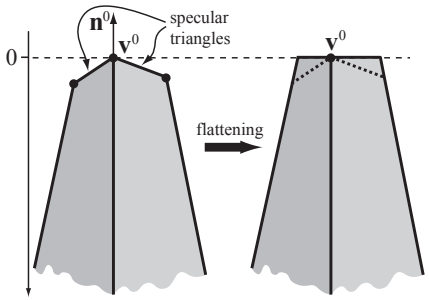

Figure 5: Flattening of triangle fan around vertex $\mathbf{v}^0$ with normal $\mathbf{n}^0$, shown in two dimensions. Note that this flattening is only used in order to compute the area function coefficients, and that the side surfaces of the volumes remain intact.

3.5 Volumetric Caustics

Scattering in participating media is described by the radiative transport equation [3], which relates the change in radiance along a direction in a medium to the loss due to extinction and the gain due to inscattered light. We will consider only homogeneous media in the following. This means that we can use a simple analytical evaluation of the scattering contribution due to a caustic beam [11]:

$$L_e(\vec{\omega}) = e^{-\sigma_t d_e} e^{-\sigma_t d_s} \Delta x \sigma_s p(\vec{\omega} \cdot \vec{\omega}') L_i(\vec{\omega}'),$$

where L_e is the radiance seen at the eye, σ_t is the extinction coefficient, σ_s is the scattering coefficient, d_e and d_s are the distances in the medium of the eye ray and the caustic beam, p is the phase function of the medium, L_i is the incident radiance of the caustic beam, and Δx is the distance that the eye ray passes through the caustic beam. In the following we will describe how we compute d_e, d_s, and Δx—the remaining values are parameters for the participating medium.

We draw a bounding volume for each caustic volume and execute a fragment program for each visited pixel, computing the intersection of the scan plane with the three caustic rays. In general the resulting triangle has curved edges, but we assume that it is linear in the case of volumetric caustics. This approximation works well as the error for the accumulated volumetric effect is negligible and does not result in any gaps between adjacent volumes. It should be noted that we still use the more precise computations for geometric receivers. The entry point $\mathbf{p}_n$ and the exit point $\mathbf{p}_f$ into the volume are computed by intersecting the triangle edges with the eye ray in the scan plane. Thus, Δx is the distance between $\mathbf{p}_n$ and $\mathbf{p}_f$. d_e and d_s are the distances from $(\mathbf{p}_n + \mathbf{p}_f)/2$ to the eye point and the specular triangle, respectively.

4 Implementation and Optimizations

In this section, our optimizations of the implementation will be described.

4.1 Vertical SIMD on GPUs

SIMD computations can be separated into horizontal and vertical computation models. Let the size of the SIMD registers be three, for example. In a horizontal model, a set of three vectors would be stored in registers like this:

$$\mathbf{v}^0 = (v_x^0, v_y^0, v_z^0), \;\; \mathbf{v}^1 = (v_x^1, v_y^1, v_z^1), \;\; \mathbf{v}^2 = (v_x^2, v_y^2, v_z^2)$$

In a vertical model, all x, y and z components are stored together in registers. This leads to the following data layout:

$$\mathbf{vx} = (v_x^0, v_x^1, v_x^2), \;\; \mathbf{vy} = (v_y^0, v_y^1, v_y^2), \;\; \mathbf{vz} = (v_z^0, v_z^1, v_z^2)$$

While this layout seems unnatural, it can be much more efficient, because the values in one register all have the same *meaning*. The same computations are applied to all three vectors in parallel. However, it can be tricky to always find three such vectors.

GPU computations are usually carried out in horizontal mode, while SIMD code for CPUs is arranged in vertical mode for optimal performance. We have applied vertical SIMD computations to the GPU with great success. This is due to the fact, that most computations operate only on the x and z components of the vectors. Such calculations can be expressed without any overhead of the unnecessary y-components in vertical mode.

Computation of the β-values in vertical mode looks like this in Cg:

```
float3 p = texRECT(pTex, wPos.xy).xyz;
// Transform p to local space here ...
float alpha = p.y;
float3 cx = alpha * rx.xyz + vx.xyz;
float3 cz = alpha * rz.xyz + vz.xyz;
float3 beta = p.x * (cz.zxy - cz.yzx)
            + p.z * (cx.yzx - cx.zxy)
            + cz.yzx * cx.zxy
            - cx.yzx * cz.zxy;
```

The texture containing receiver world space positions is bound to parameter `pTex`. The x and z coordinates of the specular triangle vertices `vx`, `vz` and the refraction vectors (`rx`, `rz`) are passed in as texture coordinates. Remember that the y-components of the refraction vectors are normalized to one and that the y-coordinates of the specular triangle vertices are zero.

4.2 Shader Optimizations

Shading language compilers usually fail to generate optimal assembly code. Optimization of the output is beneficial in most cases. Simplification of boolean expressions

and usage of special instructions are most problematic. We will describe two important optimizations here, that result in a performance improvement of 20%.

A point $\mathbf{p}$ is inside the volume if all $\beta_i \geq 0, \forall i \in [0,1,2]$ or $\beta_i \leq 0, \forall i \in [0,1,2]$. When we ignore[2] the case where one or two β-values are zero and the remaining are negative, the test can be reduced from nine assembly instructions to three. The β-vector is compared componentwise to a zero vector. A dot product of the resulting boolean vector with a vector, containing all ones, gives the number of β-values greater or equal to zero. If this value is either one or two, the tested point is not in the volume and the fragment can be discarded. Pseudo assembly code for this test is given below:

```
SGER  TEMP, BETA, {0,0,0,0};   // >=0
DP3H  TEMP.x, TEMP, {1,1,1};   // dot prod
SEQHC HC, TEMP.xxxx, {1, 2, 1, 2};
KIL   GT;   // kill fragment if outside
```

The computation of the caustic intensity requires a vector $\mathbf{m} = (1, (p'_y)^2, p'_y)$. Compiler output uses three instructions to generate this vector. Using the DST instruction, it can be computed efficiently with a single instruction:

```
DSTH  M, P.yyyy, P.yyyy;
```

4.3 Bounding Prisms

For each caustic volume, a screen-size quad could be rendered and the point-in-volume test be executed for every pixel. This would be extremely slow, and in our first implementation, we instead rasterized an axis-aligned bounding box for each volume. However, the tighter bounding volume, the fewer point-in-volume tests need to be executed. In this section, we describe how a tight bounding prism can be computed for a caustic volume. This prism is then rasterized, and the fragment shader is executed for each visited pixel.

Recall that the vertices of the specular triangle are $\mathbf{v}^i$, and for all practical applications, the volume need also be of finite length. So, assume that the vertices of the "bottom triangle" of the volume are denoted $\mathbf{b}^i$. The specular triangle will be the top of the triangular prism, and a tight plane which contains an edge will be computed for each edge of the specular triangle. Such a plane is computed as follows.

Assume, that we want to compute a plane for the edge $\mathbf{v}^0\mathbf{v}^1$. Three planes, $\mathbf{n}^i \cdot \mathbf{x} + d^i = 0$, $i \in [0,1,2]$ can be computed as:

$$\mathbf{n}^i = (\mathbf{v}^1 - \mathbf{v}^0) \times (\mathbf{b}^i - \mathbf{v}^0), \;\; d^i = -\mathbf{n}^i \cdot \mathbf{v}^0.$$

Ensure that the plane normal points outwards from the volume by flipping sign of $\mathbf{n}^i$ and d^i if the point $\mathbf{v}^2$ is in the positive half space of the plane. The plane that holds the entire caustic volume in its negative half space is the plane that we need to build our prism. For such a plane, the following must hold: $\mathbf{n}^i \cdot \mathbf{b}^k + d^i \leq 0$, for all $k \in [0,1,2]$. Note that all three planes must be tested since the volume can be self-intersecting (converging).

Once a plane has been found for each of the edges of the specular triangle, the ray of each vertex is computed as the intersection of the adjacent edges' plane equations. Finally, a bottom triangle is found by making certain that all $\mathbf{b}^i$ are inside the volume.

An example is shown in Figure 6. Notice that the bounding prism always is much tighter than using an axis-aligned bounding box. We have observed a speedup of about $2\times$ because of this.

[2]We have not experienced any visual artifacts due to this optimization.

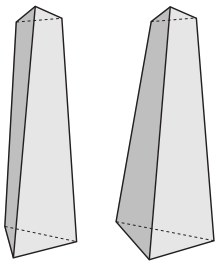

Figure 6: To the left a caustic volume is illustrated, and to the right its corresponding bounding prism is shown.

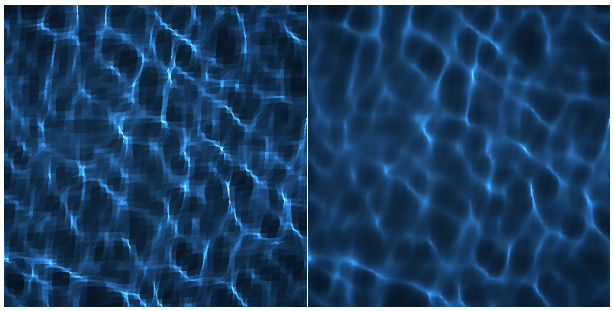

Figure 7: An ocean surface with 64×64 points rendered at 512×512 pixels. The left image was rendered at 10.7 fps with our technique without interpolation, and the right image was rendered at 8.9 fps using interpolation. As can be seen, the quality is substantially improved with our interpolation scheme.

5 Results

In all our test results, we have used a PC with an AMD Dual Athlon MP 1800 (though only one CPU was used), and a GeForce 6800 GT. In Figure 7, caustics from an ocean surface are shown when viewed from above. Notice in particular the quality improvement using our interpolation scheme.

For the ring scene, shown in Figure 1, we achieve a

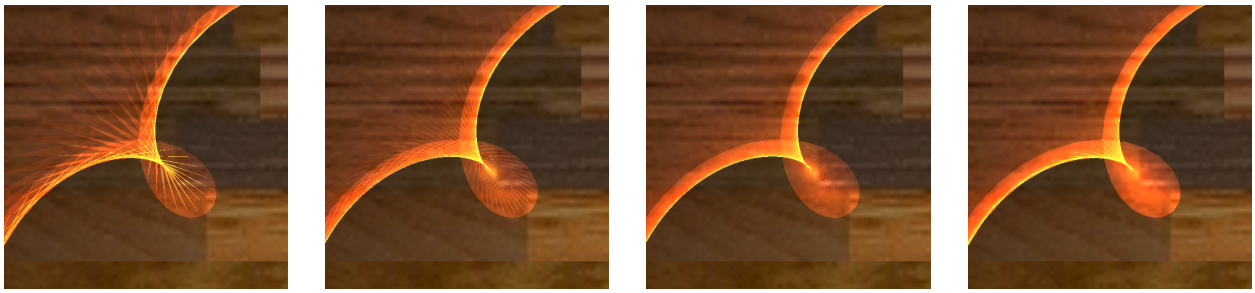

Figure 8: Zoomed-in renderings of the cardioid caustic generated by a reflective ring. The first three images from left to right show renderings with increasing generator resolution (1024, 4096 and 16384 triangles) but without interpolation. In the last image, our new interpolation technique was used with a resolution of 1024 triangles. To clearly see the artifacts in the third and fourth image, zoom in the pdf.

frame rate of about 47 fps. Static generators require almost no CPU computations. With dynamic generator geometry, the CPU consumes 15% of the rendering time for the computation of caustic rays and area function coefficients. To achieve similar image quality with no interpolation, the resolution of the ring needed to be increased by a factor of 16, which made the frame rate drop to 15 fps. Thus, the similar image quality can be obtained without interpolation by increasing the resolution. However, our interpolated version runs about $3\times$ faster, and also continues to look reasonable even when you zoom in on the caustics. This is not the case for the non-interpolated version. Zooming in on the caustic is further explored in Figure 8.

In Figure 9 all effects using our algorithm are shown. This includes caustics on the bottom and in a homogeneous participating media. We have also added dispersion by using slightly different refraction indices for R, G, and B, and rendered one pass per color. CPU time for updating the dynamic ocean generator is 5% of the total rendering time per frame.

A discoball is generating volumetric caustics in Figure 10. Each quadrilateral on the sphere generator uses the normal of the quadrilateral to generate reflection vectors from the light source. This is another example that shows that our algorithm can handle arbitrary receiving geometry.

In Figure 11 we demonstrate the advantage of our algorithm in the case of high curvature generators and receivers. Without interpolation the caustics look very blocky, while they are perfectly smooth with the new technique.

6 Conclusion and Future Work

We have presented a technique for rapidly rendering caustics using programmable graphics hardware. We avoid the blocky artifacts of previous techniques by using interpolation of the caustic intensity for each caustic triangle. We also presented a fast method for simulating volumetric caustics in homogeneous participating media, and finally we presented a number of optimizations that take advantage of programmable graphics hardware. Our current algorithm takes into account only one specular-to-diffuse bounce. More specular bounces could be included using the techniques presented by Brière and Poulin [2], but this would probably require some non-trivial clipping of the volumes.

For future work, we would like to add shadow mapping as an approximation to shadows in the caustics, since currently occlusion is not handled in our framework. We would also like to implement the entire creation of the caustic volumes in a vertex shader. Also, it would be interesting to investigate whether our interpolation scheme could be used in other rendering systems, e.g., ray tracing based algorithms.

Acknowledgements

Thanks to Marc Stamminger and Günther Greiner for initiating Manfred's visit at UCSD, and to Craig Donner for the water surface generation code. Manfred was funded by Bavaria California Technology Center. Tomas was supported by the Hans Werthén foundation, Carl Tryggers foundation, Ernhold Lundströms foundation, and the Swedish Foundation for Strategic Research. Henrik was supported by a Sloan Fellowship and the National Science Foundation under Grant No. 0305399.

References

[1] James Arvo. Backward Ray Tracing. In *Developments in Ray Tracing, SIGGRAPH '86 Course Notes*, August 1986.

[2] Normand Brière and Pierre Poulin. Adaptive Representation of Specular Light Flux. In *Graphics Interface*, pages 127–136, May 2000.

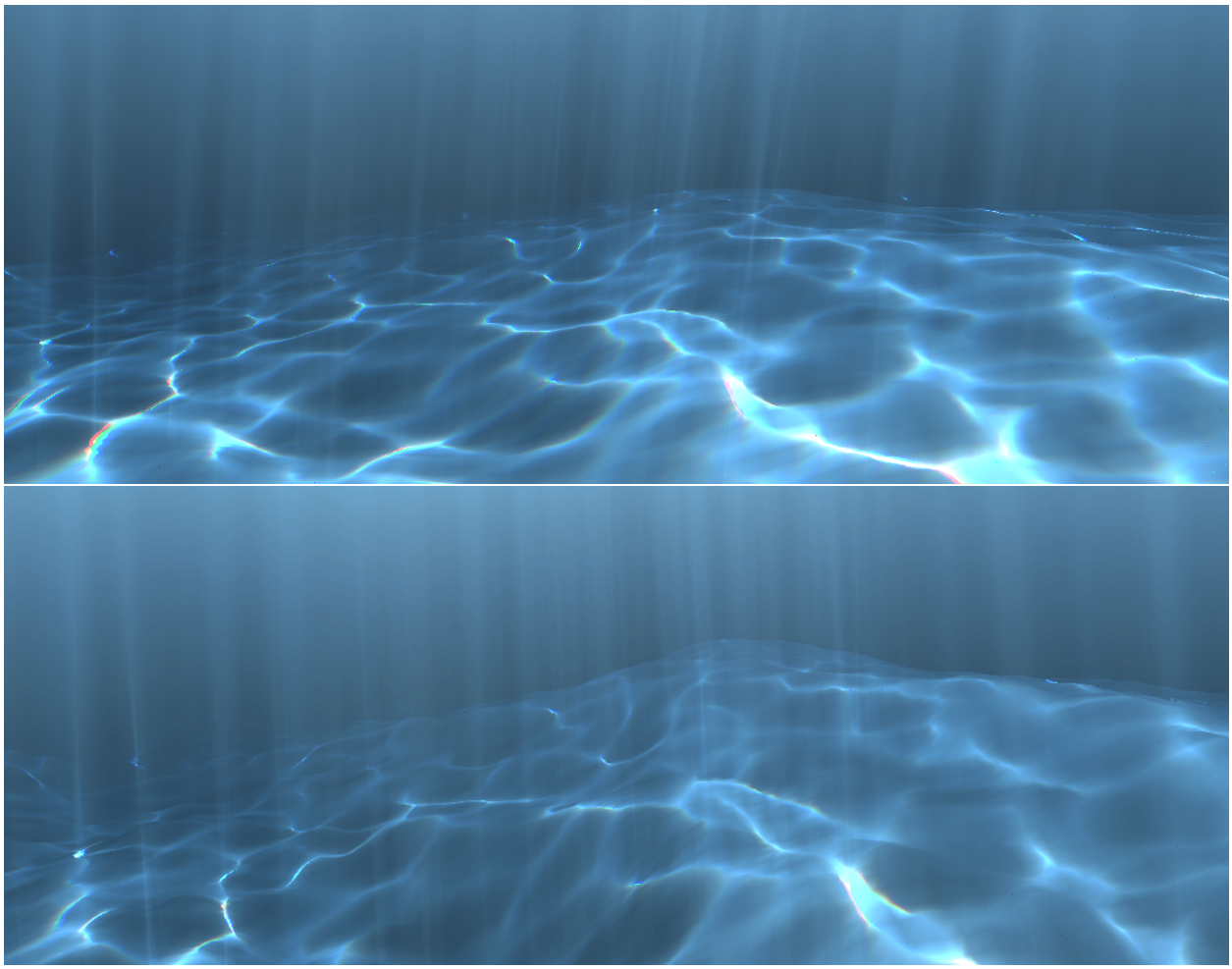

Figure 9: Two images of volumetric caustics from a 128×128 *ocean surface. The bottom image has a more pronounced bottom surface, i.e., receiver). Notice that these images have been rendered with a slight dispersion effect as well at 0.2 fps at* 1280×500 *pixels.*

[3] S. Chandrasekhar. *Radiative Transfer*. Oxford University Press, 1960.

[4] S. Collins. Adaptive Splatting for Specular to Diffuse Light Transport. In *Fifth Eurographics Workshop on Rendering*, pages 119–135, June 1994.

[5] Franklin C. Crow. Shadow Algorithms for Computer Graphics. In *Computer Graphics (SIGGRAPH 77)*, pages 242–248. ACM Press, 1977.

[6] Johannes Guenther, Ingo Wald, and Philipp Slusallek. Realtime Caustics using Distributed Photon Mapping. In *Eurographic Symposium on Rendering*, pages 111–121, 2004.

[7] Paul S. Heckbert and Pat Hanrahan. Beam Tracing Polygonal Objects. In *Computer Graphics (SIGGRAPH 84)*, pages 119–127. ACM Press, 1984.

[8] K. Iwasaki, Y. Dobashi, and T. Nishita. An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware. *Computer Graphics Forum*, 21(4):701–712, 2002.

[9] K. Iwasaki, Y. Dobashi, and T. Nishita. A Fast Rendering Method for Refractive and Reflective Caustics Due to Water Surfaces. In *EUROGRAPHICS 2003*, pages 283–291. Eurographics, 2003.

[10] Henrik Wann Jensen. Global Illumination using Photon Maps. In *Proceedings of the 7th Eurographics Workshop on Rendering*, pages 21–30. Eurographics, 1996.

[11] Henrik Wann Jensen and Per H. Christensen. Efficient Simulation of Light Transport in Scences with Participating Media using Photon Maps. In *Com-*

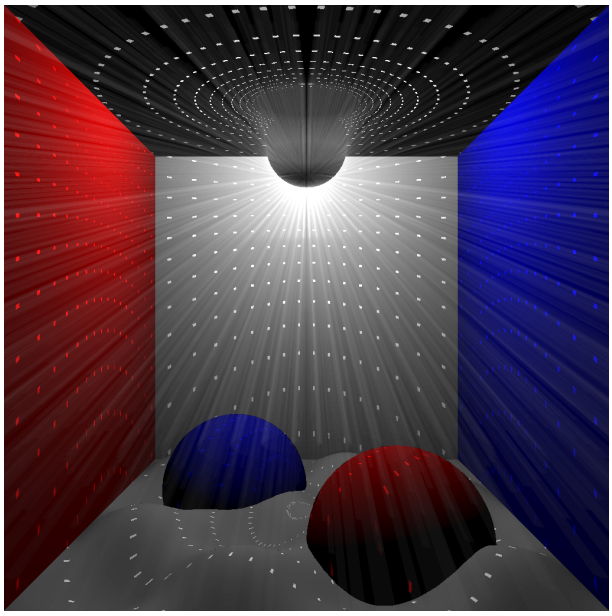

Figure 10: A discoball in a Cornell-box-like room.

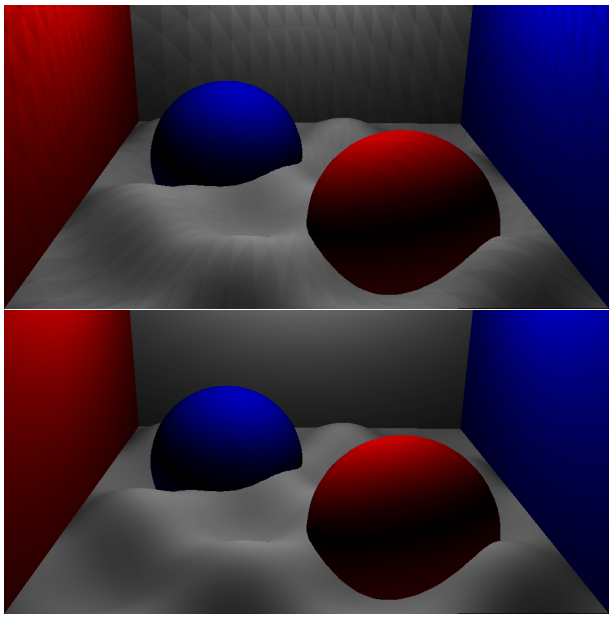

Figure 11: Caustics of a sphere with smooth surface normals at the ceiling of a Cornell-box-like room. Top: caustics with no interpolation. Bottom: smoothly interpolated caustics with our new method.

puter Graphics (SIGGRAPH 98), pages 311–320. ACM Press, 1998.

[12] Don Mitchell and Pat Hanrahan. Illumination from Curved Reflectors. In *Computer Graphics (SIGGRAPH 92)*, pages 283–291. ACM Press, 1992.

[13] Tomoyuki Nishita and Eihachiro Nakamae. Method of Displaying Optical Effects within Water using Accumulation Buffer. In *Computer Graphics (SIGGRAPH 94)*, pages 373–379. ACM Press, 1994.

[14] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon Mapping on Programmable Graphics Hardware. In *Graphics hardware*, pages 41–50. Eurographics Association, 2003.

[15] Mikio Shinya, Tokiichiro Takahashi, and Seiichiro Naito. Principles and Applications of Pencil Tracing. In *Computer Graphics (SIGGRAPH 87)*, volume 21, pages 45–54, July 1987.

[16] Jos Stam. Random Caustics: Wave Theory and Natural Textures. In *ACM SIGGRAPH Visual Proceedings*, page 151, 1996.

[17] Chris Trendall and A. James Stewart. General Calculations using Graphics Hardware, with Applications to Interactive Caustics. In *Eurographics Workshop on Rendering*, pages 287–298, June 2000.

[18] M. Wand and W. Strasser. Real-Time Caustics. *Computer Graphics Forum*, 22(3):611–611, 2003.

[19] Mark Watt. Light-Water Interaction using Backward Beam Tracing. In *Computer Graphics (SIGGRAPH 90)*, pages 377–385. ACM Press, 1990.

[20] Chris Wyman, Charles Hansen, and Peter Shirley. Interactive Caustics Using Local Precomputed Irradiance. In *Proceedings of the 2004 Pacific Conference on Computer Graphics and Applications*, pages 143–151, 2004.