Caustics Mapping: An Image-space Technique for Real-time Caustics

Musawir Shah, Jaakko Konttinen, Sumanta Pattanaik

***Abstract*— In this paper, we present a simple and practical technique for real-time rendering of caustics from reflective and refractive objects. Our algorithm, conceptually similar to shadow mapping, consists of two main parts: creation of a caustic map texture, and utilization of the map to render caustics onto non-shiny surfaces. Our approach avoids performing any expensive geometric tests, such as ray-object intersection, and involves no pre-computation; both of which are common features in previous work. The algorithm is well suited for the standard rasterization pipeline and runs entirely on the graphics hardware.**

I. Introduction

CAUSTICS are complex patterns of shimmering light that can be seen on surfaces in presence of reflective or refractive objects, for example those formed on the floor of a swimming pool in sunlight. Caustics occur when light rays from a source, such as the sun, get refracted, or reflected, and converge at a single point on a non-shiny surface. This creates the non-uniform distribution of bright and dark areas. Figure 2 shows a photograph of caustics from a glass sphere captured with a digital camera. Caustics are a highly desirable physical phenomenon in computer graphics due to their immersive visual appeal. Some very attractive results have been produced using off-line high quality rendering systems; however, real-time caustics remain open to more practical solutions. Deviating from the conventional geometry-space paradigm, which involves path tracing in a 3D scene, intersection testing, etc, we explore an image-space approach to real-time rendering of caustics. Our algorithm has the simplistic nature of shadow mapping, yet produces impressive results comparable to those created using off-line rendering. We support fully dynamic geometry, lighting, and viewing direction since there is no pre-computation involved. Furthermore, our technique does not pose any restrictions on rendering of other phenomena, such as shadows, which is the case in some previous work [3]. Our algorithm runs entirely on the graphics hardware with no computation performed on the CPU. This is an important criterion in certain applications, such as games, in which the CPU is already extensively scheduled for various tasks other than graphics.

{mali,jaakko,sumant}@cs.ucf.edu, School of Computer Science, University of Central Florida

Fig. 1. Image rendered using the caustics mapping algorithm. This result was obtained using double surface refraction (both for the appearance of the bunny as well as for the caustics) at the rate of 42 fps.

The remainder of this paper is organized as follows: a short survey of related work is presented in Section 2. Our rendering algorithm is then explained in Section 3, followed by Section 4 discussing results and limitations. We conclude with a summary of the ideas presented in the paper and provide directions for future research in Section 5.

II. Previous Work

Although caustics rendering, in general, has been subjected to a fair amount of research, a practical real-time caustics rendering does not exist for everyday applications. In this section, we look at some of the

Fig. 2. Photograph of caustics from a spherical glass paper weight using a desk lamp to emulate directional spotlighting. The caustics are formed on rough paper placed underneath the refractive object.

earlier work in offline caustics rendering and recent attempts to achieve caustics at interactive frame-rates.

For a fair amount of computational cost, accurate and extremely beautiful caustics can be produced. Introductory work using backward ray-tracing was proposed by Arvo [1], which was then pursued and extended by a number of researchers. In this method, light rays are traced from the light source into the scene as opposed to conventional ray tracing in which the rays emerge from the eye. Photon mapping, a more flexible framework, was proposed by Jensen [9] which handles caustics in a natural manner on arbitrary geometry and can also support volumetric caustics in participating media [10]. Variants and optimized versions of path tracing algorithms have been presented which utilize CPU clusters [4] and graphics hardware [12]; but the computational cost in time and resources are limiting factors in practical application of these techniques to real-time systems. Wyman et al. [19] rendered caustics at interactive frame-rates using a large shared-memory machine by pre-computing local irradiance in a scene and then sampling caustic information to render nearby surfaces. Such pre-computation steps in algorithms restrict their functionality to domains for which the pre-computation was performed and are unable to support fully dynamic scenes. Our algorithm is also based on the backward ray-tracing idea, however it does not require any pre-computation.

Wand and Straβer [16] developed an interactive caustics rendering technique by explicitly sampling points on the caustics-forming object. The receiver geometry is rendered by considering the caustic intensity contribution from each of the sample points. The authors presented results using specular caustics-forming objects, but refractive caustics can also be achieved with this technique. However, the explicit sampling hinders the scalability of the algorithm since the amount of computation done is directly proportional to the number of sample points used in rendering caustics.

Perhaps the most prominent caustics are those formed in the presence of water. Therefore, the problem of rendering underwater caustics specifically has received significant attention. In early work, Stam [13] precomputed underwater caustics textures and mapped them onto objects in the scene. Although this technique is extremely fast, the caustics produced are not correct given the shape of the water surface and the receiver geometry. Trendall and Stewart [15] have shown refractive caustics to demonstrate the use of graphics hardware for performing general purpose computations. Their intent was to perform numerical integration which they used to calculate caustic intensities on a flat receiver surface. Their method cannot support arbitrary receiver geometry and also cannot be easily extended to handle shadows.

Beam tracing has been employed to produce more physically accurate underwater optical effects, caustics in particular [5], [17]. In this technique, a light beam through a polygon of the water surface mesh is traced to the surface of a receiver object, hence projecting the polygon onto the receiver. The energy incident on the water surface polygon is used to compute the caustic intensity at the receiver, taking into account the areas of the surface and projected polygons. The intensity contributions from all the participating polygons are accumulated for the final rendering.

Nishita and Nakamae [11] present a model based on beam-tracing for rendering underwater caustics including volumetric caustics. Their idea was implemented on graphics hardware by Iwasaki et al. [7]. In a more recent publication Iwasaki et al. [8] adopt a volume rendering technique in which a volume texture is constructed for receiver objects using a number of image slices containing the projected caustic beams. The case of warped volumes which can occur in beam tracing has not been addressed in either of the above. Ernst et al. [3] manage this scenario and also present a caustic intensity interpolation scheme to reduce aliasing resulting in smoother caustics. However, their algorithm is unable to obtain shadows since it does not account for visibility. In contrast, our algorithm is able to handle shadows and in general does not impose any restrictions on rendering other phenomena.

Independently and in parallel with our work, Szirmay-Kalos et al. [14] conducted research on approximating ray-geometry intersection estimation using distant imposters which they applied to rendering caustics. Although this technique is similar to ours, there are two

main differences. We present an intersection estimation algorithm which maps to the Newton-Raphson root finding method that has faster convergence than the Secant method used by Szirmay-Kalos et al. [14]. Furthermore, a ray-plane intersection operation is performed for every iteration of their method which is not needed in our algorithm.

III. Rendering Caustics

Our rendering algorithm consists of two main phases: (i) construction of a caustic map texture, and (ii) application of the caustics map to diffuse surfaces called receivers. We will first give a brief explanation of how caustics are formed, and then relate it to our method of construction of the caustic-map.

A. Caustic formation

Caustics are formed when multiple rays of light converge at a single point. This occurs in the presence of refractive or reflective objects which cause the light rays to deviate from their initial path of propagation and converge at a common region. Therefore, to obtain caustics accurately, one must trace light rays from their source and follow their paths through refractive and off reflective surfaces. The photons eventually get deposited on nearby diffuse surfaces, called receivers, thus forming caustics as illustrated in Figure 3. Our algorithm closely emulates this physical behavior, and is capable of obtaining both refractive and reflective caustics. For reflective caustics we support a single specular bounce. In the case of refractive caustics, in addition to single surface refraction, our method also supports double surface refraction employing the image-space technique recently proposed by Wyman [18]. The algorithm supports point and directional lighting. Area lights can be accommodated by sampling a number of point lights.

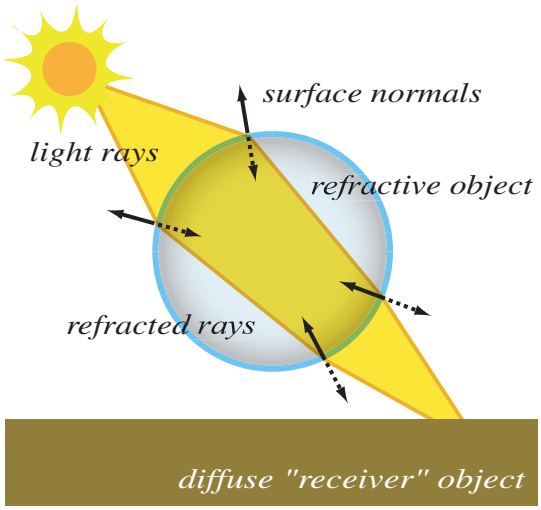

Fig. 3. Diagram showing how multiple light rays can refract through an object and converge at the same point on a diffuse surface.

B. Caustics Mapping Algorithm

Without loss of generality, we will describe rendering of caustics through refractive objects using our algorithm. A general overview of the algorithm, which closely resembles shadow mapping, is presented next along with elaborative discussion on certain parts.

Following is a stepwise breakdown of the main caustics mapping algorithm. Since the algorithm runs entirely on the GPU, it is explained in terms of render passes performed on the graphics hardware.

- **Obtain 3D positions of the receiver geometry:** The receiver geometry is rendered to a *positions texture* from the light's view. 3D world coordinates are output for each pixel instead of color. This positions texture is used for ray-intersection estimation in the next step.
- **Obtain 3D positions and surface normals of the refractive object:** The refractive object is rendered to texture from the light's view. Using multiple render targets, the surface normals and 3D world coordinates are output for each pixel. These textures are used with a grid of vertices of equal resolution, such that each vertex maps to a pixel on the texture. The vertex grid is used for the remainder of the algorithm in place of the refractive object.
- **Create caustic-map texture:** The caustic-map texture is created by splatting points onto the receiver geometry from each vertex of the refractive vertex grid along the refracted light direction. The intersection point of the refracted ray and the receiver geometry is estimated using the positions texture. This step is explained in detail in the following section.
- **Construct shadow map:** Although optional, conventional shadow mapping can easily be integrated into the caustics mapping algorithm to render images with both caustics and shadows.
- **Render final scene with caustics:** The 3D scene is rendered to the frame buffer from the camera's view. Each point of the receiver surface is projected into the light's view to compute texture coordinates for indexing the caustic-map texture. The caustic color from the texture is assigned to the pixel and augmented with diffuse shading and shadowing.

The steps mentioned above are performed every frame

in separate render passes, and thus impose no constraints on the dynamics of the scene. Furthermore, no computation is performed on the CPU; neither as a per-frame operation nor as a pre-computation.

C. Creating the Caustic-Map

Rendering of the caustics map texture lies at the heart of our algorithm. It consists of three main steps:

- Refraction of light at each vertex of the refracting surface
- Estimation of the intersection point of the refracted ray with the receiver geometry
- Estimation of caustic intensity at the intersection point

The caustic-map texture is created by rendering the refractive vertex grid from the light's view. The vertices are displaced along the refracted light direction by the estimated distance and then splatted onto the receiver geometry by rendering point primitives instead of triangles. The algorithm is implemented in a vertex shader program and proceeds as follows:

for each *vertex v* **do**
$\overrightarrow{r}$ = *Refract(LightDirectionVector)*
P = EstimateIntersection(v.position, $\overrightarrow{r}$*,*
ReceiverGeometry)
v.position = P
return *v*

Notice that multiple vertices can end up at the same position on the receiver geometry. In the pixel shader, the light intensity contribution from each vertex is accumulated using additive alpha blending. Details regarding the intensity and final caustic color computation are given in Sub-section III D.

The intersection point of the refracted ray with the receiver geometry must be computed in order to output the final position of the vertex being processed. Normally, this requires expensive ray-geometry intersection testing which is not feasible for real-time applications. We present a novel image-space algorithm for estimating the intersection point which, to our knowledge, has not been used before.

The intersection estimation algorithm utilizes the positions texture rendered in the first pass of the main caustics mapping algorithm. A schematic illustration of the procedure is shown in Figure 4. Let v be the position of the current vertex and $\overrightarrow{r}$ the normalized refracted light vector. Points along the refracted ray are thus defined as:

$$P = v + d * \overrightarrow{r} \quad (1)$$

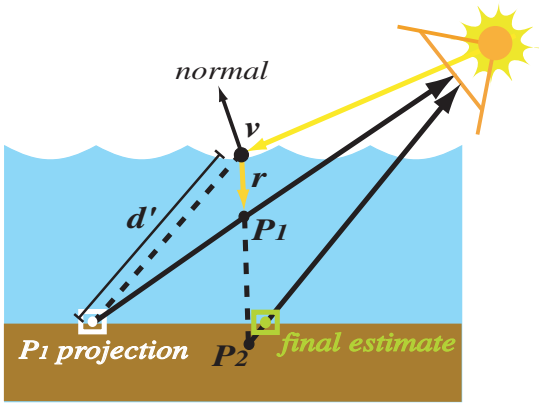

Fig. 4. Diagram of the intersection estimation algorithm. The solid-lined arrows correspond to the position texture lookups.

where d is the distance from the vertex v. Estimating the point of intersection amounts to estimating the value of d, the distance between v and the receiver geometry along $\overrightarrow{r}$. An initial value of 1 is assigned to d and a new position, P_1, is computed:

$$P_1 = v + 1 * \overrightarrow{r} \quad (2)$$

P_1 is then projected into the light's view space and used to look up the positions texture. The distance, *d'*, between v and the looked up position is used as an estimate value for d in Equation 1 to obtain a new point, P_2. Finally, P_2 is projected in to the light's view space and the positions texture is looked up once more to obtain the estimated intersection point. Following is a snippet of shader language code that computes the estimated intersection point. The inputs to this algorithm are the initial point on the ray, v, the normalized refraction direction vector, $\overrightarrow{r}$, and a texture sampler, *posTexture*, used to lookup the positions texture.

```
float3 EstimateIntersection ( float3 v, float3 r,
        sampler posTexture ) {

float3 P1 =  v + 1.0*r ;
float4 texPt =  mul( float4( P1, 1 ) , mViewProj );
float2 tc = float2(0.5*(texPt.xy/texPt.w)+float2(0.5,0.5));
tc.y = 1.0f - tc.y;
float4 recPos = tex2D(posTexture, tc);
float4 P2 =  v +distance(v,recPos.xyz)*r;
texPt =  mul( float4( P2, 1 ) , mViewProj );
tc = float2(0.5*(texPt.xy/texPt.w)+float2(0.5,0.5));
tc.y = 1.0f - tc.y;
return recPos = tex2D(posTexture, tc);
}
```

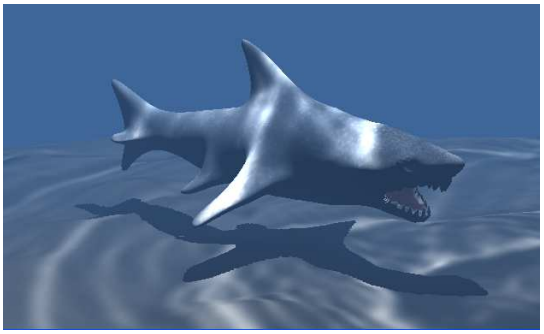

Fig. 5. Caustics on non-planar surfaces can be obtained using Caustics Mapping. The scene shows underwater caustics on a shark and a rugged seabed.

D. Convergence of Intersection Estimation Algorithm

The intersection estimation algorithm is essentially an iterative process of which a single iteration has been discussed above. For example, in the next iteration the distance between the estimated intersection point in the last iteration and *v* would be plugged into Equation 1 as a new estimate value for *d*. This iterative process was derived from the Newton-Raphson root finding method, whose convergence rate is faster than other popular methods such as the Secant and Bisection methods. In this sub-section, we map the problem of intersection estimation to a root finding problem. We then discuss the Newton-Raphson method and relate it to our intersection estimation algorithm.

In order to compute the intersection using a root finding method, we must define a function, *f(x)*, whose zero corresponds to the point of intersection of the refracted ray and the receiver geometry. Let the refracted ray be the x-axis for a discrete function, *f(x)*. *f(x)* is the distance between *x* and the projection of *x*. For each point, x_i, along the refracted ray at a distance d_i from the origin of the ray, *f(x)* is defined as follows:

$$\begin{aligned} y_i &= posTexture(x_i) \\ f(x_i) &= distance(x_i, y_i) \end{aligned} \tag{3}$$

The function *posTexture*(x_i) retrieves a 3D point from the *positions texture* by projecting x_i into the light's view and computing texture coordinates. For example, in Figure 4, the distance between P_1 and P_1 *projection* is the value for *f*(P_1).

For simplicity, *f(x)* can be re-parameterized using the distance, *d*, from the origin of the ray instead of a 3D point, *x*, along the ray: *f(v + d * $\overrightarrow{r}$)* or simply, $\hat{f}$*(d)* (recall that *v* and $\overrightarrow{r}$ are the origin and the direction of the refracted ray). Notice that when *d* is equal to the distance from the refracted ray origin to the intersection point, $\hat{f}$*(d)* is equal to 0. Therefore, the root of $\hat{f}$*(d)* directly corresponds to the distance to the intersection point. We can now define the following theorem:

Theorem 1: $\hat{f}$(d) = 0 if and only if *d* is equal to the distance along the refracted ray from the ray origin to the intersection point.

Proof: We will construct a proof by applying the definition of $\hat{f}$. Let *d* be the distance along the refracted ray from the ray origin to the intersection point. Let $p = v + d * \overrightarrow{r}$, where *v* is the ray origin and $\overrightarrow{r}$ is the ray direction. If the *positions texture* is looked up by projecting *p* into the light's view, the same point *p* is obtained. Since the distance of a point from itself is 0, the value of $\hat{f}$ must be equal to 0 by definition. ■

Thus the intersection estimation problem has been reduced to the problem of computing the root of $\hat{f}$. Although any root finding algorithm can be applied as per user preference, the Caustics Mapping algorithm uses an iterative method derived from the Newton-Raphson (NR) algorithm. The NR iteration is defined as follows:

To find a root of $\hat{f}$(d) = 0 with the initial guess d_0,

$$d_{k+1} = d_k - \frac{\hat{f}(d_k)}{\hat{f}'(d_k)} \; where \; k = 0, 1, 2, 3, ... \tag{4}$$

Note that the NR iteration requires the evaluation of the derivative function, $\hat{f}$*'(d)*. Since the function $\hat{f}$ is in a discrete form, its derivative can be computed using finite differences.

$$\hat{f}'(d) = (\hat{f}(d + \Delta) - \hat{f}(d))/\Delta \tag{5}$$

GPU code for the NR iteration is provided in the Appendix, and it can be used in place of our approximative iterative solution if desired with the added cost of computing the derivative. In certain cases where the derivative is approximated using the difference between two points specifying an interval, this iteration is referred to as the Secant method. Note that the Secant method operates on an interval of arbitrary length, whereas to compute the derivative of a function using finite difference in the NR method, the interval has to be small. Computing the derivative requires two evaluations of the function $\hat{f}$. This directly translates to two accesses of the *positions texture* per iteration. In order to reduce the overall number of texture accesses, we introduce an approximation to the NR evaluation. The NR method gives d_{k+1}, the new estimate of the root point by incrementing d_k, with a scaled version of $\hat{f}(d_k)$, the function value at d_k. The exact scale is the negative reciprocal of the derivative of the function at d_k. In our method we use the distance of the surface geometry at d_k from the origin of the refracted ray (i.e. d_0=0) as the estimate for d_{k+1}. This approximation of the NR iteration was influenced by three key observations:

- **A rough similarity to NR estimator:** The distance of the surface geometry at d_k from the origin of the refracted ray (i.e. d_0=0), can be approximated as d_k incremented by an amount related to $\hat{f}(d_k)$.
- **Correctness:** At the root, i.e. at the point of intersection, d_k is indeed equal to the distance of the surface geometry from the origin of the ray.
- **Computational efficiency:** This estimator eliminates the need for explicitly computing the function derivative and thus cuts down the number of texture lookups by half.

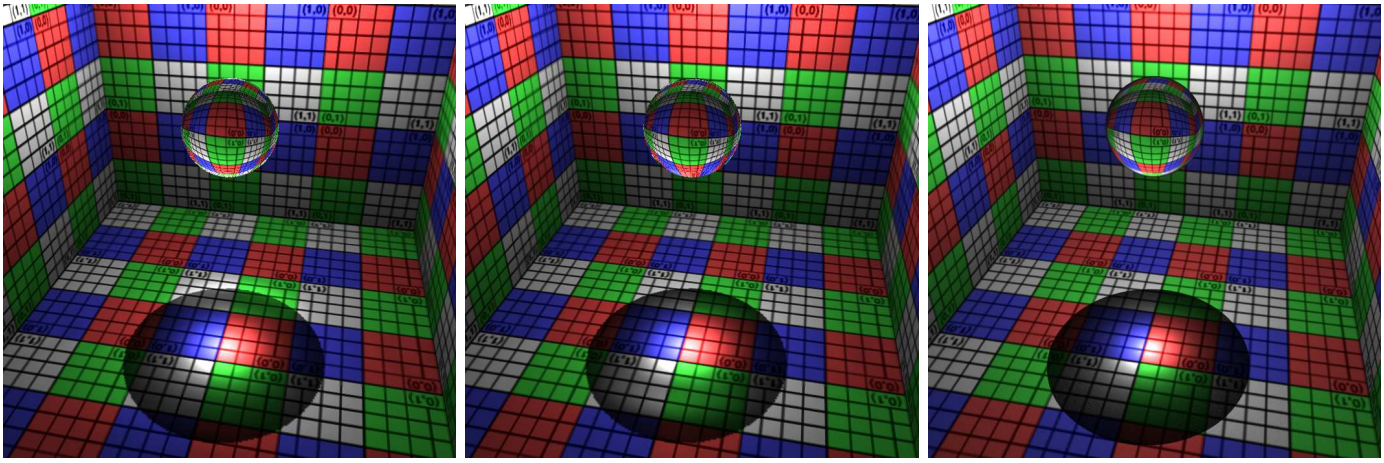

Fig. 6. Caustics from a glass sphere (from left to right) using the distance imposters method, our caustics mapping, and photon mapping. 3 iterations of the respective intersection estimation algorithms were used for both the first and the second image. The distance imposter's method produced a frame rate of 160 fps whereas caustics mapping produced 245 fps on a GeForce 7800.

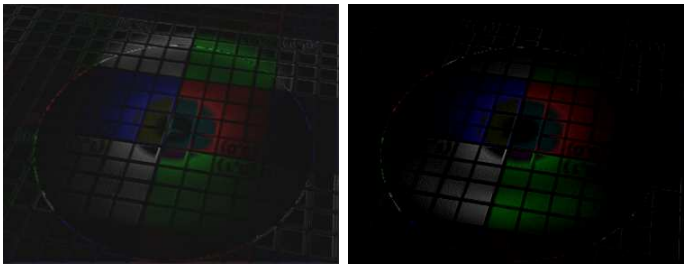

Fig. 7. Difference images of (Left) the distance imposters method, and (Right) our caustics mapping method with photon mapping. The difference images were obtained from the results shown in Figure 6.

The intersection estimation algorithm assumes that the true intersection point lies at the same distance from *v* as the current estimate point. Visually, this can be observed as a circular arc connecting P_1 and P_2, centered at *v* with radius *d'*. The assumption that the receiver surface is curved rather than planar in the local region of the intersection was motivated by the fact that no explicit intersection tests would be required in the former case. The algorithm thus incurs the cost of just a texture look-up at each iteration, unlike the distant imposters technique which additionally requires a ray-plane intersection test.

As with all iterative root finding methods, the continuity of the function is a vital criterion for convergence. Thus in order for the intersection estimation to be successful, the visibility of the receiver geometry from the light's view must be well defined. Note that when the *positions texture* lookup returns a null value, it implies that no receiver geometry was seen in that direction. If this null value is encountered in the intersection estimation algorithm, the result obtained is incorrect. In practice, the value returned is a point at infinity. Such a situation arises near the edges of the reciever geometry. However, this only means that some of the caustic splats would be lost, but it does not cause any visual artifacts in the final caustics rendering. Another limitation that is specific to the Newton-Raphson method is that in some cases it starts oscillating between two reference points and thus never converges to the true root. This problem is attributed to the fact that the Newton-Raphson method keeps only the most recent estimated point instead of an interval which contains the root point. This oscillation, however, can be detected by storing the last two estimates and then remedied by switching to a more conservative algorithm such as the Bisection method.

We have found empirically that the estimate of the intersection point improves rapidly with each iteration. The magnitude of the error and the number of iterations to convergence depends on the topology distribution of the scene. For a fairly uniform topology, the error is negligible after only 5 iterations. For the purpose of rendering caustics, we observed that only a single iteration is sufficient. Furthermore, the small error in the estimation is well sustained by our caustics mapping algorithm and thus is suitable for the purpose. It is important to note that the algorithm is not limited to rendering caustics on planar surfaces. As long as the visibility function is well defined and continuous, even fairly rugged terrains produce accurate results. Figure 5 shows under-water caustics falling on a shark and the floor, both of which have non-planar surface geometry.

E. Caustic Intensities

The intensity of the caustics formed on the diffuse receiver geometry depends on the amount of light that gets accumulated at any particular point on the receiver's surface. Since the caustics map texture is created by refracting or reflecting the light rays at each vertex, the intensity of the caustics will depend on the number of vertices which make up the caustic-forming object. If the algorithm is employed using a naive accumulation scheme, the caustics will appear to be bright or dark depending on the number of vertices of the refractive object. This undesirable phenomenon occurs due to the incorrect assumption that the light flux contribution for each vertex is the same. We combat this problem by computing

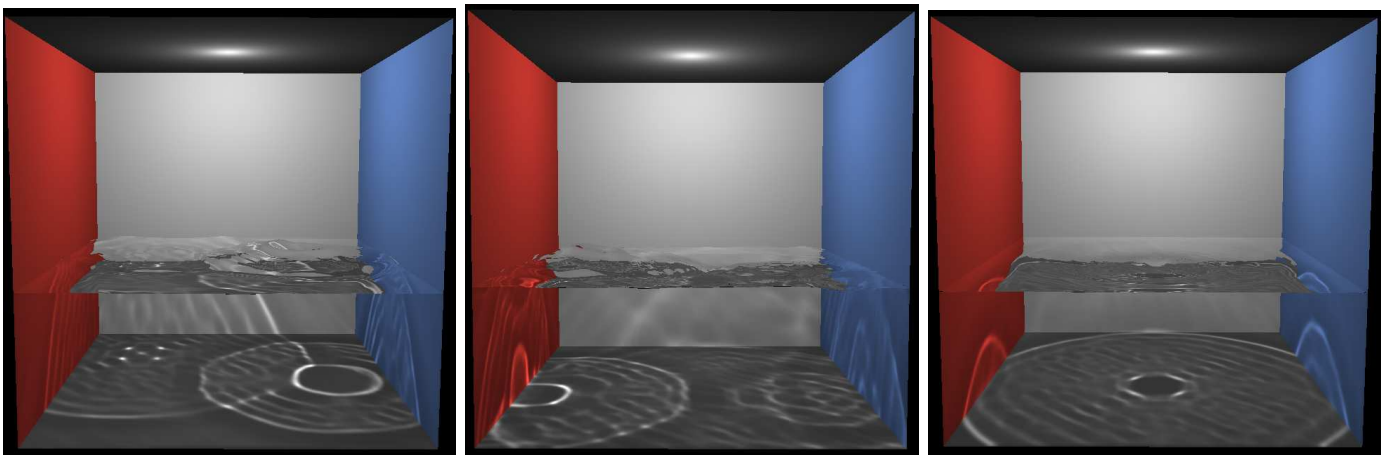

Fig. 8. Frames from a water animation demo with caustics mapping. Random "plops" are introduced in the water surface mesh at regular time intervals. Notice the caustics pattern and the corresponding shape of the water surface.

a weighting coefficient for each vertex which is then multiplied with the incident light.

In order to compute the coefficient for a vertex, the total flux contribution for that vertex must be determined by observing the total flux through the surface of the caustic-forming object visible to the light souce. Therefore, the total flux contribution for each vertex in the grid is computed as:

$$\Phi_i = \frac{1}{V}(N_i \cdot L_i) \tag{6}$$

where *V* is the total projected visible surface area of the object, N_i is the surface normal vector at the vertex, and L_i is the incident light vector. *V* is determined by counting the number of vertices of the refractive grid that are occupied by the projection of the refractive object. Since the grid resolution is the same as that of the render target texture(s) used in Step 2 of the Caustics Mapping algorithm, the number of rasterized pixels in that step gives the value for *V*. The process of determining the number of rasterized pixels is known as occlusion querying, and this feature is generally implemented in modern graphics APIs. The occlusion query simply returns the number of pixels that pass the Z-test.

It is important to note that the caustic intensity computation is performed every frame since the projected visible surface area can change. However, it does not involve any complex operations that can have a significant impact on the frame-rate.

The final color of the caustics formed through a refractive object is computed by taking into account the absorption coefficient of the object's material. This enables us to achieve colored caustics as shown in Figure 1 and Figure 13.

$$I = I_o e^{-K_a d} \tag{7}$$

where I_o is the incident light intensity, K_a is the absorption coefficient, and *d* is the distance that light travels through the refractive object. This distance is easily determined by rasterizing the back-faces of the object in an initial render pass to obtain positions of the hind points. In fact, this data is already available if double surface refraction is used [18]. For single refractive surfaces such as water, attenuation is performed over the distance between the surface and the receiver geometry.

F. Implementation Issues

The caustics mapping algorithm suffers from issues similar to shadow mapping and other image-space techniques. There are two main points of concern that must be addressed: aliasing, and view frustum limitation. The former issue is inherent in all image-space algorithms. This problem is further magnified due to the usage of point primitives rather than triangles for rendering of the caustic-map. The gaps between the point splats give a non-continous appearance to the caustics. However, if there are sufficient number of vertices in the refractive vertex grid, the gaps are significantly reduced. Figure 11 shows a comparison of the caustics produced with two different refractive grid resolutions: 64×64 and 128×128. In our implementations, we have used grids of 128×128 vertices. It is important to note that the tessellation of the mesh of refractive object has no effect on the quality of the caustics, which is solely dependent on the resolution of the refractive grid. Further antialiasing can be achieved by using textured point primitives with a gaussian fall-off for splatting.

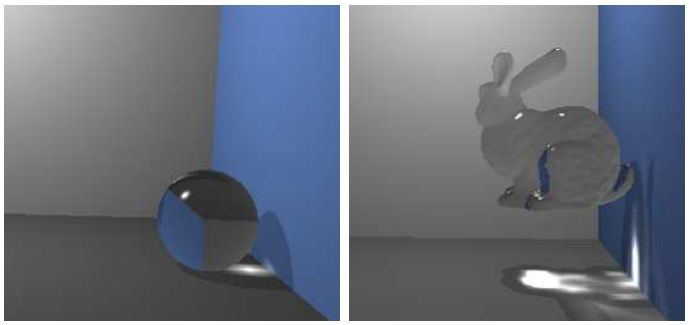

Fig. 9. Caustics rendered using our algorithm for simple objects such us spheres (245fps) as well as complex ones such as the Stanford bunny (200fps)

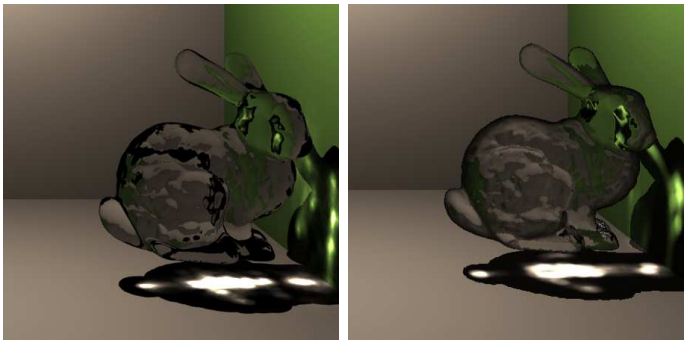

Fig. 10. Comparison of photon mapping (Left) and Caustics Mapping (Right) for a complex refractive object.

This produces continuity between the individual point splats, and was employed for generating all the images in this paper. Although not required, the caustic-map texture can also be low-pass filtered for additional smoothening of the caustics.

The second issue pertaining to the algorithm is the view frustum limitation during rasterization of the caustic-map. This problem is exactly like that of shadow mapping with point lights. In such a case an environment shadow map is created, usually with six 2D textures corresponding to each face of a cube map rather than a single texture. Similarly, in the caustics mapping algorithm if the caustics are formed outside the light's view frustum, they will not be captured on the caustic-map texture. This happens more frequently with reflective caustics than refractive ones. Using an environment caustic map solves this problem at an overhead cost of rendering extra textures. In our implementation we employed cube maps, however the dual paraboloid mapping technique proposed by Heidrich [6], which has been applied to shadow mapping for omnidirectional light sources by Brabec et al. [2], can also be utilized.

IV. Results and Limitations

The caustics mapping algorithm was developed and implemented on a 2.4GHz Intel Xeon PC with 512MB of physical memory. However, since the algorithm performs no computation on the CPU, a lower-end processor will be sufficient. We employed a GeForce 7800 graphics card and pixel shader model 3.0 because the algorithm requires texture access in the vertex shader for intersection estimation in the caustic-map generation step. The algorithm was implemented using Microsoft DirectX 9.0 SDK.

The major advantage of our algorithm is the speed at which it renders the caustics, making it very practical for utilization in games and other real-time applications. We conducted a number of tests and produced results to demonstrate the feature-set of our algorithm. Caustics from both refractive and reflective objects were generated. For refractive objects, two main categories were established: single-surface refraction, and double-surface refraction. Single-surface refraction is suitable for underwater caustics since there is only a single refraction event as light enters through the water surface and hits the floor. The result of our underwater caustics can be seen in Figure 8. It was rendered at a resolution of 640×480 at the rate of 60 frames per second. The mesh used for the water surface consisted of 100×100 vertices.

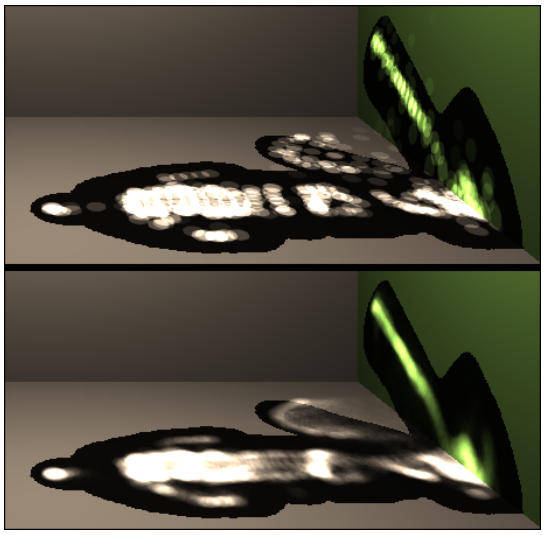

Fig. 11. Effect of refractive grid resolution. The top image was rendered using a grid resolution of 64×64, whereas the bottom image was rendered using a grid resolution of 128×128. The gaps between point splats that are visible at grid resolution 64x64, are not noticeable at grid resolution 128x128

For solid refractive objects, using double-surface refraction is more physically correct. Most of the previous work done in real-time caustics rendering is limited to single-surface refraction and cannot be easily extended to include the double-surface interaction. Our algorithm effortlessly accommodates Wyman's [18] image-space double-surface refraction. We rendered various simple and complex objects and observed the caustics that they produced. The results with caustics from a sphere, and the Stanford bunny are shown in Figure 9. Notice that the frame-rate even with double-surface refraction is quite high. The shadows in these images were obtained using conventional shadow mapping. Since we deal with point lights, a shadow cube map was employed. The resolution of both the shadow map and the caustics map was $768 \times 768 \times 6$ pixels for rendering final images of resolution 1024×768 pixels. Figure 1 shows caustics from the Stanford bunny up close, occupying most of the caustic map region. The frame-rate in this case is 42fps, which shows that the algorithm is fill-rate dependent.

Figure 10 shows a comparison of images rendered with Caustics Mapping and an offline rendering system using photon mapping. Notice that even with complex objects, such as the Stanford bunny, our algorithm produces visually convincing results. Some small differences do exist, and they are mainly attributed to the accuracy of the emulation of the refraction events that take place as light travels through the bunny to the receiver surface. For simpler geometry, such as spheres, the caustics produced using our algorithm are closer to those produced with photon mapping since the refraction events are fairly simple and can be accurately emulated using double surface refraction. Figure 6 shows a comparison of

the distant imposters method, our caustics mapping algorithm, and photon mapping using PBRT. The difference images are shown in Figure 7. The difference image on the right shows that the caustics generated by our method are similar to the caustics generated from classical photon mapping algorithm.

Caustics from reflective objects using our method are shown in Figure 12. Our caustics mapping algorithm performs better with refractive caustics than reflective ones since the error in the intersection estimation during the caustic-map generation step tends to be greater in the latter case. This is due to the fact that refraction causes small deviations in the path of the light ray, whereas reflection causes the ray to change its direction completely. Therefore for the reflection case, the visibility function is more probable to be discontinuous. We would like to improve upon our intersection estimation algorithm in future work to better handle this issue.

Another limitation of our algorithm is that it does not easily extend to volumetric caustics like some techniques proposed in previous work [3]. This is mainly due to the fact that the caustic-mapping algorithm operates in image-space. One way of achieving volumetric caustics using our algorithm is to use a number of planes perpendicular to the light ray as the caustic receivers. However this method spawns further issues relating to sampling and volumetric rendering which need to be addressed.

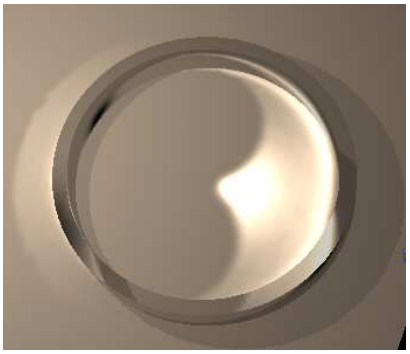

Fig. 12. Reflective caustics from a metal ring. The desired shape that is observed in real life is obtained from the caustics-mapping algorithm. Notice the circular caustic band outside the left edge of the ring.

V. Conclusion and Future Work

We have presented a practical real-time caustics rendering algorithm that runs entirely on the graphics hardware and requires no pre-computation. It emulates the light transport involved in caustics formation in the image-space; therefore it is both physically inspired and fast. The algorithm is conceptually similar to shadow mapping and integrates easily into virtually any rendering system.

We also presented a fast ray-scene intersection estimation technique which we plan to further explore and improve in future work. This technique is extremely fast and is well suited to algorithms, such as caustics mapping, which can tolerate a certain amount of error in the estimation.

Finally we would like to develop methods of applying the caustics mapping algorithm to participating media and achieve volumetric caustics.

Acknowledgment

The authors would like to thank Guillaume Francois, Kévin Boulanger, and Jared Johnson for their insightful comments, and Eric Risser for assistance with illustrations used in this paper. This work was supported in part by I2Lab, Electronic Arts and ATI Reseach.

Appendix

The following code listing is a GPU implementation of the Newton-Raphson root finding method for the intersection estimation computation. Note that this is the original, unmodified version of the Newton-Raphson algorithm. In our work, we use a modified version which does not require the computation of the derivative and is thus cheaper to evaluate. The input parameters to the function are respectively: *pos*, the origin of the refracted ray; *dir*, the direction of the refracted ray; *mVP*, the light's View-Projection matrix; *posTex*, the receiver geometry positions texture; *num_iter*, the desired number of iterations. The function *getTC()* used in the code computes texture coordinates from a 3D point by projecting it to the texture's view plane and moving it to the range [0,1], identical to computing shadow mapping coordinates. The function returns the distance along the refracted ray as an estimate of the intersection point.

```
float rayGeoNP(float3 pos, float3 dir, float4x4 mVP,
               sampler posTex, int num_iter)
{
    float eps = 0.1;
    float2 tc = float2(0.0,0.0);

    // initial guess
    float x_k = 0.10;
    for(int i=0;i<num_iter;i++)
    {
        // f(x_k)
        float3 pos_p = pos + dir*x_k;
        tc = getTC(mVP, pos_p);
        float3 newPos1 = tex2D(posTex, tc);
        float f_x_k = distance(newPos1 , pos_p);

        // f(x_k + eps)
        float3 pos_q = pos + dir*(x_k+eps);
        tc = getTC(mVP, pos_q);
        float3 newPos2 = tex2D(posTex, tc);
        float f_x_k_eps = distance(newPos2 , pos_q);

        float deriv = (f_x_k_eps - f_x_k)/eps;

        // the newton raphson iteration
        x_k = x_k - (f_x_k/deriv);
    }

    return x_k;
}
```

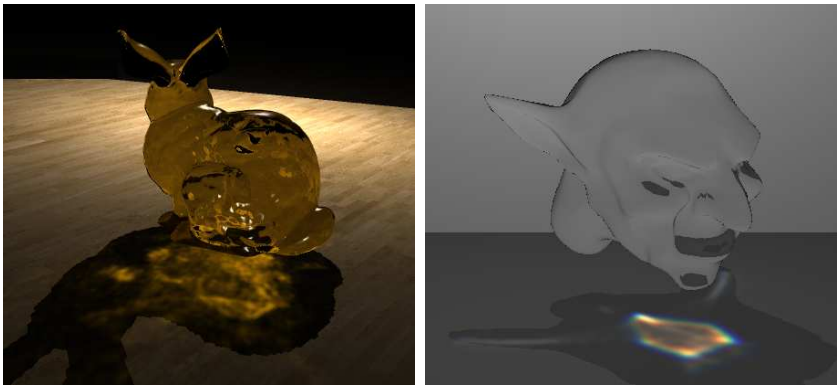

Fig. 13. (Left) Colored caustics from a refractive bunny with light attenuation in different color channels. (Right) Spectral refraction effect obtained using the caustic mapping algorithm with different refractive indices for each of the three RGB color channels.

REFERENCES

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

[2] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Shadow mapping for hemispherical and omnidirectional light sources. In *Computer Graphics International*, 2002.

[3] Manfred Ernst, Tomas Akenine-Möller, and Henrik Wann Jensen. Interactive rendering of caustics using interpolated warped volumes. In *Graphics Interface*, May 2005.

[4] Johannes Günther, Ingo Wald, and Philipp Slusallek. Realtime caustics using distributed photon mapping. In *Eurographics Symposium on Rendering*, pages 111–122, 2004.

[5] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 119–127, New York, NY, USA, 1984. ACM Press.

[6] Wolfgang Heidrich. View-independent environment maps. In *Proceedings of Eurographics/SIGGRAPH Workshop on Graphics Hardware '98*, 1998.

[7] Kei Iwasaki, Yoshinori Dobashi, and Tomoyuki Nishita. An efficient method for rendering underwater optical effects using graphics hardware. *Computer Graphics Forum*, 21(4):701–711, 2002.

[8] Kei Iwasaki, Yoshinori Dobashi, and Tomoyuki Nishita. A fast rendering method for refractive and reflective caustics due to water surfaces. In *Eurographics*, pages 283–291, 2003.

[9] Henrik Wann Jensen. Global Illumination Using Photon Maps. In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, pages 21–30, New York, NY, 1996. Springer-Verlag/Wien.

[10] Henrik Wann Jensen and Per H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. In *Computer Graphics (SIGGRAPH 98)*, pages 311–320. ACM Press, 1998.

[11] Tomoyuki Nishita and Eihachiro Nakamae. Method of displaying optical effects within water using accumulation buffer. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 373–379, New York, NY, USA, 1994. ACM Press.

[12] 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.

[13] Jos Stam. Random caustics: natural textures and wave theory revisited. In *SIGGRAPH '96: ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96*, page 150, New York, NY, USA, 1996. ACM Press.

[14] László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. Approximate ray-tracing on the GPU with distance impostors. In *Proceedings of Eurographics 2005*, 2005.

[15] C. Trendall and A. Stewart. General calculations using graphics hardware with applications to interactive caustics. In *Eurographics Workshop on Rendering*, pages 287–298, 2000.

[16] M. Wand and W. Straßer. Real-time caustics. *Computer Graphics Forum*, 22(3):611–611, 2003.

[17] Mark Watt. Light-water interaction using backward beam tracing. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 377–385, New York, NY, USA, 1990. ACM Press.

[18] Chris Wyman. An approximate image-space approach for interactive refraction. In *SIGGRAPH 2005*. ACM Press, 2005.

[19] Chris Wyman, Charles D. Hansen, and Peter Shirley. Interactive caustics using local precomputed irradiance. In *Pacific Conference on Computer Graphics and Applications*, pages 143–151, 2004.