Eurographics Symposium on Rendering (2006)
Tomas Akenine-Möller and Wolfgang Heidrich (Editors)

Interactive Screen-Space Accurate Photon Tracing on GPUs

Jens Krüger, Kai Bürger, Rüdiger Westermann

Computer Graphics & Visualization, Technische Universität München†

Abstract

Recent advances in algorithms and graphics hardware have opened the possibility to render caustics at interactive rates on commodity PCs. This paper extends on this work in that it presents a new method to directly render caustics on complex objects, to compute one or several refractions at such objects and to simulate caustics shadowing. At the core of our method is the idea to avoid the construction of photon maps by tracing photons in screen-space on programmable graphics hardware. Our algorithm is based on the rasterization of photon paths into texture maps. Intersection events are then resolved on a per-fragment basis using layered depth images. To correctly spread photon energy in screen-space we render aligned point sprites at the diffuse receivers where photons terminate. As our method does neither require any pre-processing nor an intermediate radiance representation it can efficiently deal with dynamic scenery and scenery that is modified, or even created on the GPU.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computing Methodologies]: COMPUTER GRAPHICSThree-Dimensional Graphics and Realism; I.3.3 [Computing Methodologies]: COMPUTER GRAPHICSPicture/Image Generation

1. Introduction and Related Work

Caustics are a light phenomenon that is caused by converging light, and they can appear whenever light impinges at reflecting or transmitting material. A caustic is observed as a brightness, or radiance, increase due to many light paths hitting a surface at the same position. These paths may have been reflected or refracted one or several times before impinging at the receiver. In this work, we restrict the discussion to specular-to-diffuse light transport, where reflected or refracted light hits a diffuse receiver causing light rays to terminate. In Figure 1, a number of different caustics and refractions are illustrated.

As the receiver, in general, does not know from which directions the light converges towards it, caustics cannot easily be simulated using conventional ray-tracing or scanline algorithms. As these methods backtrace the light paths from the eye point, all possible directions the light could arrive from have to be sampled at a caustics receiver. While it is impossible in general to directly render caustics effects using single pass scanline algorithms, indirect lighting effects involving specular reflections can be simulated via Monte Carlo ray-tracing [Kaj86, VG97]. As these techniques come at the expense of tracing many rays to sample the incoming energy at every receiver point, only parallel implementations [WBS*02, PMS*99] have the potential to run at interactive rates.

Instead of sampling the incoming energy for every receiver during the rendering pass, Arvo [Arv86] proposed a two pass method that first creates an illumination map containing the indirect illumination received by an object and then samples this map in the final rendering pass. The method was named backward ray-tracing because it creates the illumination map by tracing rays from the light source instead of the eye point as in conventional ray-tracing. A different strategy to create the illumination map based on the projection of area splats was later proposed by Collins [Col94]. Inspired by the work of Heckbert and Hanrahan [HH84] on beam tracing, a variation of backward ray-tracing that creates caustics polygons by projecting transmitted light beams onto diffuse receivers was suggested by Watt [Wat90]. A similar approach has been utilized by Nishita and Nakamae [NN94] for the rendering of underwater caustics.

Following the early work on two-pass ray-tracing, Jensen

† jens.krueger@in.tum.de, kai.buerger@in.tum.de, westermann@in.tum.de

Figure 1: *GPU photon tracing can be used to simulate a variety of different light effects at interactive frame rates. Even the rightmost scene including a dynamic object, shafts of light and caustics shadowing still runs at 10 fps on a 800x600 viewport.*

and Christensen [JC95] introduced the more general concept of photon maps. The key idea is to first emit energy samples into surface-aligned maps via backward ray-tracing and then reconstructing radiance estimates from these maps in the final rendering pass. It is in particular due to the computational cost for building accurate photon maps via ray-tracing that even GPU implementations [PDC*03, PBMH02] cannot achieve interactive rates for dynamic scenery of reasonable size. Only parallel implementations on multi-processor architectures [GWS04, WHS04] have shown this potential so far.

As caustics - in particular if caused by dynamic objects - are a light phenomenon that adds increasing realism and diversity to any 3D scene and rendering, much effort has been spent recently on developing techniques that can be used in interactive applications like computer games or virtual reality environments. In the following we will discuss the most fundamental concepts behind them.

1.1. GPU Caustics

Before summarizing previous work on interactive caustics rendering we will first briefly describe how to employ functionality on GPUs to efficiently construct illumination maps. This preliminary discussion is restricted to planar receivers that receive energy via one single specular indirection. As refractions at a water surface above planar grounds satisfy these assumptions, this particular example is used in the following. The purpose of this discussion is to introduce some of the basic GPU concepts used today to render caustics at interactive rates and to demonstrate the state of the art in this field. Later in the text we will present efficient techniques that are far less limited in the kind of transmitting objects and receivers they can render

On recent GPUs the *render-to-vertexbuffer* functionality allows the output of the fragment units to be directly rendered into a vertex buffer. By using this functionality caustics can be rendered into an illumination map in two passes. In the first pass, a water surface with color coded normals is rendered from the light source position and a simple fragment program calculates for each fragment the refracted line of sight. If the equation of the planar receiver is known the intersection points between these lines and the receiver can be computed analytically, too. Assuming the mapping of the illumination map to the receiver being known, the intersection points can be transformed into the local parameter space before they are output to a vertex array. In the second pass, this array is rendered as a point set into an illumination texture map, where contributions to the same texel are summed by accumulative blending [SB97]. Figure 2 shows a typical scene that can be rendered using this method.

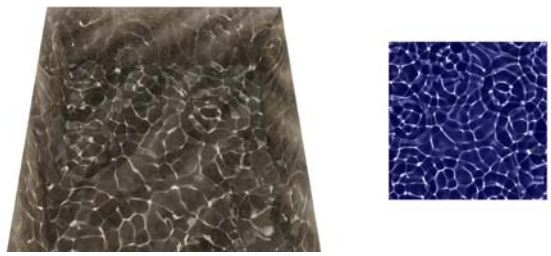

Figure 2: *Caustics on planar receivers:* 256^2 *Photons are rendered as point primitives into a cube-map. On a* $1K \times 1K$ *viewport the scene is rendered (including the construction of a* 5×128^2 *cube-map) at over 130 fps on current GPUs. The illumination map for the bottom plane is shown on the right.*

As pointed out by Wyman and Davis [WD06], the rendering of the vertex array using one pixel sized point primitives results in speckled caustics. In addition it was observed, that the rendering of points with the same intensity does not account for the spread of transferred energy over an area receiver. To overcome this limitation two different approaches were suggested. Either the intersection points of reflected or refracted light rays with the planar receiver are connected and rendered as caustics polygons with area-dependent intensity, or the intensity at the receiver is computed by adding the contributions from all photons hitting the object and by spreading these intensities in image space. Both alternatives assume adjacent primitives - either vertices in the vertex ar-

ray or pixels in image-space - belonging to the same object and being close to each other in world space.

With respect to the aforementioned GPU implementation of caustics effects two questions remain to be answered. First, how can several reflections or refractions be handled. Second, how can intersections between light rays and arbitrary, i.e. non-planar, diffuse receivers be computed. Wyman [Wym05a] proposed to approximate the exit point of a light-ray refracted several times in the interior of a focal object by a linear interpolation between two distances: The distances from the entry point to the convex hull of the object into the direction of the inverse normal and into the direction of the undistorted light ray. While the former distance is pre-computed the latter is obtained from the depth buffer at runtime. The refracted image of the surrounding scene is then obtained using a final lookup into an environment map. An extension to this approach that also accounts for geometry close to the exit point was later proposed in [Wym05b]. Although the approximation of the refracted ray is valid only if the object is convex and both intersection points along the inverse normal and the undistorted ray fall within the same polygon, in particular if the refraction is low the method gives realistic results at impressive frame rates.

Sha and Pattanaik [SKPar] estimate the intersection point between a single refracted light ray and a receiver using the undistorted image of the receiver as seen from the light source. Starting with an initial guess for the intersection point, its position is corrected by iteratively moving this point along the refracted ray in screen-space. As a consequence, the quality of this method strongly depends on the initial guess as well as on the size and the shape of the receiver.

Nishita and Nakamae [NN94] employed frame buffer hardware for blending polygonal caustics beams. This work was later extended by Iwasaki et al. [IDN02] to efficiently utilize the potential of programmable graphics hardware. Wand and Strasser [WS03] suggested to gather the radiance at sample points on a receiver by rendering the scene as seen from these points. The method effectively accounts for caustics shadowing, but it requires the scene to be rendered several times and restricts the gathering to a rather small solid angle. Larsen and Christensen [LC04] proposed to compute photon maps on the CPU, to render photons as point primitives and to filter the photon distribution in image-space on the GPU. Distance impostors were introduced in [SKALP05] to efficiently approximate the intersection points between light rays and reflections (refractions) stored in environment maps. Ernst et al. [EAMJ05] built on the concepts of polygonal caustics beams and shadow volumes [Cro77]. For every patch of the focal object a caustics volume is created, and it is used as a shadow volume that illuminates points of the scene inside that volume. The method is well suited for the rendering of underwater scenery including shafts of light, but it introduces a significant geometry load and does not account for the blocking of caustics rays by a receiver. Iwasaki et al. [IDN03] suggested an acceleration technique for caustics simulation based on the voxelization of polygonal objects into a stack of 2D slices. Approximate caustics can then be rendered by computing the intensities on these slices.

1.2. Contribution

From the above discussion it becomes clear that it is still a challenge to develop techniques that enable real-time *and* accurate rendering of caustics on and caused by complex objects. The method proposed in this work addresses these requirements in that it provides an effective means to resolve inter-object light transfer as well as several refractions at complex and dynamic polygonal objects. This is achieved by the following strategy:

- Photon rays are traced in screen-space on programmable graphics hardware. This is realized by rendering line primitives with respect to the current view. To be able to resolve intersection events at arbitrary positions along these lines they are rasterized into texture maps.
- Intersections between photon rays and objects in the scene can now be detected using layered depth maps and simple fragment operations.
- By rendering oriented point sprites at the receiver pixels in screen-space we account for the energy transport through beams of light. The use of energy splats significantly minimizes the number of photons that have to be traced.

In combination, an interactive method at high visual quality is proposed. As this method does neither require any pre-processing nor an intermediate radiance representation it can efficiently deal with dynamic scenery and scenery that is modified or even created on the GPU. Caustics shadowing is implicitly accounted for by terminating photon rays at the diffuse objects being hit. By only slight modifications the method can be used to render shafts of light.

Besides the advancements our method achieves, the following limitations are introduced by its special layout: First, as intersection events are resolved in screen-space, intersections with triangles not covering any pixel will be missed, i.e. triangles outside the view frustum, triangles parallel to the view direction and triangles below the screen resolution. Second, the accuracy of the method depends on the accuracy of the scan-conversion algorithm implemented by the rasterizer as well as the floating point precision in the GPU shader units. It is thus clear that the proposed method can produce images that are different to an exact solution. It is, on the other hand, worth noting that the proposed method significantly speeds-up photon tracing and makes it amenable to interactive and dynamic scenarios. In addition, as will be shown in a number of examples throughout the text, it achieves very plausible results without notable artifacts.

The remainder of this paper is organized as follows. In Chapter 2 we describe the rasterization of photon rays into texture maps and we show how intersections along these rays can be detected using layered depth images. The next Chapter is dedicated to rendering issues such as oriented point sprites and the simulation of shafts of light. In Chapter 4 we describe the extension of our technique to simulate several refractions in the interior of an object. We then give timing statistics for different scenarios. The paper is concluded with a discussion of future improvements.

2. Screen-Space Photon Tracing

In the following we assume, that for a set of photons emitted from the light source all specular-to-specular bounces have been resolved and a final specular-to-diffuse energy transfer is going to take place. For each photon a position along with the direction of this transfer is stored in the *transfer map*. For single refractions, for instance at a water surface, the GPU-based approach as described above can be directly used to generate this information. More complex bounces can be simulated using either the method proposed by Wyman [Wym05a] or the one we present at the end of this chapter.

The problem is to find for all photons the intersections with the receiver objects in the direction of light transfer. This can be done on a per-fragment basis by rendering caustics rays as line primitives clipped appropriately at the scene boundaries. The idea then is to process the fragments being generated during the rasterization of these lines by a pixel shader, and to only illuminate those pixels that correspond to an intersection point. Assuming that the depth image of the scene is available in a texture map, intersections can be detected by depth comparison in the pixel shader. Every fragment reads the corresponding value from the depth map and compares this value to its own depth. If both values differ less than a specified threshold the fragment color is set. Otherwise the fragment is discarded.

The envisioned algorithm has two basic problems: First, as it finds and illuminates all intersection points along the photon paths it is not able to simulate the termination of these paths at the first intersection points. While a shadow algorithm could be utilized to attenuate this effect, caustics impinging in the interior of the shadow cast and especially shadowing effects due to dynamic focal objects cannot be simulated in this way. Second, intersections with occluded geometry, which is not present in the depth map, cannot be found (see Figure 3).

2.1. Line Rasterization

To overcome the aforementioned limitations we suggest to rasterize photon rays into a texture map and to perform intersection testing for every texel in this map. The first intersection along each ray can then be determined using a log-step

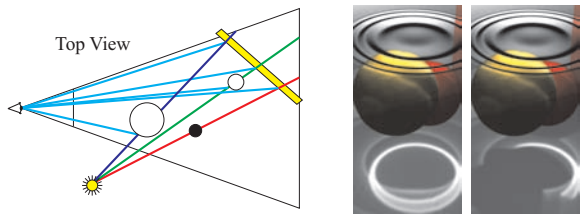

Figure 3: *If a pixel shader illuminates all line-object intersections in view-space, this yields incorrect results. In the example, five hits instead of three would be illuminated due to the black sphere being occluded in view space. In the two right images a scene is rendered without (left) and with (right) caustics shadowing. The artifacts can be clearly observed.*

texture reduce operation [KW03b]. The coordinates of the detected receiver points, now stored in the RGB components of a texture map, can finally be rendered in turn on the GPU without any read-back to the CPU.

Every photon ray is rendered into a distinct row of a 2D off-screen buffer, i.e. a texture render target (see Figure 5). This is accomplished by rendering for each photon a line primitive. Due to performance issues the set of all these lines is stored in a vertex array in GPU memory. For every line being rendered a vertex shader fetches the respective photon position $\vec{o}$ and direction $\vec{d}$ from the transfer map, and it computes the number of fragments, n_f, that would have been generated for this line by the rasterizer. This number can simply be obtained from the projection of $\vec{o}$ and $\vec{o}+t\cdot\vec{d}$ into screen-space. The shader then displaces the initial vertex coordinates such as to generate a line starting at the left column of the off-screen buffer and covering n_f fragments (see Figure 4). In addition, the screen-space position of $\vec{o}$ and $\vec{o}+t\cdot\vec{d}$ is assigned as texture coordinate to the start and the end vertex, respectively, of every line.

During scan-conversion the rasterizer interpolates the texture coordinates along the horizontal lines. It thus generates for every fragment the screen-space position it would have been mapped to if the initial caustics ray was rendered as a line primitive. A pixel shader operating on these fragments can use this position to fetch the respective value from the depth map and to compare the interpolated screen-space depth to this value. If an intersection is determined, the screen-space position of the fragment in the off-screen buffer is written to that buffer. Otherwise the fragment is discarded (see Figure 4).

After all lines have been rasterized, a texture is generated that stores in each row all the intersections between a particular photon ray and the visible objects in the scene. By applying a horizontal texture reduce operation the first intersection points are rendered into a single-column texture map. For photon rays that are almost parallel to the view di-

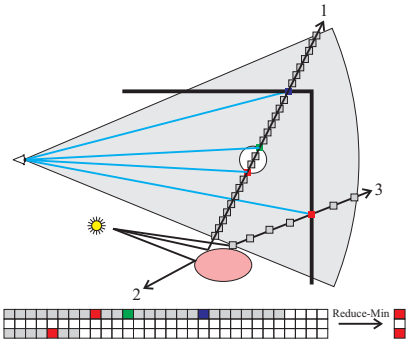

Figure 4: *The pink object at the bottom reflects the light from the light source and causes three caustics rays. These are processed by the vertex shader, which computes the length of each ray and transforms them to horizontal lines of equal length. The rendering of these lines generates the image shown at the bottom. Red, green and blue pixels indicate hits with an object, and grey cells indicate fragments that have been discarded in the pixel shader. Into white cells a fragment has never been rendered.*

rection and thus only cover a very few fragments a minimum line length is enforced. If the number of rows in the render target is less than the number of photons, the process is repeated for the remaining photons until all of them have been processed.

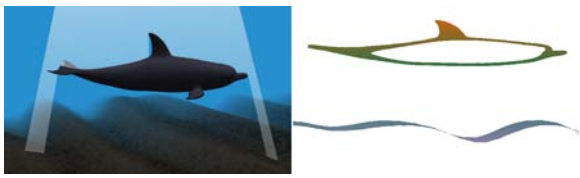

Figure 5: *The right image shows an except of the ray intersection texture while in the image the corresponding rays can be seen. For this image a very large epsilon was chosen.*

Similar to volume ray-casting on GPUs [KW03a, SSKE05], photon tracing can also be implemented as a single pixel shader being executed for each photon. In theory, the advantage of such an implementation is that the traversal process can stop as soon as an intersection with any of the objects in the scene is found. However, a comparison with the approach suggested here revealed a loss in performance of about 50%. We attribute this observation to the fact that early-out mechanisms like the early-z test or breaks in a pixel shader introduce some overhead, either a branch instruction or additional rendering passes. Moreover, since the pixel shader hardware runs in lock-step, a performance gain can only be achieved if all fragments in a contiguous array exit the program early. These observations are backed up by latest GPUBench [BFH04] results (Results for our target architecture, the GeForce 7800 GTX, are available at http://graphics.stanford.edu/projects/gpubench/results/). These results attest current GPUs a rather bad branching performance even if all fragments in a 4x4 block exit the program simultaneously.

2.2. Depth Peeling

In the presentation so far intersections with occluded geometry have been entirely ignored. The reason for this is that only one single depth map containing the depth values of the visible fragments has been considered. As can be imagined easily this limitation introduces false results and noticeable artifacts.

To be able to compute the intersection between photon rays and all objects in the scene we generate a layered depth image [SGwHS98] from the current viewing position. Therefore we employ depth-peeling [Eve01]. In the current implementation we use depth-peeling not only to generate multiple depth maps but also to generate multiple normal maps of the layered fragments. In the caustics rendering approach with lines, instead of comparing the fragment's depth to only the values in the first depth layer we now compare it to the values in the other layers as well. An intersection is indicated by at least one entry in the layered depth map that is close to the fragments depth.

To generate the layered depth image we need to know the depth complexity of the scene for the current view. The depth complexity can be determined by rendering the object once and by counting at each pixel the number of fragments falling into it during scan-conversion. The maximum over all pixels is then collected by a texture reduce-max operation. As the depth complexity of a scene can vary significantly depending on the view-point and the viewing direction, depth-peeling has to be restricted to a maximum number or passes to maintain a constant frame rate. Interestingly, our experiments have shown that it is usually sufficient to only consider up to eight depth layers. This is due to the fact that deeper layers do in general not contain a significant number of fragments, and only a few of them block any photon ray that would otherwise be seen by the viewer. In scenarios where the depth complexity is high because the scene consists of many objects, separate depth images for each object should be favored to reduce geometry load and memory overhead.

2.3. Recursive Specular-to-Specular Transfer

The proposed method enables the tracing of photons through the scene until the first intersection with any object in this scene is found. For every photon the algorithm outputs the position of the intersection point, the normal at this point and the direction of the incoming photon ray. It is thus easy to simulate subsequent specular-to-specular light transfer by reflecting or refracting the incoming ray at the intersection point, and by writing both the point coordinate and the modified direction vector into the transfer map. The algorithm is then restarted with the updated map.

3. Rendering

When light hits a diffuse surface where it is emitted equally in all directions the scene is illuminated at the point of intersection. As we have described above, for all photons these *receiver points* are encoded in the transfer map. To minimize the number of photons to be traced through the scene we assume that a given photon deposits a certain amount of energy in the surrounding of the intersection point. Similar in spirit to the distance weighted radiance estimate proposed by Jensen [Jen97] we spread the photon energy over a finite area centered around the intersection point. Surface points within this area receive an energy contribution that is inversely proportional to their distance to the center. In contrast, however, in our approach we do not gather the energy from nearby photons but we accumulate the contributions by rendering area illumination splats.

3.1. Sprite Rendering

To simulate the spread of energy at a particular point in the scene we employ point sprites. Point sprites are a means to draw points as customized textures with texture coordinates interpolated across the point. In the current scenario we can simply render one sprite for each point stored in the transfer map. By exploiting the *render-to-vertexbuffer* functionality or texture access in a vertex shader such a rendering operation can entirely be realized on the GPU. Unfortunately, point sprites come with the restriction that they are screen aligned, i.e. they resemble quadrilaterals centered at the point position and kept perpendicular to the viewing direction. As the alignment of point sprites according to the orientation of the receiver is not supported by current GPUs, a new method that overcomes this limitation is required.

Figure 6: *Sprite rendering: The images show light patterns generated by a few photons. The difference between screen-aligned textured point sprites (left) and the sprite-based ray-caster (right) is shown.*

At the core of our technique we have developed a GPU ray-tracer similar to the one proposed in [BK03] for high-quality point rendering. The idea is to compute for every pixel covered by a sprite the point seen under this pixel in the tangent plane at the receiver point. This plane is defined by the normal stored for every receiver point in the transfer map. The distance of the ray-plane intersection point to the receiver point can directly be used to estimate the amount of energy received from the photon. It also has to be considered that only *visible* surface points should be illuminated. This is accounted for by comparing the screen-space depth of the ray-plane intersection point to the value in the first layer of the layered depth image at the current pixel position. Only if both values are close together does the pixel receive an intensity contribution. In Figure 6 the quality of this approach is demonstrated.

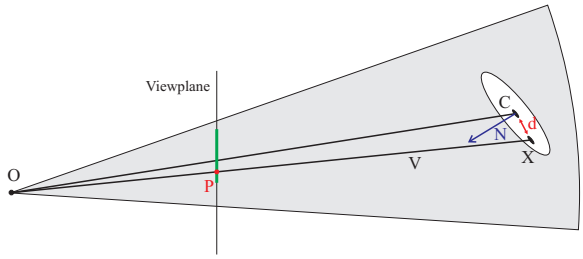

Figure 7: *Schematic view of sprite-based ray-casting. P is a pixel in the image plane, O denotes the viewing position and d is the distance of the intersection point to the receiver point. The distance is used to calculate the intensity. For a spherical intensity splat with size s and linear attenuation this is $s-d$.*

To simulate the illumination caused by a photon ray, for each of the receiver points in the transfer map a point sprite of constant size s is rendered. For every fragment generated during rasterization a pixel shader computes the intensity I as follows (based on the setup in Figure 7):

$$I = s - \left| \frac{D}{\vec{N} \cdot \vec{V}} \cdot \vec{V} - \vec{C} \right| \tag{1}$$

Here, $D = \vec{V} - \vec{C}$, $|D| = d$, $\vec{N}$ and $\vec{C}$ are constant over the sprite and can thus be computed in a vertex shader and passed as constant parameters to the fragment stage. The view direction $\vec{V}$ is also computed in a vertex shader by multiplying the pixels screen-space coordinate with the inverse projection matrix. Equation 1 is easily derived from the ray-plane intersection formula.

3.2. Rendering in Homogeneous Participating Media

Besides the fact that light illuminates surfaces, it causes another fascinating effect when passing through homogeneous participating media: Shafts of light or god-rays (see Figure 8). Adding god-rays to our approach is fairly simple as we know for every photon ray its start position at the focal object and the position where it terminates. Consequently these rays can be rendered as line primitives, which are combined in the frame buffer. As we assume the media to be homogeneous light intensity decreases exponentially with distance. This kind of attenuation can be computed in a pixel shader for every fragment that is rasterized. Contributions from multiple god-rays are accumulated in the frame buffer.

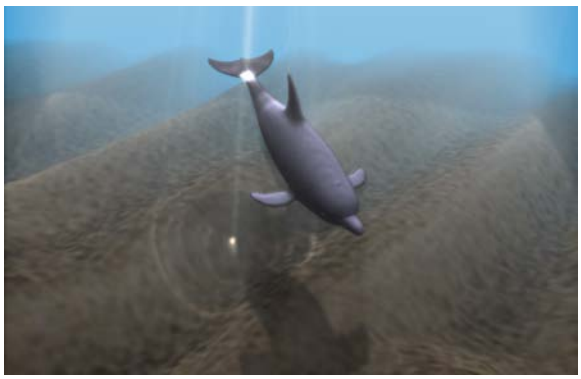

Figure 8: *Underwater scene illuminated by sun light. Shafts of light are caused by the refraction of light. In this scene the light rays converge on the flukes of the dolphin and cause a bright spot of light.*

To account for the scattering of light along the god-rays they are rendered into a separate color buffer and blurred with a Gaussian filter. As this filter has a constant screen-space support it emulates a filter with increasing support in object-space, i.e. rays further away from the viewer are blurred more than those closer to the viewer. The filtered image is finally blended over the image of the scene. As the images demonstrate, although only a very coarse approximation of the real scattering of light is computed the results look very realistic and convincing. The overhead that is introduced by filtering the god-ray image is insignificant compared to the other stages of caustics rendering.

4. Screen-Space Refraction Tracing

The idea of screen-space ray-tracing can also be applied to render refractions through transparent objects as seen by the viewer. Compared to the method suggested by Wyman [Wym05a], this method can efficiently deal with dynamic and concave geometry and it simulates refractions more accurately. On the other hand it comes at the expense of more complex fragment computations thus limiting the performance that can be achieved.

Essentially there is no difference between photon tracing in screen-space and the tracing of refracted lines of sight in screen-space. Apart from the fact that in the latter case the projected lines usually only cover a few pixels on the screen, the same ideas suggested in Section 2 can be utilized. In contrast, however, we build the tracing of refractions on the simplification that along any ray the n-th intersection point with the refracting object is found in the $(n+1)$-th depth layer. We thus assume that ray-object intersection are "monotonic" with respect to these layers. In this way the number of texture access operations to be performed at every pixel under the view-ray is equal to one. It is clear that testing against all levels of the layered depth image causes the performance

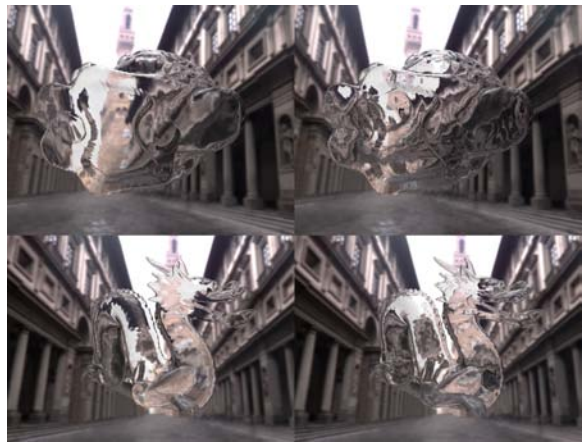

Figure 9: *Comparison between refractions through only one interface (left) and multiple refractions generated via screen-space photon tracing (right).*

to drop linearly in the depth complexity of the object. As the possibility of violating this simplification increases with the strength of the ray deflection, in such scenarios there is a higher chance to miss interior object structures.

The algorithm starts by constructing the layered depth image of the transmitting object. Visible object points along with their normals are rendered into the transfer map. We then rasterize refracted view-rays into a texture map and perform intersection testing for every texel in this map. These rays, just like the caustics rays, are then traced through the depth layers until the depth along the ray is either greater than the depth in the next layer or less than the depth in the current layer. In either case we can assume an intersection with the surface. The point in the interior of the object is considered the exit point and the normal at this point is fetched from the depth image. From this normal and the incoming ray direction the refracted exit direction is computed. This information in turn can either be used to look up an environment map or to retrace the refracted view-rays. Figure 9 shows a comparison between refractions through only one interface and refractions generated by our method.

If a layered depth image of the surrounding scene exists, intersections between the exiting ray and the scene can be detected as well. This approach has been applied in Figure 10 to simulate caustics seen through the water surface. It should be clear, however, that this is an approximation that can only simulate the refraction of light from underwater objects that are visible to the viewer in the absence of the water surface. Otherwise, the refracted object points have not been rendered and they are thus not available in the image of the scene.

Table 1 lists timings for the rendering of refracting objects using the proposed method. In these tests only the first exit point has been considered for casting view-rays into the sur-

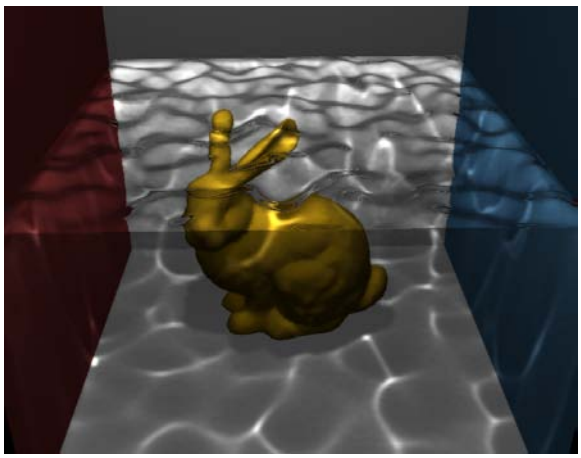

Figure 10: *A Stanford Bunny in the Cornell Box. Note how the refraction on the water surface displaces the bunnys ears.*

		Viewport resolution	
	triangles	800 x 600	1200 x 1024
Teapot	2257	600 (870)	220 (300)
Stanford Bunny	69665	220 (270)	110 (180)
Stanford Dragon Low	47794	280 (320)	120 (160)
Dragon High	202520	150 (156)	95 (105)

Table 1: *Timing statistic in fps for screen-space refraction tracing to allow a comparison to Wyman's approach [Wym05a] (fps in braces) our algorithm was set to compute only two refractions.*

rounding scene. As it can be seen, on recent graphics hardware the algorithm is only slightly slower than a highly optimized version of the one proposed in [Wym05a]. It is, on the other hand, worth noting that our technique does not require any pre-processing and is thus suitable for dynamic geometry.

5. Results

In the following we present some results of our algorithm and we give timings for different parts of it. All test were run on a single processor Pentium 4 equipped with an NVIDIA 7800 GeForce FX graphics processor. In all examples the size of the viewport was set to 800×600. Each caustics image in this paper was generated using 256×256 photons. All objects were encoded as indexed vertex arrays stored in GPU memory to enable optimal performance of depth-peeling.

Before we give timings for the major parts of our algorithm, we will first analyze the accuracy of the proposed method in more detail. Therefore, we have used the described method to trace shadow rays in screen-space. Figure 11 shows results for a single point light source and two different objects casting shadows onto a receiver. For every pixel being covered by the receiver, a photon is traced backward to the light source. Intersections with the objects between the receiver and the light source are resolved using layered depth images as described. An intersection was determined if the world space distance between the fragment of the rasterized line and a fragment coded in the layered depth map was less than 0.01. This tolerance was also used in all other examples throughout this paper.

Figure 11: *Shadows simulated by light ray tracing in screen-space. The scene was rendered on a $1K \times 1K$ viewport.*

As it can be seen, even for the complex tree the shadow on the ground is adequately sampled. Although some pixels are erroneously classified due to numerical and sampling issues, the shadow caused by very fine scale structures can still be resolved at reasonable accuracy. This observation is further evidenced by Figure 12, where the results of our approach have been compared to ray-traced shadows. From the visual point of view, both approaches yield very similar results that can hardly be distinguished from each other. Note that the artifacts in the shadow of some of the thick branches can easily be avoided by an additional in-out test considering consecutive pairs of layered depth images.

Figure 12: *Left: shadows on the floor are generated using our method. Right: shadows are ray-traced in Maya.*

Representative timings in milliseconds (ms) for caustics simulation in three example scenes are listed in Table 2: (A) In the bunny scene (Figure 10) the refraction of underwater objects at the water surface was simulated as described in Chapter 4. (B) God-rays were added to the dolphin scene (Figure 1). (C) In the pool scene (Figure 13) refractions as well as reflections at the water surface were simulated. All scenes have a depth complexity of eight.

The first column (C) shows the amount of time spent by the GPU for caustics simulation including depth-peeling, line rasterization, intersection testing and god-ray rendering. Of

all these parts intersection testing consumed over 80% of the time. In the second column (P) the time spent rendering the point sprites is given. In the last column (A) the overall performance is shown. All measurements have been repeated for different amounts of photons traced from the light source. Photon grids used in the pool demo are 1024x512, 512x256, 256x128, and 128x64. Additional results using $1K \times 1K$ photons are shown in Figure 14.

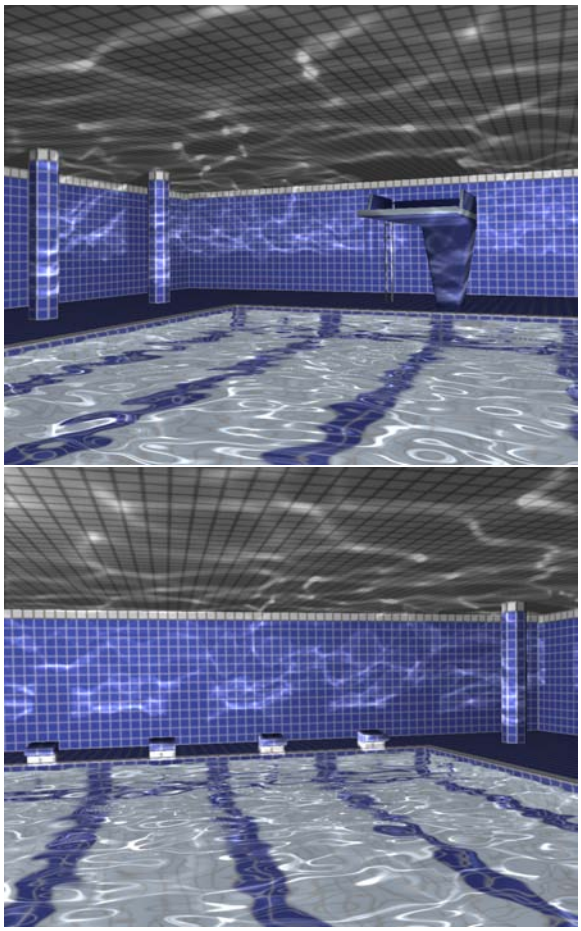

Figure 13: *Refractions and reflections at a water surface.*

	Number of photons								
	512 x 512			256 x 256			128 x 128		
	C	P	A	C	P	A	C	P	A
A	243	26	285	69	8	89	17	3	29
B	232	17	555	62	6	103	17	2	45
C	*434*	*10*	*454*	*119*	*7*	*135*	*32*	*2*	*33*

Table 2: *Timing statistics (in ms) for different scenes.*

As the timings show, even very complex light pattern can be simulated at interactive rates and convincing quality. In particular such effects as shafts of light and caustics shadowing, for instance below the bunnies and the dolphin, and behind the diving platform, add increasing realism and diversity to the 3D scenery. The images also show that even with a rather small number of photons traced through the scene the caustics effects look very realistic and do not exhibit intensity speckles. This is due to the sprite-based approach that realistically accounts for the orientation of the receiver geometry and the spread of photon energy.

6. Conclusion and Future Work

GPU photon tracing in screen-space enables interactive simulation and rendering of complex light patterns caused by refracting or reflecting objects. Even though we did not yet compare the quality of the proposed technique to that of photon mapping, our results look convincing and plausible. The ability to trace large photon sets by rasterizing lines into texture maps in combination with a novel rendering method to account for area energy splats enables visual simulation of caustics effects at high frame rates. In a number of different examples these statements have been verified. The possibility to integrate caustics shadowing, one or several refractions in the interior of complex objects and intensity splats on curved objects distinguishes the proposed GPU technique from previous approaches.

As our method does not require any pre-processing nor an intermediate radiance representation, it can efficiently deal with dynamic scenery and scenery that is modified on the GPU. In this particular respect we plan to investigate several improvements and acceleration techniques in the future.

New functionality to create geometry in a geometry shader on future graphics hardware [BG05] offers several possibilities to significantly accelerate caustics rendering. Such a shader might be used to create all lines being rasterized out of one representative line on the GPU. Another interesting research question is how to exploit such hardware for the acceleration of depth-peeling, and thus for the construction of high-resolution layered depth images. In this respect it will also be of interest to investigate the possibility to render into multiple render targets with own depth buffer.

Caustics rendering as proposed can be parallelized in a straight forward way by assigning disjoint regions in screen-space to different rendering nodes. As only the first intersection points along each photon ray have to be communicating between these nodes, the algorithm is supposed to scale almost linearly in the number of nodes.

References

[Arv86] ARVO J.: Backward ray tracing. Developments in Ray Tracing, SIGGRAPH 1986 Course Notes, 1986.

[BFH04] BUCK I., FATAHALIAN K., HANRAHAN P.: GPUBench: Evaluating GPU performance for numerical and scientifc application. In *Proceedings of the ACM*

Workshop on General-Purpose Computing on Graphics Processors (2004).

[BG05] BALAZ R., GLASSENBERG S.: DirectX and Windows Vista Presentations. http://msdn.microsoft.com/directx/archives/pdc2005/, 2005.

[BK03] BOTSCH M., KOBBELT L.: High-quality point-based rendering on modern gpus. In *Proceedings of Pacific Graphics 2003* (2003), p. 335.

[Col94] COLLINS S.: Adaptive splatting for specular to diffuse light transport. In *Proceedings of the Fifth Eurographics Workshop on Rendering* (1994), pp. 119–135.

[Cro77] CROW F. C.: Shadow algorithms for computer graphics. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques* (1977), pp. 242–248.

[EAMJ05] ERNST M., AKENINE-MÖLLER T., JENSEN H. W.: Interactive rendering of caustics using interpolated warped volumes. In *GI '05: Proceedings of the 2005 conference on Graphics interface* (2005), pp. 87–96.

[Eve01] EVERITT C.: *Interactive order-independent transparency*. Tech. rep., NVIDIA Corporation, 2001.

[GWS04] GÜNTHER J., WALD I., SLUSALLEK P.: Real-time caustics using distributed photon mapping. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering* (2004), pp. 111–121.

[HH84] HECKBERT P. S., HANRAHAN P.: Beam tracing polygonal objects. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (1984), pp. 119–127.

[IDN02] IWASAKI K., DOBASHI Y., NISHITA T.: An efficient method for rendering underwater optical effects using graphics hardware. *Computer Graphics Forum 21*, 4 (2002), 701–711.

[IDN03] IWASAKI K., DOBASHI Y., NISHITA T.: A fast rendering method for refractive and reflective caustics due to water surfaces. In *Eurographics* (2003), pp. "283–291".

[JC95] JENSEN H. W., CHRISTENSEN N. J.: Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers & Graphics 19*, 2 (1995), 215–224.

[Jen97] JENSEN H. W.: Rendering caustics on non-Lambertian surfaces. *Computer Graphics Forum* (1997), 57–64.

[Kaj86] KAJIYA J. T.: The rendering equation. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (1986), pp. 143–150.

[KW03a] KRÜGER J., WESTERMANN R.: Acceleration techniques for GPU-based volume rendering. In *Proceedings IEEE Visualization 2003* (2003).

[KW03b] KRÜGER J., WESTERMANN R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph. 22*, 3 (2003), 908–916.

[LC04] LARSEN B. D., CHRISTENSEN N.: Simulating photon mapping for real-time applications. In *Eurographics Symposium on Rendering* (2004), Henrik Wann Jensen A. K., (Ed.).

[NN94] NISHITA T., NAKAMAE E.: 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* (1994), pp. 373–379.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), pp. 703–712.

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2003), pp. 41–50.

[PMS*99] PARKER S., MARTIN W., SLOAN P.-P., SHIRLEY P., SMITS B., HANSEN C.: Interactive ray tracing. In *Symposium on Interactive 3D Graphics: Interactive 3D* (1999), pp. 119–126.

[SB97] STÜRZLINGER W., BASTOS R.: Interactive rendering of globally illuminated glossy scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques* (1997), pp. 93–102.

[SGwHS98] SHADE J., GORTLER S., WEI HE L., SZELISKI R.: Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques* (1998), pp. 231–242.

[SKALP05] SZIRMAY-KALOS L., ASZODI B., LAZANYI I., PREMECZ M.: "approximate ray-tracing on the gpu with distance impostors". *Computer Graphics Forum 24*, 3 (2005).

[SKPar] SHAH M. A., KONTTINEN J., PATTANAIK S.: Caustics mapping: An image-space technique for real-time caustics. *Transactions on Visualization and Computer Graphics (TVCG)* (to appear).

[SSKE05] STEGMAIER S., STRENGERT M., KLEIN T., ERTL T.: A simple and flexible volume rendering framework for graphics-hardware–based raycasting. In *Proceedings of the International Workshop on Volume Graphics '05* (2005), pp. 187–195.

[VG97] VEACH E., GUIBAS L. J.: Metropolis light transport. *Computer Graphics 31*, Annual Conference Series (1997), 65–76.

[Wat90] WATT M.: Light-water interaction using back-

ward beam tracing. *SIGGRAPH Comput. Graph. 24*, 4 (1990), 377–385.

[WBS*02] WALD I., BENTHIN C., SLUSALEK P., KOLLIG T., KELLER A.: Interactive global illumination using fast ray tracing. In *Eurographics Rendering Workshop* (2002), pp. 15–24.

[WD06] WYMAN C., DAVIS S.: Interactive image-space techniques for approximating caustics. In *Proceedings of the Symposium on Interactive 3D graphics* (2006).

[WHS04] WYMAN C., HANSEN C., SHIRLEY P.: Interactive caustics using local precomputed irradiance. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)* (2004), pp. 143–151.

[WS03] WAND M., STRASSER W.: Real-time caustics. In *Computer Graphics Forum* (2003), Brunet P., Fellner D., (Eds.), vol. 22(3).

[Wym05a] WYMAN C.: An approximate image-space approach for interactive refraction. *ACM Trans. Graph. 24*, 3 (2005).

[Wym05b] WYMAN C.: Interactive image-space refraction of nearby geometry. In *Proceedings of GRAPHITE* (2005), pp. 205–211.