

CHAPTER 14

Ray-Guided Volumetric Water Caustics in Single Scattering Media with DXR

Holger Gruen

NVIDIA

ABSTRACT

This chapter presents a hybrid algorithm that uses ray tracing and rasterization to render surface and volumetric caustics in single scattering participating media. The algorithm makes use of ray tracing based on DirectX Raytracing (DXR) to generate data that drives hardware tessellation to adaptively refine triangular beam volumes that are rendered to slice volumetric caustics. Further on in the rendering pipeline, ray tracing is also used to generate secondary caustics maps that store the positions of ray/scene intersections for light rays that get reflected or refracted by a water surface.

14.1 INTRODUCTION

This chapter investigates how to make use of the DirectX 12 real-time ray tracing API, DXR, to simplify current methods for rendering real-time volumetric water caustics in single scattering media. Volumetric caustics have been investigated extensively in the past [2, 5, 6, 10]. The algorithm described here uses ideas discussed in the literature and combines them with the use of DXR ray tracing and adaptive hardware tessellation.

Specifically, for rendering volumetric caustics, ray tracing is used twice in the rendering pipeline. In an initial step, ray tracing is used to compute information that then guides hardware tessellation levels for triangular beam volumes that are used to adaptively slice caustics volumes. The rendering pipeline for accumulating volumetric light that is scattered toward the eye uses all GPU shader stages, e.g., a vertex shader, a hull shader, a domain shader, a geometry shader, and a pixel shader.

The primary caustics map [7] contains the positions and surface normals of the water surface rendered from the point of view of the light. Rays are sent from these positions on the water along the refracted and reflected light directions, resulting in intersections with the scene. The positions of these intersections are stored in secondary caustics maps such as the refracted caustics map and the reflected

caustics map described in this chapter. The positions in the (primary) caustics map and the refracted caustics map are then used to define the triangular volumetric beams used during volumetric slicing.

This chapter focuses on underwater caustics from refracted light rays. Note that the algorithm described here can also be used to render caustics from light that gets reflected by the water surface and hits geometry above the water line. Also, it is possible to replace the water surface with any other transparent interface.

In underwater game scenes, volumetric lighting is often generated from visibility information encoded in a shadow map [4]. This shadow map contains, in this context, the underwater geometry rendered from the light position. As such, it delivers the intersections of the original light rays with the underwater scene through rasterization. See Figure 14-1.

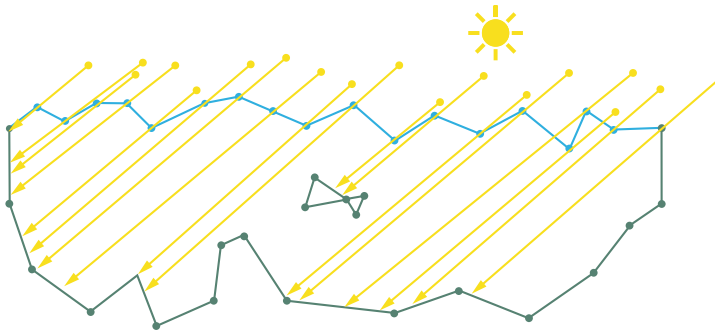


Figure 14-1. *Undisturbed light rays hitting the underwater scene.*

When a ray of light hits the water surface, some of its energy changes direction as it gets refracted by the water surface. It is therefore necessary to find the intersections of the refracted light rays with the scene. See Figure 14-2.

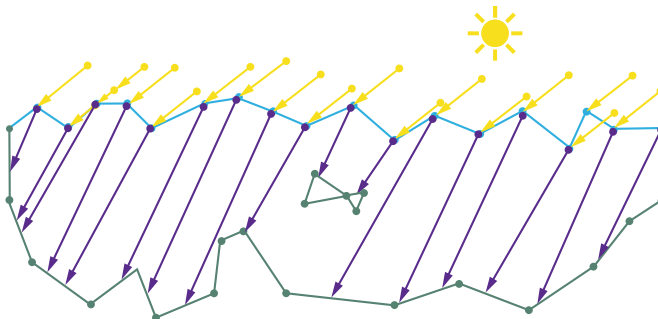


Figure 14-2. *Refracted rays (purple) hitting the underwater scene.*

A comparison of Figures 14-1 and 14-2 shows that the resulting intersection points can be very different. This difference is more pronounced if a light ray hits the water surface at a shallow angle. Refraction causes light rays to generate the typical pattern of surface caustics on the underwater geometry. In a similar manner, volumetric lighting is affected by refracted light. Several publications [5, 6, 8, 9, 10] describe how to move beyond the limits of using just a shadow map (as shown in Figure 14-1) in the context of caustics rendering.

Typically, one of the two following classes of algorithms are used:

1. *Two-dimensional image-space ray marching:*

- (a) March the primary depth buffer or the shadow map depth buffer in the pixel shader to find intersections. The problem with this approach is that refracted light rays may seem to be occluded in both the primary view and the view from the light, as shown in Figure 14-3.

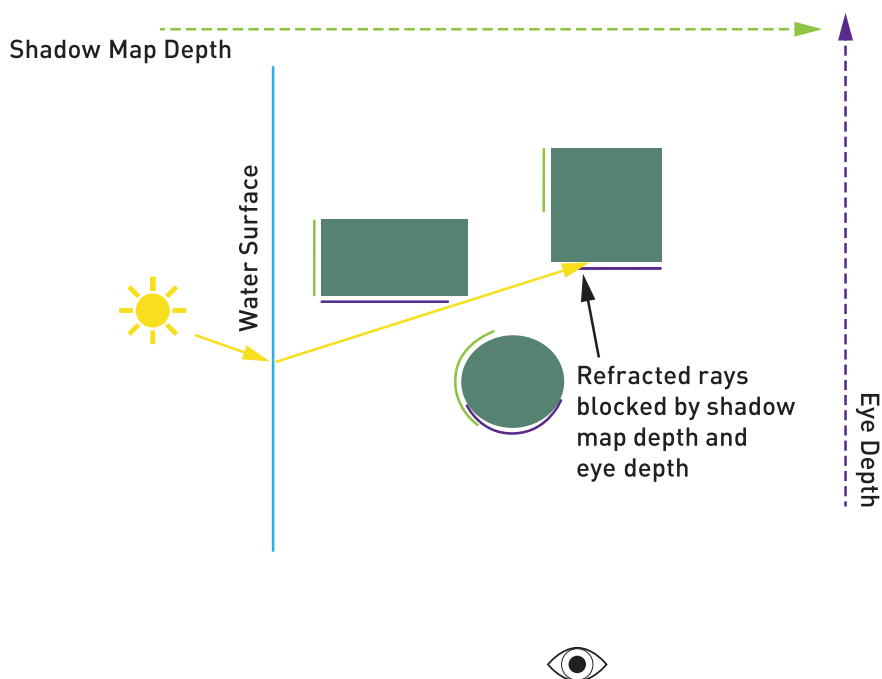


Figure 14-3. The intersection point for the refracted light ray seems to be blocked in marching both the light and eye depth maps.

- (b)** Render and march a set of images using:
- i.** Multiple depth layers of the primary depth buffer and the shadow map.
 - ii.** Multiple viewpoints of the primary depth buffer and the shadow map.
 - iii.** Distance impostors [8].

Note, however, that these methods increase the runtime cost and the memory consumption. The implementation complexity can be significantly higher than the DXR-based approach described later.

- 2.** *Three-dimensional voxel grid marching:* This class of algorithms voxelizes the underwater scene and marches the resulting grid. Dependent on grid resolution, these methods can yield impressive results. Voxelization is not a cheap operation and can be interpreted as the rasterization-side equivalent of keeping a bounding volume hierarchy up to date. Memory requirements become prohibitive quickly if high grid resolutions are required. Ray marching a sufficiently detailed 3D grid is not fast and can become prohibitively slow. Overall, the implementation complexity of voxelization methods is higher than the DXR-based approach.

The technique presented in this chapter doesn't use any of the approximate methods just described to compute the intersections of refracted light rays. Instead, it uses DXR to accurately compute where refracted light rays hit the dynamic underwater scene.

14.2 VOLUMETRIC LIGHTING AND REFRACTED LIGHT

For a general introduction to volumetric lighting computations in participating media, consult the work by Hoobler [4]. Here, we simply present the double integral that describes how much radiance L is scattered toward the eye E from a point S of the underwater scene:

$$L = \int_S^E \int_{\Omega} e^{-\tau((\omega)+|P-E|)} \sigma_s(P) p(E-P, \omega) L_{in}(P, \omega) v(P, \omega) d\omega dP \quad [1]$$

See Figure 14-4. For all points P on the half-ray from the point in the scene to the eye and for all directions of incoming refracted light Ω , the following terms are computed:

1. The extinction along the length $l(\omega)$ that the light has traveled underwater before reaching P plus the length of the path from P to the eye E . Here, τ is the extinction coefficient of the water volume—which is assumed to be constant in the remainder of this chapter.
2. The scattering coefficient $\sigma_s(P)$ at the point P .
3. The phase function $p(E - P, \omega)$ that determines how much of the light that comes in from a refracted light direction is scattered toward the eye from P .
4. The incoming radiance L_{in} at the point P along a refracted light direction.
5. The visibility v along a refracted light direction, e.g., does the refracted light ray reach the point P ?

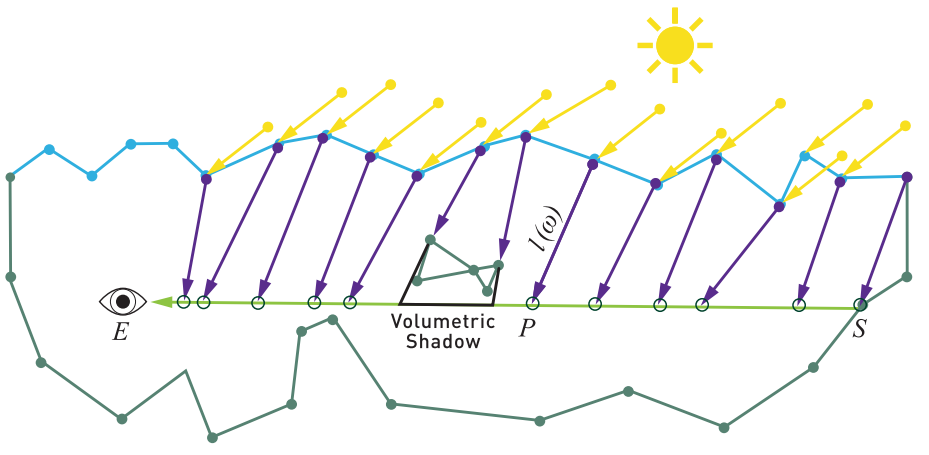


Figure 14-4. The eye E on the left looks to the right through the water. Light from above reaches various different locations along this ray, depending on the water's surface, and scatters light toward the eye.

There are two possible approximate solutions to computing the integral over all in-scattering events:

1. Use a 3D grid to accumulate discretized in-scattering events at the center of each grid cell.

A grid with a high enough resolution needs to be used to prevent leaking of volumetric light through thin scene features.

- (a) Trace enough refracted rays from their origin on the water surface to the intersection point with the underwater scene.
 - i. At each grid cell that a refracted ray enters, compute the point P on the ray that is closest to the center of the grid cell.
 - ii. Compute the phase function and the transmitted radiance that reaches the eye from this point P .
 - iii. Accumulate the transmitted radiance in the grid cell.
 - (b) For each pixel on the screen, trace a ray from the pixel to the eye. Traverse the grid on this ray and accumulate the light that reaches the eyes.
2. Create a sufficiently dense set of triangular beam volumes [2] to approximate the in-scattering integral using the graphics pipeline and additive blending.

As shown in Figure 14-5, refracted light directions can cause a triangular beam to form a non-convex volume. The algorithm proposed in Section 14.3 tries to prevent this case by using high tessellation levels in regions where the directions of refracted rays change quickly and can thus create non-convex volumes.

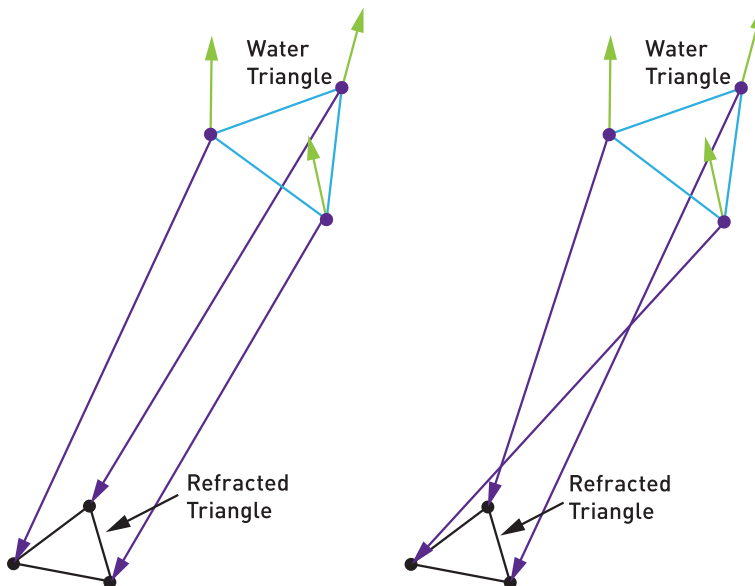


Figure 14-5. Left: the refracted triangle forms a convex volume with the water triangle. Right: the volume formed is twisted and no longer convex.

For each triangular beam, the graphics pipeline is used to render eight triangles that form the exact convex bounding volume of the beam. These triangles are generated so that their surface normals always point out of the volume.

Along a ray from the eye, the direction of refracted light changes from where the ray hits the backfacing triangles of the point to where it hits the frontfacing triangles of the volume. As a result, it is not possible to use additive blending, a positive in-scattering term at the backfacing triangles, and a negative in-scattering term at the frontfacing triangles as proposed by Golia and Jensen [3].

It is possible though, using enough small volumes, to approximate the in-scattering integral by just accumulating the in-scattering terms at the frontfacing triangles of each volume.

The demo that accompanies this chapter uses additive blending, tessellation, and a geometry shader to implement a volume slicing method that is inspired by the second approach. This is reflected in the following algorithm overview.

14.3 ALGORITHM

The following seven steps are used in the demo to render volumetric water caustics. Figure 14-6 shows an overview of these steps.

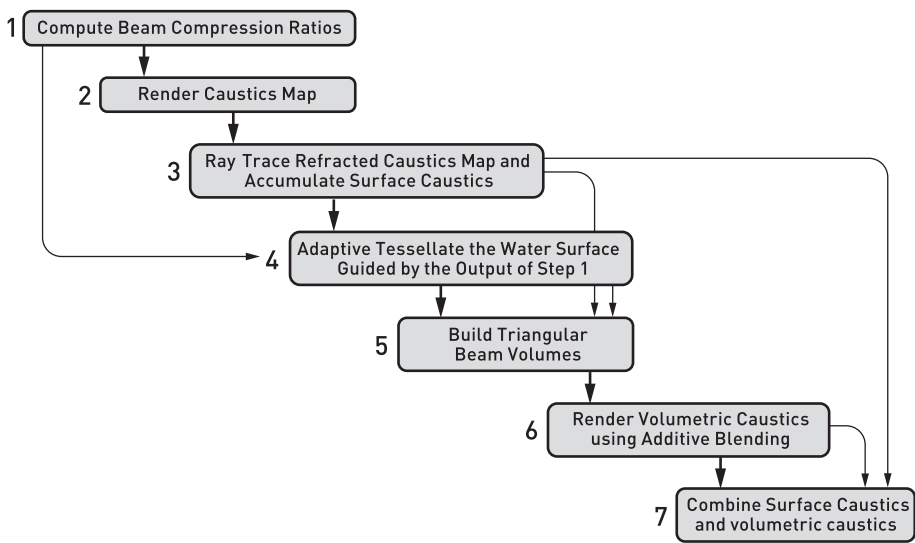


Figure 14-6. Algorithm overview.

Please note that, instead of tracing rays along the directions of refracted light rays, it is also possible to trace rays along the direction of the light rays that get reflected by the water surface and thus render reflected volumetric and surface caustics. The demo that accompanies this chapter also implements reflected surface caustics in addition to refracted volumetric and surface caustics.

14.3.1 COMPUTE BEAM COMPRESSION RATIOS

For each vertex of the water mesh that represents the geometry of the simulated water surface, a refracted ray R is constructed. This ray starts at the current position of the water vertex and points along the refracted direction of incident light.

The refracted water mesh has the same number of vertices and the same triangle count as the water surface. The positions of its vertices are computed by intersecting each ray R with the underwater geometry. Figure 14-7 depicts this process. Every blue water surface triangle generates a purple dashed triangle in the refracted water mesh.

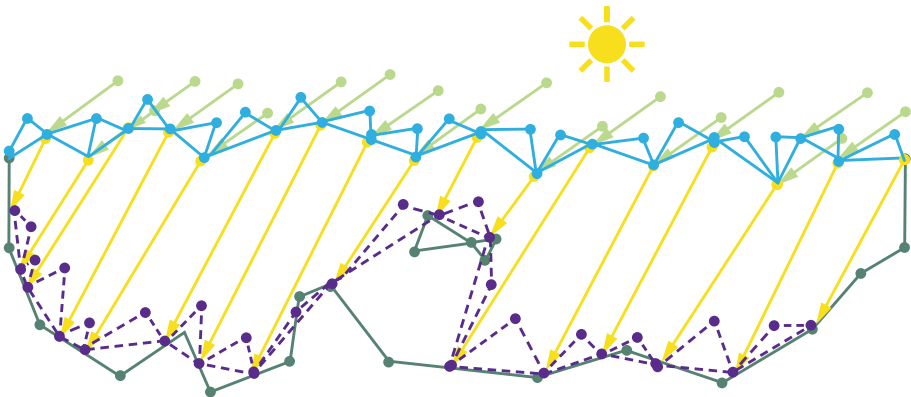


Figure 14-7. *Computing a refracted water mesh.*

Please note that the refracted water mesh does not need to be fine enough to follow every detail of the underwater geometry. It only needs to be detailed enough to facilitate the computation of a high-enough-quality compression ratio, as described below. This step can introduce an error when the water surface is not detailed enough. It is therefore necessary to refine the water surface if errors are detected.

As Figure 14-8 shows, the refraction of the light rays can either focus the light within a triangular beam or do the opposite. As a result, triangles in the refracted water mesh can have either a larger or a smaller area than their respective water triangles.

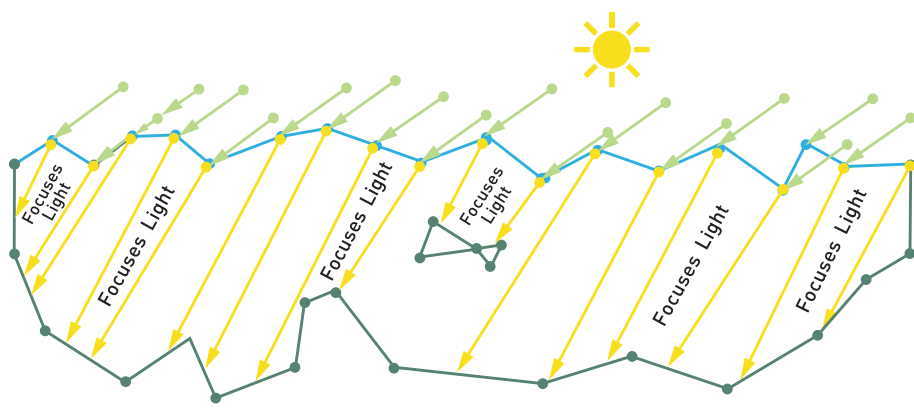


Figure 14-8. How light can focus in a refracted water mesh.

For each triangle the beam compression ratio r is computed and stored in a buffer:

$$r = \frac{a(T_w)}{a(T_r)}, \quad (2)$$

where $a()$ computes the area of a triangle, T_w is the water surface triangle, and T_r is the refracted triangle.

The original water triangles and the refracted water triangles form coarse triangular beams as shown in Figure 14-5. The compression ratio can also be thought of as a value that describes the likelihood of a triangular beam forming a non-convex volume. Consequently, the compression ratio can be used to drive the tessellation density for subdividing each coarse triangular beam into smaller beams. The idea to use the compression ratio from Equation 2 is not new and has been described in the past [3].

14.3.2 RENDER CAUSTICS MAP

In this step, two render targets are initially cleared to indicate invalid surface positions and surface normals.

Next, all water triangles are rendered with a pixel shader that writes the following values to two render targets:

1. The 3D position of the water surface.
2. The surface normal at this point of the water surface.

This is shown in Figure 14-9.

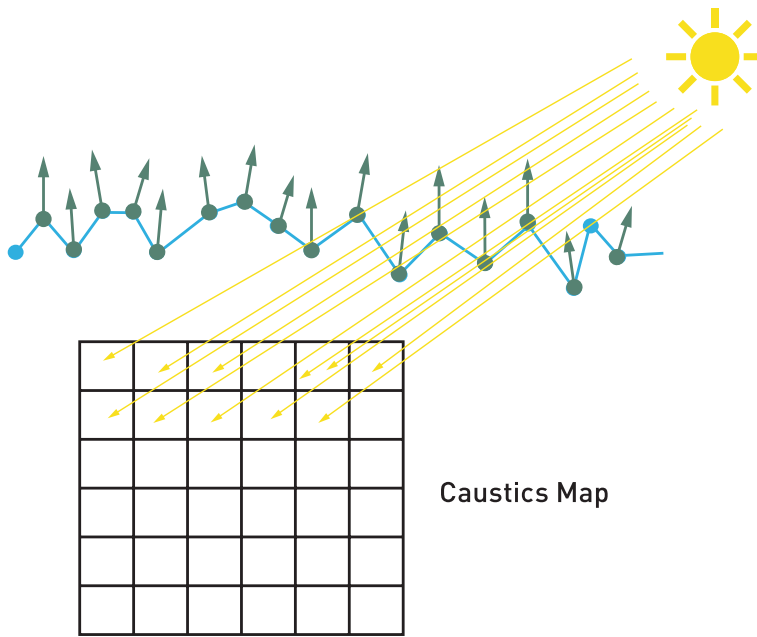


Figure 14-9. The water mesh is rendered to a caustics map as seen from the point of view of the light—the pixels of the resulting surface carry the position of the water surface and the normal of the water surface in this pixel.

14.3.3 RAY TRACE REFRACTED CAUSTICS MAP AND ACCUMULATE SURFACE CAUSTICS

This step uses DXR to trace rays for valid pixels of the caustics map rendered in step 2. The intersections with the scene are stored in a refracted caustics map. Also, the intersection positions are transformed to screen space and are used for accumulation of scattered surface caustics:

1. Trace a ray for each pixel (x, y) in the caustics map that represents a valid point on the water surface.
2. Compute the intersection of the ray with the underwater scene geometry. It is possible to cull this ray if, for example, a shadow map test reveals that the point on the water surface is shadowed by geometry above the water line.
3. Write the position of the intersection into pixel (x, y) in the refracted caustics map. See Figure 14-10.

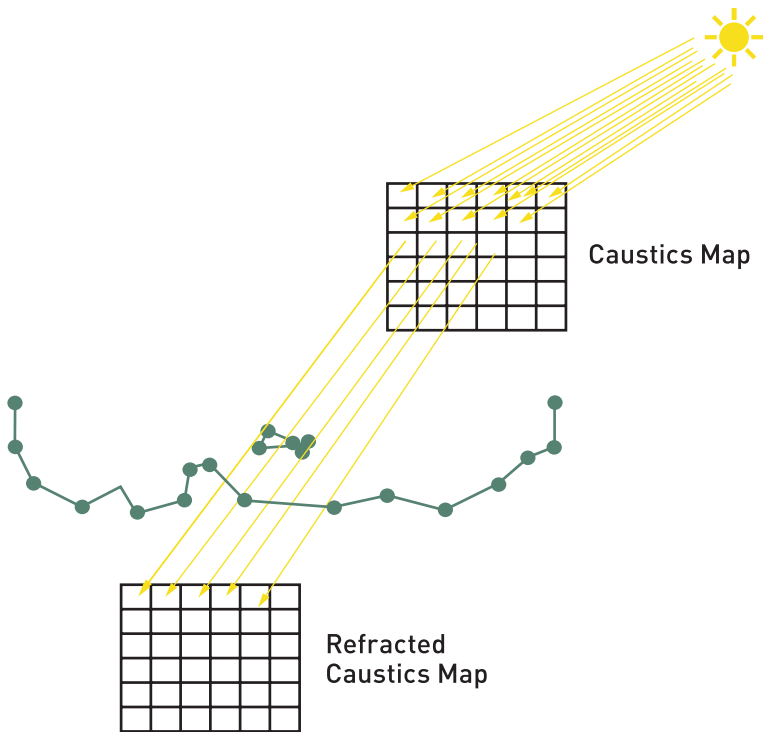


Figure 14-10. Ray tracing a refracted caustics map: send rays from the water surface positions stored in the caustics map along the refracted light directions, and store the resulting ray/scene intersections in a refracted caustics map.

4. Optionally, trace secondary rays along the reflected direction (along the scene normal) of the refracted caustics rays, and write the resulting intersection into a one-bounce caustics map at pixel (x, y) . See Figure 14-11.

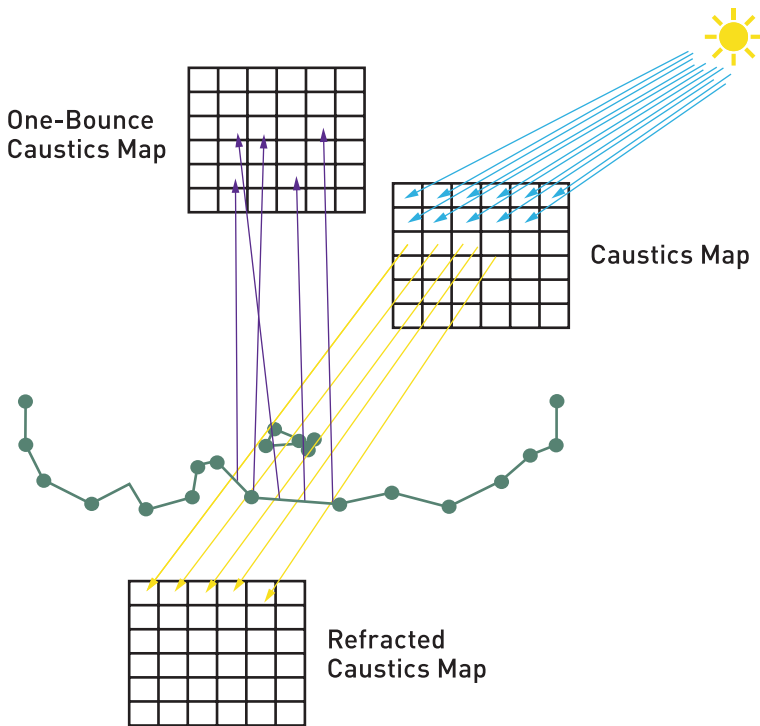


Figure 14-11. Tracing rays along the reflected direction of the caustics ray for another bounce of light, creating a one-bounce caustics map.

5. Accumulate surface caustics in an offscreen buffer.
 - (a) Project the intersection points (including the points from the optional step 4) to screen space—if the position is on the screen, use `InterlockedAdd()` to accumulate radiance in that screen location in a buffer.

To find out if the intersection corresponds to the frontmost pixel on the screen, the simplest solution is to do a depth test with a certain tolerance. Other possibilities are to also consider the G-buffer normal of the onscreen pixel and/or scale the brightness value by a function of the difference in depth. It is also possible to render a unique triangle ID into the G-buffer and to compare this ID with the primitive and instance IDs that are available in the DXR hit shaders.
 - (b) The radiance value that gets accumulated can be scaled by several factors, including the compression ratio from step 2 and/or the amount of light that has been absorbed by the distance that the ray travels through the water [1].

14.3.4 ADAPTIVELY TESSELLATE THE TRIANGLES OF THE WATER SURFACE

See Figure 14-12 for a depiction of an adaptive tessellation of a triangular beam volume.

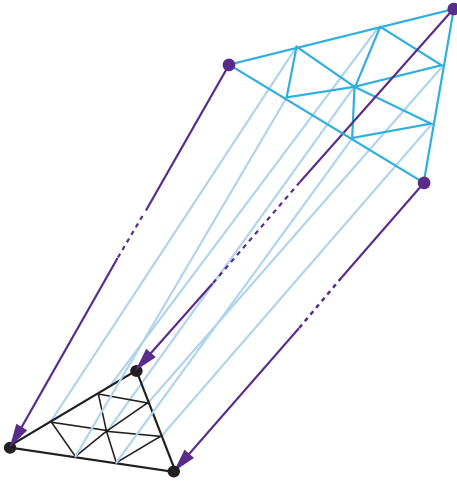


Figure 14-12. *Adaptively tessellated water triangles result in tessellated triangular beams—see step 5.*

The beam compression ratio (see Equation 2) is used to compute a tessellation factor for the water triangle that sits at the top of the triangular beam. This tessellation factor is scaled to:

1. Provide enough slices to approximate the in-scattering integral well enough.
2. Prevent the triangular beam from turning non-convex. See Figure 14-5.
3. Make sure that no volumetric light leaks through small scene features.

14.3.5 BUILD TRIANGULAR BEAM VOLUMES

Run a geometry shader to pick up the tessellated water triangles and build the triangulated hull of the corresponding triangular beam.

1. Project the 3D vertices of the incoming triangle to the (refracted) caustics map space.
2. Read the 3D positions of the triangle that forms the top cap of the volume from the caustics map.

3. Read the 3D positions of the triangle that forms the bottom cap of the volume from the refracted caustics map.
4. Build the eight triangles that form the bounding volume. See Figure 14-13. Optionally, do the same for volumes created by the refracted caustics map and the one-bounce caustics map.

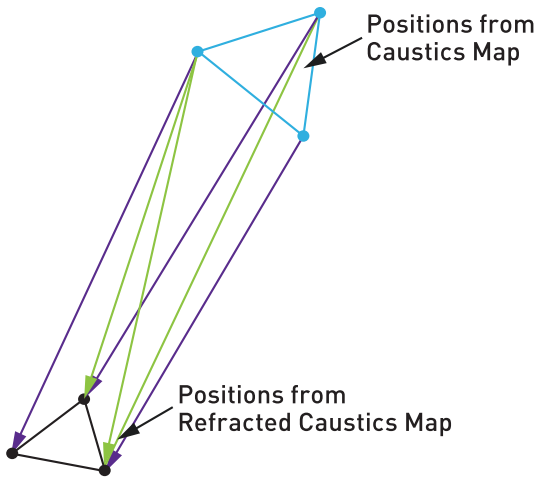


Figure 14-13. *Triangles forming a triangular beam.*

5. Compute an estimated thickness of the triangular beam at each output vertex—this way, interpolated thickness is passed to the vertex shader.
6. Compute a ray direction at every output vertex—this way, the interpolated direction is passed to the pixel shader.

14.3.6 RENDER VOLUMETRIC CAUSTICS USING ADDITIVE BLENDING

Additively blend the in-scattered light on the pixels of the frontfacing sides of each volume to a render target in the pixel shader.

1. Compute the phase function at the current 3D position given the interpolated ray direction.
2. Multiply the resulting in-scattered term by the interpolated thickness.
3. Output the result.

14.3.7 COMBINE SURFACE CAUSTICS AND VOLUMETRIC CAUSTICS

This step combines the image of the scene that has been lit by the surface caustics and a blurred version of the volumetric caustics that has been rendered using additive blending.

1. Blur/denoise the surface caustics from step 3.
2. Use the denoised surface caustics buffer to shed light on the scene, e.g., multiply it by the albedo texture of the G-buffer pixel and add it to the unlit result to produce a lit G-buffer.
3. Blur the result from step 6 slightly and add it to the lit G-buffer.

14.4 IMPLEMENTATION DETAILS

As described in Section 14.1, the DirectX 12 DXR API is used to implement all ray tracing workloads. For step 1, `DispatchRays()` is called so that each thread traces exactly one refracted ray into the scene. The resulting refracted water mesh is written to a buffer that is read by later steps and uses the same index buffer as the original water mesh.

Step 2 is implemented as a normal rasterization pass. For step 3, `DispatchRays()` is called to cast a ray for every valid pixel of the caustics map from step 2. Optionally, the shader casts additional rays along the reflected direction for surface caustics that are generated by light rays that get reflected by the water surface or the one-bounce caustics map. Accumulation of refracted/reflected light happens in a half-resolution buffer to facilitate fast denoising.

If an additional bounce of caustics is selected, yet another ray is cast in step 2 to simulate the reflection of caustics rays by the scene. The resulting intersections of these reflected rays are used to simulate indirect lighting through surface caustics and are written to another caustics map, the reflected caustics map—the buffer is sized to facilitate drawing volumetric beams for this additional bounce.

Volumetric caustics are accumulated in step 6 in a half-resolution buffer to speed up the drawing of the triangular beams. The geometry shader in step 5 creates triangular beams for the primary refracted caustics as well as for the optional additional bounce recorded in the one-bounce caustics map.

Denoising of the surface caustics buffer in step 7 is done through a set of iterated cross-bilateral blurring steps that account for differences in view-space depth, normals, and positions. Finally, surface caustics and volumetric caustics are upsampled bilaterally and get combined with the rendered scene.

14.5 RESULTS

Table 14-1 shows caustics workload timings taken in a scene for four different camera positions and light setups on an NVIDIA RTX 2080 Ti board running caustics workloads at a resolution of 1920×1080 using the official DXR API that is part of DirectX 12.

Screenshots from these four scenes are shown in Figure 14-14. All scenes run at interactive frame rates in excess of 60 FPS while casting rays from the pixels of a 2048×2048 caustics map. The timings from Table 14-1 indicate that volumetric caustics operate, in most cases, within a time span that is acceptable for integration in a modern computer game. In comparison, the work from Liktor and Dachsbacher [6] was not able to reach a performance level that made integration into games feasible.

- > The top left screenshot in Figure 14-14 shows a view from above the water line. In this screenshot refracted volumetric underwater caustics and reflected caustics that are visible above the water line are generated by the algorithm described in this chapter. The caustics workloads for this image amount to a total time of 2.9 ms.
- > The top right screenshot in Figure 14-14 shows a view from below the water line. For this scene refracted volumetric underwater caustics and a secondary volumetric bounce of light are rendered. For this scenario the volumetric bounce and high maximum tessellation factor preset cause the timing for the volumetric part of the caustics rendering to climb to 4.6 ms. These settings are currently too expensive to be used inside a game.
- > The bottom left screenshot in Figure 14-14 shows again a view from below the water line. For this scene again refracted volumetric underwater caustics and a secondary volumetric bounce of light are rendered. For this scenario, the second volumetric bounce along with a moderately high maximum tessellation factor preset cause the timing for the volumetric part of the caustics rendering to climb to a more moderate 2.1 ms. These settings are probably acceptable within a game that focuses on high-quality volumetric caustics.
- > The bottom right screenshot in Figure 14-14 shows another view from below the water line. For this scenario the second volumetric bounce along with a moderately high maximum tessellation factor preset cause the timing for the volumetric part of the caustics rendering to take only 1.4 ms. Please note how the second bounce of light casts light onto the downward-facing part of the character.

Table 14-1. Timings. All DispatchRays() include accumulative scattering.

Timings		
Screenshot	Workload	Time in ms
Top Left	Refractive + Reflective Caustics	
	DispatchRays()	0.9
	Surface Caustics Denoising	0.8
	Volumetric Slicing and Upscaling	1.2
Top Right	Refractive + Reflective Caustics + One-Bounce	
	DispatchRays()	3.0
	Surface Caustics Denoising	0.8
	Volumetric Slicing and Upscaling	4.6
Bottom Left	Refractive + Reflective Caustics + One-Bounce	
	DispatchRays()	0.9
	Surface Caustics Denoising	0.8
	Volumetric Slicing and Upscaling	2.1
Bottom Right	Refractive + Reflective Caustics + One-Bounce	
	DispatchRays()	2.1
	Surface Caustics Denoising	0.8
	Volumetric Slicing and Upscaling	1.4



Figure 14-14. Screenshots.

14.6 FUTURE WORK

In the current demo implementation, the caustics map and the refracted caustics map need to have a resolution that is high enough to capture the underwater geometry in enough detail. It would be interesting to investigate how ideas from Wyman and Nichols [10] or Liktor and Dachsbacher [6] could be used to adaptively cast rays.

Further on, instead of using the rasterization pipeline to slice the parts of the water volume that concentrate light, it could be faster to accumulate in-scattered light in a volumetric texture. For a position on a ray to the eye, the information stored in the caustics map and the refracted caustics map could be used to prevent volumetric light leaking through thin features of a scene.

14.7 DEMO

A demo that can be run on NVIDIA GPUs showcasing the proposed technique is provided in the code repository.

REFERENCES

- [1] Baboud, L., and Décoret, X. Realistic Water Volumes in Real-Time. In *Eurographics Conference on Natural Phenomena* (2006), pp. 25–32.
- [2] Ernst, M., Akenine-Möller, T., and Jensen, H. W. Interactive Rendering of Caustics Using Interpolated Warped Volumes. In *Graphics Interface* (2005), pp. 87–96.
- [3] Golias, R., and Jensen, L. S. Deep Water Animation and Rendering. https://www.gamasutra.com/view/feature/131445/deep_water_animation_and_rendering.php, 2001.
- [4] Hoobler, N. Fast, Flexible, Physically-Based Volumetric Light Scattering. https://developer.nvidia.com/sites/default/files/akamai/gameworks/downloads/papers/NVVL/Fast_Flexible_Physically-Based_Volumetric_Light_Scattering.pdf, 2016.
- [5] Hu, W., Dong, Z., Ihrke, I., Grosch, T., Yuan, G., and Seidel, H.-P. Interactive Volume Caustics in Single-Scattering Media. In *Symposium on Interactive 3D Graphics and Games* (2010), pp. 109–117.
- [6] Liktor, G., and Dachsbacher, C. Real-Time Volume Caustics with Adaptive Beam Tracing. In *Symposium on Interactive 3D Graphics and Games* (2011), pp. 47–54.

- [7] Shah, M. A., Konttinen, J., and Pattanaik, S. Caustics Mapping: An Image-Space Technique for Real-Time Caustics. *IEEE Transactions on Visualization and Computer Graphics* 13, 2 (March 2007), 272–280.
- [8] Szirmay-Kalos, L., Aszódi, B., Lazányi, I., and Premecz, M. Approximate Ray-Tracing on the GPU with Distance Impostors. *Computer Graphics Forum* 24, 3 (2005), 695–704.
- [9] Wang, R., Wang, R., Zhou, K., Pan, M., and Bao, H. An Efficient GPU-based Approach for Interactive Global Illumination. *ACM Transactions on Graphics* 28, 3 (July 2009), 91:1–91:8.
- [10] Wyman, C., and Nichols, G. Adaptive Caustic Maps Using Deferred Shading. *Computer Graphics Forum* 28, 2 (Apr. 2009), 309–318.



Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (<http://creativecommons.org/licenses/by-nc-nd/4.0/>), which permits any noncommercial use, sharing, distribution and

reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if you modified the licensed material. You do not have permission under this license to share adapted material derived from this chapter or parts of it.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.