Real-Time Rendering of Approximate Caustics under Environment Illumination

Abstract

We present a real-time GPU caustics rendering technique for dynamic scenes under environment illumination taking into account light occlusion. The dynamic scenes consist of caustic objects (reflective and/or refractive objects which produce caustics) and receiver objects (non-reflective and non-refractive), that can be translated and rotated. As the light source, we consider environment illumination (distant lights from all directions) which we approximate as a set of important directional lights. Our rendering technique is able to generate approximate caustics cast on receiver objects as well as volumetric caustics. As the preprocessing, we precompute the caustic patterns of caustic objects for several directional lights and store them in caustic images. During the rendering, we interpolate the precomputed caustic patterns based on the important directional lights which approximate the given environment illumination. The important directional lights are obtained by using our proposed environment cube map segmentation technique. Our proposed technique is able to generate real-time caustics which are visually similar to the caustic generated by using a commercial renderer mental ray.

Keywords: Caustics, Real-Time Rendering, Environment Illumination, GPU

1. Introduction

Real-time photo-realistic rendering is a major goal in computer graphics and entertainment as it enables the audience to experience and immerse into a virtual world as if it is the real world. However, it is computationally expensive to generate photo-realistic images of 3D scenes, especially dynamic 3D scenes (whose objects can be transformed such as by translation or rotation) under environment illumination. Generally, the solution involves the integration of the contributions from all light directions in the environment, with
the computation for each light direction needs to take into account several factors, such as reflections and refractions.

One of the important effects in photo-realistic rendering is caustics which are produced by reflective and/or refractive objects (caustic objects). Due to the reflective and/or refractive properties of the caustic objects, light arriving on the caustic objects are converged on some locations on surfaces hence producing caustic patterns. Generating caustic effects, however, is also computationally expensive. Thus, it is a challenge to generate real-time caustics in dynamic scenes under environment illumination. There are many researches in generating caustics in computer graphics. However, most of them generate caustic under a single light source, and some of them are not in real-time.

Figure 1: Caustics rendering using our proposed technique. First column shows caustics under one directional light source (light direction indicated by the arrow). Second column shows caustics under environment illumination. The first row shows only the cast caustics, and the second row shows cast caustics and volumetric caustics.
In this paper, we present a real-time caustics and volumetric caustics rendering technique under environment illumination taking into account light occlusion. Figure 1 shows some examples of our caustics rendering results. In our proposed rendering technique, we firstly precompute the reflective and/or refractive caustic patterns at the surrounding of caustic objects based on a set of directional lights. Afterward, we use the precomputed caustic patterns in the rendering pass in order to efficiently compute the caustic intensities at arbitrary locations. Our proposed rendering technique has some differences with the technique presented by Wyman et al. [1] and we discuss these in Sections 2 and 3. In order to achieve real-time performance for caustics rendering under environment illumination, we approximate the environment illumination as a set of directional lights using our proposed environment cube map segmentation technique (a cube map is a set of six textures, with each texture corresponds to one cube face). We show the main steps diagram of our technique in Figure 2.

To evaluate the rendering quality of our result, we compare the images generated using our technique with the images generated using mental ray (available in Maya 2010). Mental ray is a famous renderer used by movie industry to generate visual effects in some well-known movies with outstanding visual quality. As we show in Section 7.2, our technique can generate caustics which are visually similar to the caustics generated using mental ray.

The organization of this paper is as follows. In Section 2, we describe the related work. Section 3 explains the precomputation of caustic patterns for a set of directional lights. Section 4 describes how our technique uses the precomputed caustic patterns in the rendering under one and multiple directional lights. Section 5 explains how we determine light directions to represent an environment illumination and how we render approximate caustics under environment illumination. In Section 6, we present the GPU implementation of our techniques. We show the results in Section 7 and draw conclusions and future work in Section 8.

2. Related Work

There are several work in caustics rendering and they can be classified into three categories: offline (non real-time) caustic rendering, real-time caustics rendering of single reflection and/or refraction, real-time caustics rendering of multiple reflections and/or refractions.
Offline/non real-time caustics In general, photon mapping \cite{2} is the most commonly used technique in offline (non real-time) caustics rendering. In photon mapping \cite{2}, photons (packets of light energy) are shot to the scene and stored in a photon map. The photon map is used during rendering by gathering the photons around a visible point in order to estimate the caustic intensity. However, photon mapping is computationally expensive due to two reasons. During the photon shooting, it is necessary to search which triangle (assume the 3D objects are modelled as a set of triangles) is intersected by the light or photon. Hence, similar to ray tracing, it is costly to do the brute force search on all triangles in the 3D scene. To accelerate this, the triangles are arranged into an acceleration structure such as KD-tree. As for the second reason, to perform photon gathering, we need to search the local regions in the photon map for the photons. To accelerate the photon mapping, Günther et al. \cite{3} use a computer cluster and they are able to
achieve interactive rates whilst Purcell et al. [4] use a GPU and they are able to reduce the rendering time to few seconds. Zhou et al. [5] propose a real-time GPU-based kd-tree generation technique which greatly aids the photon mapping process. However, the overall rendering speed is mostly below 10 frames per second.

Ihrke et al. [6] subdivide a scene into a regular 3D grid (set of voxels). They trace a light path and process each voxel passed by the light. By using this technique, they can support caustic objects whose interiors may contain different material properties (non-homogeneous), i.e. the voxels have different index of refractions. As a result, they can simulate light path bending inside the caustic object. Sun et al. [7] compute the multi-resolution 3D grid by grouping local voxels with similar index of refraction into a larger voxel. The traversal speed is accelerated since the light may pass through a large voxel (or distance) in one step.

Real-time caustics of single reflection and/or refraction In this category, most of the generated caustics are underwater caustics [8, 9, 10] and they require computations of only a single reflection and/or refraction. This approach maps pretty well to the GPU since we can determine the first intersection with the caustic object easily by rendering the caustic object, and then we compute the refraction on each pixel in parallel by using fragment shader. In rendering underwater caustics, they mostly approximate the input water surface by subdividing the water surface into a regular grid with each cell (which is implemented as a pixel input to the fragment shader) contains the height of the water surface (height map).

Trendall and Stewart [8] simulate the water surface movement by using a wave propagation formula and the height at each point on the water surface is stored in the water surface height map. They compute light intersection and refraction with the water surface by using the fragment shader. For the second intersection (with the underwater surface), they can compute it analytically since the underwater surface is assumed to be a flat surface. Finally, they compute the caustic intensities at each point on the underwater surface by summing up the contribution of the light incident at that particular point. Later, Baboud and Decoret [9] improve the work by enabling the underwater surface to have arbitrary bumpy shapes. Hence, they use an additional height map for the underwater surface and they compute the second intersection (with the underwater surface) by using the binary search. These two works basically use a forward approach (compute light ray (or photon) path from the light source, to water surface, and finally to the underwater
Yuksel and Keyser [10] present an interesting backward approach by starting the computation from underwater surface to the water surface and finally to the light source. Hence, to compute the caustic intensity at each point on the underwater surface, they shoot several light rays/photons to several nearby regions on the water surface, compute the inverse refraction (i.e. refraction from the water medium to the air medium), and finally check if the refracted light arrives at the light source (if they do, then increase the intensity of the original point on the underwater surface). In both of these approaches, the quality of the generated caustics depends on the resolution of the height map. If the resolution of the height map is too low, then the generated caustics appear to be noisy due to undersampling.

To solve the undersampling issue, several caustic rendering techniques based on sweeping are proposed [11, 12]. For every vertex on the face of a caustic object, Ernst at al. compute the refraction or reflection direction [11]. Afterward, they sweep the face in the direction of the reflection or refraction, thus forming caustic volumes. Since the reflected or refracted light directions of the vertices might be incoherent (i.e. have different directions), the generated caustic volumes might be skewed or warped. The caustic patterns on diffuse (non-shiny or non-transparent) surface are finally formed based on the intersection between the caustic volumes and the diffuse surface. Later, Liktor and Dachsbacher [12] improve the rendering performance by implementing the sweeping process by using geometry shader.

Real-time caustics of multiple reflections and/or refractions In this category, they are able to generate real-time caustics caused by multiple reflections and/or refractions. However, computing accurate multiple reflections and/or refractions are difficult to be done in GPU since we have to do ray tracing after the first intersection in order to compute the second intersection. Hence, mostly there are two approaches in solving this problem. In the first approach, the intersection computation is approximated [13, 14, 15, 16]. In the second approach, some information necessary for real-time caustics rendering especially the light paths is precomputed [1, 17, 18].

To handle multiple refractions in the first approach, Wyman and Davis generate four temporary images (by rendering the caustic object from the light source) with each contains different information [13]: an image whose pixels containing normals of the caustic object surfaces facing the camera (and another image for the surfaces facing away from the camera), camera-surface distances of the caustic object surfaces facing the camera (and an-
other image for the surfaces facing away from the camera) which are basically height maps. They approximate the intersection by interpolating the distance between the pixels of the two camera-to-surface distance images. To compute more accurate second intersection, Liu et al. use the binary search algorithm [14] and Shah et al. use Newton-Raphson method [16]. Hu and Qin take into account the location of the second intersection points if the intersection points are on the surfaces facing the camera or facing away from the camera [15] (in the previous work, the intersection points are assumed on the surfaces facing away from the camera). Hence, they can generate more accurate refraction.

In the precomputation approach, light paths of a caustic object are precomputed and they are used in real-time rendering. The main drawback is that it is limited to the non-deformable caustic objects. Iwasaki et al. [17] precompute a light transport table which contains information about the direction and the location (on the caustic object surface) where the refracted light finally leaves the caustic object for every possible entry direction and entry point (on the caustic object). They assume that there are no objects which block the light that re-enters the caustic object (since they only store information about the first entry point and the final exiting point). Liu et al. proposed a more efficient precomputation by approximating the bundle of light paths as a set of loci called Caustic Spot Lights (CSLs) [18]. Due to high compression rate, some differences compared to a more accurate offline rendering can be observed. Wyman et al. [1] precompute the local caustics of a caustic object on uniform grids or concentric spheres (with constant radii differences between the consecutive spheres) enclosing the caustic object. They use a CPU cluster for both precomputation and rendering. In the rendering, the scene is illuminated by point or directional light sources, not environment illumination.

The existing techniques in the last category, however, mostly only support caustics under a single light source and they do not support real-time caustics rendering under environment illumination. On the other hand, our technique renders real-time caustics under environment illumination using a GPU. In contrast to the first two categories, our proposed technique can also generate caustics of multiple reflections and or refractions in real-time.

3. Precomputing Caustic Patterns
The purpose of the precomputation is to record the caustic intensities of a caustic object at the 3D local region based on several predefined directional lights. In this precomputation, caustic patterns of only several predefined directional lights are precomputed since it is impossible to record the caustic patterns of all possible directional lights. Hence, in the rendering, given an input directional light, its caustic patterns are computed by interpolating the precomputed caustic patterns of several predefined directional lights. The caustic intensities of each predefined directional light are recorded on a set of concentric spheres with varying radii (caustic spheres). We use sphere representation in storing the precomputed caustic patterns since we record caustic intensities of all directions surrounding the caustic object. Moreover, during the precomputation we can easily compute the intersections between the reflected and/or refracted light with the caustic spheres by using ray-sphere intersection computation. Caustic intensities are recorded on several concentric spheres with varying radii in order to cover the 3D space surrounding the caustic object and we precompute the caustic intensities by using photon mapping [2].

We propose the following precomputation. We sample 26 light directions based on a cube, which consist of the directions from the cube vertices (8 directions), the center of the cube edges (12 directions), and the center of the cube faces (6 directions) to the center of the cube. The predefined light directions are illustrated in Figure 3(a). Then, the reflective and the refractive caustic patterns (for each directional light) on a set of concentric spheres centered at the origin of the caustic object are computed and recorded as shown in Figure 3(b).

Unlike Wyman et al. [1] who linearly change the radii of the spheres, we suggest a quadratic function to determine the radii of the caustic spheres since caustics weaken non-linearly due to light attenuation. Thus, the changes in caustic patterns are visually more apparent near the caustic object due to the higher priority is given to the recording of caustic patterns near the object (i.e. densely sample near to the caustic object and sparsely sample far from the caustic object). As a result, the number of caustic spheres can be reduced while maintaining the visual quality. The formula for computing caustic radii is shown in Equation 1.

\[
r_i = r_{\text{min}} + (r_{\text{max}} - r_{\text{min}})((i - 1)/(s - 1))^2,
\]

where \(r_i\) is the radius of the \(i\)-th caustic sphere, \(r_{\text{min}}\) is the minimum radius, \(r_{\text{max}}\) is the maximum radius, and \(s\) is the number of caustic spheres. We set
Figure 3: (a) Predefined light directions for the precomputation are computed based on the directions from each vertex, center of each edge, and center of each face to the cube center. We show some example directions by using arrows. (b) Precomputation of caustic patterns. The incoming directional light (implemented as photons) is reflected and refracted by the caustic object.

The caustic patterns of each caustic sphere are stored in a 2D image such that the x and y axes of the image represent the longitude and latitude of the caustic sphere. The issues of storing the caustic patterns in this format are the redundant storage near the poles and the requirement of using trigonometric functions to sample the caustics during the rendering. However, the advantage is caustic spheres in latitude-longitude format can be stacked into a 3D texture (which contains several layers of 2D textures) and therefore we can efficiently sample the caustics at any point on and between the caustic spheres (or layers of a 3D texture) since GPUs can efficiently sample and interpolate texels with a single sampling instruction. Moreover, the computation of trigonometric functions in current GPUs are relatively cheap. Another option is to store the caustic patterns in other format such as cube map. However, more additional operations are needed to sample and interpolate caustic points across cube map stack or array. Moreover, not all of the current GPUs support cube map arrays.

$r_{\text{min}}$ to be slightly greater than the distance from the center of the caustic object to the nearest surface of the caustic object. Assuming the maximum dimension of the object’s bounding box $d$ is max{width, height, depth}, we set $r_{\text{max}}$ to $4d$ based on our experiment in which caustic patterns were barely noticeable on the caustic sphere with the radius of $4d$. 
From experimental results, the resolution of the caustic pattern images should be at least $512 \times 256$ in order to generate good quality caustics. The height of the caustic images is half of the width since we sample the caustic sphere $[0..2\pi]$ in longitude direction and $[0..\pi]$ in latitude direction. Generally, we suggest 16 caustic spheres ($s = 16$), thus the total memory needed to store the precomputed caustic patterns of one caustic object is $512 \times 256$ (dimension of caustic pattern image) $\times$ 3 color channels (RGB) $\times$ 16 caustic spheres $\times$ 26 light directions $= 156$ MBytes. We can precompute polychromatic caustics by setting different refractive indices for the red, green, and blue components of the directional light. We show some examples of caustic pattern images in Figure 4.

![Figure 4: Caustic patterns for gargoyle model (with refractive indices for red, green and blue components are 1.5, 1.51 and 1.52, respectively) for two different light directions.](image)

The precomputation algorithm is shown in Algorithm 1.

**Algorithm 1:** Precomputation of caustic patterns.

1. Sample predefined light directions;
2. **foreach** predefined light direction **do**
   3. Compute caustic patterns on each caustic spheres by using photon mapping \[^2\];
   4. Save the precomputed caustic patterns of each caustic sphere as an image;

4. Rendering Caustics under Directional Lights

4.1. **Single Directional Light**

Given an arbitrary light direction $L$, caustic spheres $C$ of $L$ can be computed by interpolating the precomputed caustic patterns since most likely $L$ does not coincide with either of the precomputed 26 light directions. Thus,
in our approach, we propose to choose the four precomputed light directions nearest to \( L \). Similar to Wyman et al. \cite{1}, in order to alleviate the ghosting effect (transition between two patterns, with one pattern fades out and another pattern fades in) due to the interpolation, caustic spheres of the four nearest light directions are rotated to align their directions with \( L \). Then, we blend them using bilinear interpolation to produce the caustic spheres \( C \) of \( L \). This interpolation scheme is illustrated in Figure 5.

Figure 5: Caustic spheres interpolation. For the sake of clarity, we illustrate the process in 2D using only two caustic spheres.

1. Incoming light direction in the rendering.
2. Predefined light directions and their precomputed caustic patterns (on caustic spheres).
3. a. Rotate the caustic spheres of each predefined light direction such that the predefined light directions coincide with the incoming light direction.
   b. Use bilinear interpolation to blend caustic spheres of the four nearest light directions.

The caustic intensity at a point on a surface is trilinearly interpolated (point A in Figure 6) or extrapolated (point B in Figure 6) using the intensities at the eight nearest samples in the caustic spheres \( C \) of \( L \).

Using the above method for computing the caustic intensity at an arbitrary point, we propose the following techniques to generate two types of caustics, caustics cast on receiver objects and volumetric caustics.

**Caustics cast on receiver objects** To render these caustics, firstly a temporary cube map which stores the information of receiver points (coordinate and depth value of each receiver point) is computed. The receiver point information can be computed by rendering the surrounding scene from the center of the caustic object. Caustic intensities at the receiver points (Figure 6) can then be computed based on the coordinate of the receiver points (sampled from the temporary cube map) by sampling caustic spheres \( C \) of \( L \). The caustic intensities are then multiplied with the dot product between the normals at the receiver points and the directions to the center of the caustic object (in order to take into account the cosine term in illum-
nation) and we store them in a caustic cube map. Every pixel of the caustic cube map contains information about caustic intensities in RGB channel and the depth value (is copied directly from the temporary cube map) in the A (alpha) channel of the cube map. In our experiment, the resolution of the caustic cube map is $512 \times 512 \times 6$.

The caustic cube map is then sampled in order to determine the caustics at visible points from the camera. For every visible point, we compute its local coordinates with respect to the caustic cube map and sample the corresponding entry in the caustic cube map. If the depth of the visible point is the same as the depth value stored in the cube map, we add the caustic intensity, otherwise we do not add.

**Volumetric caustics**  In the presence of participating media (assumed to be homogeneous), the volumetric caustics can be generated by using the ray casting technique [19]. A ray is cast to the scene for every pixel on the screen and the caustic intensity at each sample point on the ray is integrated. Typically, we use 128 samples on the rays. In order to check whether each sample point is blocked by occluders, its distance to the center of the caustic cube map is compared with the depth stored in the cube map. If the distance is less than or equal to the depth stored in the cube map, the caustics intensity from the caustic spheres $C$ is sampled. Otherwise, this sample point can be skipped as it is blocked by the occluders. In the integration, we consider the light attenuation from the sample point to the camera and the scattering phase function (can be computed using either Rayleigh phase function [20] or Henvey-Greenstein phase function [21]).
4.2. **Multiple Directional Lights**

To generate caustics under \( D \) numbers of directional lights, we apply the algorithm for one directional light to each light and accumulate the computed caustic patterns on the caustic cube map. Then, the accumulated caustic patterns are projected to the scene. Similarly, for volumetric caustics rendering, for every sampling point on the cast ray we sample caustic spheres \( C \) of all \( D \) directional lights and integrate them. This technique is used to render approximate cast caustics and volumetric caustics under environment illumination (Section 5). The cast caustics and volumetric caustics rendering algorithm is presented in Algorithm 2.

Note the weight in Line 5 of Algorithm 2 is set to 1.0 unless it is rendering under environment illumination (Section 5). For rendering under one directional light, Lines 2 - 6 are executed only once.

5. **Rendering Caustics under Environment Illumination**

In our technique, we represent the environment illumination as an environment cube map. Every pixel in the cube map represents a distant directional light, thus the total irradiance at a point in a scene is the sum of radiance from all pixels in the cube map. However, integrating the radiance from all pixels in the cube map during rendering is impractical. To solve this issue, we segment the environment cube map into several important light regions and represent each of them with a directional light. We then sample the important light regions in the rendering.

5.1. **Environment Cube Map Segmentation**

Real-time rendering techniques often approximate an environment illumination with several lights. Debevec [22] recursively segments the environment map (latitude-longitude format) into two regions having almost equal total radiance until a number of iterations. In our technique, to account for light occlusion, we need to represent an environment illumination as a cube map. Therefore, we cannot use Debevec’s segmentation algorithm [22]. Recently, the algorithm for segmenting environment cube map into several area light sources is proposed by Annen et al. [23]. For our purpose, we cannot use Annen et al.’s method [23] since on a cube map face that has almost homogenous radiance, the method only produces one large light region which is not suitable to be represented using only one directional light.
Algorithm 2: Rendering cast caustics and volumetric caustics.

1. Render the surrounding of a caustic object into a temporary cube map;
   /* Compute caustic spheres \( C \) and caustic cube map. */
2. \textbf{foreach} directional light \( L \) of all \( D \) directional lights \textbf{do}
   3. Choose four nearest predefined light directions;
   4. Rotate their caustic spheres and blend them;
   5. Weight them and accumulate them into the caustic spheres \( C \);
   6. Accumulate the caustic intensities to the caustic cube map;
   /* Compute cast caustics. */
   7. \textbf{foreach} visible point in the scene \textbf{do}
      8. \( \text{dist} = \) distance from the visible point to the caustic object;
      9. \textbf{if} \( \text{dist} = \) distance in the caustic cube map \textbf{then}
         10. Render the caustic point (by sampling the RGB channel of the caustic cube map);
   /* Compute volumetric caustics. */
   11. \textbf{foreach} pixel in the screen \textbf{do}
      12. Shoot a ray through that pixel to the scene;
      13. \textbf{foreach} uniform step along the ray \textbf{do}
         14. \( \text{acc} = 0; \)
         15. \( \text{dist} = \) distance between the point in the current step and the caustic object;
         16. \textbf{if} \( \text{dist} \leq \) distance in the caustic cube map \textbf{then}
            17. Sample the caustic spheres \( C \) of all \( D \) directional lights and accumulate to \( \text{acc} \);
            18. Add the pixel value in the screen with \( \text{acc} \)
For rendering caustics under environment illumination, we propose the following environment cube map segmentation algorithm for computing important light regions. We assume that the importance of a light region is determined by the total radiance of the region.

Given an environment cube map and the number of intended light regions $D$, our algorithm divides the environment cube map into $D$ important non-overlapping rectangular light regions. The algorithm starts with six regions, with each corresponds to one face of the cube map. We prioritize segmenting the region having the most total radiance until we obtain $D$ light regions. If the total radiance of a region is below a certain threshold (almost no radiance), then it will be discarded and will not be processed further. Afterward, repeatedly select the region which has the most total radiance from all faces and subdivide that region into two new regions having the same total radiance (similar idea as Debevec’s segmentation [22]).

In the subdivision we choose either to segment horizontally or vertically based on which segmentation yields two regions with the aspect ratios nearer to 1.0. The aspect ratio is the ratio of the dimension with the bigger magnitude ($\max\{\text{width, height}\}$) to the dimension with the smaller magnitude ($\min\{\text{width, height}\}$) of a rectangular region. Regions with aspect ratio closer to 1.0 are preferrable since we represent each important light region with a directional light source. For doing so, the region is segmented horizontally and vertically, and we compute their aspect ratio deviations. The deviation is computed as the sum of the squared deviation of the new regions (Equation 2).

$$Dev = (A_1 - 1.0)^2 + (A_2 - 1.0)^2,$$

with $Dev$ is the deviation, $A_1$ and $A_2$ are the aspect ratio of the two new regions from the subdivision.

We select the segmentation that yields a smaller $Dev$ and continue with performing the segmentation until $D$ light regions are obtained. For each light region (of selected $D$ regions), we store the weighted center (with the weight is the radiance of each pixel in the region) of the region as the important light direction along with the region boundary (i.e. the coordinates of the two opposite corners of the rectangular region). Figure 7 shows the results of our environment cube map segmentation algorithm.

Taking into account the light occlusion effect (details in Section 5.2), there are three approaches in computing the important light regions and their corresponding directional lights during the rendering. In the first approach,
given an environment illumination, we recompute on-the-fly the directional light for each important light region based on the unoccluded pixels of the region and in the second approach we recompute the light regions (perform the environment cube map segmentation) on-the-fly. However, these two approaches suffer from temporal incoherence and some undesirable effects such as jittering and popping-in popping-out of caustic patterns.

In the third approach, the light regions and the directional lights are precomputed without considering occluders and we use this precomputed information in the rendering. By using this approach, parts of caustic patterns become dimmer or disappear in the presence of occluders without having the temporal incoherence problems. As such, we adopt this approach for segmenting the given environment illumination.
The segmentation algorithm is shown in Algorithm 3.

**Algorithm 3: Environment cube map segmentation.**

1. Generate six regions with each is one of the cube map faces;  
   /* total_regions = 6. */
2. Discard regions whose total radiance are below threshold;  
   /* total_regions becomes \( \leq 6 \) after discarding. */
   /* Next, do the iterative segmentation until we obtain \( D \) regions. */
3. while total_regions < D do
4.     Select the region with the most total radiance;
5.     Compute \( \text{Dev} \) (Equation 2) for each segmentation direction;
6.     Choose segmentation direction (horizontal or vertical) with the lower \( \text{Dev} \) and segment the selected region into two regions;
7.     Discard regions whose total radiance are below threshold;

5.2. Directional Light Radiance Sampling Taking Into Account Occlusion

As the directional lights are derived from an environment cube map, the radiance of each directional light is the total radiance of all unoccluded pixels in the important light region (corresponding to the directional light). To compute the total radiance, we render the important light region taking into account light occlusion (by rendering the occluders as black color) into a texture (or a 2D image) and sum up the radiance of all pixels in the important light region. This computation uses the information of each important light region and its corresponding light direction computed using our environment cube map segmentation algorithm.

Since we represent the environment illumination as a cube map, it is easy to set up the sampling frustum (a frustum is a 3D rendering region with the shape of a truncated pyramid in which all visible 3D objects are inside this region) for rendering the important light regions. The rendering sampling frustum uses the boundary information of the important light regions (obtained from our environment cube map segmentation algorithm). The coordinates of the two opposite corners of the rectangular important light region are used to specify the left, right, bottom, and top boundaries of the sampling frustum. The direction of the sampling frustum is the direction from the center of the caustic object to the face of the environment cube map where the important light region lies. Figure 8 illustrates how we sample the important light and Figure 9 shows an example of environment map.
Figure 8: We sample the important light regions by rendering them. The sampling frustum for the rendering is based on the boundary information of the light region with the viewing direction is the direction to the environment cube map face containing that particular region. The occluders are rendered as black objects.

The occlusion handling under environment illumination algorithm is presented in Algorithm 4.

**Algorithm 4**: Occlusion handling.

1. **foreach** light direction $L$ **do**
2. Render the segmented region (output of Algorithm 3) corresponding to $L$;
3. Render the occluders as black color;
4. Compute the total intensity of the rendered image, which serves as the weight for $L$. (The weight is used in Line 5 of Algorithm 2);

5.3. **Caustic Computation**

Since we approximate the environment illumination as a set of directional lights, we compute the final caustics by integrating the caustic patterns of the all directional lights (with each is weighted by the total radiance of the sampled important light region of that particular light direction).
Figure 9: Environment map sampling. (a) shows the scene configuration with the environment map shown in Figure 8. (b) shows the environment map sampling by rendering 24 light regions. Note the occluders are rendered as black objects.

6. GPU Implementation

We present the GPU implementation of our caustic rendering under $D$ directional lights. We use OpenGL and Cg in the implementation. There are three visual effects in the rendering, namely refraction effect, caustics cast on diffuse surfaces, and volumetric caustics. We use the image-based method proposed by Wyman et al. [24] to render the refraction effect of the caustic object.

The caustic rendering consists of three major stages: storing the precomputed caustic spheres into memory, computing the directional light radiance, and computing caustic intensity. Storing the caustic spheres is done only once when we load the caustic object to the scene. We recompute the light radiance and caustics only if there are changes in the scene (the caustic object, surrounding objects, and/or environment light are transformed).

6.1. Storing Caustic Spheres

We store the precomputed caustic spheres (caustic images) of each light direction in a 3D texture, so that we can directly use the trilinear interpo-
lation provided internally by the graphics API when we sample the caustic spheres. We test two alternatives to store the precomputed caustic spheres of all light directions, multiple 3D textures and a single 3D texture.

**Multiple 3D textures** The first method is to use several 3D textures and each 3D texture stores the caustic spheres of one light direction. By storing the precomputed caustic spheres of each direction in a 3D texture, we need to do multiple rendering passes ($D$ passes) in order to accumulate the caustic patterns from all light directions.

**Single 3D texture** The second alternative is to store the caustic spheres of all light directions into a single 3D texture by tiling. As the result, we are able to accumulate the caustic patterns from all $D$ light directions either in single pass or multiple passes.

### 6.2. Computing Directional Light Radiance

The caustic intensity of each light direction depends on light radiance from that particular direction arriving at the caustic object. We can estimate the radiance in the presence of occluders by using the technique described in Section 5.

To account for occlusion, we render the segmented regions of the environment map from the center of the caustic object to a texture array, with each slice in the texture array corresponds to one segmented region. The rendering is done to a low resolution texture array, i.e. $32 \times 32 \times D$. Afterward, we generate low resolution textures for each slice in the texture array by using the mipmapping. Thus, we can sample the average radiance of a light region reaching the caustic object by sampling the topmost level of the mipmap. Since the radiance value that we obtain from the mipmapping is the average value, we multiply it with the number of pixels in that light region in the original environment cube map in order to get the total radiance.

### 6.3. Caustics Sampling

There are two ways for computing the caustic intensity from all lights. First, for each light we sample the four nearest precomputed caustic spheres directly (**direct sampling**). Second, we accumulate the caustic spheres of all lights beforehand (**compiled**) into compiled caustic spheres and then we sample these compiled caustic spheres in the rendering.

**Direct sampling** For each light, and for every point in the scene where caustic intensity need to be computed, we sample the caustic intensities from the caustic spheres of the four nearest precomputed light directions and
interpolate them. This method is straightforward, however, for every point in the scene we need to do the same operations (e.g. rotating the caustic spheres of the four nearest light directions for every point in the scene). This becomes a bottleneck in volumetric caustics rendering since we need to do this process for every sample point on the cast ray and it greatly decreases the performance as we need to sample many points on the cast ray, and for every point we need to iterate through all $D$ light directions.

Compiled In the compiled method, to avoid the redundant operations as the ones in the direct sampling, we accumulate the caustic spheres from all light directions into a compiled single 3D texture beforehand. For every light direction $L_i$ ($i = 1, ..., D$), we rotate and blend the caustic spheres of the four nearest light directions, multiply with the radiance of $L_i$, and accumulate them into the compiled 3D texture. In the rendering, whether casting the caustics or rendering volumetric caustics, we just need to sample the compiled 3D texture.

6.4. Rendering Passes

We can perform the rendering either in multiple passes or single passes.

Multiple passes The iteration is done in CPU, with each iteration corresponds to one directional light. In every iteration, we pass the information of one directional light such as the four nearest precomputed light directions and their rotation matrices to the fragment shader. Using this technique, we can store the precomputed caustic patterns in the GPU either in multiple 3D textures or a single 3D texture.

Single pass The iteration for all directional lights is performed in GPU. We store the necessary information computed in CPU in the Texture Buffer Object because of its efficient data storing and retrieval capability in GPU. Since Texture Buffer Object is basically a 1D array, we store the information of the lights as a series of blocks of $16 \times 4$ float numbers (for each light, 1 block for the four nearest precomputed light indices in the 3D texture, 1 block for the weights, 1 block for the light intensities, 12 blocks for the rotation matrices and 1 unused block). In the shader, we just need to loop through all the $D$ light directions, sample the information from the Texture Buffer Object, rotate and blend the caustic patterns, and accumulate them. In this technique, we can store the precomputed caustic spheres in GPU only in a single 3D texture.
7. Results

We performed the experiments on a PC with an Intel Core i7 2.67 GHz and an Nvidia GTX 285. The image size of our real-time rendering results are $1024 \times 768$ pixels.

7.1. Rendering Results using Our Technique

Figure 10 shows our rendering results under one directional light ($D = 1$ and $s = 16$) and environment illumination ($D = 24$ and $s = 16$). Since the caustics under environment illumination are computed from many light directions, they are blurred. From our experiments, the visual differences of the caustics between $D = \{32, 48\}$ and $D = 24$ (number of directional lights) were hardly noticeable. Moreover, under any number of light directions, the visual differences of $s = 16$ and $s = 32$ (number of caustic spheres) were also not very apparent. Therefore, we suggest the rendering with $D = 24$ and $s = 16$.

7.2. Visual Comparisons

Figure 11 shows the comparison of caustics rendering using our quadratic radii caustic spheres and uniform radii caustic spheres proposed by Wyman et al. [1].

As seen in Figure 11, the quadratic radii using fewer caustic spheres achieves similar visual results as the uniform radii which uses more caustic spheres. This is because light attenuates and therefore it is sufficient to densely sample near the caustic object and sparsely sample far from the caustic object. With few caustic spheres ($s = 8, 16$) circular banding artifact is visible near the caustic object in the rendering results using uniform radii caustic spheres. It happens due to the insufficient number of caustic spheres near the caustic object thus the difference of the bright caustic patterns between caustic spheres is very conspicuous. This artifact does not exist in the rendering using quadratic radii since we sample densely near the caustic object, that is the spacing between caustic spheres near the caustic object is small. Note that the artifact also does not exist in the locations far from the caustic object due to the weak caustic patterns in those locations.

We also compare the images generated using quadratic radii and uniform radii with mental ray result and we show the image differences between Figures 11(a) - 11(d) and Figure 11(e) with the brighter pixels means
Figure 10: Caustics under one directional light source with the light direction is denoted by the arrow (top row) and caustics under environment illumination (bottom row). (a) shows caustics cast to the scene and (b) shows the result with volumetric caustics.

larger difference (Figure 12). The image differences are computed by using pixel-wise Euclidean distance.

As seen in Figure 12, image differences rendered using quadratic radii with $s = 16$ caustic spheres and uniform radii with $s = 32$ exhibit less difference compared to the images rendered using quadratic radii with $s = 8$ caustic spheres and uniform radii with $s = 16$ caustic spheres. Moreover, image difference rendered using quadratic radii with $s = 16$ has slightly less difference compared to the image difference rendered using uniform radii with $s = 32$.

As for the case with light occlusion, we also show the comparison between our results and the results generated using mental ray in Figure 13 and we can see that our results are visually similar to the mental ray results. For
Figure 11: Rendering comparisons of the images generated using uniform radii caustic spheres ((a) and (b)), quadratic radii caustic spheres ((c) and (d)), and mental ray (e). We rendered (a) - (d) using 24 directional lights.
Figure 12: Image differences between Figures 11(a) - 11(d) and Figure 11(e). We scaled all the pixel differences uniformly in order to provide a better view.
example, due to the presence of the occluder above and slightly to the left of the bunny, we can see in both our and mental ray results, the caustics at the right hand side of the bunny disappear (please compare with the result without occlusion in Figure 11) as the light from the top left is blocked by the occluder. We show more rendering comparisons between the images rendered using our technique with the images rendered using mental ray in Figure 14.

As for the rendering performance, we were able to render the caustics in real-time while mental ray took about four minutes to render one image.
Figure 14: Rendering comparisons between the images generated using our technique (left column) and mental ray (right column).
7.3. Performance Comparison

We performed experiments using combinations of all possible options of the rendering techniques described in Section 6 and combinations of various numbers of directional lights $D = \{1, 16, 24, 32, 48\}$ and caustic spheres $s = \{8, 16, 32\}$ to determine the best rendering performance (in average frames per second, fps). We show the performance comparisons of rendering cast caustics in Table 1. In general, we achieve the best rendering performance by using the combination of multiple passes rendering, multiple 3D textures storage for the caustic spheres, and the compiled technique. As for the volumetric caustics rendering, the performance ratios between the alternatives have similar tendency as the performance ratios in the cast caustics rendering. For the volumetric caustics rendering with multiple passes rendering, multiple 3D textures storage for the caustic spheres, and the compiled technique, we were able to achieve 14.6 frames per second.

Table 1: Performance comparisons of various caustics rendering techniques for cast caustics rendering (in average frames per second, fps)(Note: #Dirs = number of directions, D=direct, C=compiled).

| #Dirs. | #Spheres | Multiple Passes | Single Pass | |
|--------|----------|----------------|-------------|
|        |          | Multiple Textures | Single Texture | D | C | D | C |
| 1      | 8        | 142.3          | 173.1       | 140.3 | 165.4 | 120.0 | 176.7 |
|        | 16       | 142.7          | 156.5       | 135.4 | 151.4 | 119.6 | 148.9 |
|        | 32       | 142.6          | 129.9       | 139.6 | 124.8 | 120.0 | 120.9 |
| 16     | 8        | 38.2           | 60.0        | 31.2  | 60.2  | 21.4  | 52.8  |
|        | 16       | 37.4           | 38.8        | 31.2  | 39.7  | 21.2  | 35.1  |
|        | 32       | 37.2           | 24.0        | 31.2  | 24.1  | 21.2  | 20.5  |
| 24     | 8        | 27.6           | 45.0        | 22.9  | 45.1  | 14.9  | 41.0  |
|        | 16       | 27.7           | 30.0        | 22.7  | 29.9  | 14.9  | 25.0  |
|        | 32       | 27.7           | 17.4        | 22.8  | 17.5  | 14.9  | 14.2  |
| 32     | 8        | 22.0           | 37.6        | 17.8  | 37.4  | 11.5  | 32.7  |
|        | 16       | 22.2           | 23.5        | 17.8  | 23.7  | 11.6  | 19.7  |
|        | 32       | 22.2           | 13.5        | 17.8  | 13.5  | 11.4  | 11.0  |
| 48     | 8        | 15.5           | 27.0        | 12.4  | 27.5  | 7.9   | 23.2  |
|        | 16       | 15.5           | 16.4        | 12.4  | 16.7  | 7.9   | 13.1  |
|        | 32       | 15.5           | 9.6         | 12.5  | 9.3   | 7.8   | 7.5   |
We also provide an accompanying video in this submission. Please view the submitted video for more results.

7.4. Limitations of Our Technique

Our technique can produce approximate caustics under environment illumination in real-time, however there are some limitations. First, our technique requires a large amount of memory to store the caustic spheres. Second, since we perform precomputation, the caustic objects cannot be deformed during the rendering. However, many caustic objects in real-world such as glass objects are actually non-deformable. Therefore, we can apply our method to render caustics of such objects.

8. Conclusions and Future Work

We have presented an environment cube map segmentation technique and a GPU-based rendering technique for generating approximate caustics and volumetric caustics under environment illumination. Our rendering technique supports dynamic scenes and it also takes into account light occlusion from the surrounding objects during the rendering. Based on experimental results, our rendering technique can achieve real-time performance and it can produce similar visual quality as the mental ray results.

Currently there are no shadows in the rendering results, for future work we intend to incorporate the technique in [23] for shadow rendering under environment illumination. Moreover, we also want to look into the compression of the precomputed caustic patterns. Due to the precomputation nature, our technique currently only supports non-deformable caustic objects (as the caustic patterns have to be precomputed again if there are changes in the geometry of the caustic objects). Hence, we are also interested in investigating possible ways in handling deformable caustic objects with less or without precomputation. Finally, our technique can be extended to support caustic rendering under several local area lights, since each area light can also be approximated with fewer number of lights by using our segmentation technique (Section 5.1). However, care must be taken with respect to the relative position and orientation between the area lights and the caustic object.

9. Acknowledgements

We would like to express our gratitude to Chris Wyman for the GPU implementation of the refraction effect, Paul Debevec for the environment cube
maps (Campus, Kitchen, Uffizi, and Grace Cathedral), and Stanford Computer Graphics Laboratory for the 3D models (bunny, dragon, armadillo).

References


