EUROGRAPHICS 2005 / M. Alexa and J. Marks
(Guest Editors)

Volume 24 (*2005*), *Number 3*

Approximate Ray-Tracing on the GPU with Distance Impostors

László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, Mátyás Premecz

Department of Control Engineering and Information Technology, Budapest University of Technology, Hungary
Email: szirmay@iit.bme.hu

Abstract

This paper presents a fast approximation method to obtain the point hit by a reflection or refraction ray. The calculation is based on the distance values stored in environment map texels. This approximation is used to localize environment mapped reflections and refractions, that is, to make them depend on where they occur. On the other hand, placing the eye into the light source, the method is also good to generate real-time caustics. Computing a map for each refractor surface, we can even evaluate multiple refractions without tracing rays. The method is fast and accurate if the scene consists of larger planar faces, when the results are similar to that of ray-tracing. On the other hand, the method suits very well to the GPU architecture, and can render ray-tracing and global illumination effects with few hundred frames per second. The primary application area of the proposed method is the introduction of these effects in games.

1. Introduction

When computing the light transfer, the basic operation is tracing a ray from its origin point at a direction to find that point which is the source of illumination. Current graphics processing units (GPU) trace rays of the same origin very efficiently. However, in reflection, refraction and caustic computations the rays are not so coherent, but we need to trace just a single ray from each of many origins. Although it is possible to implement such a general ray tracer on the GPU [PBMH02, PDC*03], its performance is much poorer than that of tracing a bundle of rays sharing the same origin.

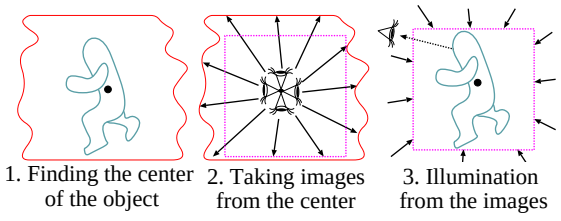


Figure 1: *Steps of environment mapping*

A GPU friendly approximation technique is *environment mapping*, which assumes that the hit point is very (infinitely) far, and thus it becomes independent of the ray origin. In this case rays can be translated to the same *reference point*, so we get that case back for which the GPU is an optimal tool. When rays originate at a given object, environment mapping takes images about the environment from the center of the object, then the environment of the object is replaced by a cube, or by a sphere, which is textured by these images (figure 1). When the incoming illumination from a direction is needed, instead of sending a ray we can look up the result from the images constituting the *environment map*. Environment mapping [BN76] has been originally proposed to render ideal mirrors in local illumination frameworks, then extended to approximate general secondary rays without expensive ray-tracing [Gre84, RTJ94, Wil01]. Environment mapping has also become a standard technique of *image based lighting* [MH84, Deb98].

A fundamental problem of environment mapping is that the environment map is the correct representation of the direction dependent illumination only at a single point, the reference point of the object. For other points, accurate results can only be expected if the distance of the point of interest from the reference point is negligible compared to the distance from the surrounding geometry. However, when the

object size and the scale of its movements are comparable with the distance from the surrounding surface, errors occur, which create the impression that the object is independent of its illuminating environment.

A compromise is needed that uses just a single environment map for the whole object, which is recomputed only if the object moved far from the reference point. Thus the computational cost is close to that of the original environment mapping method. On the other hand, the single environment map is used "intelligently" to provide different local illumination information for every point, based on the relative location from the reference point. At a given time, we associate just a single environment map with each dynamic, reflective object, but make "localized" illumination lookups when its different points are shaded. Making texture map lookups depend on the viewing direction has been suggested in the context of bump mapping [Wel04], displacement mapping [WWT*04], and in image based rendering [LH96]. Localized image based lighting has been proposed by Bjorke [Bjo04], where a proxy geometry (e.g. a sphere or a cube) of the environment is intersected by the reflection ray to obtain the visible point. Due to the fixed and simple proxy geometry, it is possible to implement ray-tracing on the pixel shader of the GPU. However, the assumption of a simple and constant environment geometry creates visible artifacts that make the proxy geometry apparent during animation.

Unlike previous environment mapping methods, we do not ray-trace the neighborhood or the proxy geometry, but rely solely on environment map lookups. The geometric information required by the localization process is also stored in the environment map, similarly to nailboards [Sch97] or layered depth impostors [DSSD99]. This information is the distance of the source of the illumination from the reference point where the environment map was taken. The proposed method picks environment map texels for a ray based on geometric information. Of course, a single map cannot always guarantee correct results if view dependent occlusions occur. As the object moves in the environment, the probability of such occlusions increases. Thus when the movement exceeds a threshold, the environment map is regenerated from the translated reference point. Since the environment map is not refreshed in every frame, the amortized cost of its generation becomes negligible.

The new environment mapping process provides exact results if the surface of the environment is a plane between the sampled points, and the approximation error is small even for non-planar surfaces as well. The algorithm is simple and can be executed by current vertex and pixel shaders at very high frame rates.

2. Localization of the environment map

The basic idea of this paper to localize environment maps is discussed using the notations of figure 2. Let us assume that center $\vec{o}$ of our coordinate system is the reference point of the environment map and we are interested in the illumination of point $\vec{x}$ from direction $\vec{R}$. We suppose that direction vector $\vec{R}$ has unit length.

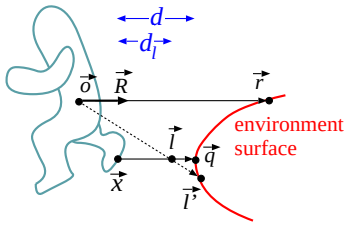


Figure 2: *Localization of the environment map having reference point $\vec{o}$. When computing the incoming radiance at point $\vec{x}$ from direction $\vec{R}$, ray tracing would select point $\vec{q}$, classical environment mapping would read the radiance of point $\vec{r}$, while the proposed method calculates $\vec{l}$ approximating the hit point on the ray and looks up the environment map in this direction obtaining the radiance of point $\vec{l}'$.*

Classical environment mapping would look up the illumination selected by direction $\vec{R}$, that is, it would use the radiance of point $\vec{r}$. However, $\vec{r}$ is usually not equal to point $\vec{q}$, which is in direction $\vec{R}$ from $\vec{x}$, and thus satisfies the following ray equation for some distance d:

$$\vec{q} = \vec{x} + \vec{R} \cdot d. \tag{1}$$

Our localization method finds an approximation of d using an iterative process working with distances between the environment and reference point $\vec{o}$. The required distance information can be computed during the generation of the environment map. While a normal environment map stores the illumination for each direction in R,G,B channels, now we also obtain the distance of the visible point for these directions and store it, for example, in the alpha channel. We call these extended environment maps as *distance impostors*.

2.1. Finding ray hit approximations

To find an initial guess for the ray hit, we first assume that the environment surface at $\vec{r}$ is perpendicular to ray direction $\vec{R}$ (figure 3). In case of perpendicular surface, the ray would hit point $\vec{p}$. Points $\vec{r}$, $\vec{x}$ and origin $\vec{o}$ define a plane, which is the base plane of figure 3. This plane also contains visible point approximation $\vec{p}$ and unit direction vector $\vec{R}$. Multiplying ray equation

$$\vec{x} + \vec{R} \cdot d_p = \vec{p} \tag{2}$$

by direction vector $\vec{R}$ and substituting $\vec{R} \cdot \vec{p} = |\vec{r}|$, which is the consequence of the perpendicular surface assumption, we can express ray parameter d_p:

$$d_p = |\vec{r}| - \vec{R} \cdot \vec{x}.$$

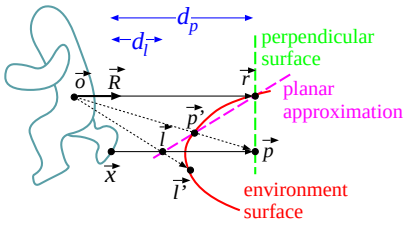


Figure 3: *Identifying first approximation point $\vec{p}$ assuming that the surface is perpendicular to $\vec{R}$, and second approximation point $\vec{l}$ supposing that the surface is planar between points $\vec{r}$ and $\vec{p}'$.*

If we used the direction of point $\vec{p}$ to lookup the environment map, we would obtain the color of point $\vec{p}'$, which is in the direction of $\vec{p}$ but is on the surface.

If the surface were a plane between $\vec{p}'$ and $\vec{r}$, then the intersection of the planar surface and the plane defined by points $\vec{r}$, $\vec{x}$ and $\vec{o}$ would be a line, and the point visible from $\vec{x}$ at direction $\vec{R}$ would be $\vec{l}$. Let us assume that the surface can be well approximated by a plane between points $\vec{r}$ and $\vec{p}'$, and find intersection $\vec{l}$ of the plane with the ray. The intersection is on the ray, thus it satisfies the following ray equation:

$$\vec{l} = \vec{x} + \vec{R} \cdot d_l. \tag{3}$$

On the other hand, point $\vec{l}$ is also on the line of $\vec{r}$ and $\vec{p}'$, thus it can be expressed as a combination of these two points with unknown weight α:

$$\vec{l} = \vec{r} \cdot \alpha + \vec{p}' \cdot (1-\alpha).$$

Note that $\vec{l}$ is not necessarily between $\vec{r}$ and $\vec{p}'$ thus α is not restricted to $[0,1]$ but can have an arbitrary value. Substituting identities $\vec{p}' = \vec{p} \cdot |\vec{p}'|/|\vec{p}|$, $\vec{r} = \vec{R} \cdot |\vec{r}|$, and equation 2, we obtain

$$\vec{l} = \vec{r} \cdot \alpha + (\vec{x} + \vec{R} \cdot d_p) \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1-\alpha) =$$

$$\vec{x} \cdot \left((1-\alpha) \cdot \frac{|\vec{p}'|}{|\vec{p}|} \right) + \vec{R} \cdot \left(d_p \cdot (1-\alpha) \cdot \frac{|\vec{p}'|}{|\vec{p}|} + \alpha \cdot |\vec{r}| \right).$$

Comparing this expression with equation 3, we obtain the following requirements for unknowns α and d_l:

$$(1-\alpha) \cdot \frac{|\vec{p}'|}{|\vec{p}|} = 1, \qquad d_p \cdot (1-\alpha) \cdot \frac{|\vec{p}'|}{|\vec{p}|} + \alpha \cdot |\vec{r}| = d_l.$$

Solving this equation, we get:

$$d_l = d_p + |\vec{r}| \cdot \left(1 - \frac{|\vec{p}|}{|\vec{p}'|} \right). \tag{4}$$

Substituting this distance into the ray equation, we obtain hit point approximation $\vec{l}$, which can be used to look up the environment map. In this way, we would obtain the radiance of point $\vec{l}'$ of figure 3.

2.2. Refinement by iteration

So far we obtained two initial guesses of the ray parameter d_p and d_l, and consequently two points $\vec{p}$ and $\vec{l}$ that are on the ray, but are not necessarily on the surface, and two other points $\vec{p}'$ and $\vec{l}'$ that are on the surface, but are not necessarily on the ray (figure 4). These initial results can be refined by an iteration process, which computes new hit point approximation $\vec{l}_{new}$ assuming that the surface is planar between sample points $\vec{p}'$ and $\vec{l}'$, and then replaces $\vec{p}$ by $\vec{l}$ and $\vec{l}$ by $\vec{l}_{new}$.

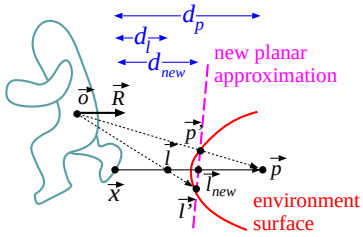


Figure 4: *Refinement by iteration.*

Since new approximation $\vec{l}_{new}$ is on the ray, it satisfies the following ray equation:

$$\vec{l}_{new} = \vec{x} + \vec{R} \cdot d_{new}. \tag{5}$$

Point $\vec{l}_{new}$ is also on the line of $\vec{l}'$ and $\vec{p}'$, thus it can be expressed as their combination with unknown weight α:

$$\vec{l}_{new} = \vec{l}' \cdot \alpha + \vec{p}' \cdot (1-\alpha).$$

Substituting identities $\vec{p}' = \vec{p} \cdot |\vec{p}'|/|\vec{p}|$ and $\vec{l}' = \vec{l} \cdot |\vec{l}'|/|\vec{l}|$, as well as the ray equation for d_p and d_l, we get:

$$\vec{l}_{new} = (\vec{x} + \vec{R} \cdot d_l) \cdot \frac{|\vec{l}'|}{|\vec{l}|} \cdot \alpha + (\vec{x} + \vec{R} \cdot d_p) \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1-\alpha).$$

Comparing this expression with equation 5, we obtain the following requirements for unknowns α and d_{new}:

$$\frac{|\vec{l}'|}{|\vec{l}|} \cdot \alpha + \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1-\alpha) = 1,$$

$$d_l \cdot \frac{|\vec{l}'|}{|\vec{l}|} \cdot \alpha + d_p \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1-\alpha) = d_{new}.$$

Solving this equation, we get:

$$d_{new} = d_l + (d_l - d_p) \cdot \frac{1 - |\vec{l}|/|\vec{l}'|}{|\vec{l}|/|\vec{l}'| - |\vec{p}|/|\vec{p}'|}. \tag{6}$$

A step of the iteration evaluates this formula and replaces d_p by d_l and d_l by d_{new} together with their associated points on the ray and on the surface.

From mathematical point of view, the proposed iteration

method solves ray equation $f(\vec{x}+\vec{R}\cdot d)=0$ with the *secant method* [Wei03], where $f(\vec{r})=0$ is the implicit equation of the environment surface, which is represented by discrete distance samples of the environment map. The secant method usually converges very quickly [Wei03], but it may not converge for high variation functions. Note that in equation 6 the absolute value of denominator $|\vec{l}|/|\vec{l}'|-|\vec{p}|/|\vec{p}'|$ can be smaller than the absolute value of enumerator $1-|\vec{l}|/|\vec{l}'|$, which means that step size $|d_l-d_p|$ may also increase. While increasing the step size where necessary improves convergence, it is also the cause of divergence.

To address the convergence issue, the basic iteration should be slightly modified. Let us first recognize that ratios $|\vec{l}|/|\vec{l}'|$ and $|\vec{p}|/|\vec{p}'|$ showing up in equation 6 express the accuracy of approximation points $\vec{l}$ and $\vec{p}$. If the point were the real ray hit, then the ratio would be 1. Values smaller than 1 indicate that the approximation point is an *undershooting*, i.e. it is in front of the surface. On the other hand, when the ratio is greater than one, the approximation is an *overshooting* since the approximation point is behind the surface. For example, in figure 4 $\vec{p}$ and $\vec{l}_{new}$ are overshooting points, while $\vec{l}$ is an undershooting.

Note that when $\vec{l}$ and $\vec{p}$ are of different types, the absolute value of denominator $|\vec{l}|/|\vec{l}'|-|\vec{p}|/|\vec{p}'|$ cannot be smaller than the absolute value of enumerator $1-|\vec{l}|/|\vec{l}'|$, which results in a step size reduction. This condition can be enforced if last approximation $\vec{l}$ is paired not necessarily with the last but one approximation, but the last approximation of opposite type. This method is equivalent to the *false position root finding method* [Wei03]. We can switch from the secant method to the false position method if we have at least one overshooting point and one undershooting approximation. Since default point $\vec{r}$ corresponds to an infinite ray parameter, which is a sure overshooting, we can use this ideal point at infinity if there are no other overshooting results. On the other hand, if there is no undershooting point, we can either follow the secant rule, use reflection point $\vec{x}$, or default point $\vec{r}$ again to substitute the undershooting point. We have found that the last option is safe and works well.

Note that even with guaranteed convergence, the proposed method is not necessarily equivalent to exact ray tracing in the limiting case. Small errors may be due to the discrete surface approximation, or to view dependent occlusions. For example, should the ray hit a point that is not visible from the reference point of the environment map, then the presented approximation scheme would obviously be unable to find that. However, when the object is curved and moving, these errors can hardly be recognized visually.

3. Environment mapping with distance impostors

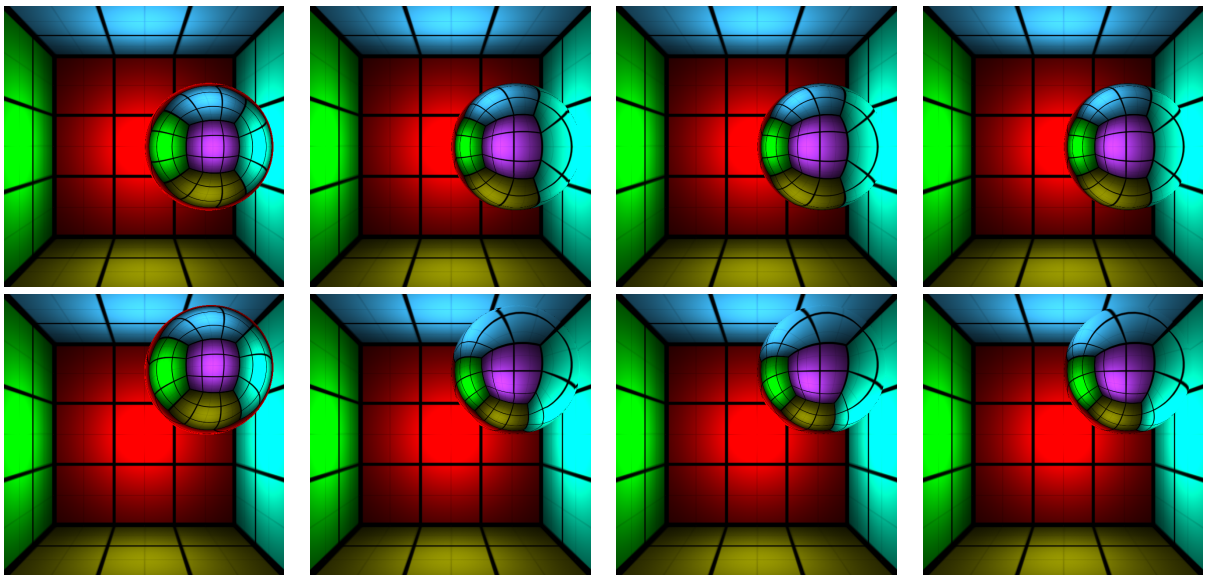
The computation of distance impostors is very similar to that of classical environment maps. The only difference is that the distance from the reference point is also calculated, which can be stored in a separate texture or in the alpha channel of the environment map. Since the distance is a non linear function of the homogeneous coordinates of the points, correct results can be obtained only by letting the pixel shader compute the distance values.

Having the distance impostor, we can place an arbitrary object in the scene and illuminate it with its environment map using custom vertex and pixel shader programs. The vertex shader transforms objects to normalized screen space, and also to the coordinate system of the environment map first applying the modeling transform, then translating to the reference point. View vector $\vec{V}$ and normal $\vec{N}$ are also obtained in world coordinates.

Having the graphics hardware computed the homogeneous division and filled the triangle with linearly interpolating all vertex data, the pixel shader is called to find ray hit $\vec{l}$ and to look up the cube map in this direction. The HLSL code of function `Hit` computing hit point approximation $\vec{l}$ with the false position method is shown below:

```
float3 Hit(float3 x, float3 R, sampler mp) {
   float rl = texCUBE(mp, R).a; // |r|
   float dp = rl - dot(x, R);
   float3 p = x + R * dp;
   float ppp = length(p)/texCUBE(mp,p).a;
   float dun =0, dov =0, pun = ppp, pov = ppp;
   if (ppp < 1) dun = dp; else dov = dp;
   float dl = max(dp + rl * (1 - ppp), 0);
   float3 l = x + R * dl;
                              // iteration
   for(int i = 0; i < NITER; i++) {
      float ddl;
      float llp = length(l)/texCUBE(mp,l).a;
      if (llp < 1) { // undershooting
         dun = dl; pun = llp;
         ddl = (dov == 0) ? rl * (1 - llp) :
               (dl-dov) * (1-llp)/(llp-pov);
      } else {       // overshooting
         dov = dl; pov = llp;
         ddl = (dun == 0) ? rl * (1 - llp) :
               (dl-dun) * (1-llp)/(llp-pun);
      }
      dl = max(dl + ddl, 0);  // avoid flip
      l = x + R * dl;
   }
   return l;
}
```

This function gets ray origin `x` and direction `R`, as well as cube map `mp` of the environment, and returns hit approximation `l`. We suppose that the distance values are stored in the alpha channel of the environment map. Ratios $|\vec{l}|/|\vec{l}'|$ and $|\vec{p}|/|\vec{p}'|$ are represented by variables `llp` and `ppp`, respectively. Note that variables `dun` and `dov` store the last undershooting and overshooting ray parameters. If there has been no such approximation, the ray parameters are zero. In this case default point $\vec{r}$ takes their roles. In order to avoid



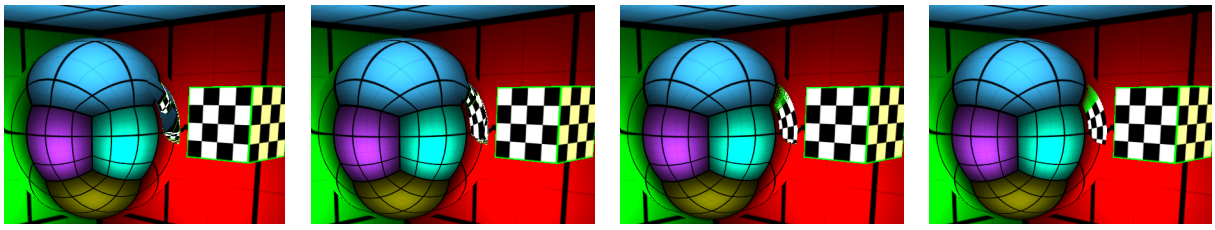
classical environment map
642 FPS

distance impostor
447 FPS

+1 iteration
323 FPS

ray traced reference

Figure 5: *Comparison of classical and localized environment map reflections with ray traced reflections placing the reference point at the center of the room and moving a reflective sphere to different locations. Note that even the initial guess made with the distance impostor is accurate almost everywhere but the corners where one iteration step is enough. The FPS values are measured with* 700×700 *resolution on an NV6800GT.*



distance impostor
430 FPS

+1 iteration
323 FPS

+4 iterations
169 FPS

+8 iterations
108 FPS

Figure 6: *A more difficult case when the room contains a box that makes the scene strongly concave and is responsible for view dependent occlusions.*

ray flipping, the algorithm limits ray parameters for the non-negative domain.

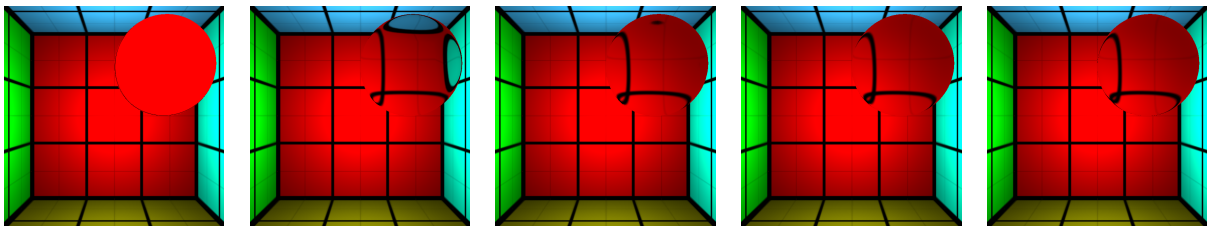
The pixel shader calls function `Hit` and looks up cube map `envmap` again to find illumination `I` of the hit point:

```
N = normalize(N); V = normalize(V);
R = reflect(V, N);        // reflection dir.
float3 l = Hit(x, R, envmap);   // ray hit
float3 I = texCUBE(envmap, l).rgb;
```

The next step is the computation of the reflection of incoming radiance `I`. If the surface is an ideal mirror, the incoming radiance should be multiplied by the Fresnel term evaluated for the angle between surface normal $\vec{N}$ and reflection direction $\vec{R}$. We applied an approximation of the Fresnel function, which is similar to Schlick's approximation [Sch93] in terms of computational cost, but can take into account not only *refraction index n* but also *extinction coefficient k*, which is essential for realistic metals [LSK05]:

$$F(\vec{N},\vec{R}) = \frac{(n-1)^2 + k^2 + 4n(1-\vec{N}\cdot\vec{R})^5}{(n+1)^2 + k^2}.$$

Figure 5 compares images rendered by the proposed method with standard environment mapping and ray tracing.



classical environment map 804 FPS | distance impostor single refraction 695 FPS | distance impostor double refraction 508 FPS | +1 iteration double refraction 249 FPS | ray traced reference

Figure 7: *Refractions of a sphere having refraction index $n = 1.1$.*

Note that for such scenes where the environment is convex from the reference point of the environment map, and there are larger planar surfaces, the new algorithm converges very quickly. In fact, even the initial guesses are usually accurate, and iteration is needed only close to edges and corners.

Figure 6 shows a difficult case where the box makes the environment surface concave and of high variation. Note that the convergence is still pretty fast, but the converged image is not exactly what we expect. We can observe that the green edge of the box is visible in a larger portion of the reflection image. This phenomenon is due to the fact that a part of the wall is not visible from the reference point of the environment map, but are expected to show up in the reflection. In such cases the algorithm can go only to the edge of the box and substitutes the reflection of the occluded points by the blurred image of the edge.

4. Multiple refractions

The proposed method can be used to simulate not only reflected but also refracted rays, just the direction computation should be changed from the law of reflection to the Snellius-Descartes law of refraction, that is, the `reflect` operation should be replaced by the `refract` operation in the pixel shader. However, tracing a refraction ray on a single level is usually not enough since the light is refracted at least twice to go through a refractor. The location of the second refraction as well as the normal vector at this point depend on the geometry of the object, which can only be analyzed by ray-tracing unless the refractor is very special (e.g. a sphere, a cylinder, etc.).

Applying distance impostors, however, we can solve this problem as well if the refractor is not strongly concave, i.e. all surface points can be seen from its center point. We create a distance impostor for each refractor, which stores the distance of the refractor surface from its center and the normal vector of the surface. If the refractor has static geometry, these impostors can be obtained during preprocessing. We call this distance impostor as the *refractor map*.

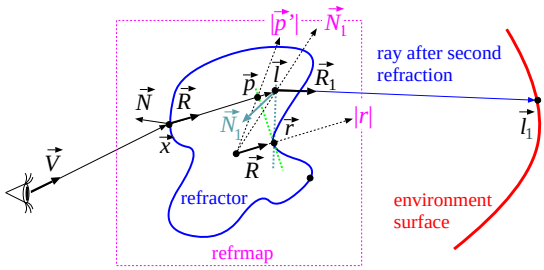


Figure 8: *Multiple refractions without iterative refinement. The ray refracts at $\vec{x}$ to direction $\vec{R}$. The refractor map is looked up in direction $\vec{R}$ to obtain $\vec{r}$. Using the perpendicular surface and planar surface assumptions we get $\vec{p}$ and $\vec{l}$, respectively. The refractor map is looked up in direction $\vec{l}$ to find normal $\vec{N}_1$ of the second refraction. Then the ray is refracted again at $\vec{l}$ using normal $\vec{N}_1$, and the process is continued with the environment map localization.*

Now let us consider point $\vec{x}$ on the surface of the refractor visible from the camera (figure 8). Using the proposed method for the refractor map, we obtain that point which is hit by the refraction ray. The normal vector at this point can be read from the distance impostor. Computing another refraction at this point and setting the origin of the refraction ray to the previously identified point, we can continue with the real environment map and find that point and color, which is visible after two refractions. The pixel shader of double refractions uses the refractor map (`refrmap`) with reference point stored in variable called `objcenter`:

```
N = normalize(N); V = normalize(V);
R = refract(V, N, 1/n); // first refraction
float3 l = Hit(x-objcenter, R, refrmap);
float3 N1 = texCUBE(refrmap, l).xyz;
R1 = refract(R, N1, n); // second refraction
   // envmap lookup
float3 l1 = Hit(l+objcenter, R1, envmap);
float3 I = texCUBE(envmap, l1); // in rad.
```

Figure 7 shows a refracting sphere rendered with classical environment map and also by the new method with single and double refractions. Note that multiple refraction has a significant effect even when the refraction index is close to 1, and also that the proposed method is quite accurate even with only one additional iteration step.

5. Application to caustics generation

The method presented so far can compute the hit point after the first (or higher order) reflection or refraction of the visibility ray. If we replace the eye by a light source, the same method can also be used to determine the first (or higher order) bounce of the light ray, thus we can compute the indirect illumination bounced off dynamic objects onto static ones.

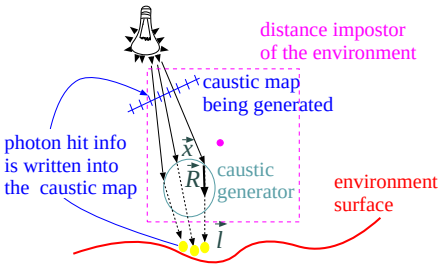


Figure 9: *Caustics generation with environment maps*

These indirect effects have a significant impact on the final image if the dynamic object is close to be an ideal reflector or refractor, when these effects show up in forms of caustic spots [Jen96, TS00, WS03]. The proposed distance impostors can thus be used to compute caustics.

When rendering the scene from the point of view of the light source, the view plane is placed between the light and the refractor (figure 9). The image on this view plane is called *caustic map*. Note that this step is very similar to the generation of depth images for shadow maps. In fact, if we combine the method with shadow mapping, we obtain caustics almost for free. However, implementing the caustic map generation separately allows us to optimally set the position and the resolution of the view plane of the caustic map.

Supposing that the surface is an ideal reflector or refractor, point $\vec{l}$ that receives the illumination of a light source after a single or multiple reflection or refraction can be obtained by the proposed approximate ray tracing, and particularly by calling the `Hit` function, after making the following substitutions: point $\vec{x}$ is visible from the light source through a caustic map pixel, $\vec{R}$ is the refraction (or reflection) of the direction from the light source onto the surface normal at $\vec{x}$. The localized environment map lookups provide an approximation of that point $\vec{l}$, which is hit by a photon after a single reflection, or alternatively, $\vec{l}$ is an approximation of the direction of the photon hit from the reference point of the environment map.

The photon hit parameters are stored in that caustic map pixel through which the primary light ray arrived at the caustic generator object. There are several alternatives to represent a photon hit. Considering that the reflected radiance caused by a photon hit is the product of the BRDF and the power of the photon, the representation of the photon hit should identify the surface point and its BRDF. A natural identification is the texture coordinates of that surface point, which is hit by the ray. A caustic map pixel stores the identification of the texture map, the u and v texture coordinates, and finally the luminance of the power of the photon. The photon power is computed from the power of the light source and the solid angle subtended by the caustic map pixel.

The identification of the u and v texture coordinates from the direction of the photon hit requires another texture lookup. Suppose that together with the environment map, we also render another map, called `uvmap`, which has the same structure, but stores the u,v coordinates and the texture id in its pixels. Having found the direction of the photon hit, this map is read to obtain the texture coordinates, which are finally written into the caustic map.

The vertex shader of caustic map generation transforms the points and illumination direction $\vec{L}$ to the coordinate system of the environment map. Then the pixel shader computes the location of the photon hit and puts it into the target pixel:

```
N = normalize(N); L = normalize(L);
R = refract(L, N, 1/n);    // or reflect ...
float3 l = Hit(x, R, envmap); // photon hit
float3 hituv = texCUBE(uvmap, l).xyz;
return float4(hituv, power); // to caustmap
```

In order to recognize those texels of the caustic map where the refractor is not visible, we initialize the caustic map with -1 alpha values. Checking the sign of the alpha later, we can decide whether or not it is a photon hit.

The generated caustic map is used to project caustic textures onto surfaces [GD01], or to modify their light map in the next rendering pass. Every photon hit should be multiplied by the BRDF, and the product is used to modulate a small filter texture, which is added to the texture of the surface. The filter texture corresponds to Gaussian filtering in texture space. In this pass we render as many small quadrilaterals (two adjacent triangles in DirectX) or point sprites as texels the caustic map has. The caustic map texels are addressed one by one with variable `caustcoord` in the vertex shader shown below. The center of these quadrilaterals is the origo, and their size depends on the support of the Gaussian filter. The vertex shader changes the coordinates of the quadrilateral vertices and centers the quadrilateral at the u,v coordinates of the photon hit in texture space if the alpha value of the caustic map texel addressed by `caustcoord` is positive, and moves the quadrilateral out of the clipping region if the alpha is negative. This approach requires the

texture memory storing the caustic map to be fed back to the vertex shader, which is possible on 3.0 compatible vertex shaders. The vertex shader of projecting caustic textures onto surfaces is as follows:

```
float4 ph = tex2Dlod(caustmap, IN.caustcoord);
OUT.filtcoord = IN.pos.xy; // filter coords
OUT.texcoord.x = ph.x + IN.pos.x / 2;
OUT.texcoord.y = ph.y - IN.pos.y / 2;
OUT.hpos.x = ph.x * 2 - 1 + IN.pos.x;
OUT.hpos.y = 1 - ph.y * 2 + IN.pos.y;
OUT.hpos.w = 1;
if (ph.a < 0) OUT.hpos.z = 2; // ignore
else          OUT.hpos.z = 0; // valid
OUT.power = ph.a;          // photon power
```

Note that the original x,y coordinates of quadrilateral vertices are copied as filter texture coordinates, and are also moved to the position of the photon hit in the texture space of the surface. The output position register (`hpos`) also stores the texture coordinates converted from $[0,1]^2$ to $[-1,1]^2$ which corresponds to rendering to this space. The `w` and `z` coordinates of the position register are used to ignore those caustic map elements which have no associated photon hit.

The pixel shader computes the color contribution as the product of the photon power, filter value and the BRDF:

```
float3 brdf = tex2d(textureid, texcoord);
float  w    = tex2d(filter, filtcoord);
return power * w * brdf;
```

The target of this rendering is the light map or the modified texture map. Note that the contribution of different photons should be added, thus we should set the blending mode to “add” before executing this phase.

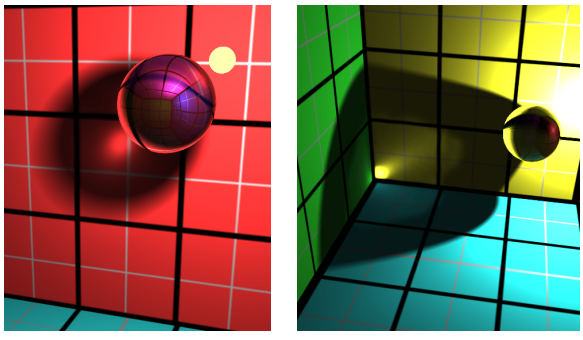


Figure 10: *Real-time caustics caused by a glass sphere ($n = 1.3$), rendered by the proposed method on 182 FPS*

Figure 10 shows the implementation of the caustics generation, when a 64×64 resolution caustic map is obtained in each frame, which is fed back to the vertex shader. Note that even with shadow, reflection, and refraction computation, the method runs with 182 FPS.

6. The complete rendering algorithm

The different techniques proposed by this paper can be combined in a complete real-time rendering algorithm. The input of this algorithm include

- the definition of the static environment in form of triangle meshes, material data, textures, and light maps having been obtained with a global illumination algorithm,
- refractor maps of those dynamic objects, which are expected to refract (or reflect) the light multiple times,
- the definition of dynamic objects set in the actual frame,
- the current position of light sources and the eye.

The image generation requires a preparation phase, and a rendering phase from the eye. The preparation phase computes the environment maps. Depending on the distribution of the dynamic objects, we may generate only a single environment map for all of them, or we may maintain a separate map for each of them. Note that this preparation phase is usually not executed in each frame, only if the object movements are large enough. If we update these maps after every 100 frames, then the cost amortizes and the slow down becomes negligible. If the scene has caustic generators, then a caustic map is obtained for each of them, and caustic maps are converted to light maps during the preparation phase.

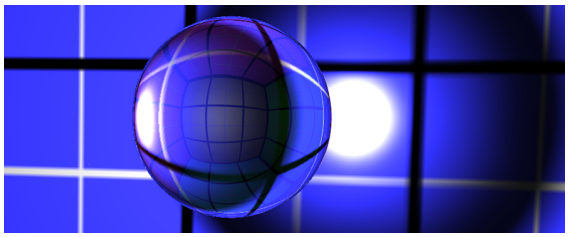


Figure 11: *Caustics seen through the refractor object.*

The final rendering phase from the eye position consists of three steps. First the static environment is rendered with their light maps making also caustics visible. Then dynamic objects are sent to the GPU, having enabled the proposed localized environment mapping and also multiple refraction computation. Note that in this way the reflection or refraction of caustics can also be generated (figure 11).

The presented algorithm has been implemented in Direct3D environment, first as a test application, and then in a game. The images of the test application are shown by figures 5, 6, 7, 10, 11, and 12. The test application computes the environment map only once to show that the proposed localization gives good results even if the objects moved significantly from their original positions. Note that this application runs typically with few hundred frames per second even in full screen mode on an NV6800GT graphics card and P4/3GHz CPU, and can maintain this speed for tens of thousand of triangles. For comparison, the peak performance

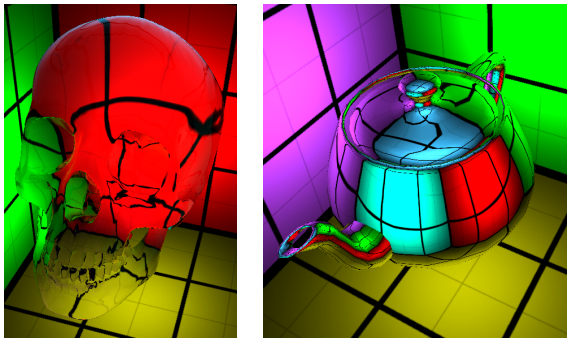


Figure 12: *Left: a glass skull ($n = 1.3$, $k = 0$) of 61000 triangles rendered on 130 FPS. Right: an alu teapot ($n = 0.5..2.3$, $k = 5..9$) of 2300 triangles rendered on 440 FPS.*

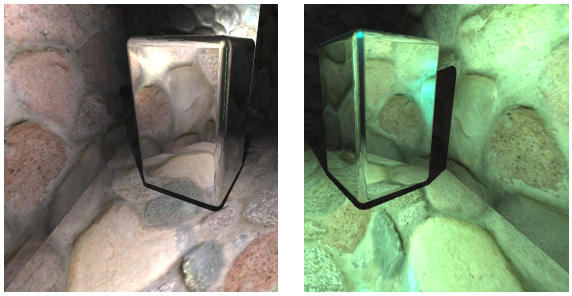


Figure 13: *A reflective box with shadows in a stone environment illuminated by dynamic lights (160 FPS)*

is 1095 FPS for the scene of figure 5 when the pixel shader executes a return instruction for the sphere.

We also included the proposed method in a game executing shadow computation, collision detection, etc. (figures 13, 14, and 15) that can run with about a hundred FPS. In this game we used $6 \times 256 \times 256$ resolution cube maps that are recomputed in every 150 msec. We have realized that the speed improves by an additional 20 percent if the distance values are separated from the color data and stored in another texture map. The reason of this behavior is that the pixel shader reads the distance values several times from different texels before the color value is fetched, and separating the distance values increases texture cache utilization.

7. Conclusions

This paper presented a localization method for environment maps, which uses the distance values stored in environment map texels. The accuracy of the initial guess can be improved by iteration. In this sense, the localization method is equivalent to approximate ray-tracing, which solves the ray equation by numerical root finding.

The proposed solution can introduce effects in games that are usually simulated by ray tracing, such as single or multiple reflections and refractions on curved surfaces, and caustics. Unlike ray tracing the proposed algorithm works with a predefined set of rays obtained by rendering the scene from the reference point, which makes a bridge between tracing rays and the incremental rendering concept of the GPU.

8. Acknowledgement

This work has been supported by OTKA (ref. No.: T042735), GameTools FP6 (IST-2-004363) project, and by the Spanish-Hungarian Fund (E-26/04).

References

[Bjo04] BJORKE K.: Image-based lighting. In *GPU Gems*, Fernando R., (Ed.). NVidia, 2004, pp. 307–322. 2

[BN76] BLINN J. F., NEWELL M. E.: Texture and reflection in computer generated images. *Communications of the ACM 19*, 10 (1976), 542–547. 1

[Deb98] DEBEVEC P.: Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *SIGGRAPH '98* (1998), pp. 189–198. 1

[DSSD99] DECORET X., SILLION F., SCHAUFLER G., DORSEY J.: Multi-layered impostors for accelerated rendering. *Computer Graphics Forum 18*, 3 (1999). 2

[GD01] GRANIER X., DRETTAKIS G.: Inremental updates for rapid glossy global illumination. *Computer Graphics Forum 20*, 3 (2001), 268–277. 7

[Gre84] GREENE N.: Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications 6*, 11 (1984), 21–29. 1

[IKSZ03] IONES A., KRUPKIN A., SBERT M., ZHUKOV S.: Fast realistic lighting for video games. *IEEE Computer Graphics and Applications 23*, 3 (2003), 54–64.

[Jen96] JENSEN H. W.: Global illumination using photon maps. In *Rendering Techniques '96* (1996), pp. 21–30. 7

[LH96] LEVOY M., HANRAHAN P.: Light field rendering. In *SIGGRAPH 96* (1996), pp. 31–42. 2

[LSK05] LAZÁNYI I., SZIRMAY-KALOS L.: Fresnel term approximations for metals. In *WSCG 2005, Short Papers* (2005), pp. 77–80. 5

[MH84] MILLER G. S., HOFFMAN C. R.: Illumination and reflection maps: Simulated objects in simulated and real environment. In *SIGGRAPH '84* (1984). 1

[PBMH02] PURCELL T., BUCK I., MARK W., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (2002), 703–712. 1

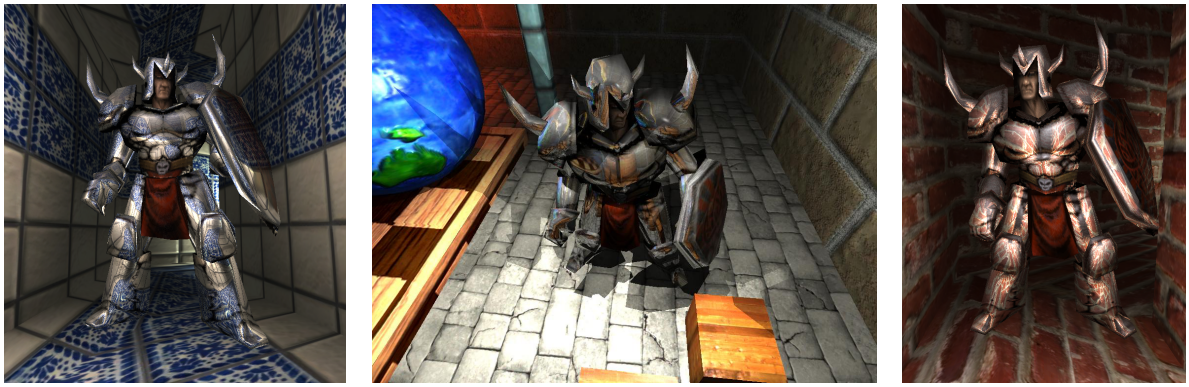


Figure 14: *A knight in reflective armor in textured environment illuminated by dynamic lights (105..130 FPS).*

Figure 15: *Reflective and refractive spheres in a game environment running with 105 and 85 FPS, respectively.*

[PDC*03] PURCELL T., DONNER T., CAMMARANO M., JENSEN H., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2003), pp. 41–50. 1

[RTJ94] REINHARD E., TIJSSEN L. U., JANSEN W.: Environment mapping for efficient sampling of the diffuse interreflection. In *Photorealistic Rendering Techniques*. Springer, 1994, pp. 410–422. 1

[Sch93] SCHLICK C.: A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering* (1993), pp. 73–83. 5

[Sch97] SCHAUFLER G.: Nailboards: A rendering primitive for image caching in dynamic scenes. In *Eurographics Workshop on Rendering* (1997), pp. 151–162. 2

[TS00] TRENDALL C., STEWART A.: General calculations using graphics hardware, with application to interactive caustics. In *Rendering Techniques 2000* (2000), pp. 287–298. 7

[Wei03] WEISSTEIN E.: *World of Mathematics*. 2003. http://mathworld.wolfram.com/Curvature.html. 4

[Wel04] WELSH T.: *Parallax Mapping with Offset Limiting: A Per Pixel Approximation of Uneven Surfaces*. Tech. rep., Infiscape Corporation, 2004. 2

[Wil01] WILKIE A.: *Photon Tracing for Complex Environments*. PhD thesis, Institute of Computer Graphics, Vienna University of Technology, 2001. 1

[WS03] WAND M., STRASSER W.: Real-time caustics. *Computer Graphics Forum 22*, 3 (2003), 611–620. 7

[WWT*04] WANG L., WANG X., TONG X., LIN S., HU S., GUO B.: View-dependent displacement mapping. *ACM Transactions on Graphics 22*, 3 (2004), 334–339. 2