Ray Tracing Effects without Tracing Rays

László Szirmay-Kalos, Barnabás Aszódi, and István Lazányi
Budapest University of Technology and Economics, Hungary

Introduction

The basic operation of rendering is tracing a ray from its origin point at a direction to find that point which is the source of illumination. The identification of the points visible from the camera requires rays that have the same origin. However, in reflection, refraction and caustic computations rays are not so coherent, but we need to trace just a single ray from each of many origins. Graphics processing units (GPU) trace rays of the same origin very efficiently by taking a "photo" from the common origin point, but they are far slower to process incoherent rays.

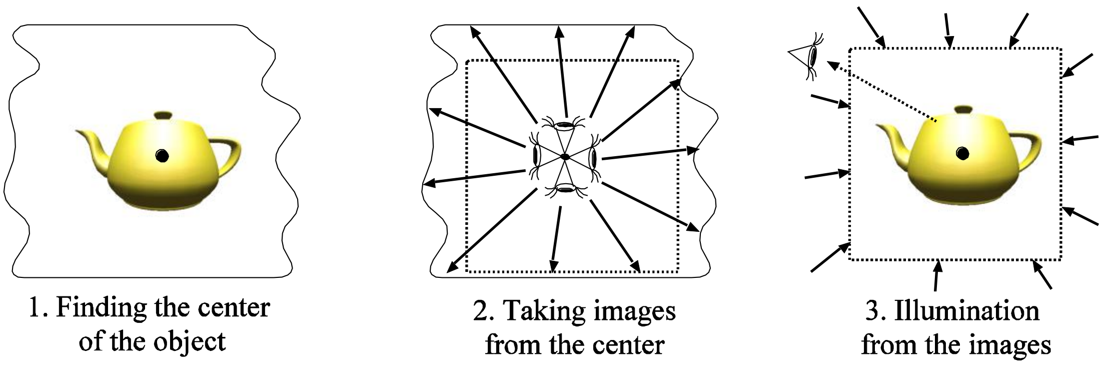

1. Finding the center of the object
2. Taking images from the center
3. Illumination from the images

Figure 1. Steps of environment mapping

A GPU friendly approximation technique to compute reflection is *environment mapping* [Blinn76], which assumes that the hit point of the ray is very far, and thus it becomes independent of the ray origin. In this case reflection rays can be supposed to share the same *reference point*, so we get that case back for which the GPU is an optimal tool. To render a reflective or refractive 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 textured by these images (figure 1). When the incoming illumination from a direction is needed, instead of sending a ray, the result is looked up in the images constituting the environment map.

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.

To attack this problem, we alter the environment map lookup to provide different local illumination information for every point, based on the relative location from the reference point. At a given time, we have just a single environment map for a reflective object, but make localized illumination lookups. Localized image based lighting has also been proposed by Bjorke [Bjorke04], 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. Unlike Bjorke's approach, we rely solely on environment map lookups. All the geometric information required by the localization process is stored in the environment map. This information is the distance of the source of the illumination from the reference point where the environment map was taken. The algorithm is simple and can be executed by current vertex and pixel shaders at very high frame rates. In the following sections we present the basic idea and the DirectX/HLSL implementation, finally we also apply the method for caustics generation.

Localization of the environment map

The idea 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.

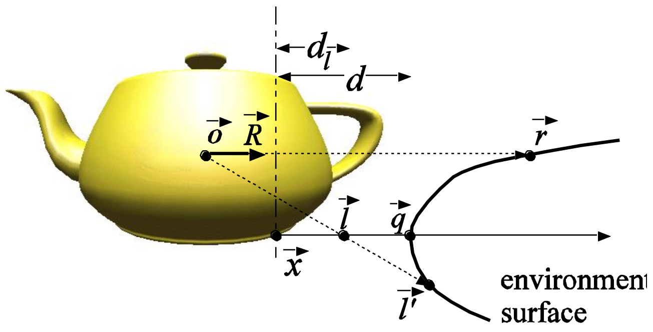


Figure 2. Notations of environment map localization. The reference point of the environment map is $\vec{o}$. We need the illumination of point $\vec{q}$ being at direction $\vec{R}$ and distance d from shaded point $\vec{x}$. The approximation method finds ray parameter d_l, and consequently point $\vec{l}$ on the ray and point $\vec{l}'$ on the surface.

When shading point $\vec{x}$ 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*.

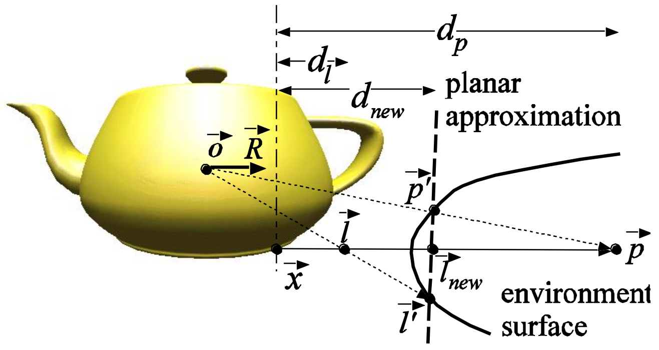


Figure 3. Iterative refinement of the ray parameter assuming that the environment is planar between points $\vec{p}'$ and $\vec{l}'$ corresponding to overshooting point $\vec{p}$ and undershooting point $\vec{l}$, respectively.

Suppose that we have 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 their projections $\vec{p}'$ and $\vec{l}'$ onto the surface from the reference point. (figure 3). The accuracy of this approximation can also be checked by reading the distance stored with the direction of $\vec{l}$ in the environment map ($|\vec{l}'|$) and comparing it with $|\vec{l}|$. If point $\vec{l}$ were on the surface, then the two distances would be equal. If visible point approximation $\vec{l}$ is in front of the surface, that is $|\vec{l}|<|\vec{l}'|$, the current approximation is an *undershooting* of distance parameter d. On the other hand, the case when point $\vec{l}$ is behind the surface ($|\vec{l}|>|\vec{l}'|$) is called *overshooting*.

If the two guesses were of different type (i.e. one is an undershooting while the other is overshooting), then the environment surface should intersect the ray between these two points. In order to find a better approximation we assume that the environment surface is a plane between the two guesses and compute the intersection $\vec{l}_{new}$ between the planar surface and the ray. Then the iteration can proceed replacing either $\vec{p}$ or $\vec{l}$ and by $\vec{l}_{new}$ depending on whether $\vec{l}_{new}$ is an overshooting or undershooting.

In order to compute the intersection point we have to consider the cases separately when projected point $\vec{l}'$ is equal to $\vec{r}$ since in this case the ray parameter becomes infinite. In this case the new ray parameter approximation is

$$d_l = d_p + |\vec{r}| \cdot \left(1 - \frac{|\vec{p}|}{|\vec{p}'|}\right). \tag{2}$$

If none of the two points is equal to $\vec{r}$, then the ray parameter corresponding to the hit point is:

$$d_{new} = d_l + (d_l - d_p) \cdot \frac{1 - |\vec{l}|/|\vec{l}'|}{|\vec{l}|/|\vec{l}'| - |\vec{p}|/|\vec{p}'|}. \tag{3}$$

Having found the new ray parameter, we can obtain the point on the ray, then looking up the environment map, the point projected onto the environment surface.

To start the iteration process we need an undershooting approximation and an overshooting approximation. Note that point $\vec{r}$ corresponds to infinite ray parameter, thus it is the projected point of a sure overshooting. On the other hand, if the object does not intersect the environment, then shaded point $\vec{x}$ is an undershooting.

From mathematical point of view, the proposed iteration method solves the ray equation with the *false position root finding method* [Weissten03], which converges surely. Note that even with guaranteed convergence, the proposed method is not necessarily equivalent to exact ray tracing in the limiting case. 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, 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.

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 object vertices (`pos`) to normalized screen space by the model-view-projection transformation (`TMVP`), and also to the coordinate system of the environment map first applying the modeling transform (`TM`), then translating to the reference point (`refpos`). View vector `V` and normal `N` are also obtained in world coordinates. Note that the normal vector is transformed with the inverse transpose of the modeling transform (`TMIT`).

```
OUT.hpos = mul(TMVP, IN.pos);       // to normalized screen space
float3 xw = mul(TM, IN.pos).xyz;    // to model space
OUT.x = xw - refpos;                // to space of environment map
OUT.N = mul(TMIT, IN.norm).xyz;     // normal vector
OUT.V = xw – eyepos;                // view vector
```

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 pun = length(x)/texCUBE(mp, x).a;      // |p|/|p’|
   float dun = 0, dov = 0, pov;
   float dl = rl * (1 – pun);                   // eq. 2
   float3 l = x + R * dl;                       // ray equation
   for(int i = 0; i < NITER; i++) {             // iteration
      float llp = length(l)/texCUBE(mp,l).a;    // |l|/|l’|
      if (llp < 0.999) {                        // undershooting
         dun = dl; pun = llp;                   // last undershooting
         dl += (dov == 0) ? rl * (1 - llp) :    // eq. 2
               (dl-dov) * (1-llp)/(llp-pov);    // eq. 3
      } else if (llp > 1.001) {                 // overshooting
         dov = dl; pov = llp;                   // last overshooting
         dl += (dl-dun) * (1-llp)/(llp-pun);    // eq. 3
      }
      l = x + R * dl;                           // ray equation
   }
   return l;                                    // computed hit point
}
```

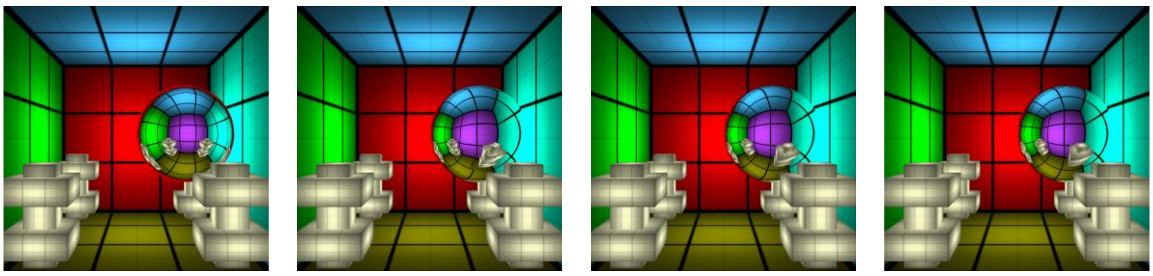
This function gets ray origin `x` and direction `R`, as well as cube map `mp`, and returns hit point approximation `l`. We suppose that the distance values are stored in the alpha channel of the environment map. Note that variables `dun` and `dov` store the last undershooting and overshooting ray parameters. If there has been no overshooting approximation, point $\vec{r}$ takes the role of the overshooting point.

The pixel shader calls function `Hit` and looks up the cube map again to find illumination `I` of the visible point, and computes the reflection by multiplying with the Fresnel function:

```
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;  // radiance of the hit point
return I * Fresnel(N, R);           // reflected radiance
```

We applied an approximation of the Fresnel function, which is similar to the Schlick’s approximation [Schlick94] 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 [Lazanyi05]:

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



Standard environment map 260 FPS | distance impostor with 1 iteration 202 FPS | distance impostor with 3 iterations 140 FPS | ray traced reference

Figure 4. 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.

Figure 4 compares the images rendered by the proposed method with standard environment mapping and ray tracing. 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. The FPS values are measured with 1024×1024 resolution on an NV6800GT.

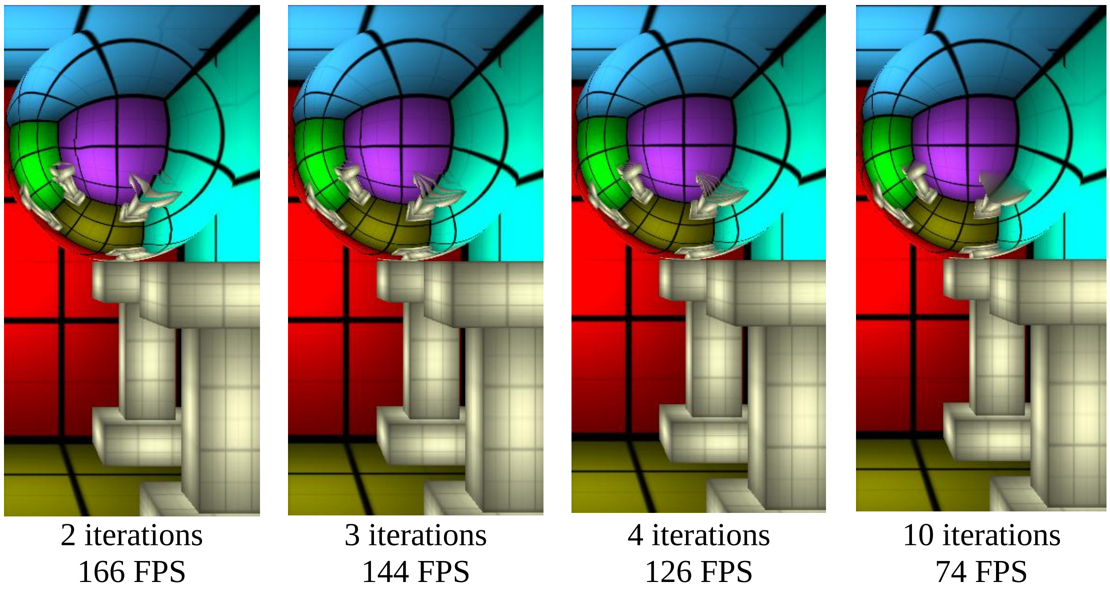


Figure 5. A more difficult case when the tops of the columns are not visible from the reference point of the environment map.

Figure 5 shows a more difficult case where the columns are bigger. Note that the convergence is still pretty fast, but the reflection of the top of the columns is not exactly what we expect. We can observe that the edge of the column is blurred, because the top of the column 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 column and substitutes the reflection of the occluded points by the blurred image of the edge.

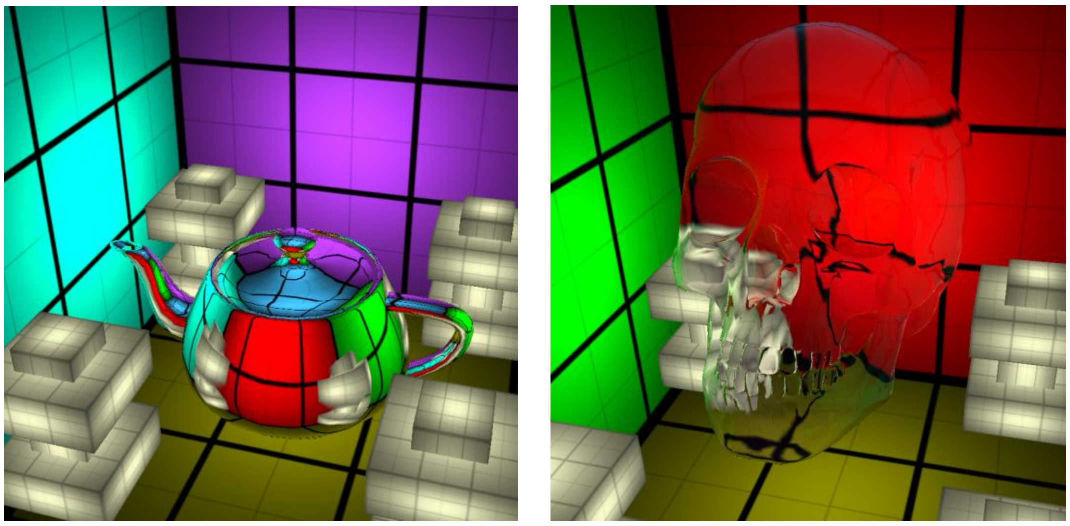


Figure 6. Reflective and refractive objects.

The proposed method can be used not only for reflection but also for refraction calculations if the `reflect` operation is replaced by the `refract` function in the pixel shader (figure 6).

Application to caustics generation

The method presented so far can compute the hit point after the 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 ideal bounce of the light ray, which is the cause of caustic effects [Szirmay05].

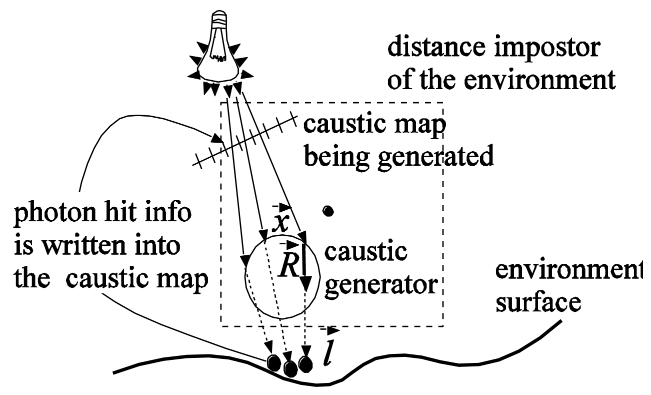


Figure 7. Caustics generation with environment maps

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 7). 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.

Supposing that the surface is an ideal reflector or refractor, point $\vec{l}$ that receives the illumination of a light source after a reflection or refraction can be obtained by the proposed approximate ray tracing, and particularly by calling the `Hit` function. 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, 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 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 L to the coordinate system of the environment map.

```
OUT.hpos = mul(TMVP, IN.pos);       // to normalized screen space
float3 xw = mul(TM, IN.pos).xyz;    // to model space
OUT.x = xw - refpos;                // to space of environment map
OUT.N = mul(TMIT, IN.norm);         // normal vector
OUT.L = xw - lightpos;              // light vector
```

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);             // hit point of the photon
float3 hituv = texCUBE(uvmap, l).xyz;     // uv of the hit point
return float4(hituv, power);              // store into 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, 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); // photon hit uv
OUT.filtcoord = IN.pos.xy;                     // filter coords
OUT.texcoord.x = ph.x + IN.pos.x / 2;          // billboard uv
OUT.texcoord.y = ph.y - IN.pos.y / 2;
OUT.hpos.x = ph.x * 2 - 1 + IN.pos.x;          // billboard address
OUT.hpos.y = 1 - ph.y * 2 + IN.pos.y;
OUT.hpos.w = 1;
if (ph.a < 0) OUT.hpos.z = 2;                  // not a hit: ignore
else          OUT.hpos.z = 0;                  // valid hit:
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.

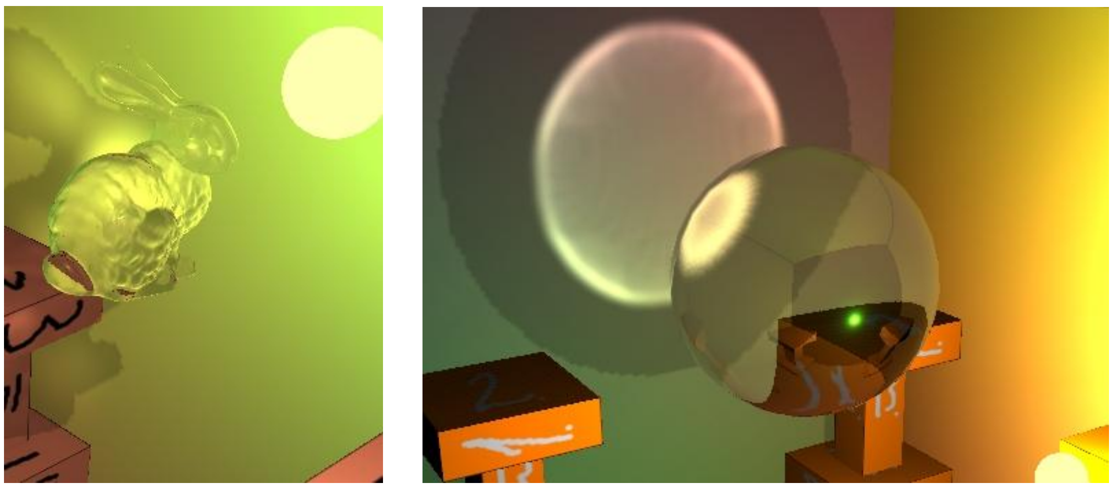


Figure 8. Real-time caustics caused by a glass sphere ($n = 1.3$), rendered by the proposed method at 50 FPS

Figure 8 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 executing 10 iterations runs with 50 FPS.

Conclusions

We proposed a localization method for environment maps, which uses the distance values stored in environment map texels. 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 reflections and refractions on curved surfaces, and caustics.

References

[Blinn76] Blinn J. F., Newell M. E. Texture and reflection in computer generated images. Communications of the ACM 19, 10 (1976), pp 542-547.

[Bjorke04] Bjorke K. Image-based lighting. In GPU Gems. Fernando R. (editor). NVidia, (2004), pp 307-322.

[Weisstein03] Weisstein E. W. Method of False Position. MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/MethodofFalsePosition.html.

[Schlick94] Schlick C. A customizable reflectance model for everyday rendering. In Fourth Eurographics Workshop on Rendering. (1993), pp. 73-83.

[Lazanyi05] Lazányi I., Szirmay-Kalos, L. Fresnel term approximations for metals. In WSCG 2005. Short papers. (2005), pp 77-80.

[Szirmay05]: Szirmay-Kalos, L. Lazányi, I., Aszódi, B., Premecz, M. Approximate ray-tracing on the GPU with distance impostors. Computer Graphics Forum 24, 3 (Eurographics 2005 Proceedings).