Real-time multi-bounce many-object ray tracing with distance-normal impostors

Peter Dancsik*
Peter Minarik†

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary

Abstract

This paper presents a fast approximate ray-tracing technique for the GPU. We extend previous distance impostor techniques to handle scenes with multiple reflective and refractive objects in real time. There are two key ideas that allow this. First, we turn around the distance impostor approach not to intersect internal rays with enclosing environmental geometry, but external rays with an object. Texture maps representing refractive objects are labelled *refractor maps*. To remedy problems with cube maps we propose a height-and-normal map representation, which we call a *refractor height map*, and describe an algorithm to find multiple refracted ray bounces within it. The second idea is the separation of static and dynamic objects. Classic distance impostors can be used for the static environment, and only those of moving objects need to be updated in every frame. Light paths passing through moving objects can be found by searching their refractor maps. We demonstrate the proposed techniques in a chess application featuring two full sets of glass pieces, with only one piece moving at a time. Shadows, reflections, refractions and caustics are simultaneously rendered in real time.

Keywords: ray tracing, distance impostors, caustics

1 Introduction

There is an ongoing effort to achieve ray tracing effects in interactive applications. Full GPU ray tracers could not produce high frame rates in a complex environment with lots of polygons. According to Whitted[8] 95 percent of the time of the ray tracing was the calculation of the intersection points. Although this number is getting lower due to the high level parallelization in the graphics card, the rule about the intersection calculation being a bottleneck still applies. If accuracy is an important requirement, calculating the intersection of the ray with the triangles is unavoidable. However, in case of interactive, frame rate sensitive, real time applications such as computer games, trading the correctness for a good approximation and better performance is always a good deal.

Let us examine the architecture. The fragment shader renders the pixels independently, allowing for massive parallellization and high performance. On the other hand, this means that the individual pixels have only local information, sufficient for the local illumination model. Conversely, in the global illumination model the color of a pixel highly depends on its environment. This kind of information could be delivered to the shader as textures. The GPU traces rays of the same origin efficiently by rasterizing the scene as seen from the origin. During rasterization the nearest intersection points can be calculated by the z-buffer mechanism. From a specific *reference point* we take pictures of the environment in all the six directions with a ninety-degree field of view angle. This technique is called environment mapping. The map is a discrete sampled representation of radiance incoming from the real environment.

The information stored in an environment map is only accurate at the reference point. The farther we go from the reference point the less the map approximates the environment from that specific point. If the map contains only environment elements from an infinite distance (such as the sky), this error vanishes. In our case the immediate environment is also in the map, thus the environment mapping gives acceptable approximations only for dot-like objects. Therefore, we need to estimate where the environment surface point actually seen at a given direction from a shaded point is, and the direction to that point from the reference point. Having this direction vector, we can read the color from the environment map to get the incoming radiance at the shaded point.

2 Previous work

It is possible for ray tracers to achieve real-time performance without approximating the intersection computation. Both CPU and GPU implementations exist. [7] reviews the state of the art of ray tracing. [9] proposes an effective implementation of the kd-tree on the GPU. Hereafter we will deal with ray tracing effects achieved by en-

*petrovdancsikov@gmail.com
†peter.minarik.84@gmail.com

vironment mapping.

In [1] the environment is approximated with a proxy geometry (for instance a sphere). We determine the intersection of the sphere and the ray starting from the given point in the given direction, after that we read the environment map at the direction of the intersection point. If the proxy does not approximate the geometry of the scene well enough or the size of the scene is much greater than the individual objects in it, the approximation does not give an accurate result.

In [5] *distance impostors* are used which store not only the environment information but the distance from the reference point as well. Knowing the distances between the visible points of the environment and the reference point, we can determine the position of an environment point seen from a given point in a given direction. As our approach is based on this work, we will elaborate on the method in Section 3.

In [6] the environment information is stored in *layered distance maps*. Taking pictures of the environment the first layer will contain the nearest intersection point of the environment and a ray started from the eye through the pixels of the screen, the second layer will contain the second nearest intersection points and so on. The distance from the reference point and other surface information are stored as well. This kind of solution is an extension of the environment map. The solution above which is using distance impostors is like a one-layered distance map. The main disadvantage of the technique is that during the calculation of the intersection points search has to be done in all the layers and when the scene changes everything has to be recalculated. Thus, the method is only real-time for a single reflective or refractive object.

Both above approximate ray tracing methods assume a single reflective or refractive object in a locally shaded environment. Multiple objects are handled by treating each one as part of the environment of the others. While this offers proper multiple reflections and refractions, in a dynamic scene, all environment maps have to be entirely updated in every frame. Even with just a few refractive objects, this will fail to be real-time.

Our approach is to store only the static environment in the environment maps thus they do not need any recalculation during runtime (see Section 5.2). To determine whether the static environment is occluded by the objects of the dynamic environment, ray-object intersections must be calculated for all dynamic objects. The intersection calculation method depends on the representation of the object. In this paper we propose two method (see Section 4 and 6).

3 Intersection computation with distance impostors

Figure 1 shows the environment around reference point $\vec{o}$. The vectors are transformed to the reference point's coordinate system. We want to compute the incoming radiance at point $\vec{x}$ from direction $\vec{R}$. Ray tracing would select the intersection point point $\vec{q}$, classical environment mapping would read the texel at direction $\vec{r}$. Reading the environment map from direction $\vec{q}$ we would get exactly the correct result. We know that point $\vec{q}$ is on the ray, therefore we must choose points that satisfy the ray equation ($\vec{x}+d\cdot\vec{R}$, $d \geq 0$).

Like in [5] we did not only store the color values in the environment map but the distances from the reference point coded into the alpha channel as well. Now we have information about the environment geometry. Maps having this kind of distance information are called *distance impostors*. We use the name *distance-normal impostor* for maps that store normal vectors as well.

Having the distance information one can tell about an approximation how well it estimates the intersection point. Such an approximation is $\vec{y}$ in Figure 1. We read the environment map's alpha channel at the direction of point $\vec{y}$ to get $\vec{y}'$. Note that ratio $|\vec{y}|/|\vec{y}'|$ expresses the accuracy of approximation point $\vec{y}$. If the ratio equals 1, point $\vec{y}$ is exactly the hit point. If the ratio is less than 1, $\vec{y}$ is an *undershooting*, i.e. the point is in front of the surface. On the other hand if the ratio is greater than 1, $\vec{y}$ is behind the surface (we call this case *overshooting*). We must choose a fast strategy to find a hit approximation using the fewest steps possible.

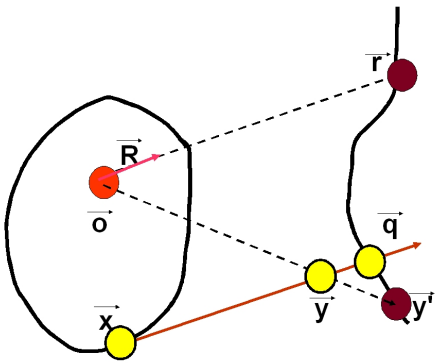

Figure 1: By querying the environment map in the direction of approximate intersection point $\vec{y}$, the result is surface point $\vec{y}'$. Storing distance values one can estimate the correctness of the approximation by calculating the ratio $|\vec{y}|/|\vec{y}'|$.

In [6] linear search is used to find the first undershooting and overshooting pair. Then secant search computes a more accurate result.

We used the method described in [5], which uses only secant search. Figure 2 shows the search process. To find

the first guess for the ray hit, we assume that the environment surface at point $\vec{r}$ is perpendicular to $\vec{R}$. We identify the intersection of the surface and the ray, that will be point $\vec{p}$. By reading the environment map's alpha channel at direction $\vec{p}$, we can get $\vec{p}'$. To get the second approximation point $\vec{l}$, we assume that the surface is planar between points $\vec{r}$ and $\vec{p}$. $\vec{l}$ is on the ray ($\vec{l} = \vec{x} + dl \cdot \vec{R}$), and on the line defined by $\vec{r}$ and $\vec{p}'$ ($\vec{l} = t \cdot \vec{r} + (1-t) \cdot \vec{p}'$). Solving the equation system we get $\vec{l}$, and compute $\vec{l}'$ from the distance value read from the environment map at direction $\vec{l}$.

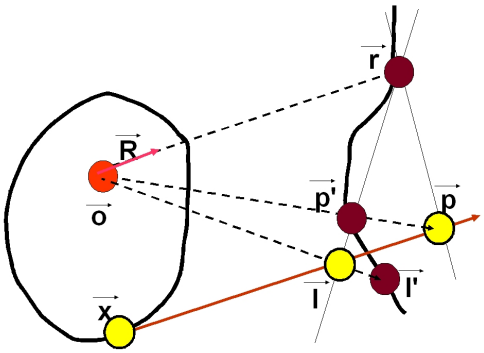

Figure 2: Finding the first two approximations in a distance impostor. Classical environment mapping would select $\vec{r}$. Assuming that the surface at $\vec{r}$ is perpendicular to $\vec{R}$, we can retrieve $\vec{p}$. Reading the environment map in direction of $\vec{p}$ we get the surface point at $\vec{p}'$. To determine $\vec{l}$ we assume that the surface is planar between $\vec{r}$ and $\vec{p}'$. This results the approximated surface point $\vec{l}'$.

The two obtained approximations can be refined by iterations. We must find the intersection of the the ray with the line defined by the last overshooting and undershooting point. Since point $\vec{r}$ corresponds to an infinite ray parameter, which is surely an overshooting point, we can use this ideal point at infinity if there are no other overshooting points. On the other hand, if there is no undershooting point, we can use $\vec{x}$ or substitute with $\vec{r}$.

At this point we can trace light paths of depth two. The primary rays are identified by rasterization, and with the presented method we can compute the secondary rays. The vertex shader calculates the world space position, the normal and the view vector. The fragment shader determines the refracted and the reflected rays and runs the search method with the rays separately. The final color is the combination of the reflected and refracted color determined by an approximation of the Fresnel function.

To compute multiple refractions we need to know the geometry of the refractor object (i.e. the position of the surface points, and the normal vectors at these points). While ray tracing efficiently determines the intersection with implicit surfaces (e.g. sphere), we can get a better approximation by using the proposed method. For each refractor we create a *refractor distance map*, which stores the distance of the surface from its center and the normal vector of the surface. If the refractor has static geometry, it is sufficient to compute the refractor map only once during preprocessing. Figure 3 shows the search process.

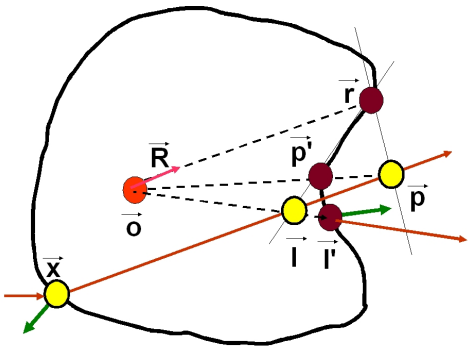

Figure 3: Search in a refractor distance map. In this case the enclosing environment is the geometry of the refractive object. The calculation of the refraction direction at the surface points is made possible by storing the normal vectors.

There are difficult cases when the search method does not find an appropriate result. This phenomenon occurs when the intersection point to be located is not visible from the reference point. That means the environment or the refractor in case of refractor distance maps is strongly concave. We deal with this problem in Section 6.

4 Ray-object intersection using refractor distance map

A *refractor distance map* is a distance-normal impostor cube map that contains the geometry information of a single refracting/reflecting object.

In the following subsection we introduce a method to search for an intersection in refractor distance maps along a ray started from the outside. Thus we will be able to compute intersection between an external ray and an object represented by a refractor distance map. That gives us the possibility to construct a multi-bounce ray tracer using exclusively refractor distance maps (see Section 4.2). In Section 5 we will present a faster way to trace rays of multiple depths wherein the environment maps do not contain the dynamic environment and intersection computation with dynamic objects is handled separately by using the following search method.

4.1 Search in refractor maps from the outside

One can determine the intersection point of a refractor and a ray from the outside knowing only the refractor distance map. In order to do this, first we have to calculate the intersection points with the bounding box of the refractor.

If there are none, then neither will there be any intersections with the refractor. If there are such points, we have to search in the refractor distance map (see Section 3). We will use a ray starting from the farthest intersection point but opposite to the direction of the ray used to calculate the intersections (see Figure 4). The search method selects a point on the surface even if the ray actually does not intersect the object itself. Because of this we say that a point (returned by the search method) is invalid when it is farther from the ray than a given small epsilon value.

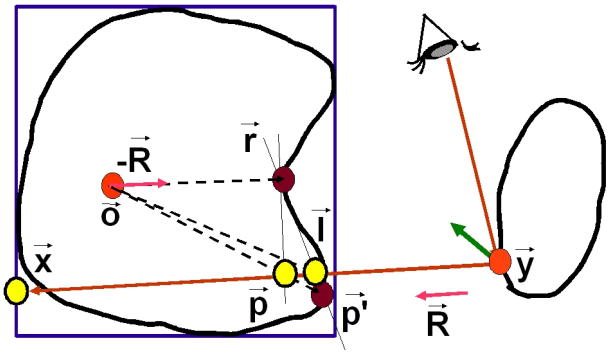

Figure 4: Computing the intersection point of a refractor and a ray ($\vec{y} + d \cdot \vec{R}$, $d \geq 0$) from the outside. The point is found as the intersection between the ray described by the ($\vec{x} - d \cdot \vec{R}$, $d \geq 0$) equation and the enclosing environment (the geometry of the refractor), where $\vec{x}$ is the farther intersection point with the bounding box.

4.2 Multiple ray bounces with refractor distance maps

The scene we are going to use as an example consists of a chessboard and 32 glass chess pieces on it. Since there are multiple refracting/reflecting objects in the scene there is a definite need for handling rays in multiple depth levels.

It is possible to calculate multiple ray bounces using the method discussed above. The main advantage of this is that all we need are the refractor distance maps. Since there are six kinds of pieces on the chessboard (pawn, rook, knight, bishop, king, queen), six refractor distance maps are enough to have. As this is not a lot, the resolution of the maps can be higher than in the case of individual environment maps for all pieces. This results in a more detailed approximation of the surfaces. As the objects have a static geometry, we can calculate these maps during the initialization and use them afterwards without the need to recalculate any of them.

The first intersection point is determined by rasterization. To get the next intersection point in the reflection direction, the search has to be done for 31 pieces in their refractor distance maps, selecting the nearest intersection. One can determine the normal vector and intersection point reading the refractor distance map. With the normal vector the reflection direction can be calculated as well, so we could continue the algorithm as described above. If there is no intersection with the refractor, we have to check whether the ray reaches the diffuse environment, meaning the chessboard in our case.

If we have N refractors $N-1$ searches are needed to calculate the intersection. The problem is that even with low Ns we will not get any real-time results. But we can speed up the algorithm by partitioning the search domain. A uniform grid aligned on the squares of the chessboard is a natural choice. To simplify the grid data structure we can also assume that there can only be one piece in one square. Even so, a full search in the grid was not feasible in a single shader. The algorithm we could implement searches for the first non-empty square along the ray and examines if there is any intersection with the refractor located there. If none was found, an intersection point with the plane of the chessboard will be determined, ignoring further squares.

5 Multiple ray bounces with multiple distance impostors

In Section 3 we re-iterated the approach described in [5] to trace rays of depth two, where the final color is the combination of the colors read from the environment map at the reflected and refracted ray directions. Note that if the environment map contains the incoming radiance of multiple bounces, we can read the contribution of complete light paths. Thus, we are able to trace rays of multiple depths reflections and refractions by computing three intersections (one in the refractor distance map to find the refracted ray, and two in the environment map along the reflected and refracted rays).

Every piece needs its own environment map since the environments they can see are significantly different. The environment map's pictures are taken from the center of the 32 pieces so we will have 32 environment maps and 6 refractor distance maps for each kind of chess piece. Because of this high number we cannot use detailed environment maps with high resolutions.

5.1 Computing the environment distance impostors

To get a distance impostor which contains the incoming radiance of multiple bounces we use the method described in [2]. First we have to render the diffuse environment such as the sky and the chessboard to each environment map. Next we render the surrounding pieces into each and every environment map. For this we search in their environment and refractor distance maps as described above.

After these the environment maps will represent at least two ray bounces. Repeating the rendering of the pieces for each environment map we will have one more ray bounce in the maps since the depth of the environment maps has increased. Repeating this step one can have an environment map of any desired depth level. The realized algorithm is represented by this pseudo code.

```
for each environment map:
    render chessboard
    render sky

for DEPTH times:
    for each environment map:
        for each surrounding chess pieces:
            load its environment map
            load its refractor map
            render the piece
```

In case of a static environment it is enough to calculate the environment maps only once during the initialization phase. If we are not lucky enough to have static environment we have to recalculate the maps. In order to save some calculation we do not need to do this every time we render a frame, only when the rate of changes makes it necessary for the authentic representation of the environment. In our scenario it is the movement of the chess pieces that causes the changes. Calculating every map for every frame the application would be unacceptably slow. For increased efficiency we used the following technique.

5.2 Moving objects

It is clear that we have to recalculate the environment maps of the moving objects (if the rate of changes makes it necessary). But for static pieces this rule does not apply. Only the static environment (non-moving pieces, chessboard, sky, etc.) will be rendered into their environment maps, so that they do not need to be updated. When rendering a non-moving piece it is not enough to search in its environment map, since a moving object could be in the way. The intersections with moving pieces have to be determined. As shown in Section 4.1, we start a search from outside of the refractor distance map of the moving objects. The intersection point at the shortest distance is required. As in chess only one piece is allowed to move at a time, calculating only one environment map dynamically is enough.

Figure 5 shows how to determine the intersecting points in the direction of refraction and reflection. If neither in direction of refraction(3) nor in direction of reflection(1) there is an intersection with the bounding box of the moving object, the intersections with the static environment are kept. In Figure 5 a search has to be executed from the reflection direction in the refractor distance map of the chess piece. There is an intersection point(4), thus three further searches(5-7) have to be done in the maps of the moving object in order to find the intersection points in the direction of refraction and reflection.

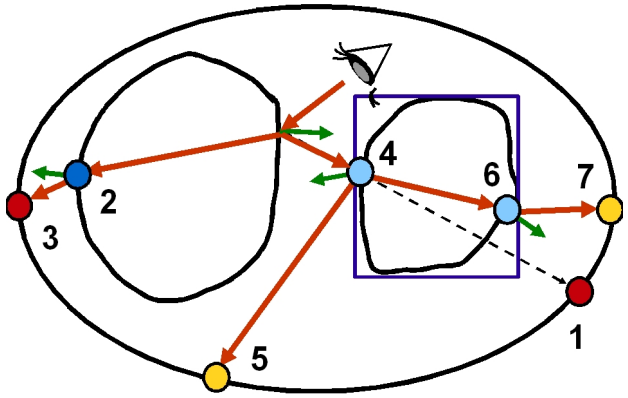

Figure 5: Separating the moving piece from the static environment. During the shading of a static chess piece the intersection points with the static environment (1, 3) could be occluded by the moving chess piece. This could be checked as proposed in Section 4.1. In our case the moving object is seen in the reflection direction (see 4), so 3 more searches are required to find intersection points 5 and 7.

6 Refractor height maps

When generating distance impostors, we store the sampled information about the surface as observed from the reference point. If the environment appears concave from the reference point there will be some part of the surface that could not be represented in the map. During ray tracing only the environment represented by the map is known. We cannot determine any intersection point if the ray intersects the surface at a point that is not represented by the map. There could be errors by searching in the environment map if we have a concave scene. If the object itself is concave the error occurs when searching the refractor distance map. Layered distance maps (see Section 2) can handle a concave environment, but that solution is too slow for a dynamic scene.

Let us take pictures from the center of a chess piece in all the six directions of it. What would we see? We realize that all of them have concave shapes. This means that there would be missing surface parts in their refractor distance map. These errors could be well seen on the screen so we have to choose another method to represent the surfaces of the chess pieces.

Let us pause a bit and take a good look at the pieces again. We could discover that all of them have at least one plane of symmetry. Taking a picture of the piece with an orthogonal camera perpendicular to the plane of symmetry and storing the distances of the surface points from the plane in a map, we would actually get a height map. Chess pieces have shapes that in pictures taken of them in the way explained above, there would be only a few or no points at all that are covered by another point of the sur-

face of the shape. As the piece is symmetric, the height map describes the whole surface of the piece. Besides the distances, we need the normal vectors as well. The refractor distance map is replaced by this height map called *refractor height map*. All we need is a method to determine the intersection point of the ray and the surface described by the height map.

Imagine that we have a height map textured onto a billboard. This billboard is put into the position of the chess piece, has the same size as the piece itself and always rotates towards the camera around the axis perpendicular to the chessboard (see Figure 6). The billboard and its height map determine a three-dimensional surface. This is a proxy surface, we do not expect perfect substitution unless we have objects with axial symmetry.

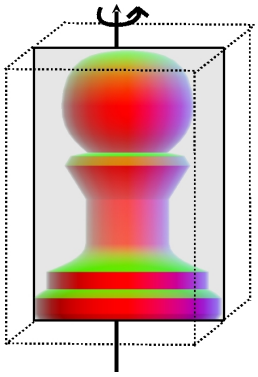

Figure 6: The refractor height map. It stores the normals of the surface points and the distances between the surface points and the plane of symmetry. It is bent on a billboard that always rotates towards the camera around the vertical axis.

The first step is to find the segment of the ray that is inside the texture domain of the height map. It is important that only that part of the geometry is used in this method that is between the plane of symmetry and the camera.

Now we want to calculate the intersection points with the plane of the billboard and with the bounding box of the proxy (see Figure 7). If the ray does not intersect the bounding box, then there will be no intersection. Let us introduce some distances from the origin of the ray. Let d_{plane} be the distance to the intersection point on the plane. d_{near} and d_{far} will be the distances to the bounding box's two intersection points ($d_{\text{near}} < d_{\text{far}}$). If d_{plane} is less than d_{near} we examine the ray segment within the bounding box (see case 3). Otherwise the segment between the two nearest points from the ray origin has to be examined (see case 1 or 2). If the segment examined is behind the billboard (case 3a) it has to be reflected to the plane in order to be able to search in the valid domain.

To find the intersection point with the height map we use *binary search* (see [3], [4]). It is simple and fast though it will not necessarily return the nearest intersection point. Nevertheless we observed no visible artifacts. This is explained by the fact that the billboard is always rotated to face the origin of the ray, thus rays never arrive at a steep angle.

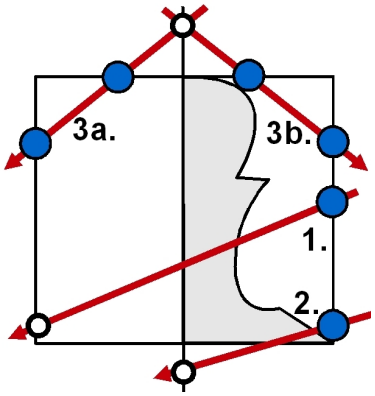

Figure 7: Locating the ray's useful segment (which is in the height map's texture space). If the segment is behind the billboard (case 3a), it will be reflected to the plane (case 3b).

Figure 8 shows an example how to calculate an intersection point. We run the search over the segment of the ray determined above. If the search proves to be successful we have an intersection point. If it does not and the search was executed between the intersection points with the bounding box, then there is no intersection point since the ray leaves the box. Otherwise the search has to continue in the space behind the plane of the billboard. That is why the ray has to be reflected to the plane of the billboard and the valid domain needs to be recalculated. The search has to be executed once again. Finally the intersection point has to be reflected to the plane of symmetry again. If there is not any intersection point this time either, the ray does not intersect the chess piece at all. If we do have the intersection point, we have to follow the refracted ray direction. The normal vector is known at the intersection point since the refractor height map contains it, so the refraction direction can be calculated. The intersection with the refracted ray can be found by repeating the same search algorithm that we used for the incoming ray.

In the algorithm described in 5.2, the refractor distance map can be replaced by the refractor height map. Search can be executed from an outer point and the calculation of multiple refractions is possible as well.

7 Caustics

Refracting and reflecting materials can focus the light passing through them or reflected on them, causing the surface to be illuminated more intensively. In our case the chessboard has to be illuminated this way. This indirect illumination effect is called caustics.

We use the method described in [5] to generate the caustics. The caustic effect has to be calculated for each chess piece individually. In the first pass we determine the set of

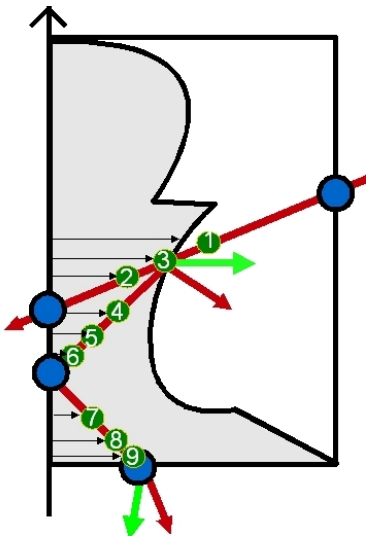

Figure 8: An example for the search process. For every binary search the first 3 steps are marked.

points on the surface that has to be brightened. For this, shooting light paths starting from the light source have to be traced. Pictures are taken of a refractor from the position of the light source. The rendered image is called a *photon map*. The intersection point with the refractor is found by rasterization. After that, multiple refraction is calculated by searching in the refractor map. For the sake of simplicity and performance we neglect the fact that the ray might interact with other pieces as well, only the intersection with the chessboard (so-called *photon hit*) is calculated. The photon hit is stored in texture space. It is easy to calculate the texture coordinates of the surface points of the chessboard, but in case of a more complex scene the environment map has to contain the texture information as well.

In the second pass caustics is rendered into the so-called light map using the information stored in the photon map. This light map will modulate the color texture of the chess board. Creation of the caustics is done by using photon splatting: around every photon hit a semi-transparent quadrilateral is drawn, which corresponds to Gaussian filtering the surroundings of the hit point in texture space. In order to accumulate the effect of the quadrilaterals drawn in the light map, additive blending mode has to be set.

For each chess piece shadow will be calculated by shadow mapping. The result goes into the light map as well. Two different light maps are used, one for the static pieces the another for the moving piece. Only the light map of the moving piece has to be updated, it is enough to calculate the light map of the static pieces only once.

8 Results

One would eventually ask if all the methods described above are worth using or not. To answer to this question we made some measurements. To determine the efficiency of these methods a chessboard scene with 33 glass pieces (a full chess set plus a sphere of glass) was rendered in a 600×600 resolution. The scene contains 30838 vertices and 47404 faces altogether. The refractor height maps were made in a 512×512 resolution, while the environment maps have 256×256 texels. For the tests an AMD Athlon 64 X2 Dual Core 4600+ CPU with 2046 MB RAM and an NVIDIA GeForce 8800GTX graphics card was used.

When there were only static pieces in the test scene, the maps did not require any recalculation during runtime thus 75 FPS was achieved. In case of a moving piece, we run several tests to compare the performance of the described methods. In the first test, every environment map contained the whole environment. As the environment maps were updated in every frame the FPS rate dropped to a 1.8 level. In the second test, with our proposed method, where the static environment and the moving piece were separated we got 11.3 FPS, which is approximately 6 times faster than the original solution. By increasing the resolution of the refractor and environment maps a better sampling quality of the environment can be achieved. Using 512×512 environment maps we got 7.5 FPS. This technique is pixel shader sensitive, so the more pixels are shaded with the glassy effect, the lower the frame rate falls.

Figure 9 shows our presented approximation compared to a classical ray tracer. In the pictures the moving piece is a reflective pawn which can be seen through a translucent sphere. A differential image shows the error of the approximation. Because of the concave environment the environment maps cannot represent the entire scene. This causes errors (labeled 1 in the figure) when searching in these maps. After computing the intersection with the moving piece we read the environment map at the reflection direction and neglect that the reflected ray might interact with the moving object itself (label 2). Error 3 occurred because the texture of the chessboard was differently sampled by the two renderers (in our implementation the texture of the chessboard is more blurry).

9 Conclusion

The results of the measurements show the effectiveness of the proposed method. By separating the moving objects from the static environment we receive interactive performance. During the measurements, the environment map of the moving piece was updated in every frame. A higher FPS rate can be achieved if this happens only when the change of the environment has reached a certain amount. Furthermore, the test scene was extremely demanding, with a high number of refractive objects. That is why we conclude that the method can be effectively used in realistic game environments on upcoming hardware.

The efficiency of the proposed method relies on the found representations for the optimal approximation of the

surfaces. In this article we introduced two possible solutions. First, we are able to describe star-shaped surfaces (convex as seen from the reference point). The second representation is applicable to surfaces that can be orthogonally projected without any overlapping to the plane of symmetry. To represent surfaces which are different from those two described above new methods need to be invented. However, one has to consider that the transformation of the surface into texture space has to be a homogeneous linear transformation, to ensure that the image of a line in world space will still be a line in texture space.

References

[1] Kevin Bjorke. Image-based lighting. *GPU Gems*, 2004.

[2] Kasper Hoy Nielsen and Niels Jorgen Christensen. Real-time recursive specular reflections on planar and curved surfaces using graphics hardware. *Journal of WSCG*, 10 (3), 2002.

[3] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. *Proceedings of SIGGRAPH 2000*, 2000.

[4] Fabio Policarpo, Manuel M. Oliveira, and J. L. D. Comba. Real-time relief mapping on arbitrary polygonal surfaces. *ACM SIGGRAPH 2005 Symposium on Interactice 3D Graphics and Games*, 2005.

[5] Laszlo Szirmay-Kalos, Barnabas Aszodi, Istvan Lazanyi, and Matyas Premecz. Approximate ray-tracing on the gpu with distance impostors. *Computer Graphics Forum 24*, 2005.

[6] Tamas Umenhoffer, Gustavo Patow, and Laszlo Szirmay-Kalos. Robust multiple specular reflections and refractions. *GPU Gems 3*, 2007.

[7] Ingo Wald, William R Mark, Johannes Gnther, Solomon Boulos, Thiago Ize, Warren Hunt, and Steven G Parker. State of the art in ray tracing animated scenes. *Eurographics 2007 State of the Art Reports*, 2007.

[8] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM. vol. 23.*, 1980.

[9] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time kd-tree construction on graphics hardware. Technical report, Microsoft Research Asia, 2008.

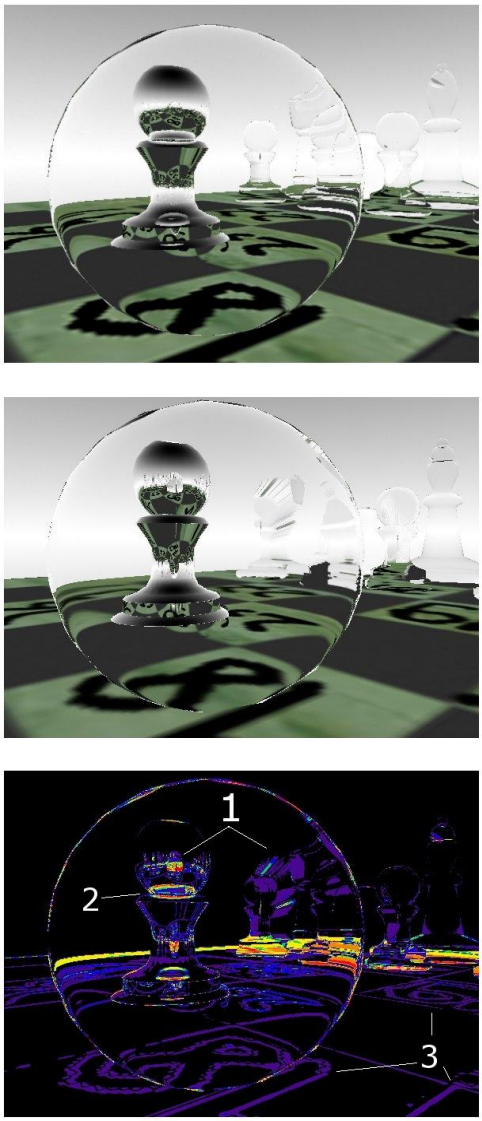

Figure 9: Comparing the proposed approximation to classical ray tracing. The top picture is rendered with the classical method, the picture in the middle shows our approximation. The third picture is a differential image, the intensity of its colors shows the degree of the difference.

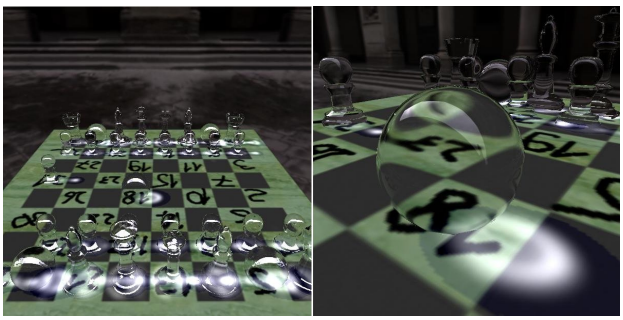

Figure 10: Left: the whole test scene. Right: double refraction using refractor height maps.