Interactive Approximate Rendering of Reflections, Refractions, and Caustics

Wei Hu and Kaihuai Qin

Abstract—Reflections, refractions, and caustics are very important for rendering global illumination images. Although many methods can be applied to generate these effects, the rendering performance is not satisfactory for interactive applications. In this paper, complex ray-object intersections are simplified so that the intersections can be computed on a GPU, and an iterative computing scheme based on the depth buffers is used for correcting the approximate results caused by the simplification. As a result, reflections and refractions of environment maps and nearby geometry can be rendered on a GPU interactively without preprocessing. We can even achieve interactive recursive reflections and refractions by using an object-impostor technique. Moreover, caustic effects caused by reflections and refractions can be rendered by placing the eye at the light. Rendered results prove that our method is sufficiently efficient to render plausible images interactively for many interactive applications.

Index Terms—Interactive rendering, reflection, refraction, caustics, GPU.

1 Introduction

In our daily life, many effects, such as reflections, refractions, shadows, and caustics, give us impressive visual enjoyment. There are several rendering techniques that can render good global illumination images, but, unfortunately, rendering these effects interactively is still one of the major challenges in computer graphics.

Ray tracing [1], path tracing [2], and photon mapping [3] are very useful techniques to render realistic global illumination effects. Although various methods have been developed for speeding up these techniques, a universal interactive method is hard to implement on a common PC.

The methods that use a GPU to compute reflected or refracted directions, and then fetch colors from an environment map or a perturbed texture, were broadly used [4], [5]. However, these methods cannot handle more than one bounce of reflections or refractions. Environment maps were also used for rendering recursive reflections in real time [6]. However, environment maps or perturbed textures cannot describe complex scenes well, especially when objects are not far away from the reflectors/refractors. Hakura and Snyder [7] proposed a hybrid rendering model that combines ray tracing with layered environment maps to render reflections and refractions, but the rendering performance is not sufficient for interactive applications.

Ofek [8] and Schmidt [9] used virtual objects to generate reflection and refraction effects. However, their method cannot deal with concave objects, and even exhibits problems when refracting through convex objects.

A method based on light fields was introduced to render reflections and refractions [10]. However, this method could only be used for static scenes, and the precomputation cost is very high.

In Wyman's methods, an approximate image-space approach for the interactive refraction of distant environments and nearby geometry was introduced. His methods allow refraction of distant environments [11] and refraction of nearby objects [12] through two interfaces. To support the two-interface refractions, preprocessing is required in Wyman's methods.

Rendering caustics is a special global illumination problem. Caustics occur if the light is reflected (or refracted) from a specular surface to a diffuse surface. Ray tracing is one of the earliest techniques to generate caustics [13]. Later, photon mapping was popular to render a variety of lighting effects [3]. Although photon mapping was implemented on graphics hardware [14], it is still far away from interactive rendering. Recently, clusters [15] and shared-memory machines [16] were used to render caustics interactively. Besides previous methods, techniques of rendering caustics interactively, even in real time [17], [18], [19], [20], [21], were developed. However, most of them deal with only one bounce reflection and refraction. Moreover, some methods cannot render complex scenes interactively and preprocessing is usually required. Besides, most of these methods cannot handle reflections and refractions interactively in a unified manner, even though these effects are inevitable in the scenes.

From the above introduction, we can see that few works have been able to render interactive reflections, refractions, and caustics of complex scenes in a unified manner. This paper presents a new technique for interactive rendering of reflections, refractions, and caustics. The new technique extends Wyman's methods [11], [12], and it can compute more accurate refractions of environment maps and nearby objects without preprocessing by using impostors [22] and depth buffer corrections. By examining reflection and refraction from a point light resource, we can also render realistic caustics interactively. Fig. 1 shows some rendering

• *The authors are with the Institute of High Performance Computing, Department of Computer Science and Technology, Tsinghua University, Beijing, China.*
E-mail: huw02@mails.tsinghua.edu.cn, qkh-dcs@tsinghua.edu.cn.

Manuscript received 24 Aug. 2005; revised 15 Jan. 2006; accepted 6 Feb. 2006; published online 8 Nov. 2006.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-0115-0805.

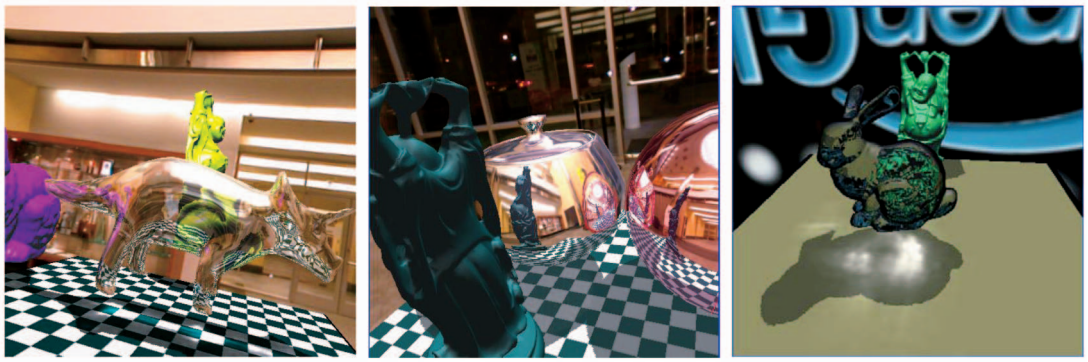

Fig. 1. Interactive rendering. The resolution of images is 512^2. Buddha has 50K triangles. Dinosaur has 21K triangles. Bunny has 70K triangles. Teapot has 2K triangles. Sphere has 1K triangles. (a) Frame rate = 18 fps. (b) Frame rate = 35 fps. (c) Frame rate = 15 fps.

results of our new method. In Fig. 1a, a bunny is reflected and a Buddha is refracted by a dinosaur, and refractive caustics are displayed. Fig. 1b shows recursive reflections. In Fig. 1c, the refractor is a 70K triangle bunny, and there are a Buddha and a plane in the scene. Note that all effects including the reflections, refractions, shadows, and caustics are rendered interactively without any preprocessing.

This paper is structured as follows: Section 2 details the methods to render reflections and refractions interactively. How to render caustics is described in Section 3. In Section 4, techniques for accelerating the rendering are introduced. Some results and comparisons are given in Section 5. Section 6 concludes this paper.

2 Interactive Reflections and Refractions

In this section, we first introduce our technique for rendering reflection and refraction from an environment map, and then a new method for rendering reflection and refraction of nearby geometry is presented.

2.1 Reflection and Refraction of an Environment Map

Fig. 2 gives a simple illustration of reflection and refraction. The eye-ray $\mathbf{V}$ hits the object at $\mathbf{P}_1$. $\mathbf{N}_1$ is the normal at $\mathbf{P}_1$. $\mathbf{V}$ is reflected in direction $\mathbf{R}$ and refracted in direction $\mathbf{T}_1$. In the fragment shader on the current graphics card, $\mathbf{V}$, $\mathbf{P}_1$, $\mathbf{N}_1$, $\mathbf{R}$, and $\mathbf{T}_1$ can be computed easily [4], and then the current pixel's color can be indexed into an environment map using $\mathbf{R}$ and $\mathbf{T}_1$ to render the reflection and refraction. However, a single bounce refraction cannot generate the accurate visual result for multibounce refractions.

$\mathbf{P}_2$, $\mathbf{N}_2$, and $\mathbf{T}_2$ can also be computed approximately using an interpolation method [11]. The approach can be performed in three basic steps as shown in Fig. 3.

This technique can render better results than the previous method [4], but it has several limitations:

- Precomputed sampling of d_{N} is necessary, so the technique does not support dynamic refractors.
- $\mathbf{P}_2$ cannot be computed accurately even for convex objects because d_{T} is an approximate value.
- $\mathbf{P}_2$ must be located on the back-facing surface, but this is not always true.
- It assumes that $n_t > n_i$.

As shown in Fig. 2, if $n_t < n_i$, $\mathbf{V}$ would be refracted in direction $\mathbf{T}_1'$, and $\mathbf{P}_2'$ would probably be located on the front-facing surface. Even if $\mathbf{P}_2'$ is located on the back-facing surface, using d_{V} and d_{N} to approximate $\mathbf{P}_2'$ would also bring on a false result.

2.1.1 Computation of $\mathbf{T}_2$

The refraction technique of our method is briefly introduced in Fig. 4. Using the new method, preprocessing is not

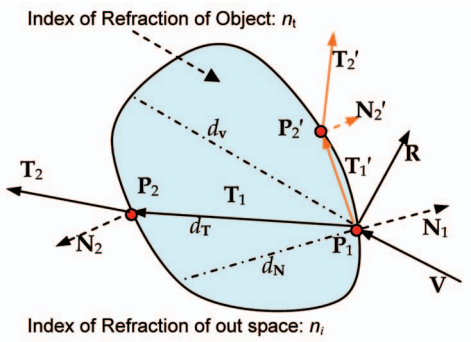

Fig. 2. Reflection and refraction on an object.

Precompute Precompute d_{N} at each vertex.
Initial pass Render normals and depths of back-facing surfaces to a buffer.
Rendering pass (in the fragment shader)
Draw front faces of refractor
{
  Compute $\mathbf{T}_1$ based on $\mathbf{V}$, $\mathbf{P}_1$ and $\mathbf{N}_1$;
  Compute d_{V} between front & back faces;
  Compute d_{T} by interpolating between d_{V} and precomputed d_{N};
  Compute $\mathbf{P}_2 = \mathbf{P}_1 + d_{\mathrm{T}}\mathbf{T}_1$;
  Project $\mathbf{P}_2$ to buffer from first pass to find $\mathbf{N}_2$;
  Compute $\mathbf{T}_2$;
}

Fig. 3. Wyman's method of reflecting environment maps.

```
Initial pass    Render normals and depths of the back-facing
                & front-facing surfaces to buffers.
Rendering pass (in the fragment shader)
Draw front-facing surface of the object.
{
    Compute T1 based on V, P1 and N1;
    Determine whether P2 is located on the back-facing or the
      front-facing surface;
    Compute dT using an iterative method;
    Compute P2 = P1 + dTT1;
    Project P2 to back-facing or front-facing buffer to find N2;
    Compute T2;
}
```

Fig. 4. Method of reflecting environment maps.

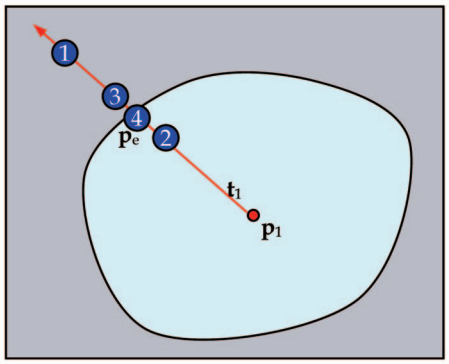

Fig. 6. Binary search in direction $\mathbf{t}_1$ from $\mathbf{p}_1$; the numbers indicate the sequence of computed test points.

required and more accurate results can be computed. That is to say, the limitations of Wyman's method are not a problem anymore in our new method. Details of our algorithm are presented as follows.

2.1.2 Determining Back-Facing or Front-Facing Surface

No matter whether $n_t < n_i$ or not, $\mathbf{P}_2$ could be located on the front-facing surfaces as shown in Fig. 5. The following method is presented to determine whether $\mathbf{P}_2$ is on the back-facing or the front-facing surface.

As shown in Fig. 5, $\mathbf{P}_e$ is the dividing point between the front-facing and the back-facing surfaces, and it is located on the same plane with the viewpoint, $\mathbf{P}_1$ and $\mathbf{P}_2$. If $\mathbf{P}_2$ is computed, θ_1 and θ_2 in Fig. 5 can be calculated. If $\theta_1 > \theta_2$, $\mathbf{P}_2$ is front-facing, else it is back-facing.

In Fig. 6, $\mathbf{p}_1$ is a pixel processed in the fragment shader. As discussed above, $\mathbf{P}_1$ and $\mathbf{T}_1$ at pixel $\mathbf{p}_1$ can be computed. Projecting $\mathbf{T}_1$ to the buffer, we can get $\mathbf{t}_1$, a 2D direction vector. In order to compute $\mathbf{P}_e$, we can first compute $\mathbf{p}_e$, which is the dividing pixel in the depth buffer. A binary search algorithm in Fig. 7 is used for computing $\mathbf{p}_e$. In this algorithm, we start with a large positive initial length (usually, 75 percent of the depth buffer size) to ensure that the first test point is outside the refractor's silhouette. The binary search proceeds with one endpoint inside and another outside the refractor's silhouette. In practice, we find that eight steps of the binary subdivision are sufficient to produce satisfactory results. This is equivalent to subdivide the initial length in 256 equally spaced intervals. Because the iterative computation is very simple, the binary search will not affect the rendering performance significantly.

Unprojecting $\mathbf{p}_e$, we can compute the position of $\mathbf{P}_e$. Therefore, we can compute θ_1 and θ_2, and determine whether $\mathbf{P}_2$ is on the front-facing or the back-facing surface. Fig. 10a shows the results for the sphere refraction.

If the refractive object is highly concave, there may be two or more $\mathbf{p}_e$ in the depth buffer, and the algorithm could obtain incorrect results. Fortunately, this limitation has few impacts on plausible rendering (see results in Section 5).

2.1.3 Computing d_T

To compute d_T, an iterative lookup technique is used: Starting with a small positive step size, d_T is adjusted according to z-values in the depth buffer and $\mathbf{P}_2$ as shown in Fig. 8. In most cases, two or three iteration steps could be enough to achieve a satisfactory result. We use a small positive value as the initial value to avoid that d_T is so large that it causes $\mathbf{P}_2$ to fall outside the refractive object's

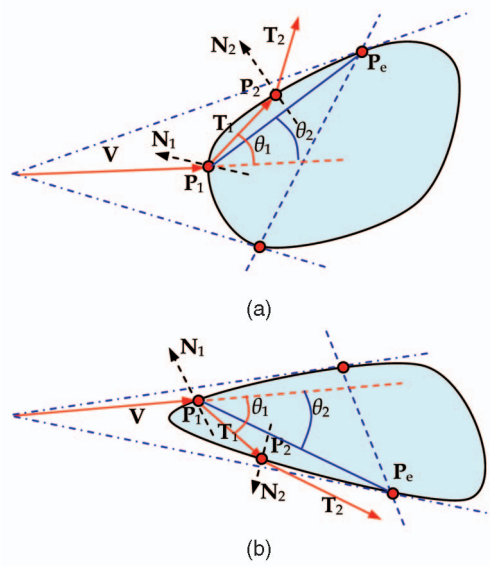

Fig. 5. $\mathbf{P}_2$ on the front-facing surfaces. (a) $n_t < n_i$ and (b) $n_t > n_i$.

```
length = a large positive value;              left = length;
do { //Binary search
    ptest = p1 + length * t1;       left = left/2.0;
    if (-1 ≤ ptest.xy ≤ 1)
        depth = tex2D(Back-facing Depth Buffer, ptest);
        //ptest is inside depth buffer
    else
        depth = 1; //ptest is outside the depth buffer
    if (depth == 1)
        length -= left;//ptest is outside refractor's silhouettes
    else
        length += left; //ptest is inside refractor's silhouettes
} while((left>Tolerance) or ((left<Tolerance) and (depth>=1.0)))
//To ensure that pe is inside the refractor's silhouettes,
//the depth value must be smaller than 1.0.
pe = ptest;
```

Fig. 7. Binary search algorithm.

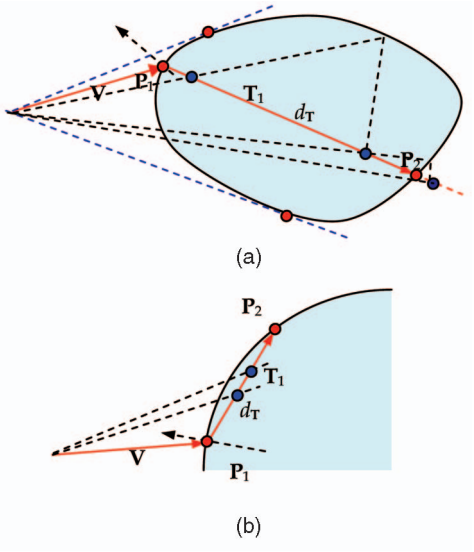

Fig. 8. Compute d_{T}. (a) $\mathbf{P}_2$ on the back-facing surface. (b) $\mathbf{P}_2$ on the front-facing surface.

```
if P2 is on the front-facing surface
    Flag = -1;   //Indicate the front-facing or back-facing
    DepthBuffer = front-facing depth buffer;
else
    Flag = 1;        DepthBuffer = back-facing depth buffer;
Steplength = a small positive value;    dT = 0;
MVP_Matrix = mul(PjMatrix, MvMatrix);
//PjMatrix is the projection matrix, and MvMatrix is the
//model view matrix of the rendered refractor.
do{
    dT += Steplength;
    //correct dT to make it closer to the accurate result
    P2 = P1 + dTT1;   tmp = Mul(MVP_Matrix, P2);
    //Project P2 to the depth buffer
    tmp.xyz /= tmp.w;  tmp.xyz += 1.0; tmp.xyz /= 2.0;
    //tmp.z is z-value of P2
    depth = tex2D(DepthBuffer, tmp.xy);
    //depth is the corresponding z-value in the depth buffer
    Steplength = Flag*(depth – tmp.z);
} while(|Steplength| > Tolerance)
```

Fig. 9. Compute d_{T}.

silhouette. The pseudocode of our algorithm is presented as shown in Fig. 9.

2.1.4 *Finding the Normal* $\mathbf{N}_2$

$\mathbf{N}_2$ is also required to compute $\mathbf{T}_2$. Since normals of the front-facing and back-facing surfaces have been rendered into the textures in the first pass, we can project $\mathbf{P}_2$ and then index into the normal textures to find the normal. It is similar to Wyman's method [11]. The difference is that $\mathbf{N}_2$ should be indexed into the front-facing or back-facing normal textures according to the position of $\mathbf{P}_2$. Fig. 10b shows the results of finding $\mathbf{N}_2$ for the sphere refraction.

After $\mathbf{T}_1$, $\mathbf{P}_2$, and $\mathbf{N}_2$ are computed, $\mathbf{T}_2$ can be easily calculated and be indexed into the environment maps (see Fig. 10c).

Obviously, d_{N} is not required anymore in our method, and neither is the assumption that $n_t > n_i$. Furthermore, no matter where $\mathbf{P}_2$ is located, we can deal with it well. In addition, because there are fewer assumptions in our method, we could render more accurate results than Wyman's method [11] as shown in Fig. 11. To illustrate the difference clearly, we use the method presented in Section 2.2 to refract the nearby plane. In theory, we could render the accurate result for a convex object. As for some concave objects, our method could render plausible results that could be seen in Section 5.

2.2 Reflection and Refraction of Nearby Geometry

Since $\mathbf{R}$ and $\mathbf{T}_2$ in Fig. 2 are computed, intersecting $\mathbf{R}$ and $\mathbf{T}_2$ with all other geometry in the scene can solve the problem of reflection and refraction of nearby geometry. However, because the intersections are computed on the fragment shader, it is difficult to input complex and dynamic scenes into the fragment shader without multipass rendering.

A technique has been proposed to refract nearby geometry by Wyman [12]. All nearby objects are rendered into one NG (Nearby Geometry) texture. On the CPU, determine the equations for a small number of planes describing the scene, and then input these planes into the fragment shader. After ray-plane intersection computation in the fragment shader, project the nearest intersection point into the NG texture, and then an iterative lookup technique is used to find the corresponding color. The technique can

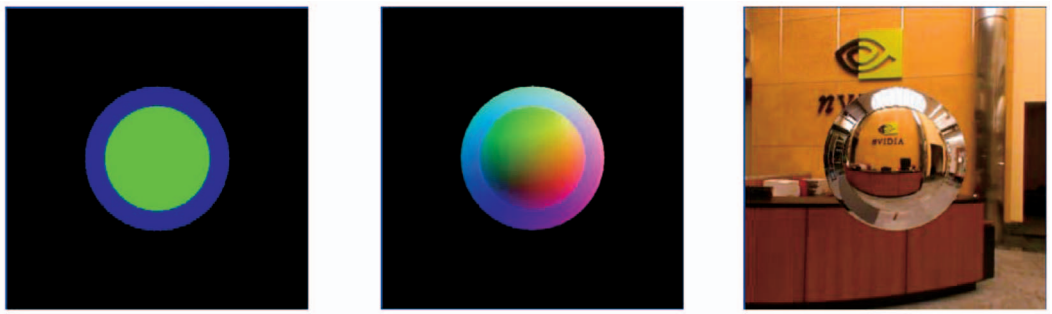

Fig. 10. Sphere refraction. Refractive index is 0.8. (a) Determining back-facing or front-facing. (b) Finding the normal $\mathbf{N}_2$. (c) Fetching color from environment maps. (The resolution of images is 512^2. Frame rate is 105 fps.)

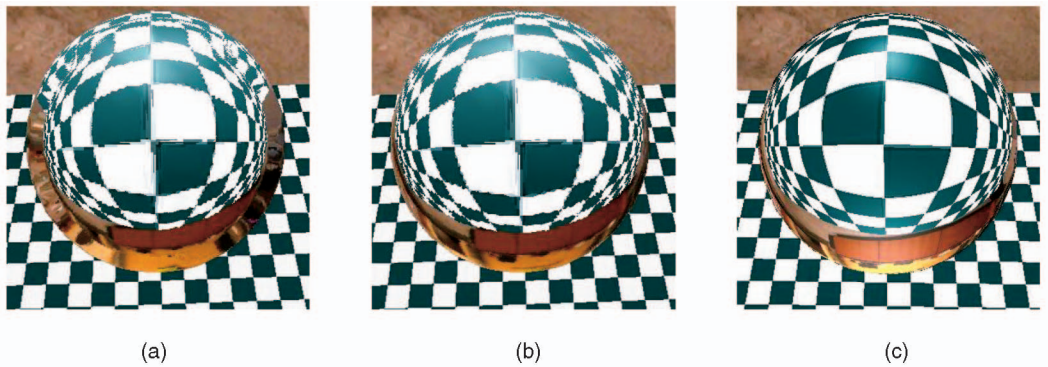

Fig. 11. Rendering the same scene using different methods. The resolution of images is 512^2. The refractive index is 1.75. Note the frame rates and edge area of the refractive spheres. (a) Wyman's method (93 fps). (b) Our new method (75 fps). (c) Ray tracing (0.133 fps).

render good results for some scenes, but it has the following limitations:

- First, reflection of nearby geometry cannot be handled using the same idea.
- Second, rendering the NG texture from the viewpoint would generate incorrect results for some scenes because the technique has an implicit assumption: The visibility among the refracted objects does not change over the surface of the refractor.

In this paper, a simple method inspired by the impostors [22] is presented to reflect and refract nearby geometry.

First, let us consider a simple scene with N rectangular planar surfaces, and we call them RECT_i $(0 < i \leq N)$. Each rectangular surface has four points, and the texture coordinates of these points are $\{0,0\}$, $\{0,1\}$, $\{1,1\}$, and $\{1,0\}$. RECT_i has its corresponding texture TEX_i. During rendering, all rectangles and textures are input into the fragment shader as parameters. After $\mathbf{R}$ or $\mathbf{T}_2$ is computed, intersect $\mathbf{R}$ or $\mathbf{T}_2$ with these rectangles. The rectangle RECT_k, whose intersection point is the closest to $\mathbf{P}_2$, is selected. We can compute the texture coordinates $\{u,v\}$ corresponding to the intersection point, and then $\{u,v\}$ can be indexed into TEX_i to determine the current pixel's color. If there is no rectangle intersected with $\mathbf{R}$ or $\mathbf{T}_2$, we can index into the environment map or the background color. Because we use the intersection point's real texture coordinates to fetch the color, instead of projecting the intersection point into the view-dependent NG texture, our method can generate the accurate reflection and refraction of a nearby rectangle.

If a more complex scene is considered, a similar technique can be applied. The major idea is to approximate a complex object using a rectangle and to render the object into the texture. Note that we must guarantee that the relationship between the rectangle and the texture is just like RECT_i and TEX_i introduced above. Therefore, the method to reflect or refract nearby rectangles can be applied. It is clear that the key technique is to determine the rectangle and texture.

2.2.1 Determining Rectangle and Texture

Since one rectangle is used to approximate a complex object, it is inevitable to generate only an approximate result. In order to obtain a better result, much attention should be focused on the technique to determine the rectangle and the corresponding texture.

In Fig. 12, there is a scene that consists of N objects $(\mathrm{OBJ}_i, 0 < i \leq N)$ and one object (OBJ_R) that can reflect and refract the light. To render reflection and refraction, we use rectangles and textures to stand in for the complex scene. For each object, we render a rectangular texture from the central point of the reflector/refractor to the object's central point with a parallel projection. Then, we can use the method of reflecting and refracting rectangular planar surfaces to render reflection and refraction of nearby

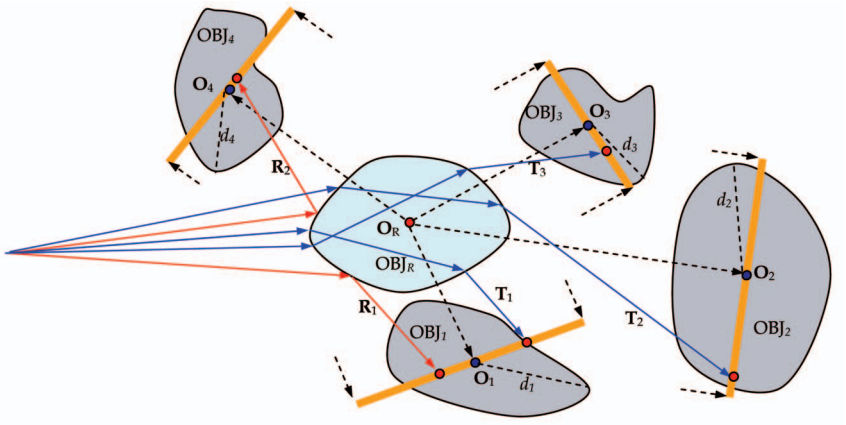

Fig. 12. Rectangles and textures to approximate complex objects.

Initial Pass

Compute center point $\mathbf{O}_R$ of OBJ_R and center point $\mathbf{O}_i$ of OBJ_i. The average value of object's vertices is used in our implementation;

Compute the largest distance (d_i) between OBJ_i's verti-ces and $\mathbf{O}_i$;

Rendering Pass

For OBJ_i

{

Calculate the vector $\mathbf{O}_R\mathbf{O}_i$;

Determine a rectangle ($RECT_i$) perpendicular to $\mathbf{O}_R\mathbf{O}_i$. The rectangle's center is $\mathbf{O}_i$. In addition, the rectangle has four equal sides whose length is $2d_i$;

Render OBJ_i into a texture (TEX_i) from $\mathbf{O}_R$ to $\mathbf{O}_i$ using parallel projection (glOrtho($-d_i$, d_i, $-d_i$, d_i, Near, Far)), where $\mathbf{O}_R\mathbf{O}_i$ be the direction of projection. Note that the up vectors of determining rectangle and rendering texture should be the same;

}

In the fragment shader

Render OBJ_R

{

Compute $\mathbf{P}_1$, $\mathbf{P}_2$, $\mathbf{R}$ and $\mathbf{T}_2$;

Ray = **R** or $\mathbf{T}_2$; **startP** = $\mathbf{P}_1$ or $\mathbf{P}_2$;

distance = MAX_DIST;

For $RECT_i$

{

tmpDist = intersect(**startP**, **Ray**, $RECT_i$, $\mathbf{P}_{rect}$);

//if Ray intersects with $RECT_i$, $\mathbf{P}_{rect}$ is the intersection //point, and tmpDist is the distance between startP and //$\mathbf{P}_{rect}$, or tmpDist = MAX_DIST.

If (*distance* > tmpDist)

{ //$RECT_i$ **is closer to startP**

k = *i*;

distance = tmpDist;

{*u*, *v*} = UV($\mathbf{P}_{rect}$, $RECT_i$);

//**compute texture coordinates of $\mathbf{P}_{rect}$**

}

}

If tmpDist < MAX_DIST

Color = Tex2D(TEXk, {u, v}); //**Ray intersects** $RECT_k$

Else

Color = TexCube(Environment Map, Ray);

}

Fig. 13. Rendering reflections and refractions of nearby geometry.

complex geometry. The detailed algorithm is shown in Fig. 13. It is clear that our method can deal with both reflection and refraction of nearby geometry.

The new method also has an assumption of visibility: The visibility of each object stays constant over the refractor. If scene geometries are not highly concave, we can render satisfactory results. For example, in Fig. 12, the intersection points between the objects and the two refracted rays ($\mathbf{T}_1$ and $\mathbf{T}_2$) cannot be rendered using the method in Wyman's method [12], because these intersection points cannot be seen from the viewpoint. Fig. 14 provides an example to illustrate this phenomenon.

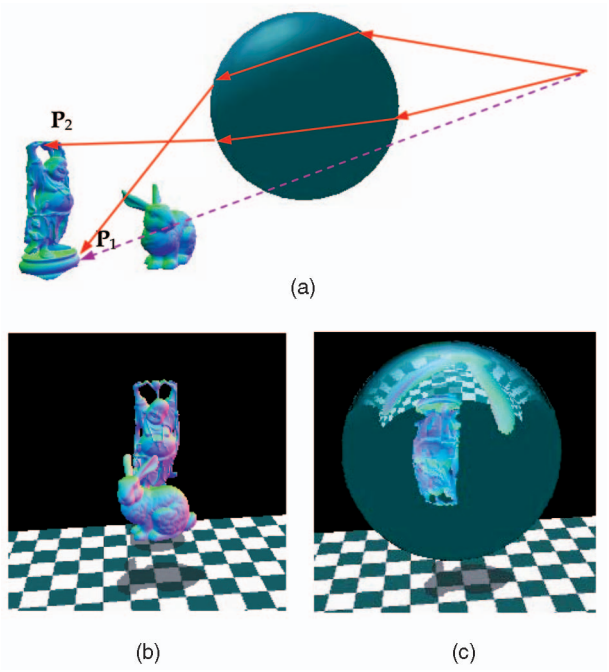

Fig. 14. $\mathbf{P}_1$ could not be seen directly from the view point if there was no refractive sphere, but it could be seen after being refracted by the sphere. (a) $\mathbf{P}_2$ could be seen no matter whether there is the refractive sphere or not. (b) If the method in [11] is used, the color of $\mathbf{P}_1$ could not be indexed from the NG texture. (c) Using the new method, we can see the correct refraction from the refractive sphere. (a) A simple scene with one refractive sphere. (b) The NG texture. (c) Our rendering result.

In addition, approximate recursive reflections and refractions can be rendered using our method. If OBJ_k in Fig. 12 is a reflector or a refractor, we render TEX_k using the above method. Therefore, we can see the recursive reflections or refractions on OBJ_R. In the same way, we can see the recursive reflections or refractions on OBJ_k too. Obviously, the distortion is inevitable, but for some cases, the plausible visual results can be rendered (see Fig. 1b and Fig. 24g).

2.2.2 The Iterative Lookup Technique

Due to approximating complex objects by rectangles, it is inevitable that we render incorrect results. In Fig. 15, the reflected or refracted ray $\mathbf{Ray}$ intersects the object at $\mathbf{P}_{\mathrm{geom}}$. In the fragment shader, $\mathbf{Ray}$ intersects the rectangle at $\mathbf{P}_{\mathrm{rect}}$, and then the color of $\mathbf{P}_{\mathrm{select}}$ is fetched from the texture due to a parallel projection. Therefore, distortion occurs especially when $\mathbf{P}_{\mathrm{geom}}$ locates far from $\mathbf{P}_{\mathrm{select}}$. A new iterative algorithm inspired by previous methods [12], [23] is presented as shown

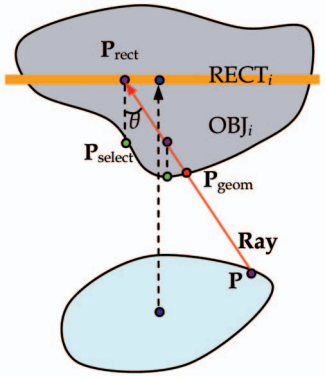

Fig. 15. Iterative computing.

```
distance = intersect(P, Ray, RECT_i, P_rect);
MVP_Matrixi = mul(PjMatrixi, MvMatrixi);
//PjMatrixi is the projection matrix of rendering TEX_i, and
//MvMatrixi is the model view matrix of rendering TEX_i
correctDepth = 0;
do{
    distance += correctDepth;
    P_new = P_2 + distance·Ray;
    tmp=Mul(ModelViewProjectMatrixparallel, P_new);
    //project P_new to TEX_DEPTH_i
    tmp.xyz /= tmp.w; tmp.xyz += 1.0; tmp.xyz /= 2.0;
    //tmp.z is z-value of P_new
    depth = tex2D(TEX_DEPTH_i, tmp.xy);
    //TEX_DEPTH_i is the z-buffer of TEX_i. depth is the z-
    //value of P_select
    correctDepth = (depth - tmp.z)/cosθ;
} while (|depth – tmp.z| > Tolerance)
Color = Tex2D(TEX_k, tmp.xy);
//{tmp.x, tmp.y} is just {u, v} in section 2.2.1
```

Fig. 16. Iterative lookup algorithm.

in Fig. 16 to reduce such distortion. In the algorithm, we can see that the iterative computation would correct $\mathbf{P}_{\text{new}}$ to close $\mathbf{P}_{\text{geom}}$ based on the depth values. $\mathbf{P}_{\text{rect}}$ is selected to be the initial point to the iterative computation in the algorithm. However, $\mathbf{P}_{\text{rect}}$ may falsely indicate that $\mathbf{Ray}$ misses the geometry, and $\mathbf{P}_{\text{rect}}$ may not generate a good result. As shown in Fig. 16, there is no intersection between $\mathbf{Ray}_1$ and RECT_i, but there are two intersection points between $\mathbf{Ray}_1$ and OBJ_i; $\mathbf{Ray}_2$ intersects RECT_i at $\mathbf{P}_{\text{rect}}$, but $\mathbf{P}_{\text{rect}}$ will generate a wrong result.

Wyman also discussed a similar problem [12], and he proposed to index into the z-buffer multiple times to find point $\mathbf{P}_{\text{new}}$ with the closest z-value between $\mathbf{P}_{\text{new}}$ and $\mathbf{P}_{\text{select}}$ as input to the iterative technique. Specifically, if d_{far} is the distance between $\mathbf{P}$ and $\mathbf{P}_{\text{rect}}$, we compute $\mathbf{P} + \alpha_j \mathbf{Ray}$ for a number of values $0 \leq \alpha_j \leq d_{\text{far}}$, and project the results into TEX_DEPTH_i. The number of values α_j used to find a good initial point for the iterative computation determines the feature size of nearby geometry guaranteed to be captured with our technique. Our method assumes that if a ray does

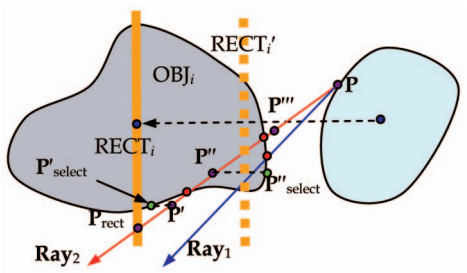

Fig. 17. Selecting a proper initial point.

not intersect with RECT_i, then it would not intersect with OBJ_i. Therefore, we cannot deal with $\mathbf{Ray}_1$ in Fig. 17. It is the limitation of our method. To reduce this problem, we can move RECT_i closer to the reflector/refractor (see RECT_i' in Fig. 17). However, it is not a general technique because the movement should be specified by the user before rendering.

In some cases, Wyman's technique is feasible to render the correct result. However, a point with closer z-value may be a worse seed. As shown in Fig. 17, if $\mathbf{P}'$ is selected as the input to the iterative algorithm, it is impossible to obtain the correct result. However, although the distance between $\mathbf{P}''$ and $\mathbf{P}''_{\text{select}}$ is larger, $\mathbf{P}''$ can generate a satisfied result.

We consider as the highest priority that the z-value of the initial point should be larger than the z-value of the corresponding $\mathbf{P}_{\text{select}}$. If we cannot find any points that meet this qualification, we then select the point with the closest z-value between $\mathbf{P}_{\text{new}}$ and $\mathbf{P}_{\text{select}}$.

Generally, using a good initial point as the input to the iterative algorithm, we can obtain a satisfactory result after two or three iteration steps. Fig. 18 gives some results of our algorithm. Two refractors have an index of refraction of 1.0, so objects should have no visual distortions after being refracted. However, since the rectangles (see Fig. 18b) are used to approximate the objects, rendered results are distorted (see Fig. 18c). After using iterative correction presented in the paper, we can reduce the problem significantly (see Fig. 18d).

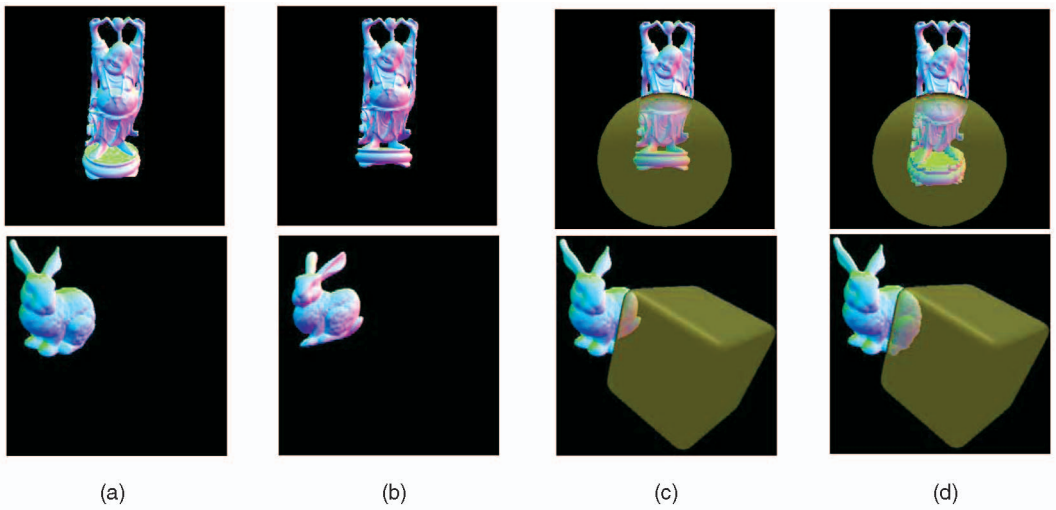

Fig. 18. Iterative corrections. The refractive index of refractor is 1.0. (a) Rendering OBJ_k from viewpoint without refractors. (b) Rendering RECT_k with texture TEX_k from viewpoint without refractors. (c) Uncorrected refraction of RECT_k. (d) Corrected refraction of RECT_k.

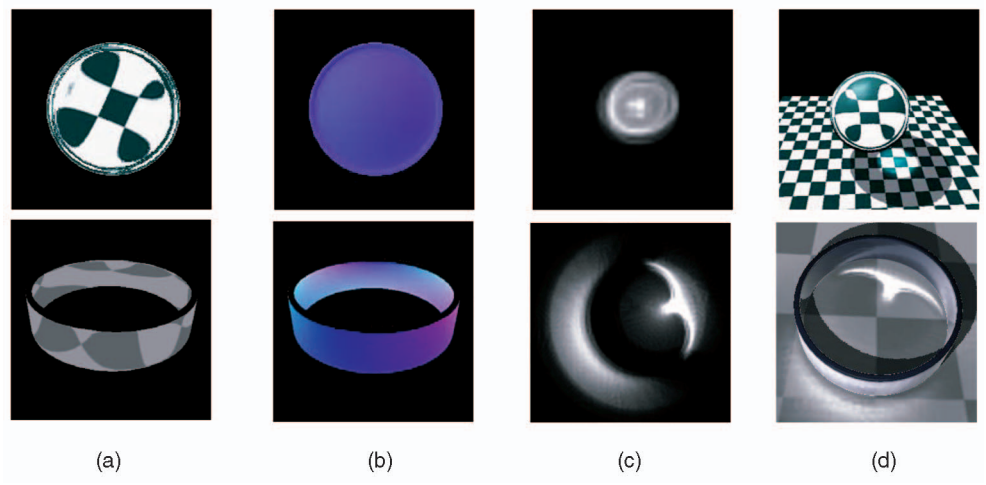

Fig. 19. Refractive caustics (top) and reflective caustics (bottom). (a) Render refraction and reflection from light. (b) Photon buffers. (c) Caustic textures. (d) Final results.

3 Interactive Caustics

Reflection and refraction can cause the focusing of light, and we call it caustics. Using the method to implement reflection and refraction of nearby geometry, we can render realistic caustics interactively. If a position on an object's surface can be seen from the light, it would receive *photons* emitted from the light. We consider that the more *times* a position can be seen from the light, the more *photons* it will receive. Therefore, caustics can be rendered using a similar method to reflection and refraction of nearby geometry.

3.1 Generating Photon Buffer

First, we render the reflector/refractor from the light position. If we render reflection and refraction of nearby geometry, in the fragment shader, the coordinates $\{u, v\}$ are used to index into the texture to determine the current pixel's color (see Fig. 19a). To render caustics, u, v, and k (index of the intersected rectangle) are output to the pixel buffer as RGB values. It means that the position $\{u, v\}$ of RECT_k can be seen from the light. The output pixel buffer is the *photon buffer* (see Fig. 19b). In the photon buffer, one pixel denotes one *photon*, and the color value of the pixel indicates the corresponding *photon* message.

3.2 Constructing Caustic Textures

For each rectangle (RECT_i, $0 < i \leq N$) that might possibly receive *photons*, we construct the corresponding *Caustic texture* ($\mathrm{TEX\_CAUSTICS}_i$, $0 < i \leq N$) from the *photon buffer*. The function *glReadPixels* is used to read the *photon buffer* from the graphics hardware to the memory. If a pixel's color is $\{u, v, k\}$ and $k > 0$, the position $\{u, v\}$ of RECT_k receives one *photon*, and then we increase the value of $\mathrm{TEX\_CAUSTICS}_k$ at position $\{u, v\}$ by one. After $\mathrm{TEX\_CAUSTICS}_i$ is constructed, each pixel contains the count of *photons* that hit the corresponding position of RECT_i.

To render caustics, it is necessary to filter $\mathrm{TEX\_CAUSTICS}_i$ ($0 < i \leq N$). Based on the photon count of the nearby pixels, we apply a filter [20] to each pixel to obtain

$$c(u,v) = s \sum_{i=-k}^{k} \sum_{j=-k}^{k} p(u+i, v+j) \sqrt{1 + 2k^2 - (i^2 + j^2)},$$

where $c(u, v)$ is the resulting color at the position (u, v), and $p(u, v)$ denotes the photon count at the position (u, v). s is a scaling value to adjust the power of the photon energies. The k value decides the size of filter and different values will generate different effects as shown in Fig. 20. Note that the larger the k value is, the longer the computing time will be, but if the k value is too small, visual effects will not be satisfied. Unless otherwise specified, k is equal to 2 and s is equal to 0.01 for rendering caustics in the paper.

This process is implemented on the GPU, and the result buffers are the caustic textures used for rendering caustics (see Fig. 19c).

3.3 Rendering Caustics

If the object (OBJ_i) corresponding to RECT_i ($0 < i \leq N$) is a rectangle, it is very simple to render caustics. Render RECT_i with texture $\mathrm{TEX\_CAUSTICS}_i$, and blend it with OBJ_i. Then, caustic effects can be rendered (see Fig. 19d).

If OBJ_i is a complex object, one more step should be performed because the simple alpha blending cannot be applied directly. The technique of shadow mapping inspires us with a simple method, called *caustic mapping*, to render caustics on a complex object. As shown in Fig. 21, while rendering OBJ_k from the eye's point-of-view, for each rasterized fragment, determine the fragment's position relative to the reflective or refractive object's center. Compute the current fragment's corresponding position in $\mathrm{TEX\_CAUSTICS}_k$ using a parallel projection, and then index into $\mathrm{TEX\_CAUSTICS}_k$ to fetch the *photon* intensity. In the end, blend *photon* intensity with the object's own color to generate the resulting color (see Fig. 22). The detailed algorithm of *caustics-mapping* is presented in Fig. 23.

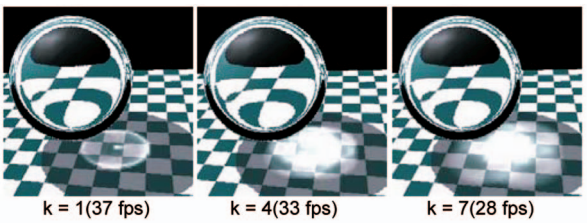

Fig. 20. Different filter sizes and corresponding frame rates.

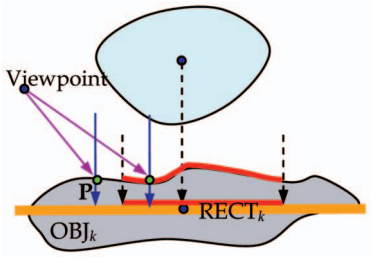

Fig. 21. Caustics mapping.

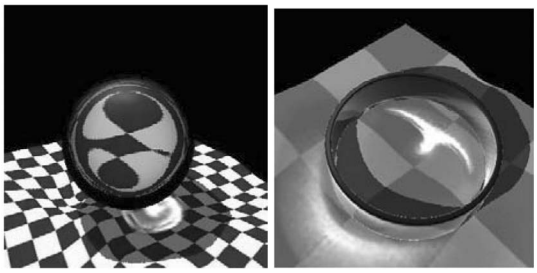

Fig. 22. Caustics on uneven objects.

```
In the fragment shader
Render OBJi
{
    Determine current fragment's XYZ position P;
    MVP_Matrixi = mul(PjMatrixi, MvMatrixi);
    //PjMatrixi is the projection matrix of rendering TEXi,
    //and MvMatrixi is the model view matrix of rendering
    //TEXi
    tmp=Mul(MVP_Matrixi, P);
    //project P to caustic texture
    tmp.xyz /= tmp.w; tmp.xyz += 1.0; tmp.xyz /= 2.0;
    intensity = tex2D(TEX_CAUSTICSi, tmp.xy);
    Blend intensity with fragment's own color;
}
```

Fig. 23. Caustic-mapping algorithm.

4 Accelerating Techniques

Generally speaking, in most cases, $\mathbf{P}_2$ would be on the back-facing surface in Section 2.1.2. To speed up the rendering, we can suppose that $\mathbf{P}_2$ is back-facing and then compute d_{T} first. If d_{T} becomes so large that it causes $\mathbf{P}_2$ to fall outside the refractive object's silhouette, we can determine that $\mathbf{P}_2$ is front-facing. This method can accelerate the rendering speed for most cases.

Iterative techniques are used in our method. Although the fragment shader of GeForce 6800/7800 can support many instructions and dynamic flow control, iterative processes will decrease the rendering performance. To solve this problem, it is necessary to reduce the number of iterations. Our method is approximated, and usually four or five iterations are enough to produce satisfying results as discussed above. Thus, we can enlarge tolerances to accelerate computing.

For a static scene, it is unnecessary to compute caustics, shadow, RECT_i, and TEX_i (Section 2.2.1) during rendering, so the rendering time can be reduced greatly. Note that all performance data in the paper are achieved under the assumption that the scenes are fully dynamic.

To construct caustic textures in Section 4.2, the function *glReadPixels* is used. As we know, this function is usually not suitable for real-time rendering. Fortunately, a high resolution is not necessary for the photon buffer because caustics are not a very precise visual effect, especially since the filter could generate plausible caustics even when the photon buffer's resolution is not high. According to our experience, 128^2 is acceptable for rendering plausible results, and reading the buffer with a resolution of 128^2 from graphics hardware will take less than 10ms. Therefore, *glReadPixels* can work well in our interactive rendering. Furthermore, Vertex Texture and MRTs (Multiple Render Targets) techniques [24] are supported by the GeForce 6800/7800 series card, so it is possible to construct caustic textures by using these techniques without reading data from GPU to the main memory.

5 Results

We have implemented our work in OpenGL on a Pentium 4, 2.8 GHz with an AGP 8x GeForce 6800 graphics card running Windows. All code is written in C++ and compiled by using Visual Studio 6. The Cg 1.4 [25] is used to generate vertex and fragment programs. All shadow effects in the paper are generated using a simple hardware shadow mapping [26] technique. The resolution of images in Fig. 24, Fig. 25, Fig. 26, Fig. 27, Fig. 28, and Fig. 29 is 512^2.

Table 1 shows some running frame rates of varying complexity scenes. To compare our method with Wyman's methods and ray tracing, we do not render the reflection and caustics. The Sphere, Teapot, Dinosaur, Buddha, Bunny, and Dragon models have 1K, 2K, 21K, 50K, 70K, and 250K triangles, respectively. Compared with Wyman's methods, our method does not have an advantage on rendering performance, but our method can render more accurate results without preprocessing. Furthermore, reflection and caustics are well supported in our method.

This paper introduces some techniques for interactive rendering, and Fig. 24 shows their impacts on rendering performance. We can see that rendering nearby geometry, depth correction, adding caustics, photon buffer size, adding reflection, and adding recursive reflection/refraction all will slow down the rendering speed.

To illustrate more comparisons between photon mapping and our new method, we rendered some images using both methods as shown in Fig. 25.

In Fig. 26, reflective and refractive caustics are cast by the ring-like object that can reflect and refract the light. Fig. 27 shows caustics on two objects, and there are two lights in the scene. Caustics through the rippling water surface are displayed in Fig. 28, and one-interface refraction is computed (see $\mathbf{P}_1$ and $\mathbf{T}_1$ in Fig. 2). The viewpoint is located under the water surface. We can see caustics on three objects: the floor, wall, and the dolphin.

Because preprocessing is not required, we can render full dynamic scenes. Fig. 29 shows a morphing sequence from a dinosaur to a leopard.

6 Conclusions and Future Work

The paper proposes a simple but effective method for rendering reflection, refraction, and caustics interactively on GPU even for complex models. The main contributions of this paper are as follows:

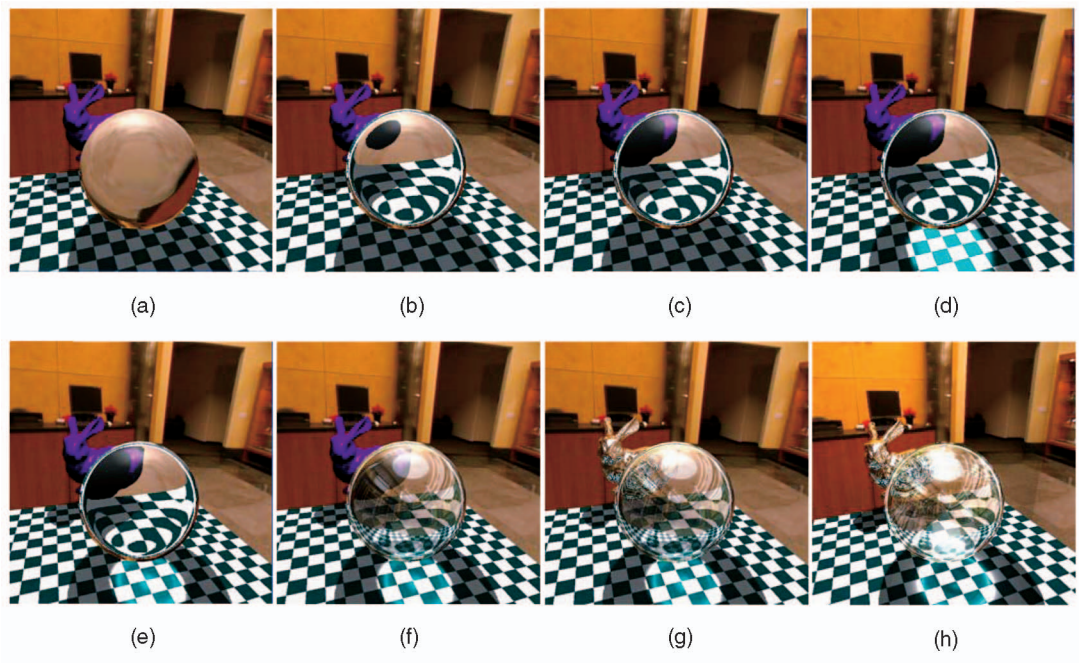

Fig. 24. Comparisons of frame rates. The refractive index of the sphere is 1.15. (a) Refraction of environment maps. (b) Uncorrected refraction of nearby geometry. (c) Corrected refraction of nearby geometry. (d) Refraction and caustics (photon buffer size is 64^2). (e) Refraction and caustics (photon buffer size is 128^2). (f) Reflection, refraction, and caustics (photon buffer size is 128^2). (g) Recursive reflection and refraction. (h) Photon mapping result of POV-Ray, which is available for free at http://www.povray.org. (a) 117 fps, (b) 57 fps, (c) 45 fps, (d) 25 fps, (e) 22 fps, (f) 18 fps, (g) 13 fps, and (h) 0.0425 fps.

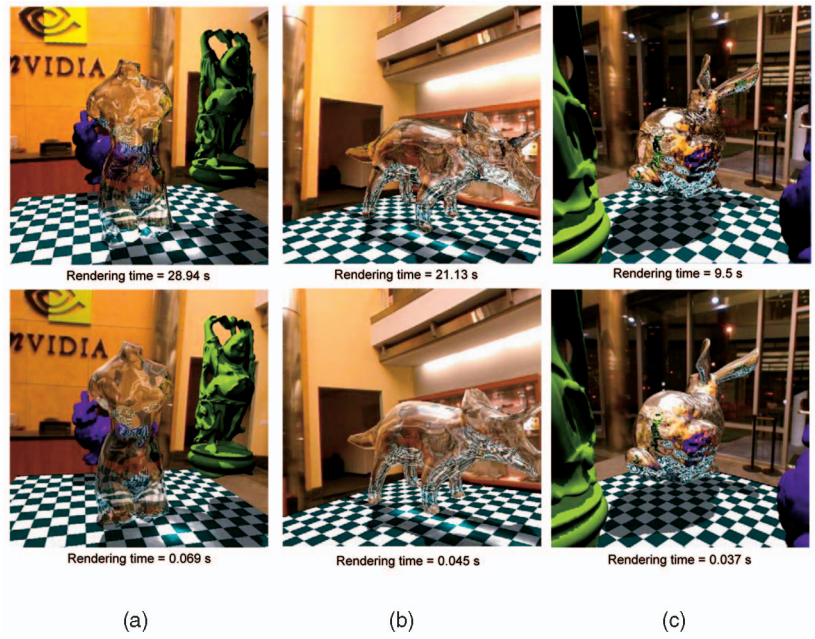

Fig. 25. Comparing photon mapping (top) with our method (bottom). The resolution of images is 512^2. A 10K triangle Venus (left) and a 21K triangle Dinosaur (center) can reflect and refract lights; A 70K triangle Bunny (right) could only reflect lights. Top images are rendered with photon mapping in POV-Ray.

- Our method can process reflection, refraction, and caustics. Except some global illumination methods, such as ray-tracing and photon-mapping, few techniques could render these effects in a united manner.
- Realistic images can be rendered interactively without any preprocessing, so our method can be applied to render fully dynamic scenes.
- As for refraction, compared with Wyman's methods [11], [12], the several limitations and restrictions do not exist in our method. Moreover, our method can render more accurate results.
- Although many techniques could render caustics interactively or even in real time, most of them only process one bounce reflection or refraction. Using our method, refraction through two surfaces is supported. In theory, since we can render approximate recursive refractions and reflections, recursive caustics can also be rendered.

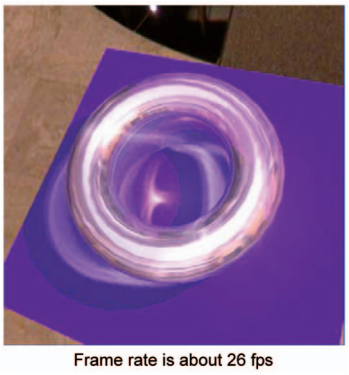

Fig. 26. Reflective and refractive caustics.

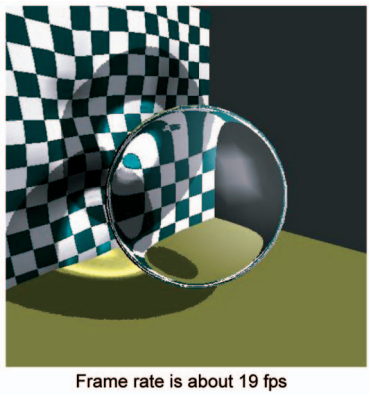

Fig. 27. Caustics on two objects.

However, our method also has some limitations:

- Using rectangles to approximate complex objects would not generate completely accurate reflection and refraction, particularly when two nearby objects are so close to each other that their texture rectangles intersect, although the iterative technique (Section 3.2) may reduce such errors.
- Just like Wyman's method [12], aliasing is unavoidable because the geometry is stored in a texture. The refractor or reflector can magnify the texels to cover many pixels (see Fig. 11).
- Each object is approximated using one rectangle and one corresponding texture. Therefore, if there are many objects in a scene, graphics hardware may not support so many textures and the intersection computation would consume too much time. To solve this problem, we can unite some objects into one object and use an environment map to present objects that are far away from reflectors and refractors.
- The recursive reflection and refraction, total internal reflection, and multi-interface refraction are still not well supported. Furthermore, for highly concave objects, some incorrect results would be rendered.

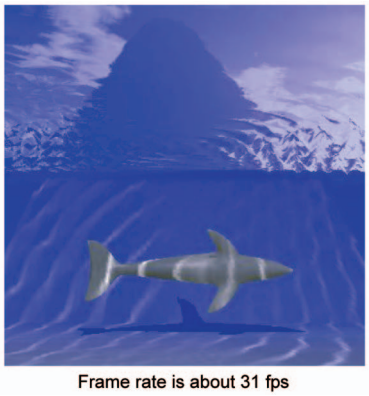

Fig. 28. Caustics under rippling water surface.

We plan to investigate several issues in the future, including more precise reflection and refraction, better handling of concave reflectors and refractors, multibounce reflection and refraction, more accurate approximations for near geometry, and more speed ups of the rendering process.

ACKNOWLEDGMENTS

The authors wish to thank the Nvidia Corporation for the cube-map textures, http://www.povray.org for the free POV-ray, and to Chris Wyman for his demo. They would also like to thank the anonymous reviewers for both their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (No. 60473112) and the Specialized Research Fund for the Doctoral Program of Higher Education of China (No. 20030003053).

Fig. 29. From Dinosaur to Leopard. Dragon has 250K triangles. Buddha has 50K triangles. Dinosaur (Leopard) has 21K triangles.

TABLE 1
Frame Rate Comparisons of Rendering Refraction for Scenes

Refractor	Nearby geometry	Refraction of environment maps				Refraction of nearby geometry		
		1-Sided Refraction	Wyman's method [10]	Our method	Ray Tracing	Wyman's method [11]	Our method	Ray Tracing
Sphere	Buddha	320 fps	157 fps	127 fps	0.126 fps	53 fps	46 fps	0.065 fps
Teapot	Bunny	310 fps	132 fps	118 fps	0.132 fps	49 fps	41 fps	0.079 fps
Dinosaur	Buddha & Dragon	252 fps	77 fps	65 fps	0.088 fps	31 fps	25 fps	0.044 fps

REFERENCES

[1] T. Whitted, "An Improved Illumination Model for Shaded Display," *Comm. ACM*, vol. 23, no. 6, pp. 343-349, 1980.

[2] J.T. Kajiya, "The Rendering Equation," *Proc. SIGGRAPH Conf.*, pp. 143-150, 1986.

[3] H.W. Jensen, "Global Illumination Using Photon Maps," *Proc. Conf. Rendering Techniques*, pp. 21-30, 1996.

[4] E. Lindholm, M. Kligard, and H. Moreton, "A User-Programmable Vertex Engine," *Proc. SIGGRAPH Conf.*, pp. 149-158, 2001.

[5] G. Oliveira, "Refractive Texture Mapping," part two, http://www.gamasutra.com/features/20001117/oliveira_01.htm, 2001.

[6] K. Nielsen and N. Christensen, "Realtime Recursive Specular Reflections on Planar and Curved Surfaces Using Graphics Hardware," *J. Winter School of Computer Graphics (WSCG)*, vol. 2, pp. 91-98, 2002.

[7] Z.S. Hakura and J.M. Snyder, "Realistic Reflections and Refractions on Graphics Hardware with Hybrid Rendering and Layered Environment Maps," *Proc. Eurographics Workshop Rendering*, 2001.

[8] E. Ofek, "Interactive Reflections on Curved Objects," *Proc. SIGGRAPH Conf.*, pp. 333-342, 1999.

[9] C.M. Schmidt, "Simulating Refraction Using Geometric Transforms," master's thesis, Computer Science Dept., Univ. of Utah, 2005

[10] W. Heidrich, H. Lensch, M. Cohen, and H. Seidel, "Light Field Techniques for Reflections and Refractions," *Proc. Eurographics Rendering Workshop*, pp. 187-196, 1999.

[11] C. Wyman, "An Approximated Image-Space Approach for Interactive Refraction," *Proc. SIGGRAPH Conf.*, 2005.

[12] C. Wyman, "Interactive Image-Space Refraction of Nearby Geometry," Technical Report UICS-TR-05-04, Computer Science Dept., Univ. of Iowa, http://www.cs.uiowa.edu/cwyman/publications/files/ISRefracWNearby/UICS-TR-05-04.pdf, 2004.

[13] J. Arvo, "Backward Ray Tracing, Developments in Ray Tracing," *Proc. SIGGRAPH 86 Course Notes*, vol. 12, Aug. 1986.

[14] T. Purcell, C. Donner, M. Cammarano, H.W. Jensen, and P. Hanrahan, "Photon Mapping on Programmable Graphics Hardware," *Proc. ACM SIGGRAPH/Eurographics Conf. Graphics Hardware*, pp. 41-50, 2003.

[15] J. Guenther, I. Wald, and P. Slusallek, "Realtime Caustics Using Distributed Photon Mapping," *Proc. Eurographics Symp. Rendering*, pp. 111-121, 2004.

[16] C. Wyman, C. Hansen, and P. Shirley, "Interactive Caustics Using Local Recomputed Irradiance," *Proc. Pacific Graphics Conf.*, 2004.

[17] M. Wand and W. Straber, "Real-Time Caustics," *Proc. Eurographics Conf.*, 2003.

[18] K. Iwasaki, Y. Dobashi, and T. Nishita, "A Fast Rendering Method for Refractive and Reflective Caustics Due to Water Surfaces," *Proc. Eurographics Conf.*, 2003.

[19] K. Iwasaki, F. Yoshimoto, and Y. Dobashi, "A Rapid Rendering Method for Caustics Arising from Refraction by Transparent Objects," *Proc. Cyber World Conf.*, 2004.

[20] B. Larsen and N. Christensen, "Simulating Photon Mapping for Real-Time Applications," *Proc. Eurographics Symp. Rendering*, 2004.

[21] P.P. Sloan, J. Kautz, and J. Snyder, "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments," *ACM Trans. Graphics*, vol. 21, no. 4, pp. 527-536, July 2002.

[22] G. Schaufler, "Image-Based Object Representation by Layered Impostors," *Proc. ACM Symp. Virtual Reality Software and Technology*, 1998.

[23] E. Ohbuchi, "A Real-Time Refraction Renderer for Volume Objects Using a Poygon-Rendering Scheme," *Proc. Computer Graphics Int'l Conf.*, pp. 190-195, 2003.

[24] Nvidia Developer Home, http://developer.nvidia.com, 2005.

[25] W. Mark, R. Glanville, K. Akeley, and M. Kilgard, "CG: A System for Programmable Graphics Hardware in a C-Like Language," *Proc. SIGGRAPH Conf.*, 2003.

[26] C. Franklin, "Shadow Algorithms for Computer Graphics," *Proc. SIGGRAPH Conf.*, pp. 242-248, 1977.

Wei Hu received the BS (1999) and the MS (2002) degrees in computer science from the Dalian University of Technology, Dalian, China. He is currently a PhD candidate in the Department of Computer Science and Technology at Tsinghua University, Beijing, China. His professional interests focus on interactive global illumination and cluster rendering.

Kaihuai Qin received the PhD and MEng degrees from the Huazhong University of Science and Technology in 1990 and 1984 and the BEng degree from the South China University of Technology in 1982. He is a professor of computer science and technology at Tsinghua University. He was a postdoctoral fellow from 1990 to 1992, then joined the Department of Computer Science and Technology of Tsinghua University as an associate professor. He was a visiting scholar at SPL, BWH, Harvard Medical School, Harvard University, Boston, Massachusetts from 1999-2000. His research interests include computer graphics, CAGD, curves and surfaces, especially subdivision surfaces and NURBS modeling, physics-based geometric modeling, wavelets, medical visualization, surgical planning and simulation, virtual reality, and intelligent and smart CAD/CAM.