Real-Time Refraction Through Deformable Objects

Manuel M. Oliveira[*]
Instituto de Informática
UFRGS

Maicon Brauwers[†]
Instituto de Informática
UFRGS

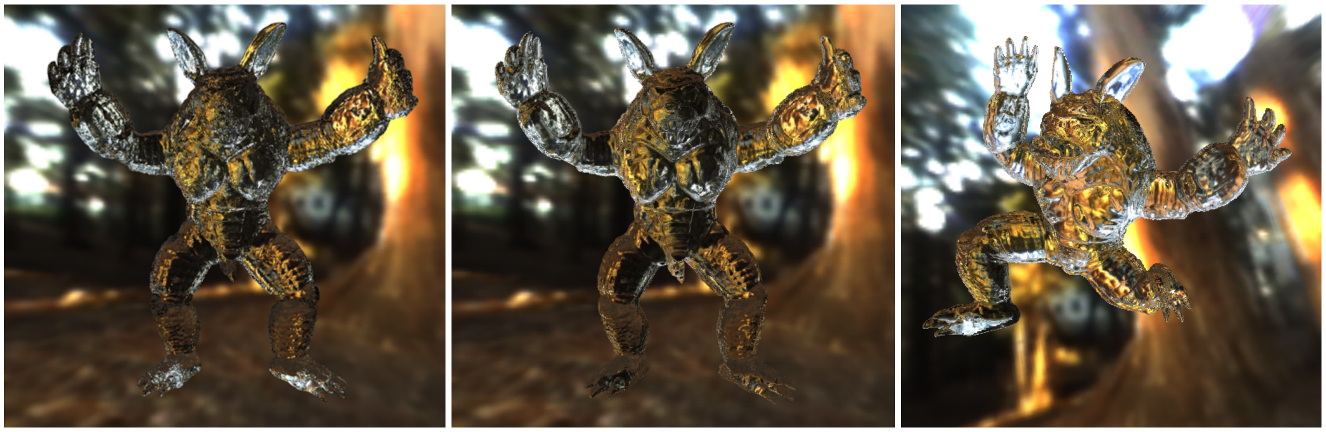

Figure 1: Armadillo model refracting a distant environment rendered using ray tracing (left) and using our technique (center). Jumping Armadillo rendered with our technique (right). Our approach requires no preprocessing, allowing real-time rendering of deforming objects.

Abstract

Light refraction is an important optical phenomenon whose simulation greatly contributes to the realism of synthesized images. Although ray tracing can correctly simulate light refraction, doing it in real time still remains a challenge. This work presents an image-space technique to simulate the refraction of distant environments in real time. Contrary to previous approaches for interactive refraction at multiple interfaces, the proposed technique does not require any preprocessing. As a result, it can be directly applied to objects undergoing shape deformations, which is a common and important feature for character animation in computer games and movies. Our approach is general in the sense that it can be used with any object representation that can be rasterized on a programmable GPU. It is based on an efficient ray-intersection procedure performed against a dynamic depth map and carried out in 2D texture space. We demonstrate the effectiveness of our approach by simulating refractions through animated characters composed of several hundred thousand polygons in real time.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: image-space refraction, real-time rendering, deformable objects, image-based rendering.

*e-mail:oliveira@inf.ufrgs.br
†e-mail:maiconb@inf.ufrgs.br

1 Introduction

The synthesis of photorealistic images has long been one of the major goals in computer graphics. In its pursuit, researchers have proposed different strategies for simulating the complex interaction between light and scene elements. For translucent objects, light refraction is an important optical phenomenon whose simulation significantly contributes to the realism of synthesized images. While ray tracing [Whitted 1980] can correctly simulate light refraction, its computational cost is prohibitive for real-time applications, even when acceleration techniques are used. Thus, several techniques for rendering approximate refractions at a single interface have been proposed for use in applications where speed tends to be more important than accuracy [Oliveira 2000; Lindholm et al. 2001; Schmidt 2003]. More recently, Wyman [Wyman 2005a] presented an interactive technique for approximating refraction of distant environments at two interfaces that produces quite impressive results. The images usually look very similar to the ones rendered with a ray tracer. However, the technique requires information about some distance measured along the normal direction at each vertex of model. This limits its use in applications where characters may undergo non-rigid transformations, vertex morphing and skinning, which are common in games.

This paper presents a technique for approximating the refraction of distant environments at two interfaces in real time. Our approach was inspired by the work of Wyman [Wyman 2005a], but contrary to previous techniques for interactive simulation of refraction at multiple interfaces [Wyman 2005a; Genevaux et al. 2006], it requires no preprocessing. As a result, it can be directly applied to models undergoing shape deformations and is general in the sense that it can be used with any object representation that can be rasterized on a programmable GPU. This flexibility is achieved using an efficient ray-intersection procedure performed against a dynamic depth map and carried out in 2D texture space. Besides supporting dynamic geometry, the proposed approach requires less memory (no need to store and send per-vertex distances to the GPU). Figure 1 illustrates the use of our technique applied to the Armadillo model on different poses. On the left, one sees an image rendered using a ray tracer [Pharr and Humphreys] and used for reference. The image in the center was rendered from the same viewpoint using our technique. Note how similar they are. Except for the feet, where total internal reflection (TIR) occurs, they are virtually indistinguishable from each other. The image on the right shows the Armadillo model during a jump, also rendered with our technique.

The main contributions of this paper include:

- A real-time technique for rendering approximate refraction of distant environments that is capable of handling any object representation that can be rasterized using a programmable GPU, including dynamically deforming models (Section 3);
- A new texture-space GPU-based algorithm for computing ray intersection against a depth map generated in perspective projection (Section 3.2). It extends previous work presented in [Policarpo et al. 2005], which is restricted to orthographic representations of depth maps (*i.e.*, height fields);

Section 2 discusses some related work and Section 3 provides the details of the proposed technique. We present some of our results in Section 4 and discuss limitations and possible extensions of the work in Section 5. Section 6 summarizes the paper.

2 Related Work

There have been a few initiatives to simulate refraction through multiple interfaces in recent years. Diefenbach and Badler [Diefenbach and Badler 1997] render refractions through planar surfaces at interactive rates using a multi-pass rendering approach. Heidrich et al. [Heidrich et al. 1999] store ray directions on a Lumigraph representation to render reflections and refractions on curved objects. The approach takes advantage of graphics hardware to accelerate the rendering, but it requires the use of pre-acquired Lumigraphs and cannot handle deforming geometry.

Hakura and Snyder [Hakura and Snyder 2001] use a hybrid approach for generating images of reflective and refractive objects that combines the use of ray tracing for local objects and hardware-supported environment maps for distant ones. While the technique produces good results, it is not sufficiently fast for real-time applications.

Guy and Soler [Guy and Soler 2004] presented a real-time technique for rendering gems that handles approximations of several optical phenomena, including refraction. Their approach takes advantage of the faceted shaped of processed gems, not being applicable to arbitrary shapes.

Genevaux et al. [Genevaux et al. 2006] simulate refractions through several interfaces for static scenes. During a preprocessing stage, light paths are evaluated by tracing rays through the refractive objects from many different entry directions. The resulting data associates each pair (object's surface point, incoming direction) to an outgoing direction which will be used to sample a distant environment map during runtime. The resulting table is then compressed using spherical harmonics and uploaded onto graphics hardware for interactive rendering. While the technique can produce nice results at interactive rates, the preprocessing stage prevents its use with objects undergoing deformations. Moreover, its memory requirements are relatively high (of the order of several dozens of megabytes). Due to the impossibility to precompute all possible light paths, the technique is prone to aliasing.

2.1 Image-Space Refraction

Wyman [Wyman 2005a] introduced an image-space technique for approximating the refraction of distant environments at two interfaces. Despite its simplicity, the approach produces very nice results. The algorithm runs on the GPU, consisting of two passes. Conceptually, it can be described as:

- First, it renders the back-facing portion of the refractive object, saving the information about normals and depth of the most distant fragments from the camera;
- The final image is obtained by rendering the front-facing portion of the object. This time, a per-fragment refracted ray T_1 (Figure 2) is computed at the first interface using the current's fragment normal (N_1) and its associated viewing direction (V). T_1 is then used to obtain the intersection point P_2 at the second interface. The coordinates of the projection of P_2 onto the camera's image plane are used to index the saved normal map and recover N_2. Finally, T_1 and N_2 are used to compute the direction of T_2, the refracted ray at the second interface, which is used to index an environment map. Figure 2 illustrates this concept.

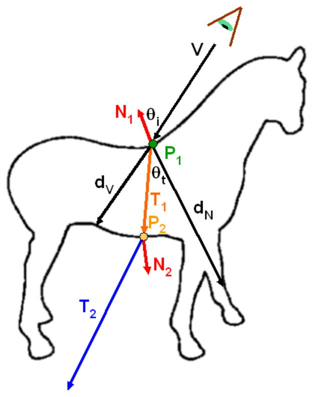

Figure 2: Refraction of distant environment through two interfaces.

Wyman [Wyman 2005a] avoids computing the intersection of T_1 with the saved depth map. Instead, he uses a heuristic to obtain an estimate $\tilde{P}_2$ for P_2. Assuming that $\eta_o \geq \eta_m$, where η_o is the index of refraction of the object and η_m is the index of refraction of the surrounding medium, then $(\eta_o/\eta_m) \in [1, \infty]$. At the extrema of this interval, T_1 would have the same direction as V and $-N_1$, respectively, which are indicated in Figure 2 by the vectors d_V and d_N. Thus, Wyman estimates the position of point $\tilde{P}_2$ along T_1 by interpolating the lengths of the vectors d_V and d_N (Figure 2) according to Equations 1 and 2. The length of d_N is the distance between the first and second interfaces considered along the normal direction of the current fragment. It is pre-computed on a per-vertex basis and stored for use during runtime, when these values are interpolated during rasterization. The length of d_V is also the distance between the first and second interfaces, but this time measured along the viewing direction. It is computed for each fragment as the difference between the front and the stored depth values.

$$\tilde{P}_2 = P_1 + \tilde{d}T_1 \tag{1}$$

where

$$\tilde{d} = \frac{\theta_t}{\theta_i} d_V + (1 - \frac{\theta_t}{\theta_i}) d_N \tag{2}$$

Note that these equations have been empirically defined and although they can produce acceptable approximations of P_2 for convex objects, the approximation error tends to grow for non-convex ones. This situation is illustrated in Figures 2 and 3. Figure 3 shows a schematic representation for the geometry (left) and normals (right) of the dragon model. The error in the estimate $\tilde{P}_2$ (left) lends, via its projection onto the camera's image plane, to the recovery of an incorrect normal $\tilde{N}_2$ used to compute T_2 and, subsequently, to index the environment map. Both N_2 and $\tilde{N}_2$ are shown on Figure 3 (right).

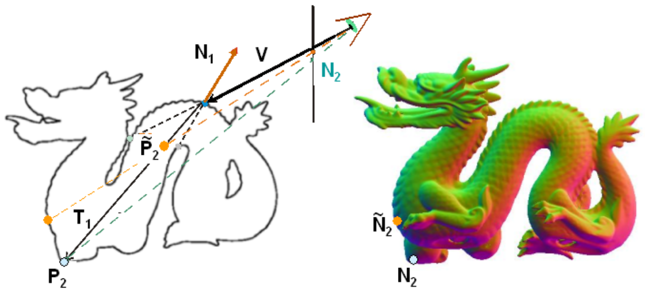

Figure 3: Schematic representation for the geometry (left) and normals (right) of the dragon model. The error in the estimate $\tilde{P}_2$ lends to incorrect sampling of the normal map at $\tilde{N}_2$ (right), causing the environment map to be sampled using an incorrect $\tilde{T}_2$ direction.

3 Refraction through Deformable Objects

Our approach supports the rendering deforming models by performing the intersection of the refracted ray T_1 (Figure 2) against the stored depth map. This approach has some advantages compared to Wyman's technique: first, it does not require any preprocessing and tends to produce better estimates for P_2 and, consequently, of N_2.

Our technique consists of three GPU passes:

1st pass : Similar to Wyman's approach, it consists of rendering the back-facing portion of the refractive object, saving the information about normals and depth of the fragments with the highest Z values;

2nd pass : It computes the depth range (minimum and maximum values) of the back-facing portion of the object by applying a parallel reduction [Harris 2005] to the depth map saved in the previous pass. At each step of the reduction, the minimum and maximum depth values from each group of four elements are saved, respectively, in the R and G channels of the output texture;

3rd pass : This is the actual rendering step, which consists of rendering the front-facing portion of the refractive object, computing the refracted rays and sampling the environment map. The position of point P_2 (Figure 2) is obtained computing the intersection of the refracted ray T_1 with the second interface represented by the perspective depth map saved in the first pass. The search is optimized by restricting it to the range computed in the second pass, and by using a new and efficient ray perspective-depth-map intersection solution entirely performed in 2D texture space (Section 3.2). The result of the intersection procedure directly provides the texture coordinates required to sample the normal map.

3.1 Saving Second Interface Normals and Depth

The first pass is straightforward and simply renders the back-facing geometry, saving normals and depth information. While normals and depth could potentially be saved on a single RGBA texture [Policarpo et al. 2005], we store them in two textures. Although this requires more memory, according to our experience, sampling the normal map using mipmapping [Williams 1983] tends to reduce noise in the final images. This can be explained by the fact that discontinuities in the normal field may cause two neighbor fragments to refract the incident rays in an incoherent manner, thus introducing undesirable artifacts. By filtering the normals, mipmapping helps to minimize this problem. On the other hand, filtering a depth map tends to introduce artifacts, which can be avoided by using a nearest neighbors sampling strategy.

Normals and depth information are saved using multiple render targets (MRTs). We use the integer GL_TEXTURE_2D texture format, as it is currently the only format supporting automatic generation of mipmaps. Thus, the X, Y and Z components of the normals expressed in camera space are stored in the R, G, B channels of the texture, after being mapped to the $[0,1]$ range. Although no mipmapping will be applied to the depth texture, OpenGL's FrameBufferObject (FBO) extension requires all textures associated to a given FBO to have the same format [OpenGL]. Thus, we also store depth information using integer values in the $[0,1]$ range, which are obtained after dividing the depth values expressed in camera space by $-Z_{far}$, where Z_{far} is the far clipping plane distance. The minus sign comes from OpenGL's right-hand coordinate system convention. Although ideally the depth normalization to the $[0,1]$ interval should be done considering the Z range found on the depth map, this information is not available during the first pass. Such an ideal depth normalization would require extra rendering passes, since the range information needs to be available before the values are stored using the 8-bit integer representation, when precision is lost. Figure 4 shows examples of a normal and a depth map captured for the Stanford bunny. The black texels correspond to regions in the image plane containing no information about the back portion of the model.

The Z test is performed using GL_GREATER to keep the fragments that are furthest from camera. This is done in accordance to Wyman's observation that in case more than two interfaces map to given a fragment, using the furthest one tends to produce images that better approximate the actual refraction [Wyman 2005a].

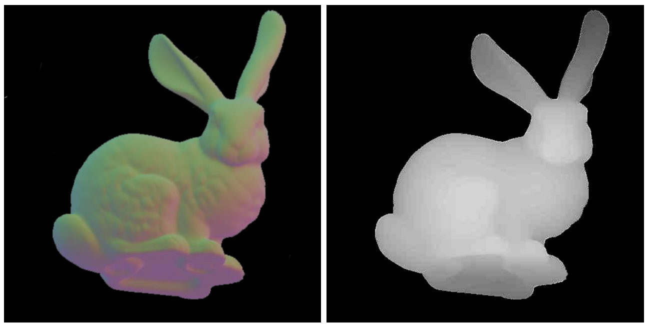

Figure 4: Normal (left) and depth maps of the back-facing geometry of the Stanford bunny as seen in perspective from a given viewpoint. Black pixels do not contain information about the projected object.

3.2 Intersecting the Second Interface

In our approach, we compute the position of point P_2 (Figure 2) by intersecting the ray refracted at the first interface (T_1) against the depth map saved during the first pass of the algorithm. Our approach differs from the GPU-based ray-height-field intersection by Policarpo et al. [Policarpo et al. 2005] in several ways: (i) we intersect the ray against dynamically generated depth maps created under perspective projection. Policarpo et al., on the other hand, used static depth maps created off-line under orthographic projection. (ii) In our approach, although the search is performed by advancing T_1 in a way that resembles a binary search, the ray can only advance, never recede. In [Policarpo et al. 2005], the intersection is computed using a linear search followed by a binary search refinement step; (iii) In our algorithm, the stored normals correspond to the back-facing portion of the model and are used for computing the direction of refracted rays, while in relief mapping [Policarpo et al. 2005] normals are used for shading visible surfaces.

By capturing and storing the depth maps under perspective projection, the linear mapping between 3D space and texture space that can be used when searching for the intersection of a ray with a height field does not hold anymore. For instance, consider the situation depicted in Figure 5. As the refracted ray T_1 advances into the scene along regularly spaced points represented by the green dots, the foreshortening caused by perspective projection requires that a variable step size be used to sample the depth texture at their corresponding projections. The crossings of the projection (dotted) lines with the image plane get closer to each other as T_1 gets further from the camera.

In Figure 5, the blue triangular region corresponds to a clipped version of the camera's view frustum, providing an approximate separation between the front and back-facing portions of the model for the given viewpoint. Z_{min} and Z_{max} indicate the minimum and maximum depth values obtained during the second pass of the algorithm.

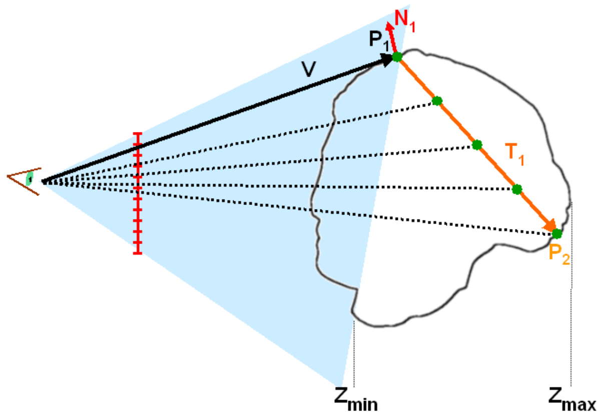

Figure 5: Nonlinear mapping of equally spaced distances traversed by the refracted ray T_1 in 3D and the lengths of their projections on the image plane. The blue region corresponds to a clipped version of the camera's view frustum, providing an approximate separation between the front and back-facing portions of the model for the given viewpoint. Z_{min} and Z_{max} are the minimum and maximum depth values from the back-facing geometry.

3.2.1 Nonlinear Search in Texture Space

This section describes a 2D texture-space approach to compute the intersection of the refracted ray T_1 with the second interface (represented by the saved depth map). This intersection procedure is performed for several fragments at a time, taking advantage of the parallel architectures of modern GPUs.

Let $S = (S_x, S_y, S_z)$ and $E = (E_x, E_y, E_z)$ be two points along the ray T_1, corresponding to the intersections of T_1 with the planes $Z = Z_{min}$ and $Z = Z_{max}$, respectively, expressed in camera space (Figure 6). Note that S and E do not need to fall inside the view frustum. Also let t_s and t_e be the 2D coordinates of the projections of S and E, respectively, on the camera's image plane. Once normalized to the $[0,1]^2$ domain, t_s and t_e define the starting and ending points for searching for an intersection between T_1 and the second interface in texture space. The coordinates of S and E are given by

$$S = P_1 + (Z_{min} - Z_{P_1})(\widehat{T_1}/\widehat{T_1}.z) \tag{3}$$

$$E = P_1 + (Z_{max} - Z_{P_1})(\widehat{T_1}/\widehat{T_1}.z) \tag{4}$$

where P_1 is the intersection of the viewing ray V with the front-facing portion of the refractive object (Figure 6), Z_{min} and Z_{max} are respectively the minimum and maximum depth values associated with the back-facing portion of the object, $\widehat{T_1}$ is the normalized vector representing the direction of the refracted ray T_1 at P_1, $\widehat{T_1}.z$ is the Z component of $\widehat{T_1}$, and Z_{P_1} is the Z coordinate of point P_1. Any point P along the segment from S to E can be expressed as the following linear combination, with $\beta \in [0,1]$

$$P = (1-\beta)S + \beta E \tag{5}$$

Likewise, any point t along the segment from t_s to t_e can be represented as

$$t = (1-\alpha)t_s + \alpha t_e \tag{6}$$

with $\alpha \in [0,1]$. However, due to the nonlinear mapping induced by the perspective distortion, $\beta \neq \alpha$, except at the end points of the interval.

Let $P = (P_x, P_y, P_z, 1)$ be the homogeneous representation of a point along T_1 with coordinates (P_x, P_y, P_z) defined in camera space, and let $\tilde{P} = (\tilde{P}_x, \tilde{P}_y, \tilde{P}_z, \tilde{P}_w) = \Pi P$, where Π is the camera's projection matrix. Similarly, let $\tilde{S} = (\tilde{S}_x, \tilde{S}_y, \tilde{S}_z, \tilde{S}_w)$ and $\tilde{E} = (\tilde{E}_x, \tilde{E}_y, \tilde{E}_z, \tilde{E}_w)$ be homogeneous representations of points S and E, respectively, under the same transformation. Recalling that pixel space and the 3D homogeneous space are related by a linear transformation [Blinn 1992], one can express the Z coordinate of a point along T_1 as

$$P_z = \frac{\tilde{P}_z}{\tilde{P}_w} = \frac{(1-\alpha)(S_z/\tilde{S}_w) + \alpha(E_z/\tilde{E}_w)}{(1-\alpha)(1/\tilde{S}_w) + \alpha(1/\tilde{E}_w)} \tag{7}$$

since $\tilde{P}_w = -P_z$ (assuming a right-hand coordinate system and the projection matrix used by OpenGL), then $\tilde{S}_w = -S_z$ and $\tilde{E}_w = -E_z$, and Equation 7 simplifies to

$$P_z = \frac{1}{(1-\alpha)(1/-S_z) + \alpha(1/-E_z)} \tag{8}$$

Equation 8 gives the Z coordinate, in camera space, of a point along the segment S-E as a function of the parameter α defined in image space. Solving Equation 8 for α, and substituting its value in Equation 6, one gets the image coordinates associated with the projection of point P on the image plane. These coordinates need to be normalized to the $[0,1]^2$ range before using them to sample the texture. The value of the parameter α is given by

$$\alpha = (\frac{1}{P_z} - k_1)k_2 \tag{9}$$

where $k_1 = (\frac{1}{S_z})$ and $k_2 = (\frac{S_z E_z}{S_z - E_z})$ are constants for a given ray T_1. Thus, given α, the texture coordinates needed to access the texture can be computed from Equation 6 as

$$texCoord = (1-\alpha)t'_s + \alpha t'_e \tag{10}$$

where $t'_s = 0.5t_s + 0.5$ and $t'_e = 0.5t_e + 0.5$. The 0.5 scaling and translation factors simply perform the mapping from the canonical projection range $[-1.0, 1.0]^2$ to the 2D texture range $[0.0, 1.0]^2$. The pseudo code shown in Algorithm 1 exemplifies the computation of the texture coordinates necessary to sample the depth texture at the projection of any point P along T_1.

```
// Given $P = (P_x, P_y, P_z)$, $S = (S_x, S_y, S_z)$ and $E = (E_x, E_y, E_z)$
// compute the constants $k_1$ and $k_2$
float $k_1 = 1/S.z$
float $k_2 = (S.z * E.z)/(S.z - E.z)$
// Project S and E to texture coordinates
float4 $\tilde{S}$ = mul(projectionMatrix, float4(S, 1.0f))
float2 $t'_s$ = ($\tilde{S}$.xy / $\tilde{S}$.w) * 0.5 + 0.5
float4 $\tilde{E}$ = mul(projectionMatrix, float4(E, 1.0f))
float2 $t'_e$ = ($\tilde{E}$.xy / $\tilde{E}$.w) * 0.5 + 0.5
// Search vector in texture coordinates
float2 $d_t = (t'_e - t'_s)$
...
// For any given point $P = (P_x, P_y, P_z)$ along the ray $T_1$
// $\alpha$ and texCoord are computed as
float $\alpha = (1/(P.z) - k_1) * k_2$
float2 $texCoord = t'_s + \alpha * d_t$
```

Algorithm 1: Pseudo code for computing the texture coordinates for sampling the depth texture at the projection of a point P along T_1.

3.3 Conservative Binary Search

In relief mapping [Policarpo et al. 2005], a two-stage approach is used to perform ray-height-field intersection in texture space: first, a linear search is used to identify the neighborhood around the intersection point, which is subsequently refined using a binary search. Since the goal of that technique is to map fine details to geometric surfaces facing the camera, the ability to intersect such delicate structures is crucial, thus justifying the cost of the linear search. When simulating refraction, however, the depth map represents surfaces hidden from the camera, making the application more tolerant to the missing of thin structures when compared to relief mapping. Moreover, when rendering non-convex refractive objects, more than two interfaces may project onto any given fragment, but our technique only stores the furthest one. That means that for the case of non-convex objects, one has only an approximate representation for the second interface. As such, we avoid the use of a linear search as it incurs in an extra cost while, according to our experience, the images produced with and without it are virtually indistinguishable from each other.

The binary search algorithm described in [Policarpo et al. 2005] is not appropriate for the kinds of depth and normal maps used in our approach for simulating refraction. While for relief mapping such maps cover the entire texture, that is not the case for the technique described in this paper. Figure 4 shows a typical pair of normal and depth maps. The black texels contain no information about the projected object, and their (R,G,B) channels are set to (0,0,0). Since the binary search computes an estimate for the intersection point, using a finite (small) number of steps, the search may end up in an invalid (*i.e.*, black) texel. Figure 6 illustrates this situation for the case of a hypothetical binary search using four steps. The light blue triangle represents the camera's field of view, whereas the white and gray triangle is the frustum defined by the object and the camera's center of projection. The gray portion of the frustum identifies the set of potential polygons that contribute to the depth and normal maps of the back-facing portion of the model. The search space covers the extension of ray T_1 from $T_1.z = Z_{min}$ to $T_1.z = Z_{max}$, which corresponds to points S and E in Figure 6. The order in which the points are visited during the binary search is indicated by the numbers 1 to 4. Although the process was converging toward the silhouette of the model, the last visited point (4) maps to a texel outside the silhouette of the object, resulting on an invalid normal ($N_2 = (0,0,0)$) and producing an undefined refracted ray direction at the second interface. This situation will manifest itself as noisy artifacts in the rendered images.

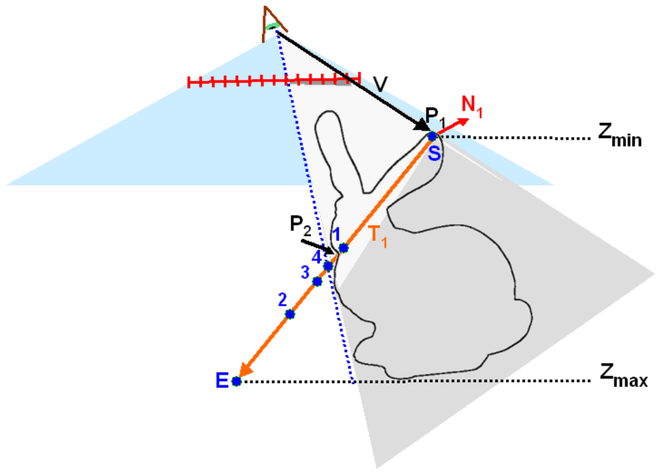

Figure 6: Problem when using a finite number of binary search steps to find P_2, the intersection of ray T_1 with the second interface. Point 4, the resulting approximation for P_2, projects onto an invalid sample on the saved depth and normal maps. The light blue region represents the complete field of view of the camera.

The inspection of Figure 6 suggests a simple solution to the problem just described: before stepping forward from P_i to P_{i+1} along T_1, one should check if P_{i+1} maps to a valid texel. In this case, one proceeds with the move; otherwise, the step size is halved and the test is repeated. We call this modified binary search *conservative binary search* (CBS). It guarantees that the estimate of the intersection point will always have a valid normal. Note that the sequence of checked points (*e.g.*, points 1 to 4 in the example shown in Figure 6) is exactly the same as for a conventional binary search, except that CBS will not move from point 1, returning it as an approximation for the actual intersection point P_2. Also note that since the stored depth values are zero outside the object's silhouette, sampling one such texel indicates that the ray has crossed the silhouette (*i.e.*, $T_1.z > storedDepth$) and T_1 should not advance. Algorithm 2 presents a pseudo code for performing the conservative binary search.

Assuming that P_2 can be exactly computed after k binary-search steps, the approximation error resulting from using only c steps of the CBS can be expressed in terms of the parameter β (see Equation 5) used for interpolation in 3D as :

$$\beta_{error} = \begin{cases} 0, & \text{if } k \leq c \\ (0.5)^c - (0.5)^k, & \text{otherwise} \end{cases}$$

Thus, for instance, $\beta_{error} \leq 3.125\%$ after 5 steps of CBS. Algorithm 2 shows a pseudocode for implementing a conservative binary search in 2D texture space.

```
// variables $k_1$, $k_2$, $t'_s$ and $d_t$ were computed in Algorithm 1
// initialize $P_2$ with $S$ and compute the search range in 3D
float3 $P_2 = S$
float3 $d_v = (E - S)$
// start the conservative binary search loop
for (i=0; i < conservative_binary_search_steps; i++) {
    $d_v * = 0.5$
    float $P.z = P_2.z + d_v.z$
    float $\alpha = (1/(P.z) - k_1) * k_2$
    float2 texCoord = $t'_s + \alpha * d_t$
    // sample depth stored in the texture
    storedDepth = f1tex2D(DepthTex,texCoord)
    storedDepth *= $Z_{far}$        // recover a positive Z value
    // check if it is ok to advance
    // invalid texels have depth = 0.0
    if ($-P.z <$ storedDepth) {
        $P_2 + = d_v$
    }
// now use texCoord to sample $N_2$ from the normal texture
// then compute $T_2$ and sample the environment map.
// Use the resulting color to shade the fragment or use a
// Fresnel approximation to combine reflection and refraction
 ...
}
```

Algorithm 2: Pseudo code for a conservative binary search in 2D texture space. The negative sign in $-P.z$ (if statement) compensates for the fact that OpenGL uses a right-hand coordinate system.

4 Results

We have implemented the refraction technique described in the paper as a set of shaders written in Cg [Mark et al. 2003]. The host program was written in C++ and OpenGL. Both depth and normal maps saved during the first pass were stored as 32-bit-per-texel RGBA textures. We used our technique to render the approximate refraction of distant environments through several geometric models and visually compared these results to the ones obtained using three other techniques: single-interface refraction (SIR), image-space refraction (ISR) [Wyman 2005a], and ray tracing (RT) [Whitted 1980]. The single-interface refraction technique was implemented as described in [Fernando and Kilgard 2003]. For ISR, we used the shaders available at Wyman's website [Wyman 2006]. The ray-traced images were generated with pbrt [Pharr and Humphreys] with 32 samples per pixel. For all images shown in the paper we set the material index of refraction to 1.2. All examples rendered with our technique, including the accompanying video, were produced using 5 steps of the conservative binary search.

Figure 1 shows refractions through the Armadillo model. On the left, one sees a ray traced image used for reference. The image in the center was rendered using our technique from the same viewpoint. Note the similarity between them. The differences, basically noticeable on the feet, are primarily due to the occurrence of total internal reflection (TIR), which is not simulated by the current version of our technique. The image on the right shows the Armadillo during a jump, also rendered using our technique. The accompanying video was recorded in real time and shows several examples of models undergoing deformations.

Figure 7 shows the dragon model rendered using the four different techniques for comparison. Figure 7 (a) shows the result produced by our technique. (b), (c) and (d) show the renderings produced with the use of image-space refraction [Wyman 2005a], ray tracing, and single-interface refraction, respectively. Although there are small differences between Figures 7 (a) and (b), they are virtually identical and also very similar to the ray-traced image in (c). Essentially, the differences between (a) and (b), and the reference image (c) are in regions where total internal reflection occurs, since the phenomenon is simulated by the ray tracer, but not by the other techniques. Although it is possible to find viewing configurations for which our technique exhibits results that are slightly closer to the image produced by the ray tracer (for some parts of the model), when compared to ISR, in general, the images produced by both technique are very similar. This can be understood considering the fact that most models are not convex (*i.e.*, they present more than two interfaces as seen from most viewpoints) and, although our approach for computing P_2 tends to be less inaccurate than the one used in [Wyman 2005a], for geometric complex objects the saved depth maps tend to store a rough approximation for the object's back surface anyway. Thus, the main practical advantages of our technique over ISR are the ability to handle dynamic models in real time and to support any object representation that can be rasterized on a programmable GPU.

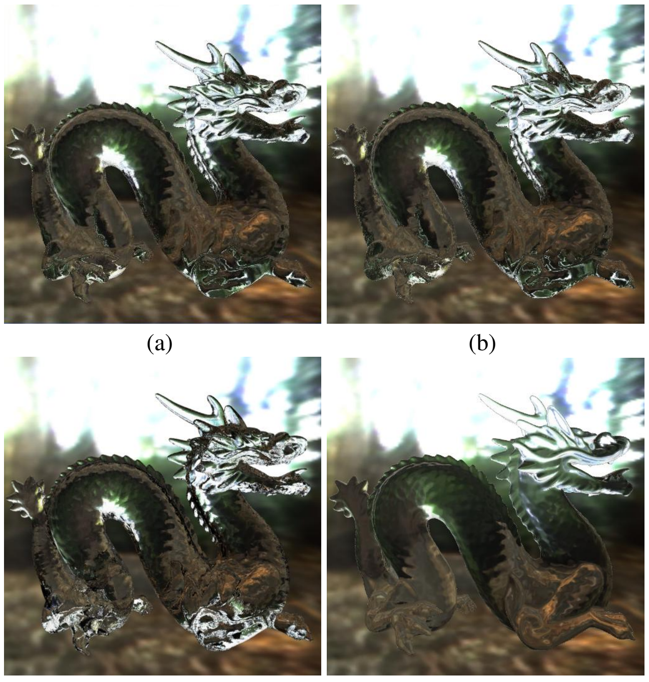

Figure 7: Dragon model rendered using four different refraction techniques: (a) Our technique; (b) Image-Space Refraction [Wyman 2005a]; (c) Ray tracing; (d) Single Interface Refraction.

Figure 8 presents two frames of an animation showing a deforming bunny. The animation was produced using a simple vertex program to perturb the coordinates of the vertices. Note the changes on the chest, head, ears and feet, which can be observed by the changes in the refracted environment. The accompanying video has a complete animation sequence of the deforming bunny. Figure 9 shows two renderings of the Laçador model, a statue of a gaucho, which is a symbol of the city of Porto Alegre, in Brazil. The image on the left was rendered using our technique, while the image on the right is a ray-traced version generated using pbrt. Note how the two images are similar. Again, the differences are mostly associated with the occurrence of TIR.

Table 1 summarizes the performance of our algorithm (DIR) for several different models and compares it with the other three techniques. The table also shows the number of polygons and the num-

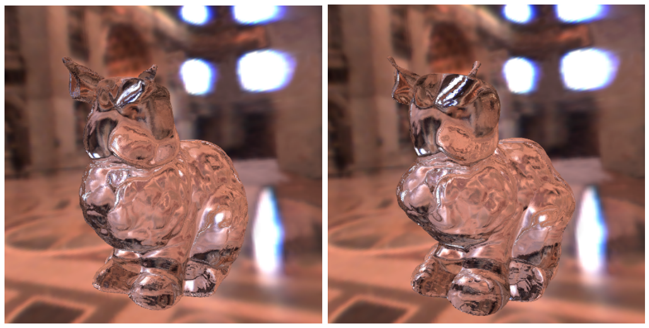

Figure 8: Rendering dynamic geometry. The images show two frames extracted from an animation showing a deforming bunny. Note the changes on the chest, head, ears and feet, which can be observed by the changes in the refracted environment.

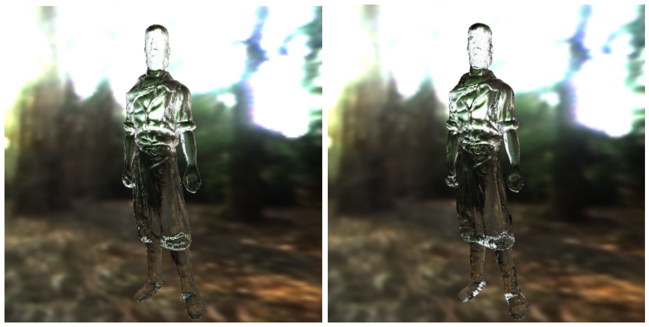

Figure 9: Laçador model rendered using our technique (left) and using ray tracing (right).

ber of pixels covered by the projection of the refractive objects on the image plane (second and third columns, respectively). The measurements were performed by rendering images with 512x512 pixels using an Athlon64 3500+ (2.21 GHz) PC with 2 GB of memory and a PCI express 16x GeForce 7800 GTX with 256 MB of memory. When rendering each object, all techniques used the same viewing configurations, as illustrated in the example of Figure 7. Each object was scaled to a maximum size that would still completely fit in the image. Note from the last two rows of Table 1 that the performances of DIR, ISR and SIR are limited by the number of polygons that need to be sent to GPU, transformed and rasterized, and not by the number of pixel covered on the screen (for these three algorithms we store the models using display lists). The graph shown in Figure 10 compares the performance (in fps) of DIR and ISR for the examples listed in Table 1. As the number of polygons increases, the performance gap between the two techniques disappear, while DIR retains its extra advantages.

The accompanying videos were recorded in real time on the same machine, also at 512x512 pixels. During editing, the videos were resized to 400x400 pixels in order to reduce storage requirements.

5 Discussion

Our technique provides only an approximate solution for rendering the refraction of distant environments. Like Wyman's technique [Wyman 2005a], ours also does not simulate total internal reflection (TIR). Whenever the computed transmission angle happens to be bigger than the critical angle, we make it equal to the critical angle, causing the refracted ray to be tangent to the surface. This strategy was proposed by Wyman [Wyman 2005a].

Table 1: Performance of our algorithm (DIR) on different models and comparison with other techniques. Performance in fps, except for RT which is expressed in seconds.

Model	polygons	pixels	DIR	ISR	SIR	RT
Buddha	50,000	56,739	288	345	601	151 s
Bunny	69,630	89,050	210	261	390	139 s
Venus	100,000	91,360	166	204	317	137 s
Laçador	153,372	37,445	156	163	311	60 s
Dragon	250,000	113,845	80	83	161	286 s
Armadillo	345,944	66,191	74	78	169	105 s

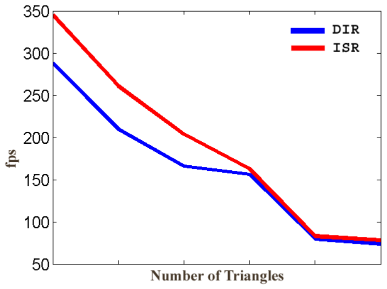

Figure 10: Performance comparison (in fps) of the techniques DIR and ISR using the data shown in Table 1. As the number of polygons increase, the performances of the two techniques converge.

Using the depth buffer to save a representation for the back portion of the model can only provide an approximation for the back surface. Simulating TIR and refractions at more than two interfaces can be achieved using a multilayer representation that stores both front and back-facing depth and normal information, similar to the one used in [Policarpo and Oliveira 2006]. In this case, however, one needs to save the depth and normal maps in perspective projection. Our ray-intersection procedure described in the paper would directly work for these extensions, simply requiring to follow the extra bouncing and refracted rays. Dynamically acquiring multiple depth layers for deforming geometry can be done using depth peeling [Mammen 1989] at the cost of additional rendering passes. Apparently, the extra cost required for computing refractions at multiple interfaces does not seem to be justifiable in general, as the results obtained with the use of techniques such as DIR and ISR are already very similar to ray-traced ones. Simulating TIR might be a more promising direction for exploration, but due to the recursive nature of the phenomenon, rays may bounce many times before escaping to the external medium, which will impact the performance of the technique.

Although uncommon, depending on the geometry of the front-facing portion of the object, the intersection of the refracted ray T_1 with the second interface may happen on the front-facing geometry. Thus, for instance, consider a configuration similar to the one depicted in Figure 6, where T_1 would hit the second interface (point P_2) still inside the white portion of the frustum. In these situations, the normal used to compute T_2 would still be sampled from the back portion of the model.

Currently, our technique does not handle refractions of nearby geometry [Wyman 2005b]. Since our technique provides good estimates for the 3D coordinates of P_2, we would like to explore this subject in the future.

6 Conclusion

We have presented a real-time technique for rendering approximate refractions of distant environments through objects. Unlike previous techniques, our approach can be directly applied to models undergoing shape deformation as well as to any object representation that can be rasterized on a GPU. Also, our approach needs no preprocessing and requires less memory than competing techniques.

We introduced an efficient GPU algorithm for computing ray intersection against a depth image represented in perspective projection. The search for the intersection is performed in 2D texture space. We have also presented *conservative binary search*, a variation of the binary search algorithm for 2D texture space that avoids sampling invalid texels. For this, we presented a bound on the error associated with its conservative estimation.

We have demonstrated the effectiveness of our approach by rendering and animating several models consisting of different number of polygons. We have shown that as the number of polygons of the refractive object increases, the performances of ISR [Wyman 2005a] and of our approach converge. Our technique, however, is more general in the sense it can be directly applied to any kind of model representation that can be rasterized.

A possible way of accelerating our technique is to avoid the parallel reduction (second pass of the algorithm), using the limits of the object's bounding box as Z_{min} and Z_{max}, since these are only used to constrain the search space. We would also like to explore the simulation of total internal reflection, as it is currently the biggest cause of differences between images rendered with our technique and ray-traced ones.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful suggestions. The Laçador model was provided by LdSM-UFRGS (http://www.ufrgs.br/ndsm/). The Armadillo, dragon and bunny models were provided by the Stanford 3D scanning repository. NVIDIA donated the GeForce 7800 GTX video card used in this research. This work was partially supported by Microsoft Brazil.

References

BLINN, J. 1992. Hyperbolic interpolation. *IEEE Computer Graphics and Applications 12*, 4, 89–94.

DIEFENBACH, P., AND BADLER, N. 1997. Multi-pass pipeline rendering: Realism for dynamic environments. In *Proc. of I3D 1997*, ACM Press, 59–70.

FERNANDO, R., AND KILGARD, M. 2003. *The Cg Tutorial. Section 7.3.* Addison Wesley, 182–188.

GENEVAUX, O., LARUE, F., AND DISCHLER, J.-M. 2006. Interactive refraction on complex static geometry using spherical harmonics. In *Proc. of I3D 2006*, 145–152.

GUY, S., AND SOLER, C. 2004. Graphics gems revisited: fast and physically-based rendering of gemstones. *ACM Trans. Graph. 23*, 3, 231–238.

HAKURA, Z. S., AND SNYDER, J. M. 2001. Realistic reflections and refractions on graphics hardware with hybrid rendering and layered environment maps. In *Proc. 12th Eurographics Workshop on Rendering Techniques*, Springer-Verlag, 289–300.

HARRIS, M. 2005. *Mapping Computaional Concepts to GPUs. In GPU Gems 2, Matt Pharr (editor).* Addison Wesley, 493–508.

HEIDRICH, W., LENSCH, H., COHEN, M., AND SEIDEL, H.-P. 1999. Light field techniqes for reflections and refractions. In *Proc. of EGWR 1999*, Springer-Verlag, 187–196.

LINDHOLM, E., KLIGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proc. SIGGRAPH 2001*, ACM Press, 149–158.

MAMMEN, A. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl. 9*, 4, 43–55.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph. 22*, 3, 896–907.

OLIVEIRA, G., 2000. Refractive texture mapping. http://www.gamasutra.com/feautures/20001117_01.htm.

OPENGL. The opengl framebuffer object extension. http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt.

PHARR, M., AND HUMPHREYS, G. Physically-based ray tracing. http://www.pbrt.org/.

POLICARPO, F., AND OLIVEIRA, M. M. 2006. Relief mapping of non-height-field surface details. In *Proc. of I3D 2006*, ACM Press, 55–62.

POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In *Proc. of I3D 2005*, ACM Press, 155–162.

SCHMIDT, C., 2003. Simulating refraction using geometric transforms. Master's thesis. CS Department. University of Utah.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6, 343–349.

WILLIAMS, L. 1983. Pyramidal parametrics. In *Siggraph 1983, Computer Graphics Proceedings*, 1–11.

WYMAN, C. 2005. An approximate image-space approach for interactive refraction. *ACM Trans. Graph. 24*, 3, 1050–1053.

WYMAN, C. 2005. Interactive image-space refraction of nearby geometry. In *Proceedings of GRAPHITE*, 205–211.

WYMAN, C., 2006. Chris wyman's website. http://www.cs.uiowa.edu/ cwyman/publications/.