Visual Comput (2005) 21: 579–590
DOI 10.1007/s00371-005-0312-4

ORIGINAL ARTICLE

Bin Chan
Wenping Wang

Geocube – GPU accelerated real-time rendering of transparency and translucency

Published online: 1 September 2005
© Springer-Verlag 2005

B. Chan (✉) · W. Wang
The University of Hong Kong
{bchan,wenpin}@cs.hku.hk

Abstract We present a new method based on GPU acceleration for real-time transparency and translucency rendering. Our method computes refraction at both the front and back sides of a transparent object, as well as internal reflection, thus delivering interactive realistic transparency effects on a commodity PC. The real-time performance is made possible by a new acceleration data structure, called *geocube*, that enables the use of GPU for fast ray-surface intersection testing. In addition, within the same framework, we introduce the novel use of the mip-map for a hierarchical representation of a sequence of key prefiltered environment maps to simulate translucency. By taking ray depth into account and using GPU to interpolate the key filtered maps to produce the desired blurring effects, we achieve real-time realistic translucency rendering of slightly scattering media that allows show-through of background details.

Keywords Real-time rendering · GPU · Transparency · Translucency

1 Introduction

Translucency is important to realistic graphics since many substances in nature are translucent. Some materials, such as human skin, plastic, even stones, that are generally considered opaque are, in fact, to some degree translucent. Many recent papers (e.g., [6] and [4]) have demonstrated the importance of translucency in image synthesis.

Transparency, being an extreme case of translucency, has proven useful to many graphics applications, from realistic game design to data visualization. For example, proper simulation of refraction is important for placing a transparent object in a 3D environment [12]. Also, in surface-based medical data visualization, the front layer of an object surface usually needs to be made semitransparent in order for objects behind to show through [10].

Simple transparency, such as that used in data visualization, can be generated with straightforward α blending, without considering light refraction [1]. Most real-time transparency shaders found today support only refraction at the front facing side of a transparent object. They produce an impression of transparency that is good enough for certain applications but far from realistic. More realistic transparency rendering entails the consideration of refraction at both the front and back sides, as well as internal refraction. Their effects are quite different, as shown in Fig. 1.

Ray tracing is a well-known technique for highly realistic simulation of both reflection and refraction in a specular environment. But, due to its intensive computation requirement, most real-time methods for ray tracing require special computing environments, such as a shared-memory machine or a cluster [9, 13, 14]. Although the rapid advance of graphics hardware (i.e., GPU) has brought about significant speedup for ray tracing [8], realistic transparency rendering based on the conventional ray-tracing approach has yet to achieve real-time performance on commodity PCs.

Translucency covers a wide range of media—from transparency to opaqueness. Translucenct objects are inherently more difficult to render than transparent ob-

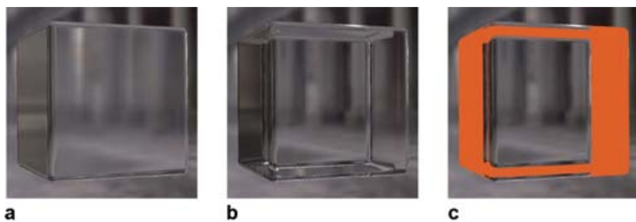


Fig. 1. a Front-face-only refraction. **b** Refraction at both front and back faces, and internal reflection. **c** Highlighted area (*orange*) involves internal reflection at least once

jects because of the scattering nature of light that causes translucency. Traditionally, translucency is rendered with global illumination techniques such as Monte Carlo ray tracing or photon mapping [5]. More recently, fast methods have been proposed [2, 4, 6, 11] to speed up its rendering; some of these methods can even achieve real-time speed.

However, these recent methods attempting real-time translucency rendering focus on subsurface scattering. A common assumption of these methods is that objects under consideration are highly scattering. Therefore, blurred color bleeding from the backside of a translucent object can be simulated but visual details in the background cannot be rendered in a properly blurred manner, especially for a thin or slightly scattering object. Figure 2 shows that the degree of blurring should depend on the thickness of a translucent object.

The present paper contains two contributions.

- We present a transparency shader for real-time transparency rendering on a commodity PC with GPU acceleration. Our method takes into account both the front and back ends of refraction, as well as internal reflection; therefore it delivers highly realistic transparency effects. We use a novel acceleration structure, called *geocube*, for object surface representation that supports a novel scheme for fast ray–surface intersection with GPU acceleration.

Fig. 2. Blurring effects of background details showing through a wedge-shaped object. The *right side* is thicker than the *left*. The image of the lamp is blurred more at the *right*

- Taking into account ray depth as computed by the transparency shader, we extend our transparency shader to render a wide range of translucency, from slightly scattering media to opaque objects. Our method is capable of producing properly blurred rendering of background details through translucent but thin objects or slighly scattering objects, as shown in Fig. 2. A novel idea here is to use mip-map for hierarchical representation of a sequence of key filtered environment maps, so that we can achieve real-time translucency rendering by using GPU to interpolate the key filtered environment maps.

2 Transparency shader

Our transparency shader considers not only an object's front-side refraction, but also the back-side refraction, as well as internal reflection up to one level. The key to efficiently computing the back-side refraction is fast computation of the exit point of a light ray entering the object; the outgoing ray at the exit point will be used as texture coordinates in an environment map for synthesizing the final image. Conventionally, the exit point is found in ray tracing by traversing through the mesh surface of an object to locate the closest ray–triangle intersections. Using spatial partition or hierarchical bounding to speed up this approach for real-time performance is currently still an active research topic. At present, directly computing ray–triangle intersections by encoding object meshes and other spatial data as textures for speedup using graphics hardware seems neither natural nor practical; it is difficult to map irregular meshes to graphics hardware features without complicated conversion, encoding, and tedious access [8].

2.1 Geocube

In this section we present a new rasterized geometry data structure, called *geocube*, for encoding an object's surface using radial distances and storing other information for quick computation of a light ray's exit point. Given an object K, we place a cube G with its center point O at an interior point of K. Here G is a cube image, i.e., its six sides are pixelized images. For the moment we suppose that K is star-shaped and that O is in the kernel of K; thus, all boundary points of K are internally visible to O. (Later we will discuss how to represent a general object with mutiple cubes and other extensions.) We sample the direction space in correspondence with the pixel centers of the cube image G. Let v be the direction from the cube center O to the surface sample point p. Then distance $d = |Op|$ from O to p is stored in the pixel location given by direction v (Fig. 3a). Therefore, a geocube provides the surface representation of an object with image resolution. A cube image

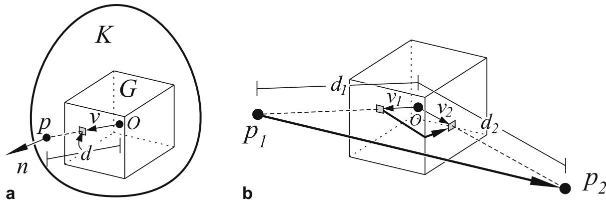


Fig. 3. Geocube definition

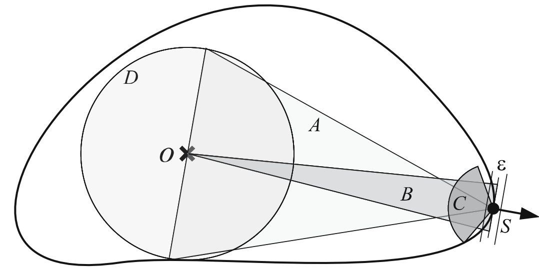


Fig. 4. Cones A, B, and C and sphere D for a surface point S. O is the geocube center

is used because it can be accessed with fast texture fetching functions of GPU. Besides the distances, the normal vectors at surface sample points are also recorded in the geocube for computing reflected or refracted ray directions.

Now we discuss how the distance values in a geocube can be used to determine the exit point of a light ray. Figure 3b shows a ray traveling inside an object from p_1 to p_2, which are two surface points. At the beginning, from the incident point p_1 we can easily find its direction, or its texture coordinates v_1. The goal now is to locate the exit point p_2 or, equivalently, its direction v_2, because the surface distance $d_2 = |Op_2|$ is recorded in the geocube. A straightforward method for finding p_2 is to move a point on the ray from p_1 incrementally forward with an appropriate step size. After every step, the distance $|Op|$ from the new moving point p to the cube center O is compared with the stored surface distance d in the geocube in the direction of p. Once a stored value d is found to be equal to $|Op|$, the exit point p_2 is found. Note that the distance value of a point falling between sample points is obtained by interpolation of the stored distance values for its nearby sample points.

In the above procedure we need to determine an appropriate step size— a too-large step size may cause overshooting and inaccuracy, while a too-small step will slow down the search. For instance, simply sampling along a ray path according to the pixel resolution of the geocube is similar to the DDA algorithm for drawing a line in 3D. This approach normally involves operations of hundreds of pixels for traversing each ray and is therefore inefficient, as observed in [8].

Based on the geocube, we use the following scheme to quickly find the exit point of a light ray on the boundary surface. In a preprocessing stage we create many simple solids lying completely inside an object K, with their union covering K (up to image resolution). Given a moving point p on a light ray r, if we find that p is inside one of these simple solids, then a quick computation is performed to send the point along the ray to a boundary point of the solid. In this way, the moving point can quickly be "propelled" to its exit point in a small number of steps. We believe there are many possible shapes for these simple solids. In our implementation, we choose to use three circular cones, called *cone A*, *cone B*, and *cone C*, for each surface sample point (Fig. 4) and record the defining parameters of these cones in the corresoponding pixel of the geocube. The following are the detailed descriptions of the three cones.

- Cone A has its apex at a surface sample point S. Its base center is at the geocube center O. The base radius is maximized subject to the degree to which the cone is inside the solid. For cone A, $\cos\theta_A$ is recorded, where θ_A is the half-apex angle. The intended function of cone A is to push a moving point near the cube center O along its light ray.
- Cone B has its apex at O and has its base center at a surface sample point S. Its base plane is the tangent plane of the boundary surface at point S. Two planes, each offset from the tangent plane at S by plus or minus half of some constant ϵ in the normal direction, are used to isolate a set of mesh triangles of the object surface near S, denoted by set V. Cone B is then set to be the smallest cone that contains set V. Like cone A, $\cos\theta_B$ is recorded for cone B, where θ_B is the half-apex angle. Since cone A's apex lies near the surface, its contribution diminishes as a moving point approaches the surface. At some point near the surface, cone B will take over to push the moving point toward its exit point.
- Cone C is not exactly a flat-based cone but a conical portion of a sphere. Its apex is at a surface sample point S, its principal axis is aligned with the surface normal vector at the sample point S, and its half-apex angle is set to be the critical angle or $arcsin(1/\eta)$, where η is the refractive index of the object material. The size of cone C, i.e., the radius R_C of the sphere containing cone C, is maximized subject to the extent that cone C lies completely inside the object. Only the radius R_C is recorded for cone C, while its half-apex angle $arcsin(1/\eta)$ is a constant for all surface sample points. The function of cone C is to ensure that any ray entering the object can travel by at least the distance R_C before reaching the surface again. Therefore, cone C is particularly useful in avoiding small initial steps when cone B is narrow.

Note that the geocube's sampling ray directions are very close to each other near the center, so a small numerical error in position calculation may cause a large quantization error in choosing the correct geocube pixels. To prevent the moving point from getting too close to the cube center, we precompute and record the largest inscribed sphere D centered at the geocube center. Thus, when a moving point is detected to have entered sphere D, it will be pushed along the ray to the surface of D.

2.2 Finding the exit point

Given a moving point p on a ray r, in every step an enclosing cone or sphere D is used to determine a step length to find a new position of p along the ray. Normally the center-to-p direction falls between the recorded directions. Then the retrieved parameters for the cones are computed by linearly interpolating the values stored in the nearby pixels in the geocube. This step of interpolation is, again, done by GPU. We compute using GPU the incremental distance values produced by both cones A and B, and also sphere D if it contains the point, and use the largest of the three values to move point p forward. The new position of p is then used to obtain a new direction to retrieve a new set of cones stored in the geocube, and these new cones will be used to further push the moving point toward its exit point.

The following is the pseudocode for the transparency shader:

```
1:  found ← false
2:  direction = position − center
3:  geocube = fetchGeoCube(direction)
4:  position+ = direction ∗ geocube.C
5:  while found ≠ true do
6:      direction = position − center
7:      if direction.magnitude < D then
8:          position+ = direction ∗ GetDDistance()
9:      else
10:         geocube = fetchGeoCube(direction)
11:         if geocube.distance < direction.magnitude then
12:             found = true
13:         else
14:             delta = max(GetADistance(), GetBDistance())
15:             position+ = direction ∗ delta
16:         end if
17:     end if
18: end while
```

In every step we need to quickly compute the incremental distance of a moving point as determined by predefined cone A, cone B, or sphere D. The formulas for computing the distances depend essentially on two variables so they can be easily evaluated by GPU using table lookup. First, cone C's value (i.e., its radius R_C) is used directly, so no formula is needed. Next we consider the circular cones (A and B) and sphere D. Suppose that moving point p on a ray r is contained in cone A or cone B. Since p is on the principal axis of the cone, by symmetry we reduce the problem to distance computation in 2D by con-

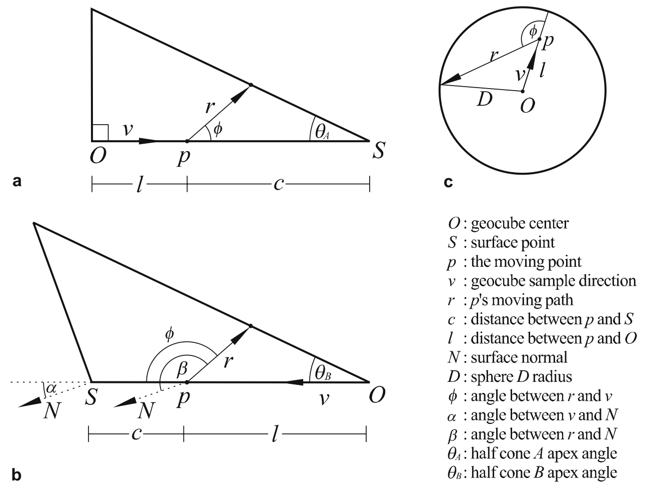


Fig. 5a–c. Computation of step distance r for cones and sphere. v is the principal axis vector. **a**, **b** and **c** show 2D sections for cones A and B and sphere D

sidering the planar section of the cone on the plane passing through the cone's principal axis and the ray r (Fig. 5). Figure 5a shows the section of cone A that is a right triangle since A is a right circular cone. Figure 5b shows the section for cone B; note that the roles of S and O are reversed and that the section of B is, in general, not a right triangle. Since the base plane of cone B is not necessarily perpendicular to its principal axis, the base normal N, in general, does not lie in the sectional plane. Finally, Fig. 5c shows the section of sphere D on the plane passing through ray r and the cube center O.

Since cone A is a special case of the more general cone B, we use the same formula to find $|r|$ for both cones A and B. We have derived two functions: $C(f, \hat{h}, \hat{a}, \hat{n}, \mu)$ for computing $|r|$ for a *unit* cone (i.e., the distance between its apex and base center is one) and $S(f, \hat{h}, \hat{v})$ for computing $|r|$ for a unit sphere, where f is the distance of point p from the base center, $\hat{h}$ is the unit ray direction, $\hat{a}$ is the unit principal axis vector from the apex to the base, $\hat{v}$ is the unit principal axis vector from the sphere center, $\hat{n}$ is the unit base normal vector, and μ is the half-apex angle of the cone. The two formulas $C(f, \hat{h}, \hat{a}, \hat{n}, \mu)$ and $S(f, \hat{h}, \hat{v})$ are given in the appendix. Note that all vector symbols with hats denote unit vectors. Based on these two functions we have

$$|r| = (c+l)\ C(l/(c+l), \hat{r}, -\hat{v}, -\hat{v}, \cos\theta_A), \quad \text{for cone } A;$$
$$|r| = (c+l)\ C(c/(c+l), \hat{r}, \hat{v}, \hat{N}, \cos\theta_B), \quad \text{for cone } B;$$
$$|r| = D\ S(l/D, \hat{r}, \hat{v}), \quad \text{for sphere } D.$$

Here $\hat{r}$ is the unit direction vector of the ray, and $\hat{v}$ is the unit direction vector from the cube center O to the surface sample point S.

2.3 Multiple cubes

One geocube can only represent a star-shaped object. Therefore, more geocubes are needed to represent a complex object. Assume that a number of cubes have been placed inside an object so every surface point is visible to at least one of these cubes' centers. It is then important to consider the switching between different cubes in relaying a moving point on a ray to its exit point. Let G_1 be the current cube for a moving point p. When p is approaching a region R that is invisible to G_1 (or the center O_1 of G_1, to be precise), cone A and cone B associated with p will become narrower and the step size of p will get smaller, as shown in Fig. 6. To avoid infinite looping, we set a minimum value δ for the step size of p so that in every step the moving point will be advanced by at least the amount δ. At the moment p steps into region R, the line segment $O_1 p$ becomes longer than the recorded surface distance in G_1 in the direction of p, and this triggers a cube-switching event—that is, we need to find a new cube G_2 that is visible to p to replace cube G_1.

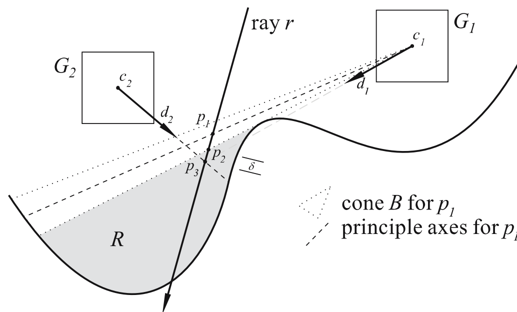


Fig. 6. A point is moving into a region invisible to the current cube G_1. p_1 is a point close to region R. Its cone B will bring p_1 to p_2. At p_2, both cones A and B degenerate into lines and cannot push the point further. A lower bound δ on the step size will be added to move the point to p_3. This causes a cube-switching event

The new cube G_2 is selected as follows. Let P be the plane passing through p and perpendicular to the direction from p to the center O_1 of the current cube G_1 (shown as the dotted line in Fig. 7a). Then we choose G_2 to be the cube whose center is nearest to the plane P among all the cubes. If there is more than one such cube having the same distance to plane P, we choose an arbitrary one to break the tie.

In precomputation, for each triangle we find among all the cubes the most suitable one to use when a ray enters the object through the triangle. Here we choose the nearest cube that the triangle is visible to. After the initial cubes for all triangles are determined, the indices of these cubes are passed as texture coordinates with the triangles.

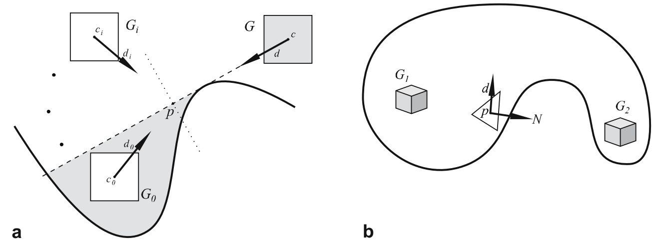


Fig. 7. a Geocube switching. G is the current geocube. Geocubes near the *dotted line* are most likely to be chosen as the new one. **b** Initial geocube selection. Both G_1 and G_2 are equally likely to be chosen by the metric. Internal visibility must be tested to eliminate G_2

3 Translucency rendering

In this section we consider extending the transparency shader described above to translucency rendering, especially for slightly scattering objects that allow show-through of background details. Assuming homogenuous scattering, the degree of the blurring effect produced by a translucent object depends on the length of a ray going through the object; recall that the ray length is readily available from our geocube-based transparency shader. We use environment map filtering to produce the effect of translucency or bluring. Thus we need to consider how a certain degree of translucency should be represented by an appropriate degree of blurring of the environment map.

The environment map filtering has been used to produce a reflection effect of object surfaces [7] in which the environment map is blurred by filtering to make an object with a mirror surface appear to have a glossy surface. It is assumed there that an object has the same surface material properties everywhere. Therefore, one single filtered environment map suffices for the whole object. Moreover, the degree of blurring in that application does not depend on the viewpoint or the object geometry.

Our idea of using a blurred environment map to simulate translucency was inspired by the work in [7], but our situation is different and more challenging. We need to model medium properties instead of surface properties. Unlike reflection, light travels into a translucent object and gets scattered more as it goes deeper. Since the optical lengths of different rays inside the object are different, we need to produce in real time different degrees of blurring (i.e., translucency) at different locations; this makes it difficult to use just a single filtered environment. Note that filtering an environment map on the fly with nonuniform filter widths is too costly for real-time rendering. Our strategy is to precompute a number of key filtered environment maps to cover a wide range of blurring effects. Each of these key maps is Gaussian filtered with a uniform filter width, and we use GPU to interpolate these key maps in real time to synthesize the final image with the desired

degree of blurring at different locations. Below we discuss two key issues in our solution: (1) generating a key filtered environment map and (2) storing all key filtered maps with minimal consumption of limited texture memory on the graphics board. Since the first task is performed offline in preprocessing, we just need to ensure that it is done with adequate accuracy. The second issue, however, is more critical to real-time performance because the compact storage of all key filtered maps is mandatory for using GPU to access and interpolate these maps.

3.1 Map filtering

The environment map we use in the shader is a cube map stored as six separate images. The six images should not be filtered independently of each other, since that would leave unproperly filtered seams between adjacent faces. So we map the cube map into a spherical map and perform filtering in a parameter domain of the spherical map. There are several ways of filtering a spherical map. One may use spherical harmonics. This approach is not ideal for low blurring filters with a narrow filter width, since that involves inaccurate manipulation of high-order terms in spherical harmonics. The spherical environment map is filtered in [7] in the spherical coordinate space, which is a rectangle-shaped panoramic map. This scheme is a bit involved since it uses different elliptic-shaped filter supports at different pixel locations to compensate for filter distortion caused by parameterization.

In our implementation, we take the following simple but approximate approach. We cover the spherical map using three different spherical coordinate systems, with their equatorial planes perpendicular to each other. Each of these spherical coordinate systems gives rise to a longitude/latitude panoramic mapping. Note that there is relatively little distortion for a large part of the image along the equator, and there are only two singularities at the two poles of the sphere in that mapping. (There is no mapping from a sphere to a rectangular image without singularity.)

Then we use the normal Gaussian filtering with a constant circular filter support (i.e., constant filter width) to filter these three panoramic maps. Each filtered image is quite accurate near the equator but becomes less accurate toward the two poles. To remove the singularities at the poles and reduce distortion off the equators, we use a weighted combination of the three filtered images to produce the final filtered map so that pixels near equators are given higher weights than pixels off the equators, and pixels near the poles make little contribution to the final filtered map. The singularties are removed becasue any pole in one map is covered by the equators in the other two maps. This approximation scheme is easy to code and produces satisfactory results in our tests. It should be pointed out that the more accurate, but slower, filtering method in [7] can also be used for our purpose, since computing the filtered kep maps is a preprocessing step that does not affect the subsequent real-time performance in translucency rendering.

3.2 Mip-map representation of key filtered maps

We use some key filtered maps to cover a sufficiently wide range of blurring, with the inbetween blurring effect computed by interpolating the key filtered maps. Generally speaking, the more key filtered maps are used, the more accurate is the interpolation. However, if we use high-quality environment maps, the amount of texture memory consumed by the key filtered maps becomes a concern. For example, a $512 \times 512 \times 6$ 16-bit-per-component floating point HDR cube map takes 12 MB to store. Hence, it is not practical to use too many full-resolution key filtered maps since most graphics boards today have only 128 MB of memory.

To reduce texture memory consumption, we observe that filtered maps with a sufficient degree of blurring do not need full-resolution representation since, intuitively, they contain fewer details than the source image. Thus, the image resolution can be reduced by half for each increasing level of blurring. A natural choice is to use $\log_2 n$ key filtered maps, where $n \times n$ is the resolution of each face of the source cube map; the full-resolution one is the source image and the smallest one is a $1 \times 1 \times 6$ cube map that contains the most blurred image. This hierarchical representation maps well to the mip-map supported by graphics boards.

An image with reduced resolution imposes a lower bound on the degree of blurring that it can accomodate. That is, when the image is less blurred than that lower bound, some image details may be suppressed by the reduced resolution (Fig. 8). We can analyze this constraint imposed by resolution reduction on the degree of blurring as follows. Suppose that I is the source image. Let I_b a Gaussian blurred image with full resolution. Let I_{bs} be the down-sampled version of I_b. Then

$$I_b = I \circ G_r$$
$$I_{bs} = I \circ G_r \circ (R \circ E),$$

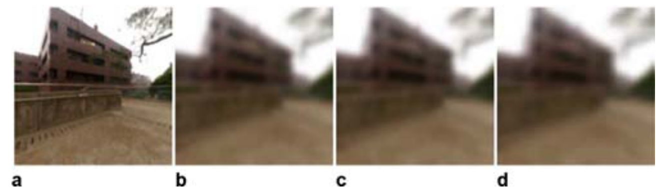


Fig. 8. a Original image (256×256). **b** Gaussian blurred with width 5. **c** Blurred and down-sampled to 64×64, no visual artifact visible. **d** Blurred and down-sampled to 32×32, slight aliasing visible. This means down-sampling from 256 to 32 cannot accommodate images blurred with Gaussian width smaller than or equal to 5

Table 1. Minimum Gaussian widths for different down-sampling ratios. The blurring angle is defined as $2r\alpha$, where α is the angle subtended by a pixel on the environment map so as to make it resolution dependent. The values shown are computed for a $512 \times 512 \times 6$ cube map. The threshold for the maximum error is 0.02

Resolution	Gaussian width r	Blurring angle
/2	4.6	1.62°
/4	10.5	3.69°
/8	21.6	7.59°
/16	43.4	15.26°
/32	86.9	30.55°
/64	173.7	61.07°

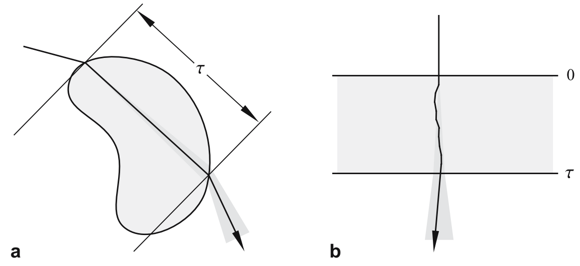


Fig. 10. a Angular distribution of radiance below the slab is described by the transmission function T_τ and the distribution above the slab by the reflection function R_τ. **b** A ray passing through a general object

where G_r is the Gaussian filter with width r, R is the down-sampling filter, and E is an enlargement filter that is a bilinear interpolation operation. Clearly, the difference between the two images I_b and I_{bs} is bounded by the difference between the two operators G_r and $G_r \circ (R \circ E)$. What the filter $R \circ E$ does is basically sample and interpolate. Therefore, when $R \circ E$ is applied to G_r, the function G_r will be sampled at regular spatial intervals dictated by the reduced resolution and inbetween values will be linearly interpolated. The output will be a piecewise linear function $G_r \circ (R \circ E)$ approximating G_r, as shown in Fig. 9. Obviously, when the resolution is fixed, the maximum error between $G_r \circ (R \circ E)$ and G_r decreases as the width r of the Gaussian filter increases. Hence, for each reduced resolution fixed by a level of mip-map, we just need to find out the smallest filter width of G_r such that the maximum error between $G_r \circ (R \circ E)$ and G_r is approximately equal to some prescribed error threshold ω, where ω is chosen such that the down-sampling's effect is hardly visually noticeable. Then these filter widths corresponding to different levels of the mip-map are used to obtain all the key filtered environment maps.

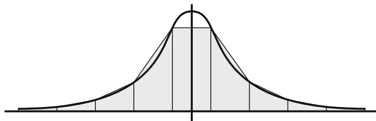


Fig. 9. Gaussian function and this down-sampled piecewise linear approximation. The actual computation is done for bivariate functions

3.3 Environment map compositing

Now we explain how to model the scattering behavior of light going through a slightly scattering medium. We consider the scattering of a light beam pointed perpendicularly to the surface of an infinitely extended slab of homogeneously scattering medium with a constant width τ (Fig. 10a). Two transfer functions are of interest to us: the *transmission function* T_τ modeling the angular distribution of the scattered light going out at the other side of the slab and the *reflection function* R_τ modeling the reflected light off the slab's surface on the same side. When rendering a general object, we determine the degree of blurring for a ray segment of length τ by the transfer functions obtained with slab thickness τ. This is only an approximation, since an object can have varied thickness and the tangent planes at the two ends of a ray segment may not be parallel to each other. This approximation is made based on the assumption that, for a slightly scattering and highly anisotropic medium, most of a light ray's energy is not scattered far away from the principal direction; thus the angular distribution of the ray through the object with optical depth τ does not deviate much from the one through a parallel slab of material of the same thickness τ (Fig. 10b).

Transfer functions for plane-parallel (i.e., slab) configurations and methods for computing these functions have been well studied in the field of radiative transfer. We use the classic results from [3]. The transfer equations are defined as integral equations based on the Henyey–Greenstein phase function and the slab thickness τ; the Henyey–Greenstein phase function $p_{HG,g}(\theta)$ describes the scattering of a light ray at an interior point of a medium, where g is its anisotropy.

The formulation we used is based directly on that in [3]. The meaning of the phase function is a bit different from many recent papers about subsurface scattering in the field of rendering. Here we assume that scattering events happen so often that it is like a continuous phenomenon inside the medium. Therefore, we can describe the effect of absorption and scattering together by a combined phase function $p'(\theta)$. In that case, there is no need for separate absorption and scattering coefficients. The amount of absorption is controlled by the value of albedo ω_0. The combined phase function we used is of the form $\omega_0\ p_{HG,g}(\theta)$, and the combined anisotropy g' is defined as usual for general phase functions.

To model a slightly scattering medium we set the value of g' to be close to 1 ($g' = 0$ gives a diffuse medium

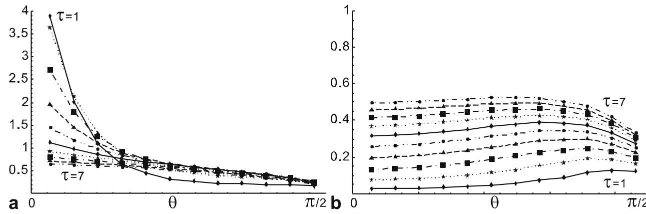


Fig. 11a,b. Transfer functions in terms of angular distribution. **a** Transmission functions for different thickness values of τ. **b** Reflection functions for different thickness values of τ. The phase function $p'(\theta)$ with anisotropy $g' = 0.8$ is used. θ is the angle of an out-going ray with the surface normal

and $g' = 1$ gives a transparent medium). By expressing $p'(\theta)$ in a Legendre expansion, we solve the integral equation for the transfer functions, which are also given in terms of Legendre expansions. We will only give the solved transfer functions for various thickness values of τ and refer the reader to [3] for the details of computation. Note that solving for the transfer functions is done offline.

Figure 11 shows the graphs of several transmission functions T_τ and reflection functions R_τ for values of thickness from $\tau = 1$ to $\tau = 7$. The transmission functions have peaks near $\theta = 0$ and drop off rapidly with increasing θ, resembling a Gaussian function (Fig. 11a). The reflection functions, on the other hand, are relatively constant for highly anisotropic media (Fig. 11b). Note that the reflected intensity increases with increasing thickness because there is more material serving as reflectors in thicker objects.

Since the transmission functions T_τ are shaped like a Gaussian function, for each sampled thickness value τ, we solve a function fitting problem to approximate T_τ by a linear transform of a Gaussian function G_r, in the form $\tilde{T}_\tau = f_t G_{\bar{\theta}} + c_t$, with appropriately determined coefficients f_t and c_t and width $\bar{\theta}$. Note that the parameters f_t and c_t and $\bar{\theta}$ are functions of the thickness τ. Assuming that each side of the cubic environment map has $n \times n$ resolution, each pixel subtends the angle $2\pi/(4n) = \pi/(2n)$, approximately. Hence, once $\bar{\theta}$ is known, its corresponding filter width (in pixel unit) for blurring is given by $2\bar{\theta}n/\pi$, and this filter is used as an index to access the mip-map of prefiltered key environment maps to produce the desired degree of blurring effect. Note that only three values, f_t, c_t, and $\bar{\theta}$, need to be recorded for representing each approximate transmission function $\bar{T}_\tau$.

Both terms in the expression $\tilde{T}_\tau = f_t G_{\bar{\theta}} + c_t$ correspond to texture colors. The $G_{\bar{\theta}}$ term refers to the color C_B in a blurred environment map, and the unit value in the constant term refers to the average color C_A of the entire environment map, i.e., the color of the most blurred map. Then the transmitted color is given by the compostion $f_t C_B + c_t C_A$.

The computation of the reflection color based on the refection function R_τ is similar to the above computation of the transmission color, but simpler, since we approximate the functions R_τ for different values of τ by different constants c_r. Then the reflected color is given by $c_r\, C_A$.

4 Implementation and results

The latest generation of graphics hardware supports floating point textures that enable us to encode data accurately as texture map. For each geocube, two four-component cube maps are used. One cube stores the geocube's surface distances and the defining parameters of cones A, B, and C. The other cube stores the coefficients of the tangent plane equation at each sample point for finding the intersection of a ray with the base of cone B. A lookup table represented as a 2D texture that encodes functions $C^*(\mu, \nu)$ and $S^*(f, \gamma)$ (see Appendix) is used for computation speedup. A four-component 1D texture map is used to store the mapping from ray depth τ to down-sampling level l, f_t, c_t, and c_r. Here l is used in the shader to access the correct mip-map level to retrieve texture color.

Some of the environment maps used are high-dynamic-range images in order to produce good visual effects for objects with high absorption. The ray-tracing algorithm is implemented in both the vertex shader and the pixel shader. Most of the job is done in the vertex stage, which brings the rays to near their exits on the boundary surface. The pixel stage is responsible for bringing the points already near the surface to the surface. The pixel shader also shades the pixels. The implementation is vertex-stage-limiting, i.e., rendering, speed depends more on the number of vertices rather than the screen resolution. All tests were run on a PC with a 3 GHz Pentium 4 and an Nvidia GeForce 6800 graphics board. We used a fixed resolution of 512×512. We measured both the time used for precomputation and run-time frame rates for a few models of different triangle counts, which are shown in Table 2. Video clips showing the real-time performance of the shader are available at `http://www.cs.hku.hk/graphicsgroup/geocube`.

The first example shows a Stanford Bunny model (Fig. 12). It simulates a semitransparent glass material

Table 2. Timing statistics for models used in the experiments. The precomputation time column shows the total time used to generate the geocubes and the number of geocubes used

Model	Tri. count	Precomput. time	FPS
Venus	40 K	249 s, 1 $64 \times 64 \times 6$ cube	23
Bunny	80 K	1085 s, 2 $64 \times 64 \times 6$ cubes	14
Isis	28 K	384 s, 2 $64 \times 64 \times 6$ cubes	25

Fig. 12. A light-absorbing but nonscattering glass Bunny. Two geocubes are used, one near the tail and one inside the head, to represent the Bunny

with absorption but no scattering. We can see that the thicker areas like the body are darker than the thinner areas such as the ears, the tail, and the paws. Despite the darkening, the background image can be seen clearly, especially near the head, because there is no light scattering and therefore no image blurring.

The second example shows a Venus model (Fig. 13). The material is light absorbing as well as slightly scattering. Figure 13 shows a comparison between a purely transparent model and a translucent one. Note that the sharpness of the background image is the same everywhere in panel a but varies from area to area in panel b because of different optical depths. The nose is so thin that it appears to be almost transparent in panel b.

The third example shows an ice Isis sculpture (Fig. 14). When ice is cooled down quickly, it traps air bubbles (which are light scatterers) in it. We do not have the accurate scattering data for ice, so we use our model to approximate the appearance only. Figure 14 shows a few different views of the ice sculpture.

The fourth example shows the blurring effects of a sphere with different degrees of scattering (Fig. 15). The materials are purely scattering (i.e., there is no absorption).

The final example is a comparison between the image qualities of geocube ray tracing and conventional ray tracing. The geocube method on GPU can render a transparent object at a speed more than two orders faster than conventional ray tracing running on GPU, but not without sacrifices. Therefore, this example is also for showing some of the limitations of the geocube algorithm.

a

b

Fig. 13. a Purely transparent Venus head. The *bright area* is the image of a light source at the back. **b** Translucent version of same model. Note the different degrees of blurring of background light on hair and face. The model is star-shaped so a single cube suffices

Figure 16a,b show two purely transparent Venus heads rendered by the geocube method and the conventional method, respectively. They look very similar, but they are not a pixel-for-pixel match because the surface represented by the geocube is rasterized. The shown-through image is shifted a little bit here and there, caused by quantization errors of the coarse geocube ($64 \times 64 \times 6$ samples) across the object. The shifts can be reduced

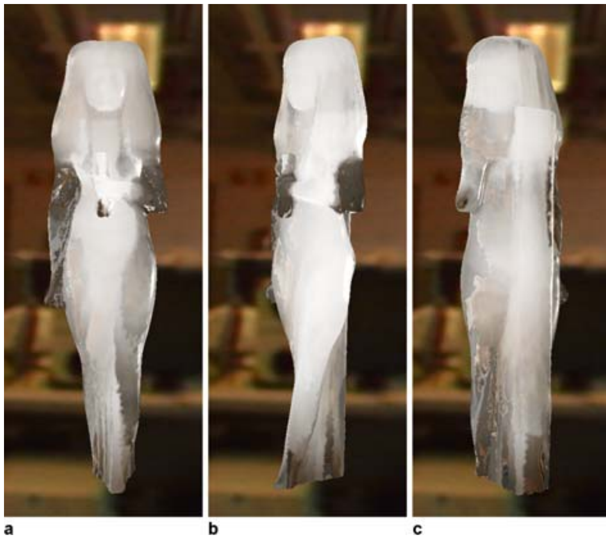


Fig. 14. An approximated ice sculpture. Two geocubes are used to represent the model. In real ice, the scattering air bubbles usually distribute unevenly across the object, so part of the ice may appear to be more transparent. We do not take that into account. The clear-and-blur differences appearing in the images are caused entirely by different optical depths

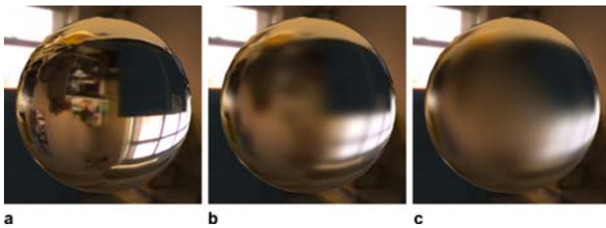


Fig. 15a–c. Different degrees of scattering. **a** Purely transparent sphere. **b** and **c** Slightly scattering spheres. Note degree of blurring is less near the silhouette than in the middle due to greater optical length in middle

by using finer geocubes, but they cannot be completely eliminated.

There are some bigger mismatches near Venus' neck. They are caused not by the quantization errors but by a limitation in the current implementation. This problem may appear when an exceptionally large number of iterations are required to propel a ray when it is near and parallel to a *rough* surface (the fracture of the Venus head model in this case). Note that such a ray travels close to the apex of all A cones and that most of the B cones it encounters are narrow because of the rough surface. Therefore, neither cone A nor cone B helps much in such situations. In the current implementation, an upper bound is set to limit the number of iterations in the shader. Such a bound is used to balance the accuracy and performance. When a ray cannot reach the surface within the limited steps, the last fetched geocube surface point will be used as the exit point. Premature exits of rays may cause very serious visual artifacts and should be minimized as much as possible. They can be avoided by setting a larger iteration upper bound. The value of the upper bound depends on the geocube resolution and the model shape. Typical values will be around 5 to 10.

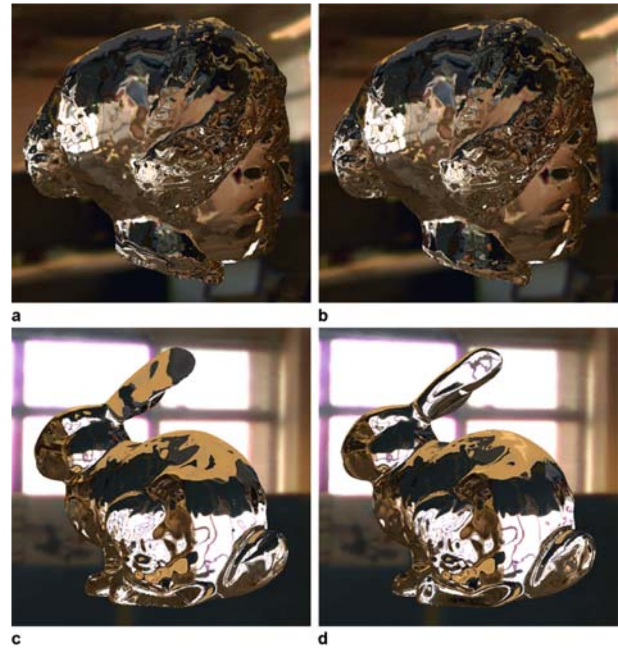


Fig. 16a–d. Image quality comparison between geocube method and conventional ray tracing. The four models are all purely transparent. **a** Geocube-ray-traced Venus head. **b** Conventionally ray-traced Venus head. **c** Geocube-ray-traced Bunny. **d** Conventionally ray-traced Bunny. For the Bunny model, the cubes are positioned such that the two ears are not fully covered to show the importance of full geocube coverage to applications like transparency rendering

Figure 16c,d shows a comparison of the Bunny model rendered by different methods. Like the Venus head model, our method generally agrees with conventional ray tracing in most regions except for the ears. This problem is caused by the inadequate placements of geocubes—a small part of the ears is not covered by any of the geocubes. Since refraction is very sensitive to surface shape, a small difference in shape will cause a big difference in the rendered image.

5 Discussions and conclusions

We have proposed a new method for rendering transparent objects and translucent objects in real time. We use a new geometric data structure, *geocube*, to represent the rasterized surface of an object so that the ray–surface intersection can be computed on GPU very efficiently. The

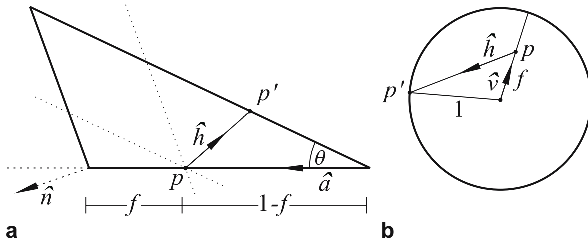


Fig. 17. Symbols' meanings for functions S and C. All vectors are unit vectors

key idea is to use a group of precomputed simple solids to quickly compute the exit points of light rays for computing the correct refraction at both front and back surfaces, as well as the internal reflection. We further combine the geocube algorithm with an environment map prefiltering technique to achieve real-time rendering of a wide range of translucent objects. Here we use the mip-map to compactly store a hierarchy of key filtered environment maps to reduce the consumption of precious texture memory.

In our implementation, the position and resolution of a geocube need to be specified by the user, with the resolution bounded by the amount of texture memory available. Multiple geocubes are needed to represent objects that are not star-shaped. In the test examples presented, the cubes are placed interactively. We have not implemented any method that can place the cubes in an optimal manner, but observe that placing the minimal number of cubes such that any surface boundary point is visible to at least one of the cube centers is similar to the Art Gallery problem in computational geometry. Therefore, it would be an interesting problem to develop an efficient, though perhaps approximate, algorithm for this problem.

A promising extension to the *geocube* is a *multilayer geocube*. That is, instead of storing a single distance value for each sampled direction, a few layers of surface distances, together with associated cones, can be stored in a single multilayer geocube to accommodate the complex shape of arbitrary objects. In this way, multiple cubes and cube switching would be eliminated, and the cube placement problem would be averted. This extension would also allow the rendering of secondary and higher-order penetration of light rays interacting with multiple general objects.

Appendix

The function $C(f, \hat{h}, \hat{a}, \hat{n}, \mu)$ is defined as

$$C(f, \hat{h}, \hat{a}, \hat{n}, \mu) = \begin{cases} (1-f)\, C^*(\mu, -\hat{h}\cdot\hat{a}), & \hat{h}\cdot\hat{n} < 0; \\ min \begin{cases} (1-f)\, C^*(\mu, -\hat{h}\cdot\hat{a}) \\ f\frac{\hat{a}\cdot\hat{n}}{\hat{h}\cdot\hat{n}} \end{cases}, & \hat{h}\cdot\hat{a} < \mu; \\ f\frac{\hat{a}\cdot\hat{n}}{\hat{h}\cdot\hat{n}}, & \text{otherwise}, \end{cases}$$

where $\mu = \cos\theta$ and

$$C^*(\mu, \gamma) = \frac{\sqrt{1-\mu^2}}{\gamma\sqrt{1-\mu^2}+\mu\sqrt{1-\gamma^2}}.$$

The function $S(f, \hat{h}, \hat{v})$ is defined as

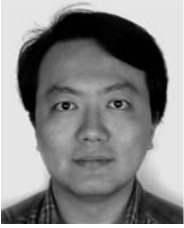
$$S(f, \hat{h}, \hat{v}) = S^*(f, \hat{h}\cdot\hat{v}),$$

where $S^*(f, \gamma) = \sqrt{1-f^2(1-\gamma^2)} - f\gamma$

C^* and S^* are both functions of two variables that can be encoded as floating point texture maps. The computation time for the two functions are therefore greatly reduced by table lookup.

References

1. Akenine-Moller T, Haines E (2003) Real-Time Rendering, 2nd edn. A.K. Peters, Wellesley, MA
2. Carr N, Hall J, Hart J (2003) GPU algorithm for radiosity and subsurface scattering. In: Proceedings of Graphics Hardware '03
3. Chandrasekhar S (1964) Radiative Transfer. Dover, New York
4. Hao X, Varshney A (2004) Real-time rendering of translucent meshes. ACM Trans Graph 23:120–142
5. Jensen H, Christensen P (1998) Efficient simulation of light transport in scences with participating media using photon maps. In: Proceedings of SIGGRAPH '98, pp 311–320
6. Jensen H, Marschner S, Levoy M, Hanrahan P (2001) A practical model for subsurface light transport. In: Proceedings of SIGGRAPH '01, pp 511–518
7. Kautz J, Vazquez P, Heidrich W, Seidel H (2000) A unified approach to prefiltered environment maps. In: Proceedings of EG Rendering Workshop '00
8. Purcell T, Buck I, Mark WR, Hanrahan P (2002) Ray tracing on programmable graphics hardware. ACM Trans Graph 21:703–712
9. Reinhard E, Smith B, Hansen C (2000) Dynamic acceleration structure for interactive ray tracing. In: Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, pp 299–306
10. Schroeder W, Martin K, Loresen B (1998) The Visualization Toolkit, 2nd edn. Prentice-Hall, Englewood Cliffs, NJ
11. Sloan P, Kautz J, Snyder J (2002) Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In: Proceedings of SIGGRAPH '02, pp 527–536
12. Smith A, Blinn J (1996) Blue screening matting. In: Proceedings of SIGGRAPH '96, pp 259–268
13. Wald I, Benthin C, Dietrich A, Slusallek P (2003) Interactive ray tracing on commodity pc clusters. In: Proceedings of EuroPar 2003, pp 499–508
14. Wald I, Slusallek P, Benthin C (2001) Interactive distributed ray tracing of highly complex models. In: Rendering Techniques 2001: 12th Eurographics Workshop on Rendering, pp 277–288



BIN CHAN is a Ph.D. candidate in the Department of Computer Science at the University of Hong Kong. He received his B.Eng. and M.Phil. in computer science from the same university, in 1995 and 1998, respectively. His research interests include real-time rendering, virtual reality, and global illumination.

WENPING WANG received his B.Sc. and M.Eng. in computer science from Shandong University, China, in 1983 and 1986, respectively, and his Ph.D. in computer science from the University of Alberta, Canada, in 1992. He is an associate professor of computer science at the University of Hong Kong, China. His research interests include computer graphics, geometric computing, and computational geometry (`http://www.cs.hku.hk/~wenping/`).