Efficient Light Transport Using Precomputed Visibility

Katja Daubert[1], Wolfgang Heidrich[2], Jan Kautz[1], Jean-Michel Dischler[3], Hans-Peter Seidel[1]

[1)] Max-Planck-Institut für Informatik
[2)] The University of British Columbia
[3)] LSIIT, UPRES-A CNRS 7005, University Louis Pasteur, Strasbourg

May 10, 2002

Abstract

Visibility computations are the most time-consuming part of global illumination algorithms. The cost is amplified by the fact that quite often identical or similar information is recomputed multiple times. In particular this is the case when multiple images of the same scene are to be generated under varying lighting conditions and/or viewpoints. But even for a single image with static illumination, the computations could be accelerated by reusing visibility information for many different light paths.

In this paper we describe a general method of precomputing, storing, and reusing visibility information for light transport in a number of different types of scenes. In particular, we consider general parametric surfaces, triangle meshes without a global parameterization, and participating media.

We also reorder the light transport in such a way that the visibility information is accessed in structured memory access patterns. This yields a method that is well suited for SIMD-style parallelization of the light transport, and can efficiently be implemented both in software and using graphics hardware. We finally demonstrate applications of the method to highly efficient precomputation of BRDFs, bidirectional texture functions, light fields, as well as near-interactive volume lighting.

Keywords: Illumination, Ray Tracing, Monte Carlo Techniques, Frame Buffer Algorithms, Texture Mapping, Reflectance & Shading Models, Volume Rendering

1 Introduction

Global illumination algorithms usually spend the majority of time on visibility computations. It therefore seems natural to try and reuse visibility information acquired at one point for different computations. For example, once the visibility between two points in the scene has been established, this information can be used for multiple light paths in which different amounts of energy are transported between the points. This is particularly advantageous in cases where multiple images with varying illumination or camera settings are to be computed.

There have been several approaches in the past where illumination information computed for one point in the scene has been reused for close by points. One example for diffuse scenes is the Irradiance Gradients method [1], and for scenes with specular objects there are techniques like photon maps [2] and density estimation [3]. Because these methods store illumination information (irradiance or incident radiance) at discrete points, it is not possible to reuse the information for changes of the light source. In addition, finding the desired information for one point in space requires a search through the data structure. This search can be performed in logarithmic expected time, but the resulting memory access patterns are irregular and can present a significant performance bottleneck.

We go a different way. Instead of storing and reusing illumination information, we directly reuse visibility information stored in a regular fashion that allows for constant time lookups. Our method is a generalization of the method by Heidrich et al. [4] for height fields to different kinds of geometry like general parametric surfaces, triangle meshes without a global parameterization, and volumes. For each case we propose efficient algorithms for computing direct and indirect illumination, which also account for shadows. Using the Method of Dependent Tests, a variant of Monte Carlo integration, we can access the visibility in a structured fashion, that allows for coherent, and therefore efficient memory access patterns in software implementations as well as for the use of graphics hardware for the light transport.

We demonstrate quality and performance of our approach by applying it to the computation of BRDFs, bidirectional texture functions (BTFs), light fields, as well as to near-interactive volume lighting.

The remainder of this paper is organized as follows. In Section 2 we review the literature related to our approach. After briefly summarizing the work by Heidrich et al. for height fields in Section 3 we discuss our method for parametric surfaces, arbitrary triangle meshes, and volumes in Sections 4, 5, and 6. We finally present applications and results of the method in Section 7, and conclude with a discussion and conclusions in Sections 8 and **??**.

2 Related Work

There have been a number of publications that describe the reuse of previously computed illumination information in global illumination algorithms. Irradiance Gradients [1] accelerate the computation of indirect light in diffuse scenes by reconstruction from irradiance samples that have been generated for other locations in close proximity of the desired surface point. The Irradiance Volume [5] represents a coarse volumetric representation of irradiance, from which the illumination at arbitrary locations can be reconstructed.

For scenes with specular objects, photons can be traced from the light sources through the scene, and stored on the objects. The incident light at arbitrary surface locations can then be reconstructed using techniques like density estimation [3] and the photon map [2].

Both the methods for diffuse and for specular surfaces store illumination information (irradiance or incident radiance) rather than visibility, and can therefore not be used to accelerate the computations in the case of changing light sources. Also the reconstruction process for any given point in the scene requires a search through the illumination data structure, which is typi-

cally the most costly part of the computation.

Other algorithms, such as finite element methods for global illumination computations, store visibility. In particular, the link structure in hierarchical and Wavelet Radiosity [6, 7] can be interpreted as a cache for visibility information. However, since this structure only represents the most relevant parts of the visibility for a given illumination situation (BF-refinement), the information typically has to be recomputed if the illumination changes.

Precomputed visibility that is completely separated from illumination and light source positions has been studied for special cases such as height fields. Horizon maps [8] represent the visibility information required for computing shadows and masking from direct light sources in height fields and bump maps. There have also been some solutions for shadows in more general geometry like folded cloth [9]. These approaches are based on sampled representations of the visibility. Other, analytic representations for general scenes like the visibility skeleton [10] suffer from a combinatorial explosion of the information with the scene complexity and from numerical instabilities.

The use of precomputed visibility for the purpose of computing the light transport between different parts of the scene has so far been limited to the case of height field geometry [4]. We expand this idea to more general geometry and participating media. By using the Method of Dependent Tests, a variant of Monte Carlo integration, we access the precomputed visibility information in a structured form. This way we achieve regular memory access patterns for efficient software rendering, as well as the possibility to compute the light transport using graphics hardware.

We demonstrate the quality and performance of our method with a number of applications ranging from near-interactive computation of indirect light in participating media over the simulation of BRDFs (similar to the methods described by Cabral et al. [11] and Westin et al. [12]) to the computation of light fields [13, 14] and BTFs [15, 16].

3 Light Transport in Height Fields

Before we introduce our light transport algorithm for computing indirect light in general polygonal scenes and participating media, we briefly review the approach presented by Heidrich et al. [4] for height fields.

The fundamental idea of this method is to calculate the visibility in a precomputation step, and to store it in a set of scattering textures S_i. In order to do this, a fixed set $D = \{d_i\}$ of sample directions on the sphere is chosen. Then a ray is shot from each grid point in the height field into each of the directions d_i and intersected with the height field geometry. A scattering texture S_i holds the intersections for all rays starting at any point in the height field in one particular direction d_i. Each of these intersections is uniquely characterized by a 2D texture coordinate (see Figure 1).

```
for each d_i ∈ D {
  for each grid point p on height field {
    ray := Ray( orig = p, dir = d_i );
    q = intersect( ray, height field);
    S_i[p] = q;
  }
}
```

Figure 1: *Pseudo code for precomputation.*

During rendering, this information is chained

together to generate a multitude of different light paths, enabling the implementation of many existing Monte Carlo algorithms. In particular, Heidrich et al. [4] apply the Method of Dependent Tests, a Monte Carlo variant in which the same sampling pattern is used for all points in the height field, to map the light transport operation to graphics hardware.

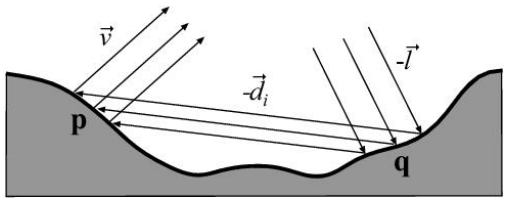


Figure 2: *When using the Method of Dependent tests, the light paths for computing the illumination at all points of the height field are identical.*

As an example, consider Figure 2, which illustrates the light paths with two reflections. Light arrives at the height field from direction $\vec{l}$, is reflected at each point in direction $-\vec{d}_i \in D$ and finally leaves the surface in the direction of the viewer $\vec{v}$. Most of the vectors are either constant across the height field, or vary only slowly (like $\vec{l}$ and $\vec{v}$ for the case of point lights and perspective views). In fact, the only strongly varying parameters in the computation are the normal at $\mathbf{p}$, the point $\mathbf{q} := S_i[\mathbf{p}]$ visible from $\mathbf{p}$ in direction $\vec{d}_i$ and the normal at $\mathbf{q}$.

The computation is split into two parts, corresponding to the reflections at $\mathbf{q}$ and later at $\mathbf{p}$. First the direct illumination of the height field in viewing direction $-\vec{d}_i$ with light arriving from $\vec{l}$ is computed by a bump mapping step[1] and stored in a texture L_d. Afterwards the second reflection is computed in a similar manner. This time the light direction is $\vec{d}_i$ and the viewing direction is $\vec{v}$, however the incoming radiance needs to be looked up in the direct illumination texture L_d. For each surface point $\mathbf{p}$ the visible point $\mathbf{q} = \mathbf{S_i}[\mathbf{p}]$ is looked up in the scattering texture corresponding to $\vec{d}_i$. $\mathbf{q}$ is then used as an index into the direct light texture L_d, yielding the light arriving at $\mathbf{p}$ in direction $-\vec{d}_i$.

This way indirect illumination of height fields is implemented as two steps of coherent light transport along one direction for all points at the same time. The mapping of this algorithm to hardware is straightforward using hardware bump mapping and dependent texture lookups (as defined for example by DirectX version 8 as well as several OpenGL extensions) for the indirection.

[1] Bump mapping here means evaluating the BRDF on the height field surface for the current light and viewing direction. We use a model very similar to Phong and evaluate it e.g. using register combiners.

4 General Parametric Surfaces

In the following, we will now extend the above algorithm to compute the indirect illumination in parametric surfaces. In order to precompute the scattering textures we first need to discretize the surface. We do this by using the parameterization to texture the surface. Then, given a texture of a certain resolution, each pixel in the texture can be easily mapped to a point on the surface. Similar to the height field precomputation step we now generate rays originating from each of these points in each of the global sample directions and intersect them with the surface. The intersection points can again be characterized by their parameter values, which we store as 2D floating point texture coordinates in separate textures for each sample direction. To compute indirect illumination, these scattering tex-

tures can now be used in the same way as for the height fields.

That is, given the scattering textures S_i and a per-texel normal, we first have to generate a texture-space representation of the direct illumination L_d. Then the indirect illumination can be computed completely in texture space using a sequence of table lookups for the light transport as depicted in Figure 3.

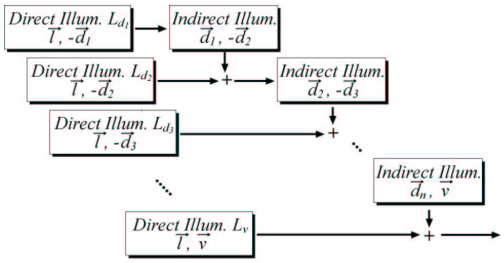


Figure 3: *Computation of multiple scattering for one light path. Each box corresponds to a texture generated by a bump mapping step. For a full illumination solution, numerous paths like this one have to be computed.*

The question then arises how to compute the direct illumination L_d, for which a shadow test is required. For the height field case, Heidrich et al. [4] propose a hardware-friendly representation of horizon maps [8] that allows for efficient shadow tests for arbitrary light directions. Obviously, a horizon approach, no matter in which representation, will not work in the case of general parametric surfaces, since the light directions can consist of several disjoint regions. Similarly, representations like the shadow map [17] will not work, because these are valid only for specific light positions. Finally, analytic representations like the visibility skeleton [10] will be infeasible due to the combinatorial explosion in the complexity.

We therefore propose the following shadow algorithm that is similar in spirit to the horizon map in that it represents an approximation of the shadowing information for all light directions and positions. In contrast to horizon maps, however, it works for arbitrary geometries. In a precomputation step we partition the sphere of possible light directions into several regions by choosing some uniformly distributed directions c_i and defining the regions around them. Then we compute the fraction of solid angle not blocked by other parts of the surface for each region. In order to do this we can reuse the visibility information already computed. For each of the directions d_i we determine to which of the regions it belongs, and then compute the fraction of these directions that do not hit other parts of the surface. Pseudo code for precomputing the fractions can be found in Figure 5. The results of this step for one height field point are illustrated in Figure 4.

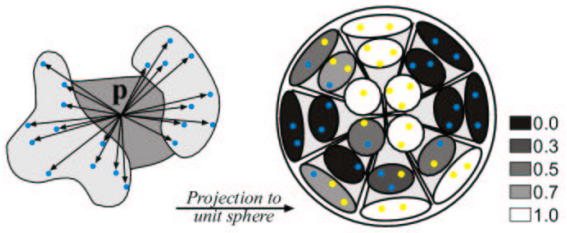


Figure 4: *Left: results of scattering precomputation for p (only hits are drawn). Right: Projection of hits (blue) and misses (yellow) to unit sphere. Color of shadow regions (cones) corresponds to value of fraction – dark: high number of hits, light: high number of misses.*

After having computed the fractions for all points and all shadow regions, we can store the results in a texture with one channel per region. During rendering, the shadow test for a given light direction can be performed by computing a weighted sum of the fractions for all directions to avoid quantization artifacts. For the weights

```
for each d_i ∈ D
  nearest[d_i] = find c_i nearest to d_i;

for each grid point p on height field {
  for each d_i
    increment total[nearest[d_i]];
    if S_i[p] is valid point
      increment light[nearest[d_i]];
  for each c_i
    fraction[p, c_i] = light[c_i] / total[c_i];
}
```

Figure 5: *Computing the fractions. d_i are the directions used for the visibility precomputation, c_i are the directions of the shadow region i.*

we use cosine powers of the angle between the true light direction and the various c_i.

These weights are chosen to be easily implementable using graphics hardware such as the register combiner feature on new PC boards. For hardware rendering, we code the shadow information into RGBA textures, in such a way that we have one texture for four directional regions. Depending on how many simultaneous textures and combiner stages a given graphics hardware supports, we can check a number of directions at once. For every shadow texture we also need one vector of weights, and by loading two of these textures and weight vectors into a single combiner stage, we can compute the weighted sum for eight directions in one stage using two textures. Usually several passes are needed to compute the whole sum, the results of which are added using the blending operation. In our implementation we used 32 shadow directions on hardware that supports two simultaneous textures, and can therefore compute self shadowing for a given direction in four passes. The result is a texture with values ranging from zero (totally shadowed) to one (fully lit) for each point on the surface. This value can then be used to attenuate the result of a direct light computation.

Figure 6 shows a piecewise parametric surface without (left) and with shadows (right), computed by our algorithm. In both images the light source is located above and to the left of the object.

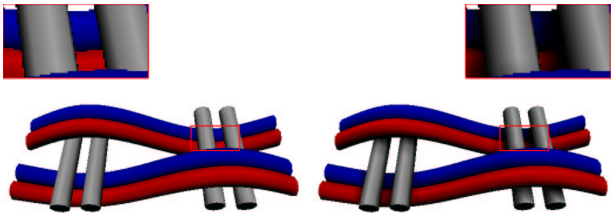


Figure 6: *Piecewise parametric surface without shadows (left) and with shadows computed by our shadow algorithm (right). Top corners show closeups of marked regions.*

5 Arbitrary Triangle Meshes

One advantage of this shadow algorithm is that once we have the scattering information, the shadow computation takes place in some texture space. It is therefore well suited also for application to arbitrary triangles meshes, provided we find a way to efficiently index surface locations on these meshes.

One possibility is to reduce the problem to parametric surfaces by finding a parameterization for the triangle mesh. For example, we can use the MAPS algorithm [18] and first reduce the fine mesh to a coarse triangle mesh. These coarse triangles have an inherent parameterization of their own. Then the vertices are reinserted, thereby assigning them parameter values dependent on their position on the coarse triangles. After completion the mesh consists of as many global parameterizations as there were coarse triangles.

We can then merge these parameterizations into a single large parameter space to hold the

scattering information. At this point it is possible to apply the same shadowing algorithm as for parametric surfaces to generate the direct illumination map L_d for the whole mesh. From there on, we again use only texture space computations to calculate the indirect illumination using scattering textures S_i that are parameterized in the given global texture space for the mesh.

On the other hand, if we have a very fine, uniform mesh to start with, it may be sufficient to compute the illumination only at the vertices. In this case it is not necessary to generate a global parameterization and resample the surface into texture space. Instead of using a true texture for representing the samples, we use a simple lookup table, in which each vertex in the mesh is mapped to one table entry as depicted in Figure 7.

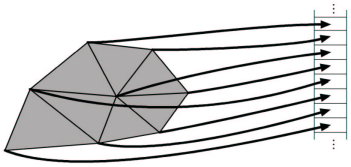


Figure 7: *Each vertex is mapped to an entry of a lookup table.*

The shadow data structure will then contain the same information as described in Section 4, but now for every vertex in the mesh, that is, for every entry in the table.

For the scattering information, the visibility is computed in a similar fashion as for the parametric surfaces, with one important difference. Since we no longer have a parameterization for the surface, we cannot store exact intersections of rays with the surface, but rather have to quantize the intersections to entries represented in the table. This corresponds to storing the vertex closest to the ray intersection rather than the true intersection point. This will also slightly alter the direction d_i, a fact we can ignore if the mesh is fine enough. As the error depends heavily on the density of the tessellation, we can apply local refinements like subdivision or simple vertex insertion on the triangle to compensate for under-tessellated regions in the original mesh.

To use graphics hardware, we code the one-dimensional tables representing the shadow information and the scattering information into two-dimensional texture maps. Since we do not have the connectivity information that we get from a parameterization of the surface, we cannot interpolate in the illumination textures L_d during the light transport phase. However, once we have computed the indirect illumination by using table lookups as before, we have obtained per-vertex illumination that can be interpolated across the triangle mesh using Gouraud shading.

6 Volumes and Participating Media

As a final scenario for our approach to light transport we would like to consider an algorithm for computing the scattering of light in participating media. To this end, we generalize the texture based volume rendering for emission-absorption volumes described by Cabral et al. [19].

The original texture-based volume rendering algorithm works as follows. The volume is sliced with planes parallel to the image plane, yielding a stack of polygons. Each vertex receives texture coordinates corresponding to the 3D location of the vertex within the volume. Using 3D texture mapping hardware, each polygon is thus textured with the corresponding volume slice. These slices are rendered back-to-front using al-

pha blending to simulate the emission and absorption in the volume (see [19] for details).

We extend this basic algorithm to direct and indirect illumination in participating media. In analogy to the surface case we use the term "direct illumination" for light emitted from a point or directional light source that is scattered exactly once before it hits the eye. "Indirect illumination" is therefore light that is reflected more than once inside the medium. For this paper, we restrict ourselves to *isotropic* media [20], that is, to media whose phase function is given as $p(\cos\alpha)$, where α is the angle between the incident and the exitant light directions.

6.1 Direct Illumination in Participating Media

We begin the discussion by describing an efficient, hardware-accelerated algorithm for rendering direct illumination at every point of a volume. Note that this step is analogous to the shadowing step in the surface algorithms.

In other words, we need to compute, for every voxel in the volume, the direct light arriving from the light source, that is, the radiance leaving the light source into the direction of the voxel minus the absorbed and outscattered parts. A texture mapping approach to achieve this is to render the volume from the light source by intersecting it with planes as before, but here the planes are rendered front-to-back. Every newly rendered slice is weighted by the accumulated transparency of the previously rendered slices, which represent the portions of the volume closer to the light source. The result of the rendering of every individual slice is a sampled representation of the direct illumination arriving at this slice of the volume.

To avoid resampling and representing regions that are not covered by the actual volume, we align the volumetric representation of the direct illumination with the voxel grid of the original volume. This is achieved by aligning the slices in parallel to one of the axes of the coordinate system rather than to the image plane while rendering from the light source.

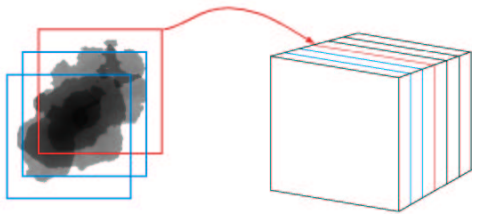


Figure 8: *The volume slices representing the direct illumination in the volume result from a front-to-back rendering from the light source.*

Using 3D texture mapping hardware, the direct illumination volume can therefore be generated as follows. First, we find the largest component of the light direction $\vec{l}$. Then, we slice the volume into planes perpendicular to the corresponding coordinate axis. We render these slices front-to-back, thereby multiplying them with the transparencies of the previous slices that have been accumulated in the framebuffer. After rendering every individual slice, copy the region of the framebuffer covered by it as a new slice into the 3D texture representing the direct illumination. This copying between framebuffer and texture memory is quite efficient on modern graphics hardware and does not result in the serious performance penalties of framebuffer readbacks to main memory.

The result of the algorithm just described is a 3D texture containing the direct illumination $L_d(\mathbf{x}, \vec{l}_\mathbf{x})$ resulting from a point light source in direction $\vec{l}_\mathbf{x}$ at every voxel $\mathbf{x}$. Using this texture, the original texture based volume render-

ing algorithm can be modified to include direct light. To this end we sort the slices back-to-front, and render every slice in two different passes. In the first pass, we have to take care of the absorption in the volume. This is done in the same way the original texture based volume rendering algorithm handles absorption: the slice is texture mapped with the local density $\rho(\mathbf{x})$, which is interpolated from a volume texture. In a back to front rendering of the slices using alpha-blending, the previously rendered parts are attenuated by the absorbed parts.

In a second pass, the slice is rendered again, this time to add in the direct illumination. This local interaction of light is described as

$$L_l(\mathbf{x}, \vec{v}) = \rho(\mathbf{x}) \frac{1}{4\pi} \int_{S^2} p(\langle \vec{v} | \vec{\omega}_i \rangle) \cdot L_d(\mathbf{x}, \vec{\omega}_i) d\omega_i. \tag{1}$$

For the case of directional or point light sources that we want to consider here, we get $L_d(\mathbf{x}, \vec{\omega}_i) = \delta(\vec{\omega}_i, \vec{l}) \cdot L_d(\mathbf{x}, \vec{l})$, and Equation 1 simplifies to

$$L_l(\mathbf{x}, \vec{v}) = \rho(\mathbf{x}) \frac{1}{4\pi} p(\langle \vec{v} | \vec{l} \rangle) \cdot L_d(\mathbf{x}, \vec{l}). \tag{2}$$

Thus, the slice is texture-mapped with the 3D texture $L_d(\mathbf{x}, \vec{l}_{\mathbf{x}})$, multiplied with the local volume density $\rho(\mathbf{x})$ take from the volume data set, and weighted by the phase function $p(\cos \alpha)$, which is either constant (corresponding to directional light and orthographic viewer), or can be interpolated as an additional 1D texture or per-vertex color. The result of this pass is added to the result of the first pass in the framebuffer. An example of the direct illumination generated with this method can be seen on the left side of Figure 14.

6.2 Scattering in Participating Media

Now that we have an algorithm for rendering the *direct* illumination in a volume, we can again use the idea of precomputing visibility information to accelerate the computation of *indirect* illumination (scattering) in participating media. We assume that the direct illumination $L_d(\mathbf{x}, \vec{l}_{\mathbf{x}})$ is given on a regular grid that is axis-aligned with the original volume data, as discussed above. However, the grid used for the direct illumination does not need to have the same resolution as the original volume data.

To compute the actual indirect illumination in the medium, we first require visibility information for the scattering process, similar to the case of geometry. Again, we randomly define a global set of directions $D = \{\vec{d}_i\}$. Then we compute one *volume* of visibility information for each direction. Each point $\mathbf{x}$ in such a scattering volume S_i represents the coordinates of another point $\mathbf{y}$ with $\mathbf{y} = \mathbf{x} + k \cdot \vec{d}_i$. The distance k is determined by sampling the volume along the ray in fixed step sizes, and at each sample making a stochastic decision based on the local volume density as to whether or not the ray is scattered at that sample location. If we decide that light is scattered, we store the coordinate of the sample as the point visible from the origin of the ray, as well as the percentage of light that arrives at the scattering point from the origin of the ray without being absorbed or scattered. Otherwise, we continue traversing the ray through the volume. This method in essence resembles volume rendering by stochastic ray-tracing, but the difference is again that we apply the method of dependent tests by reusing this same sampling pattern for multiple light paths, rather than generating a different pattern for every path.

Given a direct illumination volume L_d along

with the scattering volumes S_i, the actual light transport for the indirect light is then similar to the case of parametric surfaces, except that we have to use the phase function of the volume instead of a BRDF in order to model the local interactions of light with matter. We use the precomputed visibility information for the volume scattering for transporting light from one position in the volume into a given global direction. Note that in contrast to the surface algorithms, the visibility information is now stochastic, due to ray-casting approach employed in the precomputation phase. The core of the algorithm is again to use a simple table lookup in texture space to find the point visible from another point in a certain direction. The right side of Figure 14 shows the result of such a rendering, while the left side shows the result of direct illumination only.

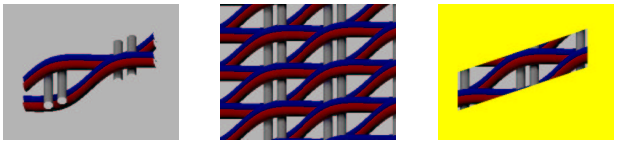
The difference in the light transport for participating media to the algorithms for surfaces is mostly the fact that all tables, including the direct illumination L_d and the scattering textures S_i are now 3-dimensional. As a consequence, the table lookups for the light transport also have to be 3-dimensional. Unfortunately, 3-dimensional dependent texture lookups are not available on current hardware or in specifications of future hardware like DirectX 8. Therefore it is at the moment necessary to implement the table lookups in software. In Sections 7 and 8 we discuss the performance implications of this. We will see that the method can be quite fast even in software implementations.

7 Applications and Results

For the implementation of our methods we chose two different platforms. The methods described in Sections 3, 4, and 5 were implemented on SGI workstations as well as PCs with GeForce class graphics cards, while the methods from Section 6 were only implemented on an Octane VPro, which supports 3D textures and provides enough texture RAM for our volume rendering algorithm.

7.1 Efficient Simulation of BRDFs

As a first application of our method we consider the simulation of BRDFs. We used our methods for generating BRDF samples for several different micro geometries by first computing shadowing and indirect illumination in texture space as described above. We then rendered an orthographic image of the geometry from the viewing direction to handle occlusion. For the BRDF computation we assume periodic micro geometry, which means we also have to handle occlusions between several periods of the geometry. Rather than replicating the geometry to account for this kind of occlusion, we simply replicate the 2D image of one period, and composite multiple copies back to front. A BRDF sample is then obtained by averaging over the area covered by one copy of the micro geometry. Figure 9 demonstrates the acquisition process. If we sam-



(a) (b) (c)

Figure 9: *(a) micro geometry after lighting computation (b) replication and composition of the 2D image. (c) only the area visible through the yellow window is averaged (only pixels with alpha $\neq$ 0)*

ple light and viewing directions over the sphere rather than the hemisphere, we can also account for transmission, yielding a bidirectional scattering distribution function, or BSDF. In this case it is usually advisable to also store a transparency value for each direction, which accounts for the Dirac peak of light passing straight through the material. To obtain this transparency, we generate an alpha mask in the framebuffer during the rendering of the geometry. This mask represents pixels that are actually covered by the geometry. During averaging of the BRDF sample, the ratio of covered and uncovered pixels is taken into account for generating the transparency.

The computation time for one BRDF sample depends mostly on the texture size and on the number of directions used for computing the indirect light. In our case we used 128 sample directions for the integration and were able to compute a single sample for a 32x32 texture in 0.7 seconds, or for a 64x64 texture in 2.3 seconds. Due to the cost of traditional simulation algorithms, the usual approach of simulating BRDFs with a virtual gonioreflectometer is to acquire only a small number of samples, and to project those into a basis like Spherical Harmonics [12] or cosine lobes [21] in order to arrive at a smooth BRDF representation. We found that this method blurred out a lot of the detail for some of our more complex micro-geometry, and therefore obtained a more dense sampling with 10000 samples. For a model fitting into a 64x32 texture the simulation therefore takes slightly more than four hours. We took samples for all combinations of 200 viewing and lighting directions distributed on the whole sphere.

The resulting tabular BRDFs were then used in a ray tracer to generate the scene in Figure 15. The sofa's BRDF was computed from the model depicted in Figure 15*c*, consisting of about 3400 vertices. The resulting BRDF is more or less diffuse with a slight color shift from green to blue for different viewing angles. The satin BRDF of the cushion and the tablecloth was computed from the model shown in Figure 15*a*. We used a specular value of $k_s = 0.3$ and an exponent of $N = 8$ for the micro BRDF that also shows up as a specular highlight in the simulated BRDF. The red curtains were made of a woven material modeled with the piecewise parametric surface shown in Figure 15*d*. We aligned the tangents of the curtain model in such a way that the grey cylinders of the micro geometry run horizontally across the curtain. The resulting BRDF is anisotropic and shows clear color shifts to red and blue, respectively, for grazing viewing angles. Also note how the BRDF becomes less transparent for these angles. This behavior is even more prominent for the BRDF generated from the micro geometry shown in Figure 15*b*, which we used for the almost transparent curtains in front of the windows. Note how the curtains are nearly invisible for orthogonal viewing directions and only become grey and less transparent for grazing angles.

Model	Size	Time in sec		Memory in kB	
		Scat.	Shad.	Scat.	Shad.
c	64x64	82.46	85.47	2051	257
d	64x32	18.91	15.11	4099	513

Table 1: *Precomputation times for 128 scattering directions and 32 shadow regions for the models in Figure 15c and d.*

The timings for the precomputation of scattering textures and shadow fractions for the models *c* and *d* in Figure 15 can be taken from Table 1. The set of scattering directions consisted of 128 directions uniformly distributed over the sphere.

We divided the sphere into 32 regions to determine fractional visibility for the shadow computations. The column marked "Size" refers to the amount of texture space used up by the models.

We also used the method described by Kautz and McCool [22] to render the BRDFs in hardware. After reparameterizing the BRDF, this method approximates the 4D BRDF by a product of two 2D functions, each of which is sampled and stored in a texture map. During rendering the product is computed using blending operations. The torus in Figure 10 was rendered with the same BRDF as the red curtains in Figure 15 (the transmission of the material is ignored here). Note the color shifts to red and blue for the grazing angles.

Figure 10: *Frame from hardware rendering of BRDFs as described in [22] using BRDF computed from model in Figure 15d. Note the color shifts at grazing angles.*

7.2 Generation of Bidirectional Texture Functions (BTFs)

By slightly modifying the algorithm for computing BRDF samples sketched above, we can also compute samples for BTFs [15, 16]. Again we compute the direct and indirect light, as well as shadows and repeat the scene to account for occlusion. However, we store a whole image per combination of one viewing and one lighting direction, instead of only an averaged BRDF sample. Figure 11 shows four sample images computed by our method. Since BTFs are six-dimensional functions, a faithful representation is very demanding in terms of memory and bandwidth. For instance, writing a computed image for the model in Figure 15*b* takes nearly as long as the illumination computation. This makes the generation of BTFs almost a factor of 2 slower than the BRDF simulation.

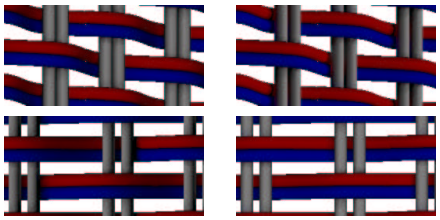


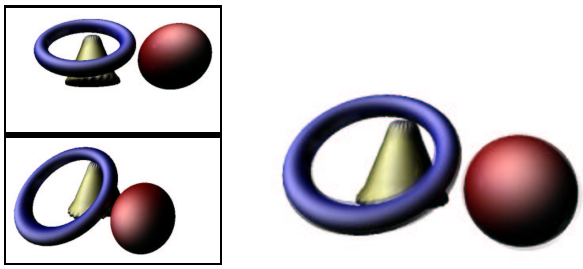
Figure 11: *Four BTF samples generated for two different light and two different viewing directions.*

In [23] we present a technique for creating and rendering the spatially variant BRDF of textiles. To capture the necessary data, we slightly modified the algorithm for computing BTF samples and render the geometric model of a stitch or weave for a set of different light and viewing directions. After fitting a specialized BRDF model to the data, we are able to generate realistic images of clothing, as can be seen in Figure 12.

7.3 Computation of Light Fields

Another application for our method is to generate light fields of globally illuminated scenes. Light fields consist of a high number of images of a scene for different view points. The information in these images can be used to render the scene at interactive rates, completely independent of its complexity. In comparison to the BRDF and the BTF, the illumination remains

Figure 12: *Data for spatially variant BRDF was obtained with algorithm similar to Section 7.2*



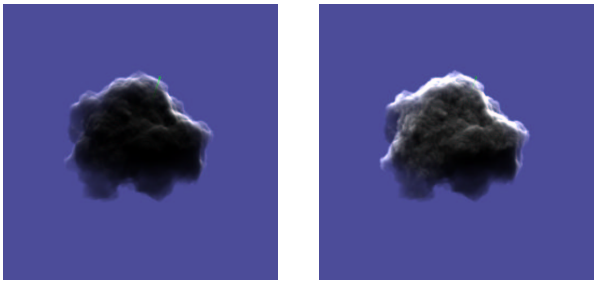
(a) *(b)*

Figure 13: *(a) slices of the light field computed using our methods. (b) rendering of the light field*

the same over all images, only the camera location changes. Therefore it is sufficient to compute the direct, indirect light and shadows once for a scene similar to the one depicted in Figure 13.[2] We then rerender the scene from different camera locations, storing separate images for each eye point, see Figure 13(a). Similar to the applications described above we manage to achieve extremely low computation times by reusing visibility information, allowing us to generate a light field of the resolution 256^2x16^2 in less than two minutes. Figure 13(b) shows a snapshot from the rendering of this light field.

7.4 Participating Media

As mentioned above, the algorithm for participating media has been implemented for SGI VPro, which has enough texture memory for our purposes, and supports 3D textures. On this platform, computing the direct illumination L_d as described in Section 6.1 and displaying it as described in Section 6.2 is interactive, at about 12 frames per second for the 256^3 cloud data in Figure 14. If we assume a constant phase function $p(\alpha) = c$, L_d does not need to be recomputed if the light source remains fixed. In this case, we achieve about 20 fps. for displaying the direct illumination in the volume as described in Section 6.2.



(a) (b)

Figure 14: *Volume lighting. (a) direct illumination only. (b) direct illumination and one-bounce indirect illumination.*

For the computation of the indirect illumination it is advisable to reduce the resolution of the grid in order to save memory. For the cloud we used a grid size of 64^3, with precomputed visibility for 64 directions, yielding 64 MB of scattering information that can be generated

[2] Especially in this application we might want to compute the lighting for more than one light source. To do so, we have to modify the first part of the algorithm (see Section 3), the computation of L_d, to accumulate the illumination of all light sources in direction $-\vec{d}_i$. The second step (scattering) remains the same as for one light source.

in about 17 minutes by volume ray-casting on a 300 MHz R10k. As described in Section 6.2, we perform the light transport for the volume rendering algorithm in software. This means that for computing one-bounce indirect illumination in the participating medium, we have to traverse these 64 MB of data in software, and perform 16 million table lookups. This takes about 1.6 seconds, with about 0.7 seconds spent on cache misses (we measured this by comparing the rendering times to a variant of the algorithm that performed the same operations on constant data). We expect that these costs could be reduced by adding hardware support for this kind of 3D table lookup, because the graphics chips have a higher bandwidth to the texture RAM than the CPUs have to the main memory. In addition, graphics chips are able to employ more aggressive caching and prefetching schemes than general-purpose CPUs.

In the special case of a constant phase function we only need to recompute the indirect illumination once for every change of the light source. In that case the indirect illumination can be merged into one volume with the direct illumination and displayed at 12 fps.

8 Discussion and Conclusions

As we have seen in the previous section, our approach allows for the efficient computation of illumination for different lighting and viewing situations, using precomputed visibility information. The precomputation times are quite moderate, and we can amortize them over many different light transports, for example to generate different light paths for a single image, or to compute many different images with varying illumination and changing camera positions.

The efficiency of our algorithms is not only due to the fact that we reuse visibility information that would have had to be recomputed with other approaches. Another reason is that we use regular data structures, which allow us to reduce the light transport operator to simple, constant time table lookups. As a result we obtain regularized memory access patterns, which already yield a high performance in software implementations. However, we have argued that we can achieve additional performance benefits by mapping the light transport onto graphics hardware. This in turn is possible because similar table lookups are performed for all points on the surface or in the volume, which leads to a SIMD-style parallelism.

Since most computations take place in texture space, we do not make much use of specific hardware features like geometric transformations and scan-conversion of geometric primitives. In fact these are only used for the final display of the scene after the illumination has already been computed. The use of graphics hardware is still interesting for this kind of algorithm, mostly because it is optimized for high bandwidths to texture and framebuffer RAM with specific caching schemes specifically designed for this kind of lookup process.

Of course the final display of the illuminated scene could easily be replaced by other algorithms like interactive ray-tracing, while keeping our algorithms for light transport based on precomputed visibility. In this case we would only need a high-bandwidth architecture optimized for performing the table lookups.

Furthermore, our algorithm should be easy to parallelize on multiprocessor systems or clusters. One way would be to use different CPUs to compute different light paths. Each CPU then only needs to know the visibility textures correspond-

Figure 15: *Using our methods, BRDFs can efficiently be computed for different micro geometry (a-d). These BRDFs can then be used, e.g. in a ray tracer, to correspondingly shade objects in a scene: the cushion and tablecloth exhibit the BRDF computed from (a) (satin), (b) was used for the nearly transparent curtains over the windows, the sofa's BRDF was computed from (c), and (d) was used for the red curtains.*

ing to directions that are comprised in its paths.

In this work we have demonstrated the feasibility, quality, and performance of the proposed method by applying it to the simulation of BRDFs, the computation of light fields and BTFs, and near-interactive illumination in participating media. We think that these applications show the possibilities of utilizing our method also for other scenarios, such as the efficient illumination of animation sequences.

References

[1] G. Ward and P. Heckbert. Irradiance Gradients. *Third Eurographics Workshop on Rendering*, pages 85–98, May 1992.

[2] H. W. Jensen. Global Illumination using Photon Maps. In *Eurographics Rendering Workshop 1996*, pages 21–30. Eurographics, 1996.

[3] P. Shirley, B. Wade, P. Hubbard, D. Zareski, B. Walter, and D. Greenberg. Global Illumination via Density Estimation. In *Eurographics Rendering Workshop 1995*. Eurographics, June 1995.

[4] W. Heidrich, K. Daubert, J. Kautz, and H.-P. Seidel. Illuminating Micro Geometry Based on Precomputed Visibility. In *Computer Graphics (SIGGRAPH '00 Proceedings)*, pages 455–464, July 2000.

[5] G. Greger, P. Shirley, P. Hubbard, and D. Greenberg. The Irradiance Volume. *IEEE Computer Graphics & Applications*, 18(2):32–43, March–April 1998.

[6] P. Hanrahan, D. Salzman, and L. Aupperle. A Rapid Hierarchical Radiosity Algorithm. In

Computer Graphics (SIGGRAPH '91 Proceedings), pages 197–206, July 1991.

[7] S. Gortler, P. Schröder, M. Cohen, and P. Hanrahan. Wavelet Radiosity. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 221–230, August 1993.

[8] N. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, 4(2):109–117, July 1988.

[9] J. Stewart. Computing Visibility from Folded Surfaces. *Computers and Graphics*, 23(5):693–702, October 1999.

[10] F. Durand, G. Drettakis, and C. Puech. The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 89–100, August 1997.

[11] B. Cabral, N. Max, and R. Springmeyer. Bidirectional Reflection Functions From Surface Bump Maps. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 273–281, July 1987.

[12] S. Westin, J. Arvo, and K. Torrance. Predicting Reflectance Functions From Complex Surfaces. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 255–264, July 1992.

[13] M. Levoy and P. Hanrahan. Light Field Rendering. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 31–42, August 1996.

[14] S. Gortler, R. Grzeszczuk, R. Szelinski, and M. Cohen. The Lumigraph. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 43–54, August 1996.

[15] K. Dana, B. van Ginneken, S. Nayar, and J. Koenderink. Reflectance and Texture of Real World Surfaces. *ACM Transactions on Graphics*, 18(1):1–34, January 1999.

[16] J.-M. Dischler. Efficiently Rendering Macro Geometric Surface Structures with Bi-Directional Texture Functions. In *Rendering Techniques '98 (Proc. of Eurographics Workshop on Rendering)*, pages 169–180, June 1998.

[17] L. Williams. Casting Curved Shadows on Curved Surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, August 1978.

[18] A. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: Multiresolution Adaptive Parameterization of Surfaces. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 95–104, July 1998.

[19] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *1994 Symposium on Volume Visualization*, pages 91–98, October 1994.

[20] J. Kajiya and B. Von Herzen. Ray Tracing Volume Densities. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pages 165–174, July 1984.

[21] E. Lafortune, S. Foo, K. Torrance, and D. Greenberg. Non-Linear Approximation of Reflectance Functions. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 117–126, August 1997.

[22] J. Kautz and M. McCool. Interactive Rendering with Arbitrary BRDFs using Separable Approximations. In *Rendering Techniques '99 (Proc. of Eurographics Workshop on Rendering)*, pages 247 – 260, June 1999.

[23] K. Daubert, H. Lensch, W. Heidrich, and H.-P. Seidel. Efficient Cloth Modeling and Rendering. In S. Gortler and K. Myszkowski, editors, *Rendering Techniques*, pages 63–70. Springer, June 2001.