Creating light atlases with multi-bounce indirect illumination

Randolf Schärfig · Marc Stamminger · Kai Hormann

Abstract

Indirect illumination is an essential part of realistically rendering virtual scenes. In this paper we present a new method for computing multi-bounce indirect illumination for diffuse surfaces which is particularly well-suited for indoor scenes with complex occlusion, where an appropriate simulation of the indirect illumination is extremely important. The technique presented in this paper combines the benefits of shooting methods with the concept of photon mapping to compute a convincing light map for the scene with full diffuse lighting effects. The main idea is to carry out a multi-bounce light distribution almost entirely on the GPU using a shooting approach with virtual point lights. The final result is then stored in a texture atlas by projecting the energy from each virtual point light into the texels visible from its perspective. The technique uses only few resources on the graphics card and is flexible in the sense that it can easily be adjusted for either quality or speed, allowing the user to create convincing results in a matter of seconds or minutes, depending on the scene complexity.

Citation Info

Journal
Computers & Graphics
Volume
55, April 2016
Pages
97–107

1 Introduction

Although indirect illumination is essential for realistically rendering virtual scenes, it is rarely computed on the fly in interactive computer graphics, due to the expensive computations that are required. Instead, indirect illumination is usually either precomputed or created by artists and stored in a special texture map called *light map*. This resembles indirect lighting well enough to improve the visual quality of the scene but cannot react to changes in the lighting conditions. Although there are techniques that can be used to illuminate objects in real time by indirect light, they require the indirect lighting situation of the surrounding scene to be known.

One technique that is used excessively in computer games and other interactive graphic programs nowadays is based on the concept of *ambient occlusion* or its modern variant, *screen space ambient occlusion* [20]. It approximates the amount of light that a certain surface point is likely to receive from the ambient light in the scene by checking how strongly it is occluded by nearby surfaces. This value is then used to manipulate the constant ambient brightness term for the corresponding surface point, which gives a more realistic look to the scene, but it is static and does not react to changes in the lighting conditions. This effect increases the realism of a scene, but does not introduce the effect of light shining around corners due to multi-bounce reflections or colour bleeding. However, the latter can be achieved by a recent extension [28], which precomputes the possible light paths and stores them in the texture atlas of the scene. Moreover, in combination with *precomputed radiance transfer* [25], the previously mentioned techniques give very good approximative results for scenes with moving objects.

All these techniques only provide a rough approximation of the global illumination in order to improve the visual quality of rendered scenes, but they are not able to react to changes in the lighting situation. Other techniques, that achieve interactive rates, can approximate indirect illumination in a scene by computing only one or two light bounces, but this is insufficient for complex scenes, like the one in Figure 10, because it prevents the light to spread out far enough.

Other approaches, like *photon mapping* [12], achieve perfectly good indirect illumination quality, but at high computational costs, so that they are not suitable even for generating physically correct previews. The rendering time can be reduced dramatically by carrying out the computations in parallel on the GPU [18], albeit at the cost of compromising the quality of the result.

In general, most indirect illumination techniques perform the rather costly lighting computations in every frame instead of storing the results and reusing them for subsequent frames, which wastes valuable computational power, in particular if the scene is static and has little changes in the lighting conditions.

1.1 Contribution

In this paper we present a novel approach for approximating the indirect illumination of a virtual scene with multiple light bounces on the GPU and storing it in the *light atlas* of the scene. This approach is not intended to compute direct lighting effects, which can be generated more efficiently with other standard techniques. Furthermore we restrict the computation to purely diffuse surfaces. Our multi-bounce method captures the phenomenological properties of the exact diffuse light distribution with high quality in a couple of seconds up to a few minutes, depending on scene complexity and the desired quality of the result. Since many scenes require significantly more than just one or two light bounces to produce realistically looking results (see Figure 10), we separate the basic light distribution from the gathering of these results into the final visible result.

We first scatter the light energy among *virtual point lights* (VPLs) that are evenly spread over the scene geometry, similar to the technique described by Lehtinen et al. [15]. This iteratively distributes the light over the scene and gives a rough approximation of the energy distribution (Section 3). Hence, the VPLs act like the patches in classical shooting-gathering approaches [1, 3].

As the results computed in this first step are only intermediate and never visible to the user, an approximation is perfectly fine. In the second step, however, we create the illumination that the user perceives and therefore the result needs to be visually appealing and smooth, capturing all the phenomenological effects of soft shadows and brightness gradients. Since all VPLs used in the first step store the amount of light that they distributed during the entire process, we can now use this information to store this energy directly into a light atlas that covers the whole scene and stores the radiance of the surface points in its texels (Section 4). This light atlas is then used for rendering the scene with indirect illumination. The light atlas does not store the direct lighting effects and in all our examples we consider the indirect illumination only.

Combining these two techniques, we achieve short computation times by distributing the light only coarsely in step one and get a high quality by switching to a high resolution rendering in step two when computing the results that are used to finally display the scene (Section 5). Furthermore, the rendering can be performed in the background while the user is navigating the scene. That enables an early judgement of the lighting situation and the user can cancel the computation early on and change the lighting to better fit the intended effect.

Our algorithm, which is restricted to diffuse surfaces, computes realistically looking results for complex scenes. The results are smooth and the render time does not depend on the number of direct lights. Possible applications, where these conditions apply include architectural light design, where most surfaces are diffuse and quick previews help speeding up the process of placing lights.

The main contributions of this paper are:

- approximating a physically correct low-resolution light distribution with *multiple bounces*;
- converting the result into a high-resolution light distribution and transferring it to a *light atlas*.

2 Related Work

Most indirect illumination techniques aim at perfect results without considering to compute them at interactive frame rates [7, 11, 13]. Since our goal is the exact opposite, we focus on other real-time techniques only.

The first approach of computing indirect illumination was through the radiosity equation formulated by Cohen and Greenberg [2]. This radiosity method describes an energy equilibrium within a closed scene assuming that all emissions and reflections are ideally diffuse and the light distribution is computed iteratively over the discretized surface of the scene. Then shooting and gathering approaches were introduced and even ported on the graphics card by Coombe et al. [3]. This technique uses the aforementioned approach for the coarse distribution of light within the scene, followed by a highly precise method for storing the lighting results, therefore changing the geometry of the scene and creating a higher tessellation. In our approach we only use the shooting method in the beginning of the light distribution and then switch to a scene texture atlas for storing the high resolution results. Therefore, our approach does neither depend on nor does it change the scene geometry. Szécsi et al. [26] propose to precompute the expensive integrals needed for indirect illumination for all but one variable, the light position, and store them in a texture atlas of the scene. During rendering, the radiance of the visible scene pixels is determined by evaluating this precomputed data for the given light positions. Although the results look very promising, the amount of memory needed is

rather big, even for scenes without complex occluder configurations. Our technique can handle complex scenes using only a light atlas, which is required anyway.

Kristensen et al. [14] presented an algorithm that is able to indirectly relight scenes in real-time. This is achieved by extending the idea of precomputed radiance transfer and introducing the idea of unstructured light clouds to account for local lighting. The indirect illumination of a static scene is precomputed for a very dense light cloud, which is then compressed to contain just a small number of lights. While rendering the scene, its indirect illumination is approximated by interpolating the precomputed radiosity of the closest lights in the light cloud. This requires a relatively high number of values stored for every vertex of the scene and can only compute the lighting at these scene vertices. Therefore, the quality of the resulting image depends strongly on the tessellation of the scene. The technique gives nice results in static scenes with moving lights, but it cannot handle scene changes or large moving objects that strongly change the scene's lighting situation. Instead, our technique uses a high resolution texture to store the indirect shadows, which allows us to handle even high frequency lighting details. Moreover, our technique does not require any further computations once the light distribution is stored, which in turn reduces the time needed for rendering every frame.

A technique that reaches interactive results for single-bounce indirect illumination was introduced by Dachsbacher and Stamminger [6]. The authors use *reflective shadow maps* generated from the direct light source and a certain number of pixels in this shadow map act as new point light sources to illuminate the scene. This is done in the screen space of the final scene rendering, using the geometry information of each visible pixel as well as the information given for all the point light sources. Due to the nature of this technique it can only compute one-bounce indirect lighting and it does not compute the visibility. Hence, it cannot create indirect shadows, while our technique creates such soft shadows naturally. Prutkin et al. [21] suggest to speed up global illumination through reflective shadow maps by clustering the VPLs and treat them as single area lights. Lensing and Broll [16] apply reflective shadow maps, but reduce the number of computations by first clustering visible points, depending on their geometric properties and then computing only one radiosity value for all similar pixels. Dong et al. [8] compute indirect illumination through clustered visibility. VPLs are clustered into area lights and soft-shadow techniques are used to compute the illumination of visible screen pixels. The above techniques introduce colour bleeding and soft shadows, but only from those points which are directly illuminated. They are fast because they use a very low number of VPLs, which in turn might be problematic in situations where many direct lights illuminate non-overlapping parts of the scene.

An approach that computes double-bounce indirect lighting was introduced by Crassin et al. [4] and can be seen as an extension of reflective shadow maps. Instead of a shadow map, this method uses a sparse octree to store the radiosity from direct light sources. Then the radiosity is gathered for every visible pixel in the final image from this octree. That makes this technique effectively a double-bounce algorithm with rather coarse approximations in the gathering and the light distribution. Yet, this algorithm is able to handle indirect shadows in contrast to reflective shadow maps [6]. Nevertheless, this technique can only handle up to two light bounces, while our technique distributes the light with multiple bounces until the distribution is physically plausible.

A technique that combines the ideas in [14] with the concept of *light-cuts* is proposed by Ritschel et al. [22]. They compute indirect lighting on glossy surfaces by storing coherent surface shadow maps which allows for fast visibility tests within a scene even with moving objects. For indirect lighting, the authors use the idea of virtual point lights and light cuts to compute the illumination distribution within the scene. Although this technique produces good results, it is useful only in scenes that are nicely illuminated by a few (one or two) light bounces, while our technique targets at scenes that have many occlusions and need a high number of light bounces to produce a realistic result.

Lehtinen et al. [15] describe a method for computing indirect illumination that is decoupled from the underlying geometry, since it only computes and stores the light transport for randomly scattered points in the scene. For later evaluating other points, they use scattered data approximation. The authors also define a hierarchy over these points and store the difference between the current and the next level. This technique is similar to ours regarding the initial light distribution, but we use a texture atlas for the final visible result to capture even small lighting details.

The technique proposed by McGuire and Luebke [19] uses a photon mapping approach to compute indirect lighting. The authors describe an algorithm that first uses the graphics card to distribute initial photons from a light source using the GPU-projection for the expensive visibility tests. The data is then read back to the CPU where it is processed with standard ray-tracing techniques and the results are recorded in an octree structure that stores the radiance. This data structure is then sent to the graphics card again

for computing the final gathering. While this technique creates a reasonable ambience that is adequate for computer games, the results look rather uniform and lack highly detailed shadow effects, due to the low number of photons and the wide spread of the corresponding photon-volumes.

Other techniques [10, 11] use the extreme parallel capability of modern GPUs to accelerate ray-tracing approaches. They cluster different light rays according to their direction and then use the GPU to compute the visibility for all these bundles simultaneously. This makes the computation much faster than on the CPU, but still does not achieve interactive frame rates. Our technique creates similar quality, but in shorter time.

Luksch et al. [17] propose a technique for computing the light atlas of a scene at almost interactive speed. They partition the scene into polygons and distribute the light energy among virtual light sources, one for each polygon. In a final gathering step, they render the light atlas and collect the radiosity for each texel from the direct and virtual light sources. Instead, we use a shooting method to distribute the radiosity from all VPLs into the light atlas.

Jensen and Christensen [13] introduce the idea of photon mapping, which is a stochastic approach to indirect illumination. Algorithms based on photon mapping shoot energy into the scene, starting at the light sources. Whenever one of the particles (photons) hits a surface element in the scene, the energy that the photon transports is stored at this point in a *photon map* and this surface point is treated as a new light source: it shoots photons into the scene, which carry the amount of light that is reflected at this point. After the light distribution is finished, that is, when all new photons carry an amount of energy below a certain threshold, the photon map is evaluated from the viewpoint of the camera, giving each surface element a certain brightness that depends on the radiosity stored in the photon map at this position. While this technique is still state of the art in terms of quality, the time required for shooting the high amount of photons that is necessary for good and realistic results is very high. Furthermore, this method, as well as other stochastic approaches [10, 11, 12], requires a noise filter, since the results are randomly distributed and therefore scattered points. Our technique distributes the light in a smooth way that has a lot of similarity with photon mapping, but it creates results much faster. Due to the different approach in storing the final result, our technique is also very flexible, allowing the user to trade quality for shorter rendering times on a wide range. With our technique, previews that give a good and physically plausible idea of the final scene illumination can be created in the order of seconds or minutes.

3 Coarse Light Distribution

We assume that the scene consists of diffuse surfaces and is illuminated by a number of direct light sources. The goal of the first step of our method is to quickly approximate the indirect light distribution over the scene in a physically correct manner using multiple bounces. We achieve this by discretizing the scene with uniformly distributed VPLs (Section 3.1) in a precomputing step, where each VPL represents a small surface patch. The irradiance per VPL is then determined in two steps, first by distributing the power from the direct light sources to all VPLs with a standard shooting method [24] (Section 3.2) and then by iteratively distributing the irradiance among the VPLs (Section 3.3). The visibility between two patches is determined by using sample points associated with each VPL to get a more detailed distribution at low costs. The main part of the algorithm is parallelized and runs on the GPU, but the priority queue, which is needed for choosing the next VPL during the light distribution stage, is implemented on the CPU.

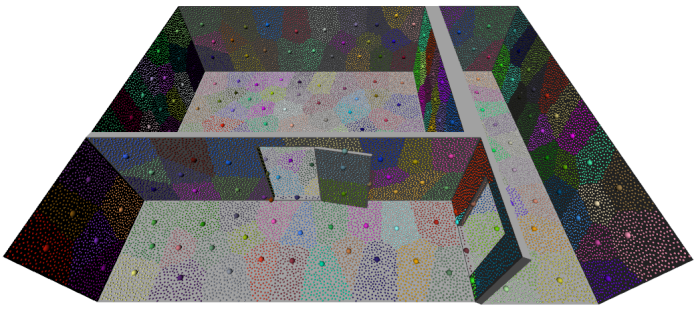


Figure 1: Distribution of VPLs (big dots) and associated sample points (small dots). In this example, the density of sample points is 30 times higher than the density of VPLs, so that the patches represented by each VPL become visible. In practice, it is sufficient to use about 10 sample points per VPL.

3.1 Scene Discretization

We start by uniformly distributing n VPLs $V_1,\dots,V_n$ over the scene with the desired density (see Figure 1) and storing the normals $N_1,\dots,N_n$ and the reflection coefficients $\rho_1,\dots,\rho_n$ of the patches P_j represented by each VPL V_j. The distribution of VPLs is done in a pre-processing step that divides the scene into almost planar patches. Each patch is then triangulated with a pre-determined triangle area size, using the *triangle* tool [23]. The interior vertices of this triangulation are then used as VPL positions. This guarantees each VPL to have a certain offset to nearby corners, which could otherwise lead to artefacts in the light drawing stage. Furthermore, we create sample points associated to each VPL, where each V_j has m_j sample points S_{jk}, $k=1,\dots,m_j$. The sample points are generated in the same way as the VPLs, but with higher density. The latter ensures that we can approximate the area A_j of patch P_j by the number of sample points,

$$A_j \approx m_j C, \tag{1}$$

where

$$C = \frac{A}{m_1+\dots+m_n},$$

with A denoting the overall surface area of the scene.

3.2 Initial Direct Light Distribution

We now assume that the scene is illuminated by k direct light sources $L_1,\dots,L_k$ with emitting powers $\tilde{E}_1,\dots,\tilde{E}_k$, each represented by a light patch with area $\tilde{A}_i$. Note that we use the tilde accent in this section to distinguish quantities that correspond to the direct light sources L_i from those that belong to the patches P_j, which are represented by the VPLs V_j. To distribute the light of the L_i to the VPLs, we compute the initial reflected power of P_j as

$$E_j = \rho_j \sum_{i=1}^{k} \tilde{F}_{ij}\tilde{E}_i, \qquad j=1,\dots,n, \tag{2}$$

where $\tilde{F}_{ij}$ denotes the form factor that describes the geometric relationship between L_i and P_j. Under the classical assumption that the patches are small and not too close to the light sources, it can be approximated by

$$\tilde{F}_{ij} = A_j \frac{\langle l_{ij}, n_i\rangle \cdot \langle -l_{ij}, N_j\rangle}{\pi {d_{ij}}^2} \cdot \mathcal{V}_{ij}, \tag{3}$$

where n_i is the direction of light source L_i, N_j is the normal vector of patch P_j, l_{ij} is the normalized light vector from L_i to V_j, d_{ij} is the distance between L_i and V_j, and $\mathcal{V}_{ij}$ is a visibility factor that describes what percentage of the patch represented by V_j receive light from L_i.

Equation (2) is derived from the standard radiosity equation

$$B_j = \rho_j \sum_{i=1}^{k} \tilde{F}_{ji}\tilde{B}_i, \qquad j=1,\dots,n,$$

by replacing the emitted radiosities $\tilde{B}_i$ of the light sources and the reflected radiosities B_j of the patches with the respective powers $\tilde{E}_i = \tilde{B}_i\tilde{A}_i$ and $E_j = B_jA_j$, and by exploiting the reciprocity of the form factors, $\tilde{F}_{ij}\tilde{A}_i = \tilde{F}_{ji}A_j$.

In order to implement this strategy on the GPU, we store the data in two VBOs. The *patch VBO* contains the list of VPLs and stores for each V_j its position in world coordinates, the normal N_j, the number m_j of associated sample points, and an offset o_j into the sample VBO. This *sample VBO* is just the list of sample point positions, grouped by VPLs, so that the position of S_{jk} can be accessed with the index o_j+k. In addition, we need a *transform feedback buffer* (TFB) that contains the *incoming* radiosities or *irradiances*

$$I_j = \frac{B_j}{\rho_j} = \frac{E_j}{\rho_j A_j} \tag{4}$$

of the patches P_j and is initialized with zero values. Note that we use the irradiance I_j instead of the power E_j or the radiosity B_j for efficiency reasons. Dividing both sides of (2) by ρ_jA_j, we get

$$I_j = \sum_{i=1}^{k} \tilde{F}'_{ij}\tilde{E}_i, \qquad j=1,\dots,n, \tag{5}$$

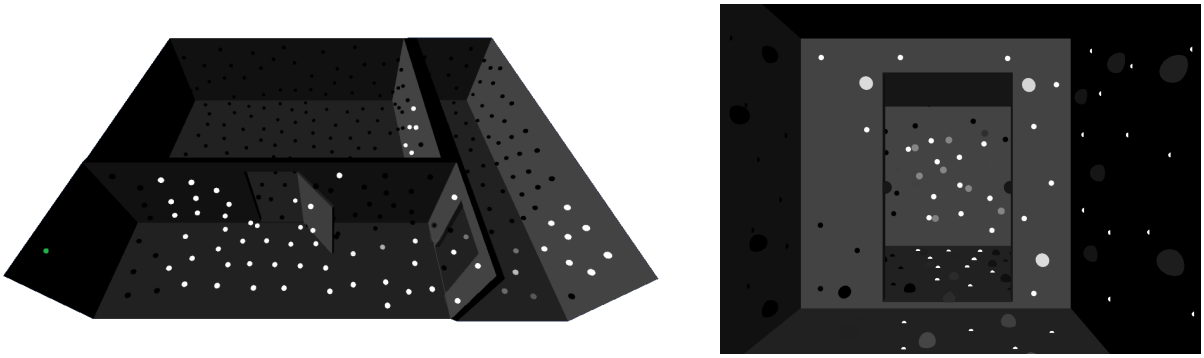


Figure 2: Light distribution from the active VPL V_i (green dot) to all other VPLs. The left image visualizes the VPLs that are occluded as black dots, and the brightness of the other dots corresponds to the amount of radiosity that each VPL receives from V_i. The right image shows a close-up to the door on the right, with the sample points (small dots) in addition to the VPLs. Sample points appear in black or white, depending on their visibility. Note how the ratio of visible and occluded sample points influences the brightness of the corresponding VPL.

with the scaled form factor

$$\tilde{F}'_{ij} = \frac{\tilde{F}_{ij}}{A_j} = \frac{\langle l_{ij}, n_i \rangle \cdot \langle -l_{ij}, N_j \rangle}{\pi {d_{ij}}^2} \cdot \mathcal{V}_{ij}. \tag{6}$$

The advantage of computing (5) instead of (2) is that we do not need to access ρ_j. Moreover, we see that I_j is independent of the specific light patch areas $\tilde{A}_i$. Therefore, we do not need to specify the actual light patches and can describe each light source solely by its position, direction, and emitting power.

The direct light distribution is now implemented as a shooting method by iterating over the L_i. For each L_i we first render the scene from the light source's point of view and store the depth values in the FBO D. We then handle all VPLs in parallel by processing the patch VBO with a vertex shader. This shader determines the contribution of L_i to I_j and accumulates these values in the TFB, thus computing (5). While most terms of the form factor $\tilde{F}'_{ij}$ in (6) can be derived from the information associated with L_i and V_j, the visibility factor needs to be determined on the fly to account for changes in the scene configuration, like moving objects. To this end, we adapt the hemicube algorithm [2] and project the sample points S_{jk} of V_j, which are looked up in the sample VBO, into the scene, by multiplying them with the current modelview-projection-matrix. We then test the transformed z-components z_{jk} against the values in D, and compute $\mathcal{V}_{ij}$ as

$$\mathcal{V}_{ij} = \frac{1}{m_j} \sum_{k=1}^{m_j} \delta_k, \qquad \delta_k = \begin{cases} 1, & \text{if } z_{jk} \leq \text{depth in } D, \\ 0, & \text{if } z_{jk} > \text{depth in } D, \end{cases}$$

which is essentially shadow mapping for each of the sample points.

3.3 Iterative Light Distribution

We finally distribute these initial radiosities among the VPLs with an adapted version of the classical progressive refinement algorithm [1]. We start by setting for each V_i the unshot irradiance ΔI_i to I_i from (5) and creating a priority queue of VPLs, sorted by the VPLs unshot power ΔE_i. According to (1) and (4), these values can be approximated as

$$\Delta E_i \approx \rho_i m_i \Delta I_i C, \tag{7}$$

and we compute and update them on the CPU after each shooting step. We then shoot the unshot irradiance of the first element V_i of the queue into the scene (see Figure 2) by computing

$$\begin{aligned} R &\leftarrow \rho_i F_{ji} \Delta I_i, \\ \Delta I_j &\leftarrow \Delta I_j + R, \\ I_j &\leftarrow I_j + R, \end{aligned} \tag{8}$$

for $j = 1, \ldots, n$, set ΔI_i to zero, and update the priority queue. This process is iterated until the largest ΔE_i is smaller than some threshold. Here, the form factor F_{ij} describes the percentage of the power exchange from

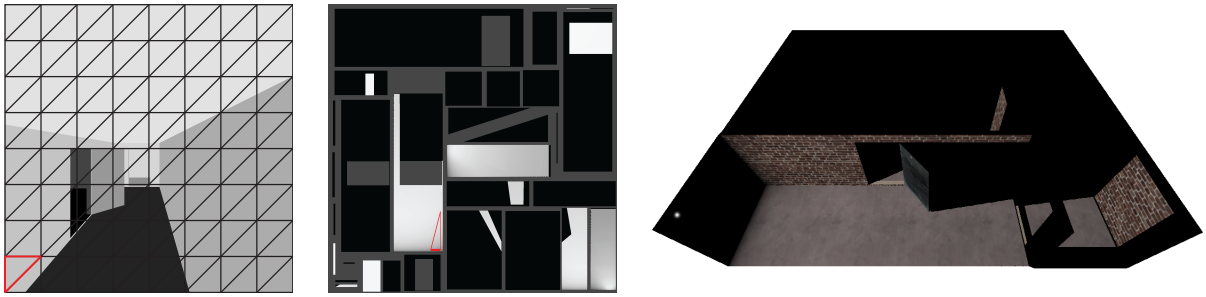


Figure 3: The light is distributed from a VPL into the scene by first rendering the scene from the viewpoint of the VPL and overlapping it with a 2D triangle mesh (left). Each triangle is then projected into the light atlas $\mathcal{I}$ (centre), and the light distribution is computed for all covered texels (right).

P_i to P_j and can be approximated similar as above by

$$F_{ij} = F'_{ij} A_j, \qquad F'_{ij} = \frac{\langle l_{ij}, N_i \rangle \cdot \langle -l_{ij}, N_j \rangle}{\pi {d_{ij}}^2} \cdot \mathcal{V}_{ij},$$

where l_{ij} is the normalized vector from V_i to V_j and d_{ij} is the distance between these two VPLs. Note that the scaled form factors F'_{ij} are symmetric in i and j, so that the right hand side in (8) simplifies to

$$\rho_i F_{ji} \Delta I_i = F'_{ji} \rho_i A_i \Delta I_i = F'_{ij} \Delta E_i.$$

The unshot power ΔE_i is taken from the priority queue, and the scaled form factors F'_{ij} and the visibility factor $\mathcal{V}_{ij}$ are computed as in Section 3.2 using the sample points S_{jk} associated with the receiving VPL V_j. The only difference is that we now iterate only over VPLs and no longer over direct light sources.

As in Section 3.2, the whole computation is done in parallel on the GPU and therefore very fast. The CPU is involved only for updating ΔE_i, sorting the priority queue, and selecting the next VPL for distributing its unshot radiosity. Since the form factors are computed on the fly, we avoid the $O(n^2)$ storage overhead.

4 Filling the Light Atlas

After having distributed the radiosity among the VPLs, the goal of the second step of our method is to create a high resolution light distribution and store it in the light atlas $\mathcal{I}$. We achieve this by shooting the light from each VPL into the scene and accumulating the irradiance for all texels in the light atlas. The main idea of this algorithm is simple (Section 4.1), but special attention needs to be paid to seams and shadow edges (Section 4.2). We then improve both quality and timings by adding a per-texel soft-shadow algorithm (Section 4.3).

4.1 Final Shooting Step

For each texel τ of the light atlas $\mathcal{I}$ with coordinates (u, v) we compute its irradiance as

$$\mathcal{I}(u,v) = \sum_{i=1}^{n} F'_i(u,v) E_i, \tag{9}$$

where $F'_i(u,v)$ is the scaled form factor between V_i and the scene point $P(u,v)$ which corresponds to the texel τ. As in the first step of the algorithm (Section 3) we implement the computation of the irradiance in (9) as a shooting method by looping over the VPLs and accumulating their contributions in $\mathcal{I}$, but with one important difference regarding the visibility test.

During the distribution of light among the VPLs, the number of light receivers is relatively small and it is efficient to determine the visibility for each VPL. In this final step, however, the resolution of the light atlas is usually high, hence testing each texel for visibility is costly. Therefore, we propose a different strategy for finding the texels that are visible from the current VPL V_i. We overlap the scene, as seen from V_i, with a triangle mesh and project each triangle first onto the scene geometry and from there into $\mathcal{I}$ (see Figure 3). The projected triangles thus cover exactly that part of $\mathcal{I}$ which contains the texels that receive light from V_i.

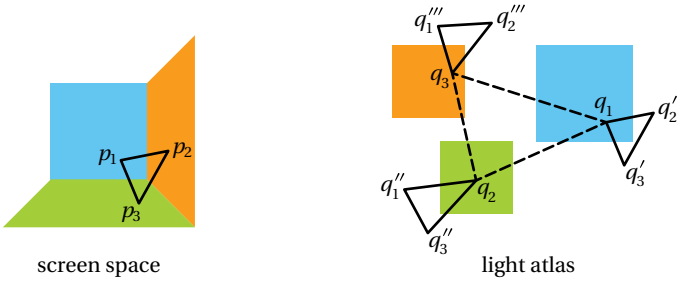


Figure 4: The vertices of a triangle that overlaps a seam (left) get projected to texture coordinates in different charts of the light atlas. As the charts may be packed arbitrarily, the resulting texture triangle (dashed) does not cover the correct texels, which in turn are the ones inside the three black triangles (right). A similar situation occurs at shadow edges.

In order to implement this strategy on the GPU, we first place a camera at the position of V_i, looking in the direction of N_i. We then create a uniform 2D triangle mesh M that covers the camera's viewport and render the scene into a geometry buffer $G = [T, N, D]$ with three layers. These layers store the texture coordinates, the normals with respect to the local coordinate system of V_i, and the distance to the camera of the underlying scene geometry, respectively. This render step effectively performs the first part of the projection mentioned above, albeit in the inverse direction. That is, the scene is projected onto M rather than the other way around. We now draw M using a vertex shader that samples the texture coordinate buffer T at the coordinates (x, y) of each mesh vertex and replaces the latter with $T(x, y)$. This turns each mesh triangle into a triangle in $\mathcal{I}$ and completes the projection process. We also sample the normal buffer N and the distance buffer D at (x, y) and send these values as vertex attributes further down the pipeline. In the fragment shader we therefore have all the relevant information for computing and accumulating the contributions $F'_i(u, v)E_i$ of each V_i to the texel at (u, v), except for the light vector l. Since the light vectors are constant with respect to the local coordinate systems of the VPLs, we precompute them for each pixel of a texture with the same resolution as G, sample this texture at (x, y) in the vertex shader, and introduce this value as an additional vertex attribute. Note that this procedure automatically resolves the visibility test, because only visible texels are considered.

In order to correctly distribute the light of the VPL into the whole scene, we would have to set up the camera with a large opening angle, such that all visible parts of the scene are covered. But as this induces high distortions in the projection, we consider instead a hemicube to cover the 180° view from the VPL's position and use the geometry shader to replicate each triangle of M five times, once for each side of the hemicube.

4.2 Handling Seams and Shadow Edges

The two-step projection process described in the previous section works as long as a triangle does not straddle a seam or a shadow edge. In these cases, the vertices of the triangle get projected into different charts in $\mathcal{I}$ so that the resulting texture triangle is wrong (see Figure 4). To resolve this problem, we propose to replicate and project those triangles into each chart involved by reconstructing the missing coordinates. This requires an additional layer I in G which stores in each pixel the chart ID of the underlying scene geometry.

The challenge here is that not all the vertex coordinates of the projected triangles can be read from the geometry buffer G. For example, in the situation shown in Figure 4, the copy of the triangle $[p_1, p_2, p_3]$, which is projected into the first chart, is missing the vertices q'_2 and q'_3.

We resolve this problem by using Taylor's theorem to approximate these missing vertices as

$$q'_k \approx q_1 + \nabla T(p_1)(p_k - p_1), \qquad k = 2, 3.$$

However, this approximation is correct only if T varies linearly, but due to the perspective correction, which is applied to T by the GPU when the scene is rendered into G, this is not the case. Therefore, we replace the layer T of G with the layer U, which stores the texture coordinates divided by the homogeneous w-coordinate. We turn off the perspective correction for this layer and instruct the GPU to also create the layer ∇U with the screen-space derivatives of U. Moreover, we add the layer W, which contains the reciprocal w-coordinates without perspective correction, and its derivatives ∇W to G. With these layers at hand, we can now compute the projections $q_k = T(p_k)$ of the mesh vertices p_k as

$$q_k = \frac{U(p_k)}{W(p_k)}, \qquad k = 1, 2, 3, \tag{10}$$

Figure 5: It may happen that a triangle of M overlaps parts of the scene (blue) that belong to charts that are different to to the ones corresponding to the triangle vertices. This can happen, for example, at the corner of a door frame (left) or in the case of two pillars in front of a wall (right). In these cases, the overlapped texels in the blue chart do not receive any light.

and the missing vertices of the projected triangles as

$$q'_k = \frac{U(p_1)+\nabla U(p_1)(p_k-p_1)}{W(p_1)+\nabla W(p_1)(p_k-p_1)}, \qquad k=2,3, \tag{11}$$

and similarly for q''_k and q'''_k.

The multiple projection strategy is implemented with a geometry shader, which processes the triangles of M in parallel. For each triangle $[p_1,p_2,p_3]$ of M, the shader first samples I to get the chart IDs of the vertices. If the chart IDs are all identical, then we replace the coordinates p_k with q_k in (10) and handle the projected triangle $[q_1,q_2,q_3]$ as described in Section 4.1. Otherwise, we create as many projected triangles as there are different chart IDs and reconstruct the missing vertices using (11). Since these texture triangles also cover parts of the light atlas outside the corresponding chart (see Figure 4), we add a stencil test in the fragment shader which checks if the fragment is contained in the correct chart and otherwise discards it.

Although the described technique is very stable, it can lead to artefacts in certain situations. For example, even if all vertices of a triangle of M lie in the same chart, the triangle may still overlap a part of the scene which corresponds to a different chart (see Figure 5), and then this part does not receive any light. A similar situation can occur if the vertices of the triangle lie in different charts. In our experiments we observed that this happens rarely and does not lead to recognizable artefacts in the final indirect illumination solution, because of all the other VPLs that contribute light to the problematic regions. Moreover, the likelihood of such situations can be reduced by increasing the tessellation of M.

4.3 Creating Soft Shadows

Figure 7 shows an example of the overall light distribution computed with the technique described so far and illustrates that realistic shadows are achieved only if M consists of a large number of triangles. The reason for this behaviour is explained in Figure 6, where the triangle $t=[p_1,p_2,p_3]$ of M straddles a shadow edge, and p_1 lies on scene object O_1, which casts a shadow onto scene object O_2 that contains p_2 and p_3. In this situation, the technique from the previous section creates and distributes light into the two extrapolated, projected triangles $[q_1,q'_2,q'_3]$ and $[q''_1,q_2,q_3]$ in the light atlas $\mathcal{I}$, which correspond to the scene triangles $[p_1,p'_2,p'_3]$ and $[p''_1,p_2,p_3]$. The first triangle extends the part of t belonging to the shadow caster O_1, and it is clipped to the correct region by the stencil test, because the shadow edge is also a boundary edge of O_1's chart in $\mathcal{I}$. The second triangle extends the part of t belonging to the shadow receiver O_2 and ranges into the shadow of O_1, thus causing light to spread into a region, which is not supposed to be illuminated. Clearly, this effect diminishes and becomes negligible if the resolution of M is high so that the projected triangles are sufficiently small.

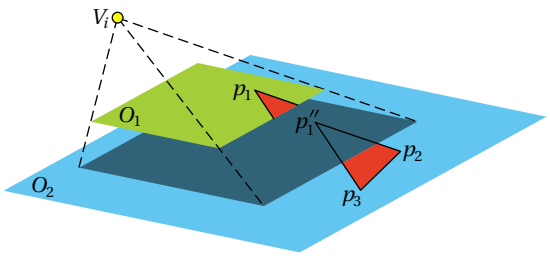


Figure 6: The red triangle of M is projected onto two different objects of the scene, and the part that lies in the blue object is extrapolated into the shadow of the green object.

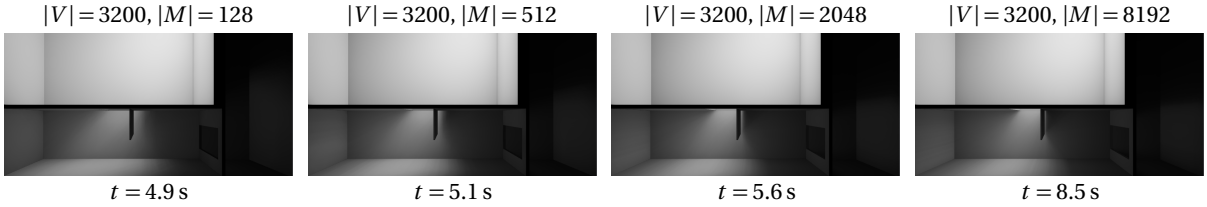


Figure 7: Light distribution using 3200 VPLs and M with 128, 512, 2048 and 8192 triangles (from left to right) for the projection step. Note that the shadow of the door appears correct only if the resolution of M is sufficiently high so that the projected triangles are small.

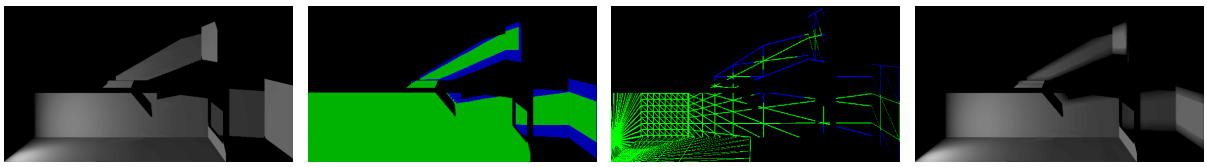


Figure 8: Light distribution for one VPL with the technique from Section 4.2 (left) and with soft shadows (right). The pictures in the centre show the projection of M with 512 triangles into the scene, where the green parts are illuminated correctly, while the blue parts are shadowed and receive light incorrectly due to the extrapolation process.

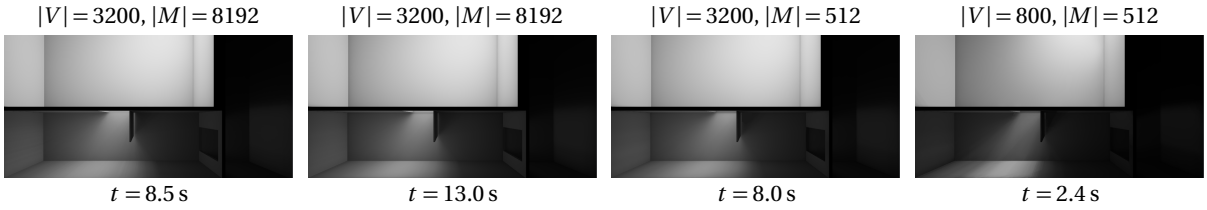


Figure 9: Without soft shadows (left), a high resolution of M is required to create the correct shadow effect of the door (cf. Figure 7). Using our soft shadows with $K = 5$ and the same resolution of M creates a computational overhead, but this can be counterbalanced by decreasing the resolution of M without sacrificing the quality of the result (centre). Moreover, with soft shadows we achieve the same quality with less VPLs, which in turn reduces the computation time (right).

However, since the resolution of M has an impact on the overall timing (see Figure 7), we propose to adapt shadow mapping to create realistic shadows even if M consists of few triangles only. To this end, remember that the fragment shader has access to the distance D between the camera and the scene point $P(u,v)$ that corresponds to the texel at (u,v). Note that for the texels in the extrapolated part of a projected triangle, this value may not be correct. For example, in Figure 6, the distance value D for the texel at q_1'' is not $\|V_i - p_1''\|$, but the smaller distance $\|V_i - p_1\|$. However, we can use this to our advantage, because it allows us to detect points that lie in the shadow of another object. All we need to do is to add another layer ∇D to the geometry buffer G, which contains the derivatives of D, so that the depth value at q_1'' can be approximated as in (11) by the geometry shader. Hence, the texel is shadowed, if the extrapolated depth value is larger than D.

To further improve the quality, we use the approach of Fernando [9] to create soft shadows by considering the neighbourhood of $P(u,v)$. This requires us to introduce the coordinates (x,y) of the mesh vertices as additional vertex attributes, so that the fragment shader has access to the interpolated coordinates $(\bar{x},\bar{y})$. We then sample D at $(\bar{x},\bar{y})$ and the neighbouring coordinates and compare the values $D(\bar{x}+\delta_x,\bar{y}+\delta_y)$ for $\delta_x,\delta_y \in \{-K,\ldots,K\}$, where K is the size of the sampling kernel, with the extrapolated depth value at $P(u,v)$, to find out what percentage of the neighbourhood of $P(u,v)$ are shadowed. The irradiance value for the current texel is then multiplied with this percentage before being drawn into the light atlas. Figure 8 illustrates this approach for one VPL, and Figure 9 shows the overall effect for the example in Figure 7.

5 Results

All results mentioned in this section feature one spot light source and were computed on an Intel i5-3350P with 8GB RAM and an NVidia GeForce GTX 680. After first demonstrating the quality of our technique (Section 5.1), we discuss the timings and their dependence on various parameters (Section 5.2), and finally show that our approach can also be used for simulating the effect of big area lights (Section 5.3).

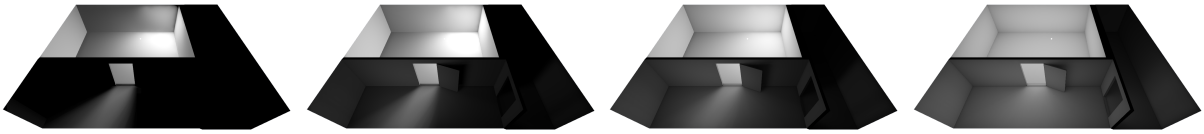


Figure 10: Indirect light distribution inside a test scene with 1, 2, 5, and 20 light bounces (from left to right) and 1000 VPLs.

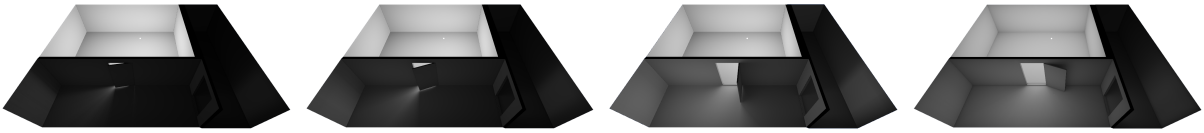
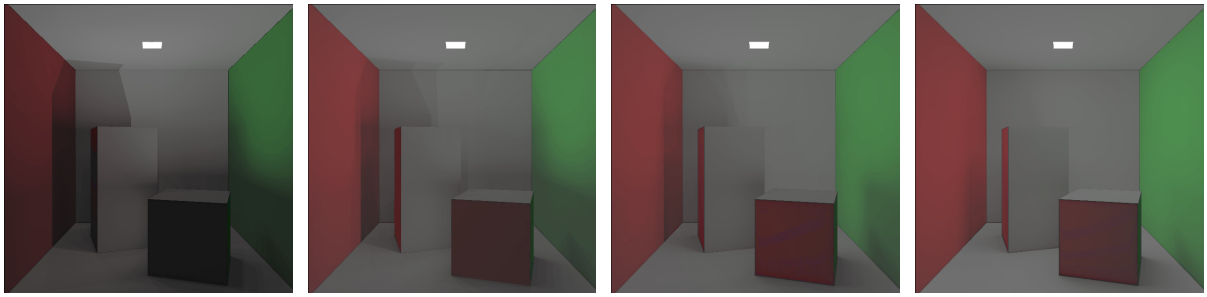


Figure 11: Indirect light distribution for the scene from Figure 10, with the door open by 10°, 20°, 90°, and 170° (from left to right).



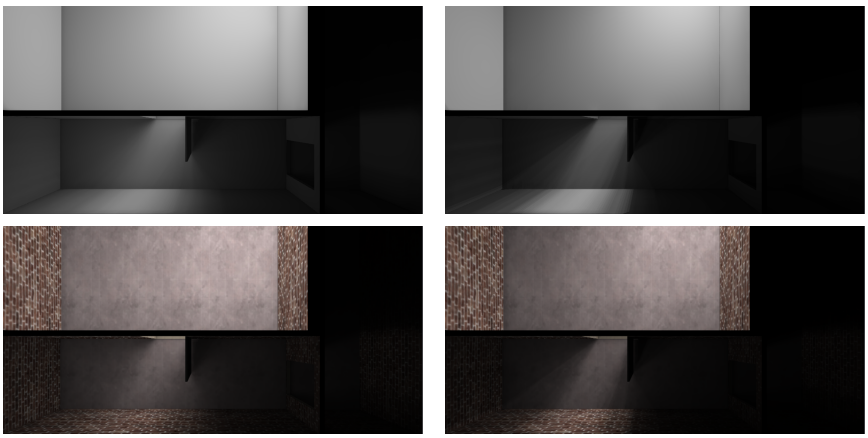
$|V| = 100$, $t = 0.4$ s $|V| = 200$, $t = 0.9$ s $|V| = 400$, $t = 1.2$ s $|V| = 1600$, $t = 4.5$ s

Figure 12: Light distribution for the Cornell box with different numbers of VPLs.

5.1 Quality

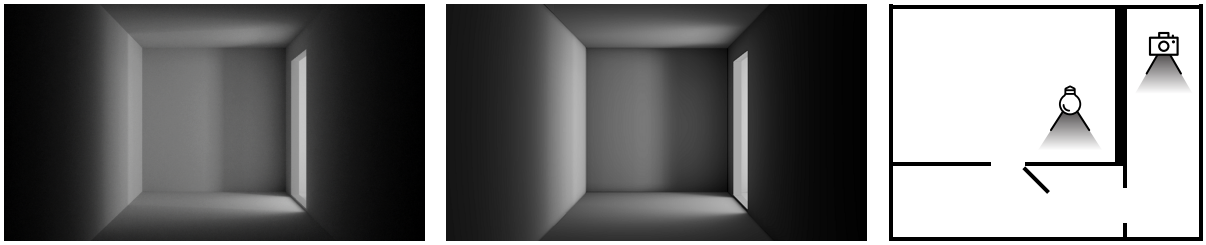
Figure 10 demonstrates that the ability of our method to distribute light with more than two bounces significantly helps to create a realistic indirect illumination effect. Our tests suggest that five light bounces are a good compromise between quality and speed, and we use this setting in all our examples, unless stated otherwise. Figure 11 shows the same test scene, but with the door open at different angles. Note how our technique captures the lighting effect at the bottom of the door and how light is propagated by multiple bounces into the corridor on the right, even for small opening angles.

While the number of light bounces increases the realism of the result, it is the number of VPLs, which impacts the visual quality of the result, as shown in Figure 12. Even though the global lighting situation and



$|V| = 1600$, $t = 4.5$ s $|V| = 400$, $t = 1.5$ s

Figure 13: Light distribution for the scene from Figure 10 with different numbers of VPLs. Using only 400 VPLs (right) creates visible artefacts in the light distribution, but they are barely noticeable once the scene is rendered with texture.



$t = 15\,\mathrm{h}$ $t = 4.5\,\mathrm{s}$ scene configuration

Figure 14: Comparison of the results using *LuxRender* (left) and our technique (middle) for the corridor scene, with a spot light source and the camera as shown in the sketch (right).

colour bleeding appear to be correct for a small number of VPLs, using too few of them can lead to sharp, unnatural shadow edges and other small artefacts. Both are correctly smoothed out for a sufficiently large number of VPLs, depending on the scene geometry. In general, the more occluders a scene contains, the more VPLs are needed to achieve good results, and the same holds for the situation where light must to be distributed through a small hole or gap (see Figure 18).

Figures 13 and 16 further show that the number of VPLs can be decreased for textured scenes, because the texture tends to hide the apparent small artefacts of the indirect light distribution.

Figure 14 shows a comparison between our technique and the "ground truth", as computed by letting *LuxRender*'s path tracer run for 15 hours. The results are very similar, despite the fact that our approximation to the ground truth was computed in 4.5 seconds only. Note that in this example the light reaches the shown part of the scene only after three light bounces. Therefore, other methods with comparable timings, for example [6, 4, 22], would generate an entirely black image.

5.2 Timings

The overall runtime of our algorithm depends on a number of parameters: the number of light bounces and VPLs, the resolutions of the geometry buffer G and the 2D triangle mesh M, and the scene complexity. The resolution of the light atlas $\mathcal{I}$ does not influence the performance, because we draw only those texels of $\mathcal{I}$ which really receive light and this procedure is not fragment-bound.

The number of light bounces is relevant only for the first step of the algorithm and affects the runtime linearly. The resolutions of G and M are relevant only for the second step of the algorithm, and while the timings are almost independent of the resolution of G, they depend linearly on the resolution of M (see Figure 7). In addition, both steps depend on the number of VPLs, and while the runtime of the first step scales quadratically in this parameter, the second step depends only linearly on the VPL count.

Figure 9 illustrates that our texel-accurate soft shadow technique effectually reduces the overall rendering time, because it leads to convincing results even for small numbers of VPLs and triangle meshes M with low resolution. In order for the soft shadow technique to be effective, the resolution of G needs to be higher than the resolution of M. Our tests suggest that setting the resolution of M to $16 \times 16 \times 2 = 512$ triangles, the resolution of G to 256×256, and the kernel size K to 5 is a good compromise between quality and speed, and we used these settings in all examples, unless stated otherwise.

Table 1 shows that the average cost of the first step of our algorithm further depends on the complexity of the scene, because the visibility test requires to render the whole scene into a depth buffer for each VPL. The

	timings per VPL [ms]				timings for different numbers of VPLs [s]				
scene	fill D	distribution	fill G	draw $\mathcal{I}$	100	400	1600	6400	25600
Corridor	0.00141	0.000115	0.28	5.05	0.705	**1.546**	4.539	16.986	159.631
Cornell box	0.00139	0.000119	0.24	5.02	0.495	**1.218**	4.571	19.512	128.697
Sponza	0.00258	0.000116	2.10	8.04	—	—	7.732	**29.159**	228.252

Table 1: Timings for our test scenes. The left half of the table shows the average runtime per VPL for filling the depth buffer and distributing the light from one VPL to another (first step), as well as filling the geometry buffer and drawing into the light atlas (second step). The right half of the table shows the overall runtime of our algorithm, where the timing corresponding to the smallest number of VPLs that gives realistic results is marked in boldface.

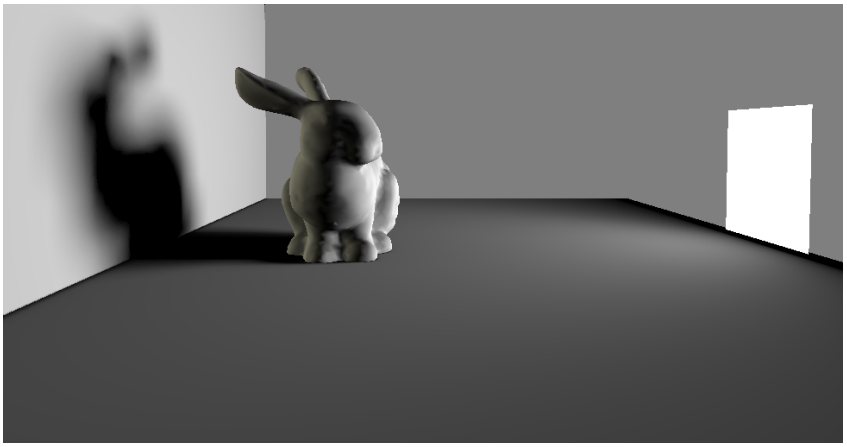


Figure 15: Area light effect, created with our technique using 100 VPLs in 0.6 seconds.

cost for the latter ranges from 0.0014 ms per VPL and per light bounce for the corridor scene in Figures 10 and 11 and the Cornell box in Figure 12 to 0.0026 ms for the Sponza scene in Figure 16. Instead, the time for distributing light from one VPL to another is constant for all scenes. However, while all other parts affect the overall runtime linearly, this part has a quadratic influence on the total computational cost. Filling the geometry buffer G in the second step of our algorithm also depends on the scene geometry. Fortunately, filling G is not very costly, and so this does not affect the overall runtime significantly. The latter is mainly determined by drawing the indirect illumination into the light atlas $\mathcal{I}$, and this part is independent of the scene complexity. An additional overhead is noticeable in both steps for the Sponza scene, because this scene comes with one light atlas for each of the three floors of the scene. Overall, the runtime per VPL for the Sponza scene is about twice as high, compared to the corridor scene and the Cornell box, and about 16 times as many VPLs are needed to achieve realistic results, because of the complexity of this scene.

5.3 Area Lights

Our technique for rendering VPLs into the light atlas can also be used to create realistic area light effects, as shown in Figure 15. We first distribute n VPLs uniformly over the surface of an area light with power E. Then we assign to each VPL the power E/n and fill the light atlas as explained in Section 4. In contrast to simple soft shadow algorithms, this method also correctly handles the shape of the area light, as well as the distances between the area light, the shadow caster, and the shadow receiver.

6 Conclusion

Apart from the fact that our indirect illumination method handles multiple bounces and efficiently creates realistic results, it is worth noticing that it is particularly well-suited for static scenes. In this situation, the light atlas is computed in a preprocessing step and can then be used to navigate the scene in real time.

Furthermore, our technique is fast enough to give an artist the possibility to create the desired overall lighting effect by interactively changing the numbers and positions of the light sources. In this setting, interactivity can be achieved by reducing the number of VPLs, and even though the rendering result with too few VPLs may exhibit artefacts, it still gives the user a realistic preview of how the result will look when a greater number of VPLs is used. It should even be possible to compute and display the result progressively with improving quality by increasing the number of VPLs on the fly, but it remains future work to further explore this idea.

The closest competitor for our algorithm is the method by Luksch et al. [17], which also computes realistic results at interactive rates. However, the main difference is the creation of the light atlas, which is more efficient in our technique, because we only address those texels in the light atlas which receive light, instead of looping over all texels for each VPL. In fact, while they report that the runtime depends linearly on the resolution of the light atlas, we did not experience any practical influence of that resolution on the overall performance, even in the case of the Sponza scene, which uses three light atlases, each with a resolution of 8000×8000 texels.

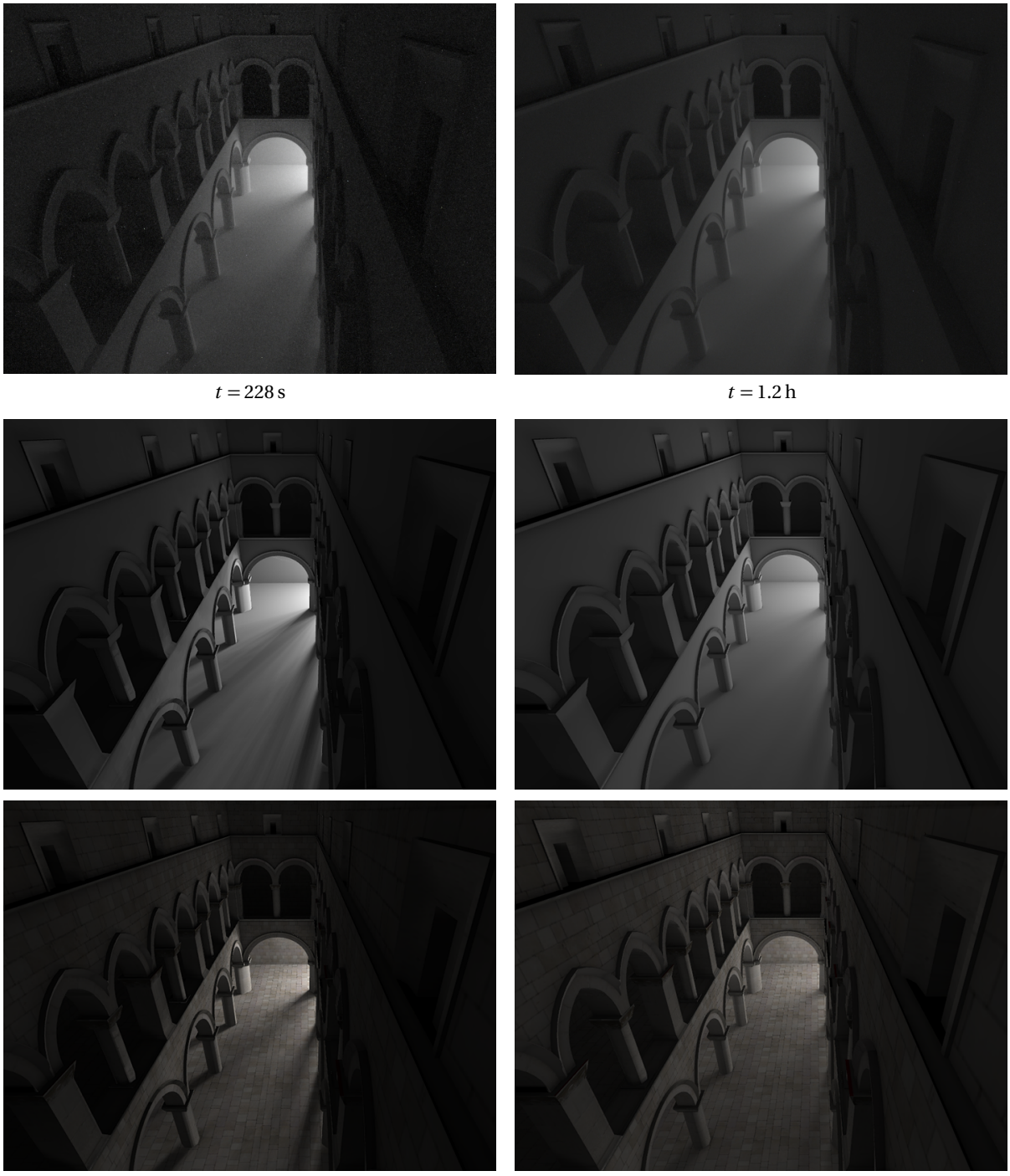


Figure 16: The top row shows the Sponza scene rendered by *LuxRender* in about 4 minutes (left) and more than one hour (right). The images below show the results of our technique with different numbers of VPLs for comparison, and the last row shows the same results with textures.

Conceptually, our technique is somewhat similar to photon mapping. In the first part, the light is distributed with multiple bounces, but instead of shooting it randomly in any possible direction, the light exchange is restricted to the fixed VPL positions. In the second part, this discrete light distribution is extrapolated to the whole scene, but while a photon mapper requires a rather expensive averaging step, our projection strategy automatically generates a smooth light distribution in the light atlas. Consequently, our approach is more efficient and can illuminate the Sponza scene with high quality in less than five minutes, whereas *LuxRender*,

Figure 17: Indirect illumination of the Sponza scene as computed by our technique with 6400 VPLs and three light bounces in 29 seconds.

a state-of-the-art path-tracer, produces insufficient quality in the same time and takes more than an hour to deliver a result with similar quality (see Figure 16). Notice that all phenomenological effects of the ground truth are present in our result. Figure 17 shows another example of the Sponza scene, which was computed in less than a minute and still captures the main characteristic indirect illumination effects.

6.1 Design Choices

In principle, it would be possible to speed up our approach by clustering the VPLs [5], so that the light is distributed from a much smaller number of sources. This would reduce the number of required rendering passes, but the small number of clusters would create artefacts similar to those in Figure 18, even if soft shadow techniques are used to smooth out the shadow edges.

Another option for accelerating the first step of our method would be to precompute and store the form factors between VPLs with the visibility factor included. However, this would restrict the technique to work with static scenes only, whereas the current approach is flexible enough to also handle moving objects, which is one of our goals for future work.

We further chose to precompute and distribute the VPLs uniformly instead of creating randomly distributed VPLs on the fly [8], because we observed that the uniform distribution leads to smoother shadows and a higher overall rendering quality.

Finally, we decided to store irradiance instead of radiosity in the light atlas, because we assume that the multiplication with the diffuse reflection coefficient is computed when the scene is rendered. In particular, this is the correct approach for textured scenes, where the texture itself provides a pixel-accurate description of the reflection coefficient.

6.2 Future Work

Apart from the progressive computation mentioned above, another interesting line of future work is dealing with dynamic lighting situations. We believe that this can be done quite efficiently with our framework as follows. Whenever the lighting of the scene changes, we first redistribute the light among the VPLs as explained in Section 3. In the second step, we then update the light atlas by shooting and accumulating the differences between the old and the new irradiance values at the VPLs, but only for those VPLs for which this difference is above some threshold. Since this step dominates the overall runtime and we expect the number of VPLs that need to be considered to be small, this should take only a fraction of the time required for recomputing the light atlas from scratch.

Furthermore it would be interesting to explore the possibility of distributing the VPLs not uniformly but adaptively in the scene. For example, the reason for the shadow artefacts in Figure 18 is that only a small fraction of the uniformly distributed 400 VPLs is involved in distributing light through the door, especially when it is open by only 10°. Detecting such situations and adaptively placing more VPLs in areas where they are needed to create smooth shadows will probably reduce the overall number of VPLs significantly, thus resulting in more favourable rendering times.

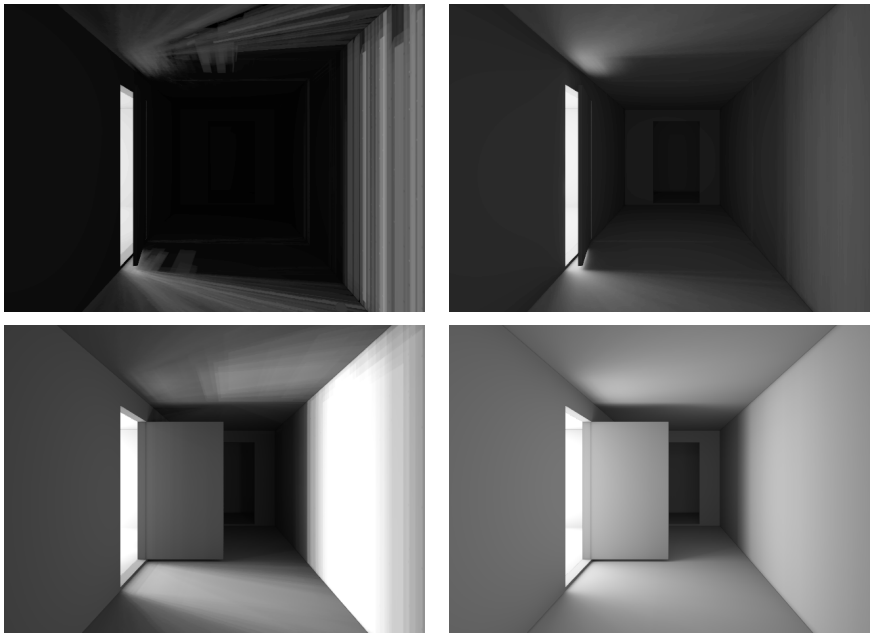


Figure 18: Using only 400 VPLs for the scene from Figure 11 with the door open by 10° (top) and 90° (bottom) leads to shadow artefacts (left), and it requires 10000 or 1600 VPLs, respectively, to get visually smooth results (right).

Our technique could also be extended to render caustics and handle specular surfaces by combining it with the idea of *caustic triangles* [27]. The computation would then be done just once for the refractors and the caustics would be stored directly in the light atlas for rendering.

Acknowledgements

We would like to thank the reviewers and in particular the associate editor, László Szirmay-Kalos, for their valuable comments which significantly helped to improve this paper.

References

[1] M. F. Cohen, S. E. Chen, J. R. Wallace, and D. P. Greenberg. A progressive refinement approach to fast radiosity image generation. *ACM SIGGRAPH Computer Graphics*, 22(4):75–84, Aug. 1988. Proceedings of SIGGRAPH.

[2] M. F. Cohen and D. P. Greenberg. The hemi-cube: a radiosity solution for complex environments. *ACM SIGGRAPH Computer Graphics*, 19(3):31–40, July 1985. Proceedings of SIGGRAPH.

[3] G. Coombe, M. J. Harris, and A. Lastra. Radiosity on graphics hardware. In *Proceedings of Graphics Interface*, pages 161–168, London, ON, May 2004.

[4] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum*, 30(7):1921–1930, Sept. 2011. Proceedings of Pacific Graphics.

[5] C. Dachsbacher, J. Křivánek, M. Hašan, A. Arbree, B. Walter, and J. Novák. Scalable realistic rendering with many-light methods. *Computer Graphics Forum*, 33(1):88–104, Feb. 2014.

[6] C. Dachsbacher and M. Stamminger. Reflective shadow maps. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, pages 203–213, Washington, D.C., Apr. 2005.

[7] H. Dammertz, A. Keller, and H. P. A. Lensch. Progressive point-light-based global illumination. *Computer Graphics Forum*, 29(8):2504–2515, Dec. 2010.

[8] Z. Dong, T. Grosch, T. Ritschel, J. Kautz, and H.-P. Seidel. Real-time indirect illumination with clustered visibility. In *Proceedings of Vision, Modeling, and Visualization*, pages 187–196, Braunschweig, Germany, Nov. 2009.

[9] R. Fernando. Percentage-closer soft shadows. In *Proceedings of SIGGRAPH*, Sketches, page #35:1, Los Angeles, CA, July 2005.

[10] T. Hachisuka. High-quality global illumination rendering using rasterization. In M. Pharr, editor, *GPU Gems 2*, chapter 38, pages 615–633. Addison-Wesley, 2005.

[11] J. Hermes, N. Henrich, T. Grosch, and S. Müller. Global illumination using parallel global ray-bundles. In *Proceedings of Vision, Modeling, and Visualization*, pages 65–72, Siegen, Germany, Nov. 2010.

[12] H. W. Jensen. Global illumination using photon maps. In X. Pueyo and P. Schröder, editors, *Rendering Techniques '96*, pages 21–30. Springer, 1996.

[13] H. W. Jensen and N. J. Christensen. Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers & Graphics*, 19(2):215–224, Mar.–Apr. 1995.

[14] A. W. Kristensen, T. Akenine-Möller, and H. W. Jensen. Precomputed local radiance transfer for real-time lighting design. *ACM Transactions on Graphics*, 24(3):1208–1215, July 2005. Proceedings of SIGGRAPH.

[15] J. Lehtinen, M. Zwicker, E. Turquin, J. Kontkanen, F. Durand, F. X. Sillion, and T. Aila. A meshless hierarchical representation for light transport. *ACM Transactions on Graphics*, 27:Article 37, 9 pages, Aug. 2008. Proceedings of SIGGRAPH.

[16] P. Lensing and W. Broll. Efficient shading of indirect illumination applying reflective shadow maps. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, pages 95–102, Orlando, FL, Mar. 2013.

[17] C. Luksch, R. F. Tobler, R. Habel, M. Schwärzler, and M. Wimmer. Fast light-map computation with virtual polygon lights. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, pages 87–94, Orlando, FL, Mar. 2013.

[18] M. Mara, D. Luebke, and M. McGuire. Toward practical real-time photon mapping: Efficient GPU density estimation. In *Proceedings of the ACM Symposium on Interactive 3D Graphics and Games*, pages 71–78, Orlando, FL, Mar. 2013.

[19] M. McGuire and D. Luebke. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of High Performance Graphics*, pages 77–89, New Orleans, LA, Aug. 2009.

[20] M. Mittring. Finding next gen: CryEngine 2. In *Advanced Real-Time Rendering in 3D Graphics and Games*, number 28 in SIGGRAPH Course Notes, pages 97–121. ACM Press, Aug. 2007.

[21] R. Prutkin, A. Kaplanyan, and C. Dachsbacher. Reflective shadow map clustering for real-time global illumination. In *Proceedings of Eurographics*, Short Papers, pages 9–12, Cagliari, Italy, May 2012.

[22] T. Ritschel, T. Grosch, J. Kautz, and H.-P. Seidel. Interactive global illumination based on coherent surface shadow maps. In *Proceedings of Graphics Interface*, pages 185–192, Windsor, ON, May 2008.

[23] J. R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and elaunay triangulator. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer, 1996.

[24] F. X. Sillion and C. Puech. *Radiosity and Global Illumination.* The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann Publishers, 1994.

[25] P.-P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics*, 21:527–536, July 2002. Proceedings of SIGGRAPH.

[26] L. Szécsi, L. Szirmay-Kalos, and M. Sbert. Light animation with precomputed light paths on the GPU. In *Proceedings of Graphics Interface*, pages 187–194, Quebec City, QC, June 2006.

[27] T. Umenhoffer, G. Patow, and L. Szirmay-Kalos. Caustic triangles on the GPU. In *Proceedings of Computer Graphics International*, pages 222–227, Istanbul, Turkey, June 2008.

[28] T. Umenhoffer, B. Tóth, and L. Szirmay-Kalos. Efficient methods for ambient lighting. In *Proceedings of the 25th Spring Conference on Computer Graphics*, pages 87–94, Budmerice, Slovakia, Apr. 2009.