Pacific Graphics 2008
T. Igarashi, N. Max, and F. Sillion
(Guest Editors)

Volume 27 (2008), Number 7

Interactive Global Illumination for Deformable Geometry in CUDA

Arne Schmitz, Markus Tavenrath and Leif Kobbelt

Computer Graphics Group, RWTH Aachen University

Abstract

Interactive global illumination for fully deformable scenes with dynamic relighting is currently a very elusive goal in the area of realistic rendering. In this work we propose a highly efficient and scalable system that is based on explicit visibility calculations. The rendering equation defines the light exchange between surfaces, which we approximate by subsampling. By utilizing the power of modern parallel GPUs using the CUDA framework we achieve interactive frame rates. Since we update the global illumination continuously in an asynchronous fashion, we maintain interactivity at all times for moderately complex scenes. We show that we can achieve higher frame rates for scenes with moving light sources, diffuse indirect illumination and dynamic geometry than other current methods, while maintaining a high image quality.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism: Radiosity

Figure 1: *Four of our test scenes. In clockwise order (with vertex count, overall FPS, GI updates):* TORSO *(7.1k, 70 fps, 0.4s),* TEAROOM *(11.8k, 75 fps, 0.5s),* CONFERENCE *(123k, 11 fps, 14.7s),* IPHI *(60k, 12 fps, 7.9s).*

1. Introduction

To have interactive, full global illumination in dynamic or deformable scenarios is considered a great achievement in rendering. In the last years there were several approaches that solve at least a part of this problem.

Our contribution in this work is that we show that with today's highly parallel multi-core CPUs and GPUs, it is possible to implement an interactive system for diffuse global illumination solutions, using simple and efficient algorithms. We have implemented our work on the NVIDIA CUDA platform, due to the easy development on the platform. Our system allows for rapid light exchange, using simple and very efficient data structures that fully exploit the power of the CUDA platform. Also we have achieved to maintain interactivity in deformable scenes, with moving, non-rigid objects and with moving light sources. This enables our system to be used both in interactive relighting applications as also in future game engines. Although we use the CUDA platform, our system is general enough to be easily ported to other parallel compute platforms such as the Cell architecture, as well as other upcoming systems. We concentrated on CUDA, since it is already available and offers the highest degree of parallelization available.

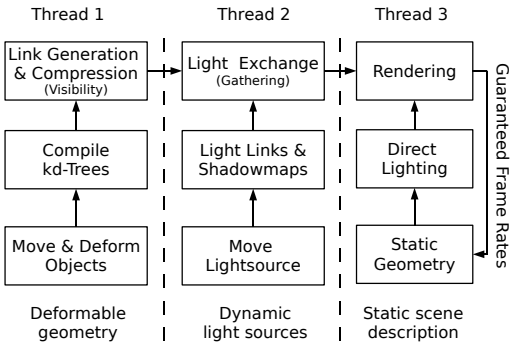

Figure 1: *The system is split into three parts: Handling of dynamic objects, handling of dynamic light sources, and rendering. Dynamic objects and light sources are handled asynchronously, to allow for high rendering performance.*

Our algorithm is based on explicit visibility computation. For typical medium-scale scenes we can achieve a rendering performance that is up to 2× higher than other current methods using implicit visibility. Also we can render more complex scenes than those methods. We also show that the performance scales with the number of GPUs in the computer system. If the compute power of the GPUs is doubled, the turnaround time for a full global illumination solution is approximately halved. For interactive lighting with static geometry we are about one order of magnitude faster than a comparable solution using implicit visibility.

1.1. Related Work

There are numerous works published in the field of Global Illumination. We will concentrate here on related work that is dealing with the real time aspect of it, and also on work that supports dynamic geometry.

When looking at real time ray-tracing, one can find vast amounts of works that perform Whitted-style ray-tracing and ray-casting. Notable here are the works of Popov, Shevtsov, and Horn [PGSS07, SSK07, HSHH07] where the main focus is on efficient *k*d-tree building and traversal. Furthermore Wald [WH] showed how to build *k*d-trees efficiently. In a similar work, Wächter et al. [WK06] showed how to use a variation of the *k*d-tree to make building and traversal more efficient. However, all those works and related ones only do simple ray-tracing without diffuse inter-reflections.

Quite recently Dachsbacher et al. [DSDD07] presented an approach that computes an approximate solution by transmitting negative radiance through discrete bins of finite size. The underlying assumption is that radiance is invariant along rays, which is not the case for finite bins, so that some bias is introduced. Similar to that is Dong et al. [DKTS07], where a hierarchy is constructed that determines visibility implicitly. Both approaches allow diffuse reflections and indirect light bounces and can handle dynamic scenes. Scenes with dynamic elements have also been covered in other works, for example Wald et al. [WBS07] use bounding volume hierarchies, since they can be updated quite fast.

An approach similar to ours is Havran et al. [HBHS05], in which they store photon paths and evaluate them when needed. We extend this by storing not only paths but a generalized link structure, that is evaluated similar to traditional radiosity methods, but instead we propagate radiance. Other methods also use a discrete ray sampling of the hemisphere on a surface point to evaluate the radiance [CPC84, WRC]. There exist modernized versions of those algorithms, like the work by Gautron et al. [GKBP05], which map onto current GPU hardware. An approach by Heidrich et al. [HDKS00] uses precomputed visibility, which is related to our approach. Their work only handles local visibility of static geometry, but allows for indirect illumination.

Kristensen et al. developed a competing approach for interactive lighting [KAMJ05], which requires precomputations on a computer cluster but renders very fast. Another interactive lighting approach is by Yue et al. [YIDN07], which also requires pre-computations. Other methods related to this are PRT-based methods which use spherical harmonics or wavelets to store radiance [KTHS06, SLS05]. However dynamic scenes cannot be handled in an easy way and the pre-computations take some time.

2. System Overview

For interactive global illumination systems, it is important to identify the main bottlenecks which are most time consuming. With global illumination using explicit visibility we have several compute-intensive parts.

First, we have to build a spatial search structure. In our case this is a *k*d-tree, for accelerating the visibility tests. Building a *k*d-tree is computationally expensive, the lower bound being $\Omega(n \log n)$. Similar costs occur for determining the visibility in the scene. This is done by shooting rays, each of which has $O(\log n)$ complexity. If mutual visibility of all elements is to be computed, $O(n^2 \log n)$ time is needed. Since this is prohibitive for large scenes, we use an undersampling to reduce the complexity. This results in a link-structure which determines the possible paths the light can take in the scene.

The second most expensive step in the rendering pipeline is computing the light exchange. Light is propagated through the scene by gathering on the link-structure. Also, for interactive light design we support static geometry and moving light sources. This allows us to update only the links from the light source to the static scene geometry. Furthermore, GPUs can efficiently compute the direct illumination, which can be used to lessen the workload on other parts of the pipeline.

The rendering is the last step, and, although it is also computationally expensive, can be done efficiently on consumer graphics hardware. So this is not a great concern for implementing an efficient global illumination pipeline.

Our proposed system uses this information to provide three distinct algorithmic parts to maintain a guaranteed frame rate. This allows for maximum flexibility in the user's interaction with the system. Figure 1 shows the structure of our proposed rendering pipeline. It achieves interactive, guaranteed frame rates by splitting the workload into three parts for deformable geometry, interactive relighting, and rendering of the scene using OpenGL.

The stages presented in Figure 1 run partially on the CPU and partially on the GPU. The first stage for dynamic geometry, including kd-tree compilation and link compression runs on the CPU, while the visibility tests run on the GPU. In the second stage, light link sampling and the light exchange run on the GPU. The light exchange only uses one GPU at the moment, since it is the fastest part of the algorithm and synchronization of multiple GPUs is less efficient. The third stage consists of rendering the direct light on the GPU. We manage to get both the CPU and the GPUs to be utilized evenly, with the main parts being the kd-tree compilation on the CPU and the visibility computation on the GPU.

2.1. Architecture

Our work uses the NVIDIA CUDA computing platform [NVI07] for link generation and light exchange, while rendering is performed using OpenGL. The system is generalized in such a way that it is easily portable to other compute platforms. The CUDA system allows for a much simpler development process, compared to traditional shading languages like GLSL or HLSL, since the code is mostly standard C and allows easy debugging.

Most parts of the algorithm run asynchronously to allow for interactive frame rates. Updates to the scene's illumination are computed in the background. In section 4.1 we will discuss the scalability of our approach. This is an important aspect of our system, since we can easily take advantage of more computing power, in the form of more CPU cores or GPUs. With the current trend of more cores on the chip our system will scale well with future hardware generations.

2.2. Explicit Visibility

We chose to use explicit visibility, in contrast to some recent approaches. The alternative to use implicit visibility as described in Dachsbacher et al. [DSDD07] would allow us to omit the *k*d-tree and the expensive ray-queries. However the convergence of these algorithms depend on the depth complexity of the scene. That is as many iterations are needed to converge to a correct solution as there are layers of occluding objects. Also the link structure of implicit visibility algorithms can consume quite a lot of memory.

3. Algorithm

This section explains the different stages of our algorithm. The main steps being *k*d-tree construction, link generation, initial radiance updates and rendering.

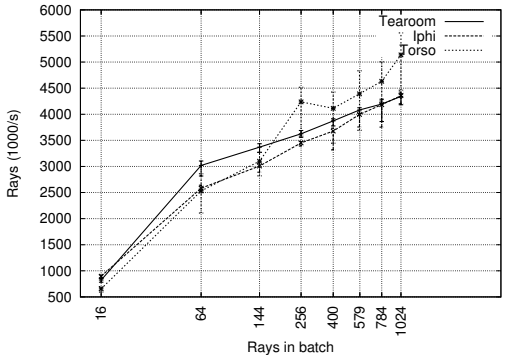

Figure 2: *This graph shows how the number of rays shot at each vertex influences the number of rays shot per second for different scenes. The more rays are shot per vertex, the more coherent those rays are, and the higher the throughput.*

3.1. *k*d-Tree Construction

For dynamic scenes, a fast construction of the *k*d-tree is important. At the same time the ray-traversal during link construction needs to be fast. Because of this, we use the surface area heuristic (SAH) during tree construction [MB90]. Since the SAH is costly, we utilize initial clustering in the same way as Shevtsov et al. [SSK07]. In the first $\log_2 N_C$ levels of the *k*d-tree we use an object median split along the longest axis of the current node's bounding box, where N_C denotes the number of processing cores of the host CPU. After these first levels, we have N_C leaf nodes. After that we do parallel subtree construction using the SAH heuristic. It should be noted that quite recently Zhuo et al. [ZHWG08] presented a method to build the *k*d-tree efficiently on the GPU, which could probably make our system even faster.

After the tree has been constructed, we use the optimization algorithm of Popov et al. [PGSS07], to generate ropes at the leafs of the tree, to accelerate traversal.

3.2. Ray Shooting and Link Generation

Using the *k*d-tree created in this way we achieve between 0.5 and 5 million ray-traversals per second for highly incoherent secondary rays. The worst case happens in badly tesselated scenes with triangles of greatly varying size, while uniformly tesselated shapes tend to produce better trees.

The basis of our algorithm is the surface parameterization of the rendering equation [DBB06]:

$$L(x \leftarrow \Theta) = L_e(x \leftarrow \Theta) + \int_A f_r(y, \psi \leftrightarrow \overrightarrow{yz}) \, L(y \leftarrow \overrightarrow{yz}) V(y,z) G(y,z) dA_z \quad (1)$$

With $y = r(x, \Theta)$ and V being the visibility term and G the geometry term. The goal of the ray shooting step is to create a radiance link structure. Every link is defined between vertices and weighted with the visibility and the geometry term

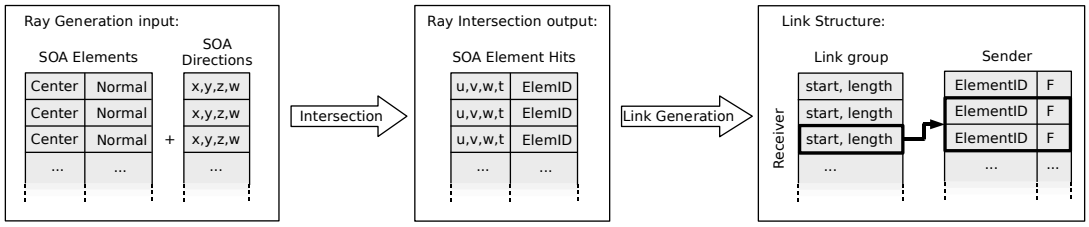

Figure 3: *This figure depicts the ray generation pipeline. Elements consisting of a vertex and a normal form the input. We provide a discrete set of directions that are to be sampled (left). After the ray intersection from each vertex with each direction, we get an array of element hits (middle) that is transformed into the final link structure (right). Every element now has a link group of variable length associated, which points into the large link array.*

respectively. By repeatedly gathering radiance on the link structure, we achieve the evaluation of equation (1) by iteration. The initial values are computed by explicit lighting, as described in section 3.3 and the gathering is described in section 3.3. After gathering, the radiance values are interpolated by the OpenGL shader during the rendering step.

There are two ways to shoot the rays. One is to traverse them synchronously, in step with the rendering pipeline. This leads to lower frame rates, since the ray shooting is very expensive. Therefore we perform asynchronous ray shooting, which results in an interactive rendering of the scene, while a full global illumination solution is computed at a lower frame rate and handed to the OpenGL renderer when finished.

Ray Generation The ray shooting is divided into batches that are sent to all available CUDA devices. The CPU is used for creating these batches and later evaluating them and for computing the final vertex colors. Every batch consists of a number of elements (i.e. vertices), their local coordinate basis, described by their normal vector, and a set C_Ω of ray directions to be sampled. In most examples we choose $n_C = 256$ distinct ray directions, covering the hemisphere in a cosine-weighted fashion. The directions have to be transferred only once, since they are transformed into the local basis of every element when needed.

Every processed batch returns an array of n_C hit-structures that contain the element id that was hit, and the barycentric coordinates of the hit, as also the distance t on the ray. From these hit structures, we generate the final link structure. Every element is associated with a variably sized link group, that points into the link array. See Figure 3 for a detailed visualization of the data structures used.

We are computing radiance on vertices by building vertex-to-vertex-links. But in the shooting stage we intersect with triangles. We need to distribute the information from the shooting stage to the vertices of the mesh. Consider that we have hit the point $x = uA + vB + wC$ on a triangle. We generate three links, weighted with their barycentric coordinates. This weighting is necessary to satisfy the energy balance, since we sample only n_C directions on the hemisphere.

The hits found in this process are used to build the link structure. However this produces a large number of redundant links, since many rays hit the same triangle and therefore links to certain vertices are very redundant. Hence we combine sets of links where both source and destination vertex are the same, which results in compression ratios of about 3 to 5 times. This is done by a straightforward algorithm on the CPU after the shooting stage. First we iterate over all links of a vertex and accumulate the geometry terms G for specific peer vertices in an array. In a second iteration we loop over all vertices found in the first iteration and generate new links with the accumulated weighted geometry terms.

A great advantage of our system is, that the total number of links scales at most linearly with the number of elements used, since every element only has a constant number of links. The link compression is computationally rather inexpensive, so that it pays off in the light exchange step and allows for high update rates for moving light sources.

There are some notable observations when implementing this on current CUDA hardware. First of all memory access is very expensive on recent graphics devices, since GPUs have only very small caches for the individual threads in the multiprocessor. Also just read-only memory is cached. For example the G80 series of GPUs provides 8192 bytes of cache for textures, which is very small. Since we store our tree in texture memory, the rays that traverse the tree should be as coherent as possible, to maximize the cache hit rate. One limitation of the CUDA architecture is that the dimensions of a texture are limited. Therefore we store our linear data structures in 2D textures.

For maximizing the coherent number of rays, we use a concentric map that projects samples from a square to a disc. See Shirley et al. [SC97] for details on this map. We sample the square in blocks of size 4×4, use the map to get samples on a disc and then project the samples onto the hemisphere above the element. Also note that for maximum efficiency you should take two such 4×4 blocks, resulting in 32 relatively coherent rays. This maps best to the CUDA architecture where 32 threads are always running in lock-step in a so

called warp. In every warp, threads have to follow the same code-path. Diverging code paths have an impact on runtime.

Another implication of this architecture is that the more rays are shot in the hemisphere, the higher the ray-throughput gets. This is because the 4×4 ray blocks get more local and more coherent. That is also why packetizing the rays in a classic sense does not scale well on GPUs. Coherence of the four border rays of a packet will always be lower than when traversing all the rays explicitly. Also splitting packets is very expensive on GPUs, because of the lock-stepping of the warps. We noticed that on the G80 architecture, shooting 16×16 rays per vertex with 4×4 blocks was a good compromise between total run-time, number of rays shot per second and total number of links generated. Figure 2 shows this dependency of throughput to number of rays per vertex.

Scheduling and Balancing Building the *k*d-tree and link generation is run asynchronously. Depending on the scene complexity we achieve one to five complete global illumination updates per second, for medium complex scenes. Highly complex scenes, like the CONFERENCE scene with more than 100*k* elements, can be handled with updates to the lighting situation around every 0.4 seconds, and full updates including dynamic geometry every 12 seconds.

To allow for better performance, we use a load balancing on the CUDA devices. In a system with two or more CUDA devices, the individual ray batches take different amount of time for traversal. Our load balancing scheduler distributes the ray-batches in such a way, that all installed CUDA devices take the same time to process the batches, based on their performance in the last batch.

3.3. Updating the Light Source

Although the latency for a complete global illumination update for the whole scene is already quite low in our system, we achieve very high frame rates for moving light sources. Similar to Kristensen et al. [KAMJ05] we can interactively move light sources, but instead of hours to weeks of precomputation we can handle many scenes in fractions of a second, if the geometry of the scene changes. Implicit visibility methods like Dachsbacher et al. [DSDD07] also allow for interactive light source manipulation, but we still have a faster turnaround time, if the whole scene changes, plus we achieve higher frame rates when only the light source is moving, see Section 5 for more information.

Lightlink Sampling We support up to eight fully dynamic directional or point light sources, which use cube shadow maps with adaptive percentage closer filtering for smooth edges. Also the light sources support full high dynamic range RGB color, for enhanced flexibility during lighting design. Using multi-pass rendering it would be possible to support arbitrary numbers of light-sources, albeit with loss of performance. Area lights are also easy to implement in our system, one just has to change the ray origins of the light samples to cover the surface of the light source.

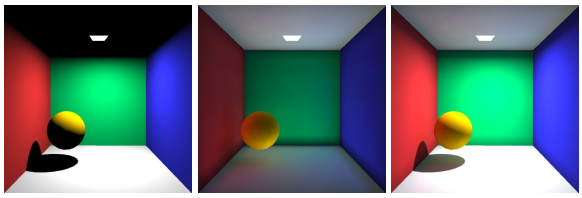

Figure 4: *We compute the direct lighting via a simple OpenGL shader (left), the indirect lighting via our light exchange (middle), and the combined image is a sum of the two first images, computed in the shader (right).*

Whenever a light source is moved, we establish explicit visibility of the light source with all elements in the scene. This is done by shooting rays from the light source to the element. This allows us to compute the geometry and visibility term between both, and we then initialize the radiance at each vertex. The geometry term is defined as:

$$G(x,y) = \frac{\cos\theta_x \cos\theta_y}{r_{xy}^2} \tag{2}$$

Since we only sample the vertices of the mesh, we are prone to aliasing and undersampling, especially in respect to the light sources. Therefore we do an oversampling of the visibility term, by also shooting rays from the lightsource to selected points on the triangles in the one-ring around the target vertex. We use a 1:4 subdivision scheme to determine the additional sampling points.

After determining the geometry term from each light source to all elements in the scene, we initialize the elements' outgoing radiance with the light they received from the light sources, multiplied by their BRDF.

Radiance Propagation by Gathering After each element has received its initial outgoing radiance, we start to propagate the radiance through the scene by means of gathering. We just iterate over all elements, and collect the radiance from all its links. The radiance is summed to the total incoming radiance, weighted by the geometry and visibility term:

$$I_{total,x,i+1} = I_{total,x,i} + \sum_y I_{outgoing,y} \cdot V(x,y) G(x,y) \tag{3}$$

The new outgoing radiance of an element is simply the incoming radiance from all incoming links $x \leftarrow y$ multiplied by the diffuse reflectance f_d:

$$I_{outgoing,x} = \sum_{\{y|x \leftarrow y\}} I_{outgoing,y} \cdot f_d \tag{4}$$

One such summing over all elements results in one light propagation bounce. In most scenes, four iterations result in a sufficient light distribution. Thus the radiance gathering itself is one of the cheapest parts of our algorithm.

Since we use a subsampling of the path space, we are

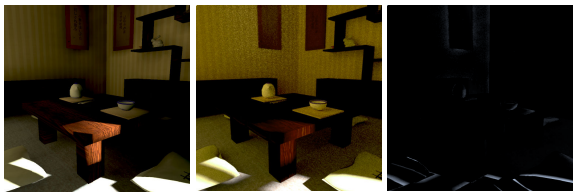

Figure 5: *Left: our method, Middle: Luxrender unbiased renderer. Right: absolute difference of radiance values. Differences are at shadow boundaries. Also under the tea bowl some light paths were missed by our algorithm.*

prone to the same noise artifacts as similar algorithms, like distributed ray-tracing by Cook et al. [CPC84]. However, in our case we can easily perform a radiance smoothing step on the vertices:

$$I_{smooth,v} = \frac{1}{|N(v)|+1}\left(I_v + \sum_{v_i \in N(v)} I_{v_i}\right) \tag{5}$$

Where $N(v)$ is the one-ring of vertex v. This eliminates practically all noise that stems from the path space subsampling. One has to be careful not to smooth over feature edges, since there the radiance will probably be non-continuous. So instead of smoothing over $N(v)$, we smooth over $\hat{N}(v)$, which is defined as:

$$\hat{N}(v) = \left\{v_i | v_i \in N(v) \wedge n_v \cdot n_{v_i} > \cos\alpha_f\right\} \tag{6}$$

Where α_f is a threshold angle for feature edges, which can be user defined per mesh or per scene. These feature edges are also used for the direct lighting, to produce sharp edges on meshes. Note that our work is dependent on the tesselation of the scene, since higher tesselation will allow for a more accurate resolution of the indirect lighting.

3.4. Rendering

We download the computed total incoming radiance values from CUDA, and use it as input for our vertex buffer based mesh rendering. Since our elements coincide with the rendered vertices, this is a one to one mapping.

The surface shading is based on a simple split of the rendering equation into the direct and indirect illumination. The direct illumination is evaluated by a simple diffuse or glossy BRDF combined with cube mapped shadow maps for each light source. The shadow maps are rendered with percentage closer filtering, which smoothes out any aliasing artifacts. The indirect radiance is then added to that. Figure 4 shows how the two parts of the rendering equation are put together.

4. Discussion

The two most important aspects to discuss about a global illumination system are performance and image quality. In respect to performance, the scalability of is very important. Specific timings will be discussed in Section 5, while in this section we will discuss computational complexity.

Figure 6: *Our system is even able to handle large models. Shown are direct and indirect light (one and four bounces).*

4.1. Scalability

One of the big problems of global illumination is, that when one uses finite element methods, that the problem is equivalent to the n-body problem, as shown by Hanrahan et al. [HSA91]. This implies that naive solutions will always take $O(n^2)$ to complete, where in the following n will denote the number of elements in the scene (in our case vertices). This can be solved for small scenes on current hardware, but obviously this does not scale well. Using hierarchical approaches, this can be pushed down to $O(n \log n)$ in both time an space complexity.

Time and Space Complexity Our approach has linear space complexity, linear time complexity for light exchange and a log-linear time complexity for visibility and link construction. So for dynamic scenes we need to consider the following costs. First we need to build the kd-tree, which can be accomplished in $t_{kd} = O(n \log n)$ time. The shooting step for generating the links between elements takes $t_{shoot} = O(n_C \cdot n \log n)$, where n_C is the number of sampled directions. Computing the light links takes $t_{light} = O(n_L \cdot n \log n)$ time, where n_L is the number of light sources in the scene. The light exchange itself is performed in $t_{radiance} = O(n_B \cdot n_C \cdot n)$ time, where n_B is the number of bounces. This is the greatest advantage of our approach compared to other works, since our method scales linearly with the number of elements in the scene. Other methods, as shown in the next paragraph, usually take at least log-linear time. We can now see that computing the whole global illumination solution from scratch takes

$$\begin{aligned} t_{full} &= t_{kd} + t_{shoot} + t_{light} + t_{radiance} && (7)\\ &= O((1 + n_C + n_L) n \log n + n_B \cdot n_C \cdot n) && (8) \end{aligned}$$

For medium sized scenes of a few ten-thousand triangles, we achieve still interactive rates for t_{full}. And as discussed earlier, light updates are much cheaper:

$$\begin{aligned} t_{fulllight} &= t_{light} + t_{radiance} && (9)\\ &= O(L \cdot n \log n) + O(n_B \cdot n_C \cdot n) && (10) \end{aligned}$$

Scalability Compared With Other Methods In comparison, the work by Dachsbacher et al. [DSDD07] is a bit more difficult to analyze. They also use a discrete set of directions, for which they produce links. Due to the implicit visibility, their link structure contains $O(n^2)$ links in the worst case, because bins can contain more than one link. This is dealt with

by using a link hierarchy. The authors do not state an accurate value for the space complexity, but due to the hierarchy it might be $\Omega(n \log n)$. The complexity of constructing the link structure is completely unknown, but their initial linking phase takes $13s$ for the oriental scene, which is equivalent to our TEAROOM scene. Our algorithm takes $t_{full} = 1.1s$, and renders the scene with 65 frames per second, compared to 7.5 fps in the work of Dachsbacher et al. [DSDD07]. In both cases full indirect lighting is update in each frame. Their net number of links in this scene is 776k, while ours is 2019k. But still our light exchange is computed much faster. We need $6ms$ in total, while their algorithm takes $111ms$. The tests were performed on similar machines, albeit our CPU ran at 2.4GHz, versus 3.0GHz. In both cases the graphics cards used were GeForce 8800 GTX.

A second work that produces similar results as our approach is [DKTS07]. They use implicit visibility, which is computed during construction of a link hierarchy. They support only diffuse indirect illumination and dynamic models. The complexity of their method is unclear. They state an approximate complexity of $O(n \log n)$. When comparing both systems, we noticed, that our approach can compete with theirs. However they only cite timings for small scenes. For a 2670 vertex scene their approach achieves 7.53 fps. We achieve 6.3 GI updates per second on a scene with the same complexity. However it has to be noted, that their approach uses only one bounce of indirect illumination. More bounces are more costly than in our approach.

4.2. Image Quality

Since we are sampling radiance at vertices, the computed values are interpolated over the surface by the graphics card. A higher tesselation will result in a more accurate interpolation. Also, the more rays are used at each vertex, the better the radiance estimate will be.

A very important aspect of our system is that it handles indirect illumination very well. Since the light exchange is computed so rapidly, the computation of many light bounces is possible. In scenes like TEAROOM large parts of the scene are illuminated indirectly.

For proof of a good convergence of our algorithm, we compared our method with an unbiased path tracer called Luxrender, which is based on the well known PBRT by Pharr et al [PH04]. As can be seen from figure 5 our method converges to a correct solution. The main differences are mostly the shadow boundaries, which is due to the usage of shadow maps, but also some difficult light paths may not be found by our algorithm. But the absolute difference in radiance values is marginal. The reference solution was computed in 24 hours time with approximately 10,000 samples per pixel.

5. Results

We have mainly used five different scenes to test our system. The geometric complexity of the scenes, and the complexity of the resulting *k*d-trees are listed in Table 1. All scenes were built from scratch, except for the import of the more complex meshes.

Scene	Vtx $\cdot 10^3$	kd-nodes $\cdot 10^3$	kd-nodes MB	Links$\cdot 10^6$ raw	Links$\cdot 10^6$ cmp
COLORLAMP	1.6	1037	0.4	3.6	0.5
TORSO	7.1	17.5	1.7	3.4	0.8
TEAROOM	11.9	14.2	1.5	6.7	2.0
IPHI	76.5	109	9.4	42	12
CONFERENCE	122.8	297	30	73	14

Table 1: *The scenes used for testing. The number of links is shown in raw, and compressed form.*

A detailed performance comparison of the test scenes can be found in Table 2. There we used two systems to test if our system scales well with the number of GPUs in the system. As can be seen from the table, the number of rays shot and the waiting time for the first frame (which equals t_{full}, see section 4.1) directly reflects the number of GPUs installed. Ray throughput almost doubles, for the general link generation. Especially note that the TORSO scene is fully dynamic and achieves very high frame rates. We also ran our algorithm with an emulated 4 GPUs, which worked well. True benchmarks on a 4 GPU machine will be done in the future.

Our results show that we render faster than competing methods and we can handle models one magnitude larger than said methods (see Figure 6). Furthermore we showed that our system is scalable with future GPU and CPU generations.

6. Conclusion

We have shown that with the CUDA framework, fast and scalable diffuse global illumination in deformable scenes can be computed. Our work improves the possible performance especially for interactive relighting of diffuse scenes.

The next step will be to further reduce the number of links that are needed, for example by using a hierarchy. We are currently working on support for glossy indirect illumination, which will be implemented shortly.

The current implementation renders only point lights, directed or omnidirectional, with filtered cube shadow maps. Area lights are possible and are intended as future work.

Furthermore we want to evaluate how the temporal appearance of the asynchronous global illumination can be improved. For fast moving objects the indirect illumination information might be too much out of date and lead to unpleasant visual artifacts. This will be a focus of improvement.

Last, we also want to see how well our approach works on other parallel systems, like the Cell platform. The much bigger local memory of the Cell will probably allow for more complex operations, although the Cell has much fewer cores than a typical CUDA device.

Scene	Sys	kd-Tree	Gen. Links		Light links		Bounces	First	FPS
		ms	ms	10^3 rays/s	ms	10^3 rays/s	ms	frame	
COLORLAMP	A	8	833	1930	4	393	3	0.8s	54.0
	B	9	433	3709	5	314	2	0.5s	42.3
TORSO	A	111	413	4420	15	476	4	0.5s	65.0
	B	78	256	7121	19	397	4	0.4s	70.0
TEAROOM	A	71	680	4472	13	915	6	0.8s	65.0
	B	53	400	7595	15	793	6	0.5s	75.0
IPHI	A	35	11382	1721	92	832	54	12.4s	12.5
	B	47	5422	3614	103	746	56	7.9s	12.6
CONFERENCE	A	173	21606	1454	472	260	57	25.2s	10.0
	B	218	11403	2755	516	207	61	14.7s	11.5

Table 2: *This table shows the performance of systems A) Core2Duo 2.4 GHz, GeForce 8800 Ultra, B) Core2Quad 2.4 GHz, Dual GeForce 8800 Ultra. The time for the first frame, which equals the time for a full global illumination update, is reduced significantly, because the ray throughput almost doubles. All scenes use 3 bounces of indirect light.*

Acknowledgements

Thanks to Bassam Kurdali for the Mancandy model used in the TORSO scene.

References

[CPC84] COOK R. L., PORTER T., CARPENTER L.: Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 137–145.

[DBB06] DUTRÉ P., BALA K., BEKAERT P.: *Advanced Global Illumination, 2nd Edition*. A K Peters Ltd., 2006.

[DKTS07] DONG Z., KAUTZ J., THEOBALT C., SEIDEL H.-P.: Interactive global illumination using implicit visibility. In *Pacific Conference on Computer Graphics and Applications* (Washington, DC, USA, 2007), IEEE Computer Society.

[DSDD07] DACHSBACHER C., STAMMINGER M., DRETTAKIS G., DURAND F.: Implicit Visibility and Antiradiance for Interactive Global Illumination. *ACM Transactions on Graphics (SIGGRAPH) 26*, 3 (August 2007).

[GKBP05] GAUTRON P., KŘIVÁNEK J., BOUATOUCH K., PATTANAIK S.: Radiance cache splatting: a GPU-friendly global illumination algorithm. In *SIGGRAPH '05* (2005), ACM, p. 36.

[HBHS05] HAVRAN V., BITTNER J., HERZOG R., SEIDEL H.-P.: Ray maps for global illumination. In *EGSR* (2005).

[HDKS00] HEIDRICH W., DAUBERT K., KAUTZ J., SEIDEL H.-P.: Illuminating micro geometry based on precomputed visibility. In *SIGGRAPH '00* (2000), ACM, pp. 455–464.

[HSA91] HANRAHAN P., SALZMAN D., AUPPERLE L.: A rapid hierarchical radiosity algorithm. In *SIGGRAPH '91* (New York, NY, USA, 1991), ACM, pp. 197–206.

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *I3D '07* (New York, NY, USA, 2007), ACM, pp. 167–174.

[KAMJ05] KRISTENSEN A. W., AKENINE-MÖLLER T., JENSEN H. W.: Precomputed local radiance transfer for real-time lighting design. *ACM Trans. Graph. 24*, 3 (2005).

[KTHS06] KONTKANEN J., TURQUIN E., HOLZSCHUCH N., SILLION F.: Wavelet radiance transport for interactive indirect lighting. In *Rendering Techniques 2006 (EGSR)* (jun 2006).

[MB90] MACDONALD D. J., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Vis. Comput. 6*, 3 (1990).

[NVI07] NVIDIA CORPORATION: *NVIDIA CUDA Programming Guide*, 2007.

[PGSS07] POPOV S., GÜNTHER J., SEIDEL H.-P., SLUSALLEK P.: Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (Sept. 2007).

[PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[SC97] SHIRLEY P., CHIU K.: A low distortion map between disk and square. *journal of graphics tools 2*, 3 (1997), 45–52.

[SLS05] SLOAN P.-P., LUNA B., SNYDER J.: Local, deformable precomputed radiance transfer. *ACM Trans. Graph. 24*, 3 (2005).

[SSK07] SHEVTSOV M., SOUPIKOV A., KAPUSTIN A.: Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. *Computer Graphics Forum 26*, 3 (Sep 2007).

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26*, 1 (2007).

[WH] WALD I., HAVRAN V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Interactive Ray Tracing 2006*.

[WK06] WÄCHTER C., KELLER A.: Instant Ray Tracing: The Bounding Interval Hierarchy. In *Symposium on Rendering* (2006), pp. 139–149.

[WRC] WARD G. J., RUBINSTEIN F. M., CLEAR R. D.: A ray tracing solution for diffuse interreflection. In *SIGGRAPH '88*, ACM.

[YIDN07] YUE Y., IWASAKI K., DOBASHI Y., NISHITA T.: Global illumination for interactive lighting design using light path pre-computation and hierarchical histogram estimation. *Pacific Graphics* (2007), 87–98.

[ZHWG08] ZHOU K., HOU Q., WANG R., GUO B.: Real-Time KD-tree Construction on Graphics Hardware. *Technical Report* (2008).