Eurographics Symposium on Rendering (2004)
A. Keller, H. W. Jensen (Editors)

Simulating Photon Mapping for Real-time Applications

Bent Dalgaard Larsen † and Niels Jørgen Christensen ‡

Technical University of Denmark

Abstract

This paper introduces a novel method for simulating photon mapping for real-time applications. First we introduce a new method for selectively redistributing photons. Then we describe a method for selectively updating the indirect illumination. The indirect illumination is calculated using a new GPU accelerated final gathering method and the illumination is then stored in light maps. Caustic photons are traced on the CPU and then drawn using points in the framebuffer, and finally filtered using the GPU. Both diffuse and non-diffuse surfaces can be handled by calculating the direct illumination on the GPU and the photon tracing on the CPU. We achieve real-time frame rates for dynamic scenes.

Keywords: photon mapping, real-time, global illumination, graphics hardware

1. Introduction and Previous Work

Global illumination methods such as radiosity ([GTGB84]) and photon mapping ([Jen01]) tend to be slow. Therefore, many attempts have been made to optimize the calculation of global illumination.

Recently, concepts for graphics hardware acceleration of ray-tracing have been suggested ([CHH02]). In [PDC*03] the photon mapping algorithm was implemented almost entirely on graphics hardware. These solutions have proven that it is possible to implement complex algorithms on the GPUs. However, so far these implementations are not faster than CPU based implementations. In [PDC*03] the final gathering step is omitted and consequently the image quality is low. The final gathering step is the most computational expensive step and even though this step is omitted the frame rates are only near interactive when using progressive updates.

Another research direction is to use a great number of CPUs ([WKB*02]). However, such a setup is mostly available at universities.

In [TPWG02], a method for calculating interactive global illumination is described. The scene is rendered using graphics hardware and the illumination is stored in a hierarchical patch structure in object space. At first, the top level is used but based on priorities calculated in camera space the patches are then subdivided. The illumination is calculated using a path tracing algorithm and the illumination of only 10 to 100 patches can be calculated per second on each CPU. As a result of this low number up to 16 CPUs are used. When the camera or objects are moved quickly artefacts will occur.

In [Kel97], instant radiosity is introduced. The technique is based on Quasi-Monte-Carlo (QMC) walk of photons. As each photon hits a surface, a point light source is created which is rendered using graphics hardware. The final image is created by rendering the scene a large number of times and adding the images using the accumulation buffer. Therefore when the view is changed, the entire process has to be started over. Halton sequences are used to distribute the photons and these are updated based on an age criterion.

In [DBMS02], photons are divided into groups and re-emitted selectively based on dynamic objects collision with a bounding volume around photon groups. This solution demonstrates interactive frame rates in a purely diffuse environment. The photons are also distributed using a QMC method. Photon energies are stored at the nearest vertices in the scene, and the scene therefore has to be tesselated heavily. The photon energies are also used for the direct illumination, which makes the direct illumination very inaccurate. The weakness of using selective photon tracing using this approach is that the final gathering step is not not include.

† bdl@imm.dtu.dk
‡ njc@imm.dtu.dk

The final gathering step is necessary for calculating accurate indirect illumination ([Jen01]).

Real-time caustics have been approximated in [WS03], but the solution is limited to single specular interaction and assumes that the light sources are far away from the specular reflectors. Furthermore, occlusion between the light source and the specular reflector are not handle.

Thus, several attempts have been made to obtain an interactive global illumination solution. However, no general solution has been obtained that can run on commodity hardware. Severe limitations exist either on the computer platform, in the scene complexity, the generality of the global solution, or the dynamics of the scene.

However, for individual elements in global illumination, solutions have been proposed for dynamic scenes on commodity hardware. Real-time direct lighting has been known for many years, but has been made more versatile recently by the introduction of programmable hardware. Real-time soft shadows can be approximated using different techniques ([HLHS03]). Specular interreflection between planar surfaces, between curved surfaces, and between mixtures of planar and curved surfaces can also be handled in real-time ([NC02]).

This paper introduces a new method for simulating photon mapping in real-time for dynamic scenes on commodity hardware. We will do this by exploiting the main idea behind photon mapping, namely that the different elements in global illumination can be calculated individually. The direct, specular, caustics, and indirect illuminations are simply added together. Parts of the global illumination solution can be derived from the results mentioned above, and we will need to find specific methods for the following four problems that tend to be computationally intensive for a real-time application:

- First, we improve photon emission from light sources in dynamic scenes. Specifically, we introduce an improved method for exploiting the frame-to-frame coherence for moving objects in a scene.
- Secondly, we introduce a strategy for an approximated reconstruction of the full illumination of the scene after the photon emission.
- Thirdly, we use this reconstruction to find the indirect illumination in a scene using a hardware optimized fast gathering method.
- Fourthly, we introduce an optimized solution for caustics also based on an adequate reconstruction method.

Our proposed methods follow the photon mapping strategy closely ([Jen01]).

The photon emission (Section 2), reconstruction of the photon map (Section 3), final gathering (Section 4), and caustics (Section 5) solve our four main problems. Section 6 handles the remaining contributions (such as direct illumination, shadows, and specular inter-reflections) and Section 7 integrates the individual contributions into a final solution.

2. Selective Photon Emission

In traditional photon mapping all photons are traced in a first pass. But tracing all photons each frame in a real-time application is too computationally expensive. The selective photon tracing introduced in [DBMS02] solves this problem in some ways as only intelligently selected photons are re-emitted each frame. They give each photon a fixed initial direction using QMC Halton sequences, and the photons from the light source are divided into groups where the initial direction of each photon is similar. Each group contains an equal amount of photons with equal energies. The photons are traced on the CPU. We find this to be an attractive strategy, but we distribute the photons a bit differently. We do this to avoid the weaknesses of the method which are described in the introduction.

We enumerate the photons in each group, and for each frame only one photon from the group is traced. In the next frame, a new photon from the same group is selected. This is done in a Round-Robin fashion. The path of each photon is stored, and if the new path diverges from the previous path, all photons from the group will be marked for redistribution. In this way, more effort is spent in areas where the scene is modified. It is also guaranteed that minor changes will eventually be seen by a photon. It may nevertheless take more time to discover minor changes than major changes. This is the case as only few photons from the group may be invalidated and the group may act for a longer time as if no changes have occured in its domain. Major changes will be registered faster as many or all photons from a particular group will be invalidated. Photon bounces are handled by using Russian Roulette ([Jen01]).

The photons are stored as *photon-hits* on the surfaces along the path of the traced photon. The complete path of each photon is also stored. In this way, it is easy to remove the photon-hits from the scene in constant time if the path of the photon is invalidated. It is also faster to determine whether the photon path has been invalidated. Each surface also has pointers to the photon-hits that have hit that particular surface, making it faster to determine the total amount of photon energies on a surface. The extra storage needed per photon is an energy amount for each hit, and a pointer from the photon-hit to the surface, and a pointer from the surface to the photon-hit. The average length of a path is fairly short when using Russian Roulette. The memory overhead for storing the photon path is therefore not substantial.

As a result of our chosen strategy of storing the photon paths, a moving light source will cause all photon paths to be invalidated each frame.

3. Approximated Reconstruction of the Full Illumination

The photons distributed as described in the previous section carry information about the full illumination in the scene. These photons are used for reconstructing an approximation of the full illumination. This process is described in the following.

If enough photons are distributed, the photons will represent the correct illumination in the scene, and they can be calculated directly by using density estimation ([SWH*95]). The problem is that "enough" photons are many millions and it is almost impossible to completely remove the noise. Therefore, the photons are used to reconstruct an approximation of the full illumination, and then this approximation is used in the final gathering step to calculate a smooth and accurate approximation of the indirect illumination. The number of photons needed when performing the final gathering is many times smaller than the photons needed for density estimation ([Jen01]).

In order to approximate the full illumination we compute the irradiance using an N-nearest neighbor query of the photons, and the total energy of these photons is divided by the area that they span. The radiance of the surface is stored in reconstruction texture maps applied to the surfaces. We call these texture maps approximated illumination maps (AIMs).

Kd-trees are fast for N-nearest neighbor queries. In order to build the kd-tree, only photons that are located on surfaces which could possibly contribute to the irradiance of the current surface are considered. We use a technique similar to the one presented in [LC03]. In [LC03] the division into surfaces is done automatically, however in our implementation we have performed the division by hand using a 3D modelling tool. When a photon hits a surface, it is not stored in a global structure but in a structure belonging to the surface that was hit (see the discussion for an analysis of this choice). A surface can contain an arbitrary number of polygons. Figure 1 shows a Cornell box with a bumpy floor and unique colors for each surface.

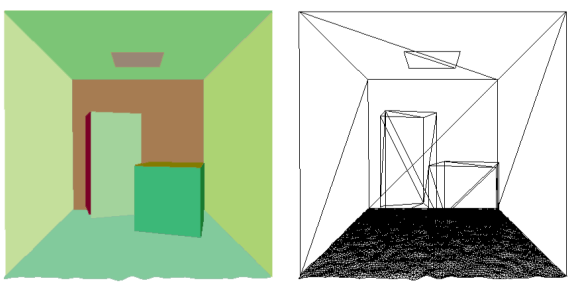

Figure 1: Left: A unique color is displayed for each surface, Right: Same scene shown using wireframe (13.000 polygons and 16 surfaces

Updating all the AIMs for each frame is computationally expensive and undesirable. Therefore we use a delta value for each surface to control when its AIM should be updated. The value (Δ_f) is a delta value for the full approximated illumination. The Δ_f value is affected by the energy of any photon that is removed or stored on a surface. Only when Δ_f is larger than a small threshold value, the AIM should be updated.

In practice it can be necessary to limit the amount of work done per frame. In our implementation we first handle surfaces with high Δ_f divided by the surface area. We only update a limited amount of surfaces per frame.

As only one photon map from a single surface is utilized at any point in time, it is only necessary to create a small photon map in the memory. This makes the memory requirement smaller than if one global photon map containing all the photons had been used.

4. Indirect Illumination using Hardware Optimized Final Gathering

A hardware optimized final gathering method was introduced in [CG85], namely the hemi-cube. This method needs to render the scene 5 times for each resulting *final gather value*. The 5 renderings are one for each side of the hemi-cube. The front side of the hemi-cube contributes with about 56% of the value, while each of the side planes each contribute with about 11% of the value. In [SP89] a method is introduced that only uses the front plane of the hemi-cube and then enlarges this front plane. In this way, a more accurate solution can be achieved, as we only use the front plane. If e.g. the front side is enlarged to double side length, thus making the total area 4 times larger, this accounts for about 86% of the total incoming directions (See Figure 2).

Figure 2: Final gathering using a single plane

We use this method for the final gathering step and render the scene with the AIMs. Each pixel of this rendering must be multiplied with the fraction of the hemisphere that it spans and a cosine factor in order to calculate the irradiance. The irradiance is defined as:

$$E = \int_{\Omega} L(\omega)(n \cdot \omega) d\omega \qquad (1)$$

where E is the irradiance, L is the incoming light, n is the normal, ω is the direction of the incoming light, and $d\omega$ is the solid angle.

The cosine weighted area on the hemisphere which a fragment on the rendered surface of the rendered plane spans is

defined as:

$$F_h(x,y) = \frac{1}{\pi(x^2+y^2+1)^2}\Delta A_f \quad (2)$$

where F_h is the cosine weighted area of the hemisphere and ΔA_f is the area of a fragment. The (x,y) components are the distances from the center of the plane([CG85]). The irradiance can then be calculated as:

$$E = \sum_{x,y} p(x,y)F_h(x,y) \quad (3)$$

where $p(x,y)$ is the pixel value of the rendered plane. The irradiance value should be divided by the percentage of the hemisphere which the rendered plane spans in order to compensate for the missing areas.

The calculation above can be implemented on the GPU. The way we have implemented this is first to render the scene to a pixel buffer (pbuffer). Then this pbuffer is used as input to a fragment program which multiplies each pixel with a value calculated using Equation 2. Then the resulting summation is calculated using hardware MIP map functionality (we use the OpenGL extension SGIS_generate_mipmap). The MIP map function calculates the average of a texture in the topmost level of the MIP map. Therefore we multiply each pixel by the total number of rendered pixels. This is performed in the fragment program which is run before the MIP map is executed. We copy this final pixel to a texture that is applied to the surface in the scene. We call this texture the indirect illumination map (IIM) (In our implementation they have the same resolution as the AIMs). All the steps are executed on the GPU, and no expensive read-back to the CPU is necessary.

Calculating the radiance for the indirect illumination can be done in several ways. If a texture is applied to the surface, the irradiance should be stored in the IIM and multiplied with the texture during rendering. But if the surface has a uniform color, the radiance could just as well be stored directly in the IIM. This can be done by pre-multiplying the irradiance values with the reflectance of the surface in the fragment program.

Displaying the illumination is often done in real-time applications by using texture maps which are also called light maps. Light maps usually contain both direct and indirect illumination and they are often coarse. Since the indirect illumination usually changes slowly over a surface it is possible to use an even coarser texture.

It is noted that when we use this approach, only diffuse reconstruction of the indirect illumination can be handled.

Many techniques can be used for applying a texture with uniform size to a model, and several full automatic methods have been proposed. In our implementation, we have applied the textures manually by using a 3D modelling tool.

As with the AIMs, it is computationally expensive to update all the IIMs for each frame. Therefore, we introduce Δ_i which is the delta value for the indirect illumination. This value is similar to Δ_f except that it is only affected by photons that have bounced at least once. As with the AIMs the surface with the highest Δ_i will have its IIM updated first.

In our implementation, we restrict the number of texels which can be updated per frame in order to keep a high frame rate.

We use two textures for the IIMs. One that is visualized and one that is being updated (double buffering). When the update of a texture is done, the two textures are switched using a blend between the textures. This is done in order to avoid popping. But for the indirect illumination to be updated this is a trade-off between popping and lag. We have therefore set the blend function to be fairly fast. Whether a quick or a slow blend should be used depends on the application.

5. Caustics

Caustics arise when a photon hits a diffuse surface after having been specularly reflected one or several times directly from the light source. When using photon mapping, caustic photons are traced in the same way as with the photons used for the indirect illumination. But photons are only traced in directions where known caustics generators (specular reflectors) are located ([Jen01]). When the photon hits a diffuse surface, its location is stored. Our method is based on this strategy, which means that we do not have the limitations of the real-time caustic method described in [WS03].

We distribute the photons evenly using QMC Halton sequences in order to lower the noise and avoid flickering. The photons are traced by using a standard CPU ray-tracer. We store the photons in a simple list. We do not divide the photons into groups as with the indirect illumination because caustics are very localized.

In order to reconstruct the caustics, we do the following. First we draw the scene by using the color black to a pbuffer with the depth buffer enabled. This is done in order to get the right occlusion of the photons. Then all the photons are drawn additively as points by using blending. Afterwards the pbuffer contains a count of the photons that have hit a particular pixel. This pbuffer is used as a texture, which makes it possible for a fragment program to read the color value of the current pixel from the previous rendering. A screen size polygon with this texture is therefore drawn by using a fragment program.

Furthermore it is also possible to read the photon count. Based on the photon count of the nearby pixels, a filter is applied to the pixels:

$$c(x,y) = s\sum_{i=-k}^{k}\sum_{j=-k}^{k} t(x+i,y+j)\sqrt{1+2k^2-(i^2+j^2)} \quad (4)$$

where $c(x,y)$ is the resulting color at position (x,y) and $t(x,y)$ is the texture value at position (x,y). s is a scaling value that adjusts power of the photon energies. We use a filter of size $7x7$ (i.e. $k = 3$).

This is a screen space filtering while photon mapping traditionally uses filtering in world space. If the depth buffer is also made available to the fragment program (which is possible on the GPUs of today), it is possible to calculate the world space position of the photon by multiplying the depth with the inverse modelview projection matrix (after rescaling to the canonical view volume). In this way, it is possible to perform the caustic filtering in world space, which will produce more correct caustics. Even though this will create more accurate caustics, it will also make the fragment program longer and consequently slower as this calculation has to be done for all fragments. In our case that would be 49 times. We have therefore chosen only to implement screen space filtering.

By using this method, it is possible to count 255 photons (8 bits) at each pixel, and this will be sufficient in most cases. When counting the photons in the framebuffer it is assumed that all lights and all caustics generators have the same color, otherwise a floating point pbuffer is needed.

Tracing all photons in every frame may not be necessary. If the application run at e.g. 30 fps, it may not be notable if each photon is only retraced every second to every tenth frame. If the specular object is moved fast, it will be possible to see a trail after the specular object. Whether this is visually acceptable depends on the application. In our experience, the delayed photon distribution does not disturb the visual appearance, and if the object moves slowly it is hard to notice.

The implementation of equation 4 is a time consuming fragment program. When using a filter of size $7x7$, the summations are unrolled to a program that has about 400 lines of assembly code and 49 lookups in the texture. Therefore it is desirable to limit the use of the fragment program to areas where caustic photons are actually present. This is done by using the following method.

Before the photons are drawn to the pbuffer, stenciling is enabled. The stencil is set to increment on zpass. When the photons are drawn, they are also marked in the stencil buffer. Then the screen is divided into a number of grid cells. For each grid cell, an occlusion query is started as a quad is drawn. The stencil function is set only to let pixel be written to the pbuffer if the stencil value in the pbuffer is greater than zero. When the quad is drawn, the occlusion query will return the number of pixels that were modified. If no pixels were modified, no photons need to be filtered in this area. In this way, the inexpensive occlusion query can identify the areas that have caustic photons. Often the caustic only fills a few percent of the screen. The process is illustrated in Figure 3.

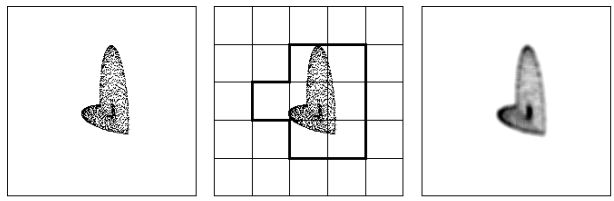

Figure 3: Left: photons are drawn on the screen. Middle: Areas on the screen are tested for photons. Right: Areas are filtered in screen space.

6. Direct Illumination, Shadows and Specular Interreflections

By using modern graphics hardware, it is possible to evaluate the shading of each pixel individually ([MGA03]). In our implementation, we use a fragment program for calculating the direct light, and we use stencil buffer shadow volumes for calculating the shadows. Our implementation uses hard shadows but real time hardware rendered soft shadows could just as well have been used as described in [JCLP04] or one from [HLHS03] could be selected. Any soft shadow algorithm is equally well suited as the shadow rendering is done in a separate pass. When combining the light contributions the shadow is applied by using a screen size texture.

We use a dynamic environment map for specular reflections. This is done by using a cube-map and for each frame the scene is rendered six times in order to update the sides of the cube. Multiple interreflections could have been used as well ([NC02]).

7. Combining the Light Contributions

Combining the various contributions is an additive process. We create a separate pbuffer for the shadows and another for the caustics. When the scene is rendered, the direct illumination is calculated for each pixel and multiplied by the content in the shadow pbuffer. The texture value for the indirect illumination is sampled in the IIMs which are applied to each surface. These values are added to the final color along with the caustic's value (see Figure 4). In this way, we combine the contributions in a final pass.

The formula is as follows:

$$L = L_{indirect} + L_{caustics} + L_{specular} + L_{direct} * shadow \quad (5)$$

When several lights are present in the scene the formula is as follows:

$$L = L_{indirect} + L_{caustics} + L_{specular} + \sum_{i=0}^{lights} L_{direct}(i) * shadow(i) \quad (6)$$

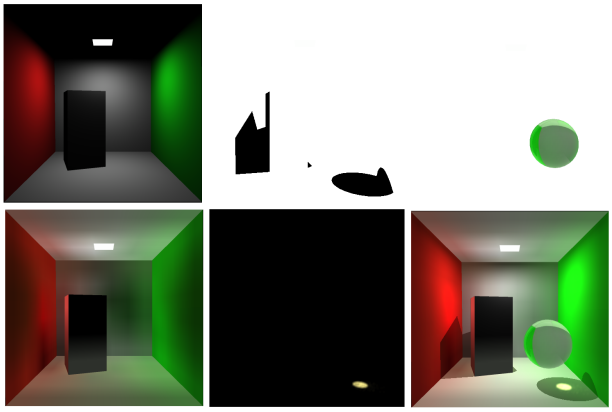

Figure 4: Top left: The direct illumination without shadows. Top middle: Shadows. Top right: Reflections. Bottom left: Indirect illumination. Bottom middle: Caustics. Bottom right: Complete illumination

8. Results

We have implemented our application on a Pentium 4, 2.4 Ghz with a GeForceFx 5950 graphics card running Windows. All code has been written in C++ and compiled by using Visual Studio 6. Cg was used for all vertex and fragment programs ([MGA03]). Our photon-tracer utilizes a standard axis-aligned BSP-tree.

Each dynamic object in the scenes uses a separate BSP-tree. Any photon traced in the scene is therefore tested for intersection with all BSP-trees.

The scene in Figure 4 with indirect illumination and caustics runs a 35+ fps. The cube and the sphere are dynamic objects. 10000 photons are used for the caustics, and they are completely updated over 8 frames. For the indirect illumination, 77 photon groups are used each with 40 photons. A maximum of 20 texels per frame are updated by using final gathering. The big surfaces on the walls have textures of 5 by 5 texels for both the AIM and IIM. In total, the scene has 140 texels for the AIMs and similarly 140 texels for the IIMs. The scene is rendered at a resolution of 512 by 512 pixels and all pbuffers are also 512 by 512 pixels. The environment map is rendered as a cube-map and the scene is rendered 6 times per frame. Each side in the cube-map has a resolution of 128 by 128 pixels. The memory used for storing the photons and their paths is in this case the number of photons multiplied by 3 floats for the energies and the size of two pointers multiplied by the average path length, which in our case is approximately 2. This gives a total memory requirement of approximately 120 Kb.

If we consider two unconnected rooms, and modifications only occur in one of the rooms, then no updates will be necessary in the other room. Only minimal computational power will be spent in the other room as no photons will be invalidated. This case is shown in the top-most scene in Figure 5.

Photons	Irradiance lookup (100 photons)	Balancing time
300	0.023 ms	0.10 ms
500	0.027 ms	0.18 ms
1000	0.029 ms	0.39 ms
2000	0.034 ms	0.84 ms

Table 1: Timings for balancing a kd-tree with photons

When the rooms are connected, updates made in one of the rooms will now affect the illumination in both rooms (see middle image in Figure 5). The bottom-most image in Figure 5 shows a scene, where the right room is illuminated primarily by indirect illumination.

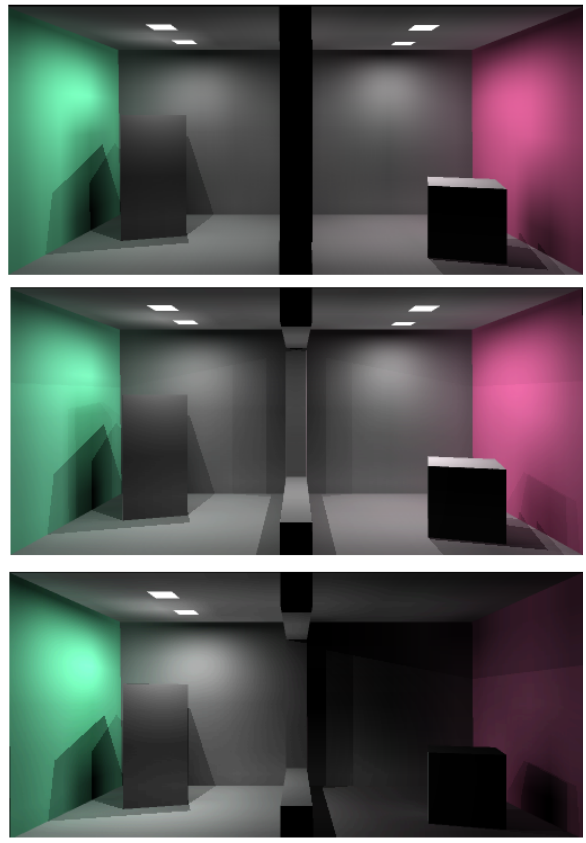

Figure 5: Top: A scene with two divided rooms each with two light sources. Middle: The rooms have been connected. Bottom: Two of the light sources have been turned off.

Balancing a kd-tree for fast searching is computationally cheap when the number of photons in the kd-tree is low. In Table 1, timings for balancing a kd-tree are shown. The time for finding the nearest 100 photons is also shown.

When the indirect illumination is updated it is important

Render size	Fragments	Polygons in scene	Timings
8	8x8	34	0.75 ms
16	16x16	34	0.76 ms
32	32x32	34	0.78 ms
8	8x8	13,000	0.90 ms
16	16x16	13,000	0.91 ms
32	32x32	13,000	0.97 ms
8	8x8	133,000	1.55 ms
16	16x16	133,000	1.57 ms
32	32x32	133,000	1.59 ms

Table 2: GPU final gathering timings

that the final gathering step is fast. We have measured how much time a single final gathering takes. A single final gathering includes a rendering of the scene using textures of the approximated illumination, a fragment processing of each pixel, summation of the pixels using hardware MIP map generation, and a copy of the final pixel to a texture (see section 4). We have timed these steps using scenes with different polygon count. The scenes with 13,000 and 133,000 polygons were made by subdividing the surfaces and making the surfaces more bumpy. The results can be observed in Table 2. It is noted that the timings are not very sensitive to the number of polygons in the scene.

Another option would be to use traditional ray-tracing for the final gathering step and send the calculated value to the graphics hardware. Our timings show that copying a single texture value from the CPU to the GPU takes 0.65 ms. Tracing 1024 (32x32) rays in a scene with one dynamic object (i.e. two BSP-trees) and 8000 polygons takes 16.7 ms using our implementation of the axis-aligned BSP tree. Furthermore, the radiance should be calculated at the surface that each ray hits and a final cosine weighted summation should be performed. Using our measurements, it can therefore be concluded that our hardware optimized final gathering method is many times faster than a ray-tracer based approach.

The filtering of caustics in screen space is optimized by using the occlusion query which is described in a previous section. Our timing of the occlusion query shows that it takes 0.33 ms to run 100 occlusion queries over an area of 512 times 512 pixels. Using our GPU caustics filter on an area of 512 times 512 takes 37.6 ms while only filtering 1% of this area takes 0.38 ms. In a typical scene, the caustics fills less than 5% of the screen (see Figure 6). The time spent on the occlusion query is therefore well worth the extra effort.

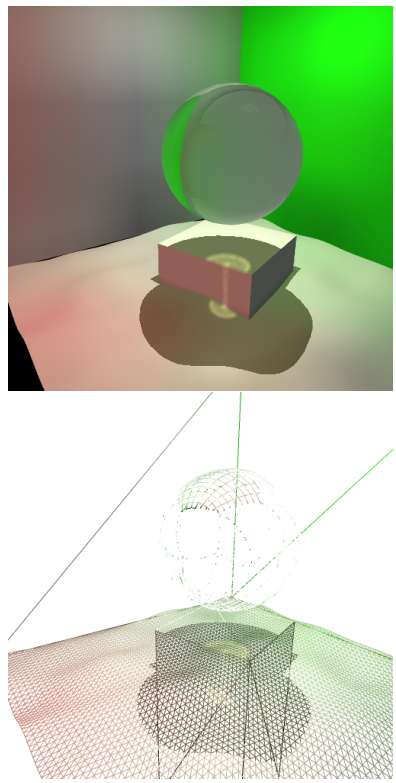

Figure 6: Top:Caustics being cast from a dynamic object onto another dynamic object and a bumpy floor. Bottom: The same scene shown using wireframe

9. Discussion and Future Work

Many methods can be used for optimizing a 3D application. Popular methods are culling, LOD, tri-stripping, and front-to-back render order, but many others can be used ([AMH02]). These all work well with our new methods.

The final gathering method renders the entire visible scene using the center of the texel that should be updated as the camera point. Since all pixels of this rendering are averaged using MIP mapping before they are used, it is not necessary to render an extremely accurate image, and the lowest level of detail for all objects in the scene might as well be chosen. Culling algorithms should of course also be enabled using the final gathering renderings.

Tracing photons is done using an axis-aligned BSP-tree. If the scene is divided up into a number of cells e.g. using portals the BSP-tree can just as well be divided up into several trees. But since photon tracing using a BSP-tree only takes $O(\log n)$ it may not be desirable to split the BSP-tree.

Both our method for final gathering and photon tracing scales well with regard to the number of polygons in the scene. The limiting factor is the movement of objects in the scene that causes photons to be invalidated and the indirect

illumination to be updated. Particularly the accuracy of the indirect illumination, i.e. the number of texels in the IIMs, is a limiting factor. Consequently, making the scene larger e.g. with more rooms and floors, will not affect the lag and frame rate if modifications occur locally. But the computation time will be affected heavily if the objects get more detailed and more and smaller texels have to be used for representing the illumination. Our method scales well with the size of the scene but not with a lot of fine details in the geometry. Nevertheless, a fractal floor, as demonstrated in our example, can be handled appropriately. Avoiding many small texels in the IIMs is an area of future research.

As each light source has a fixed number of photon groups it may be expensive to trace a single photon from each group every frame if many light sources are present in the scene. It may therefore be desirable to trace fewer photons from each light source per frame. Whether to trace a photon from each photon group or trace fewer photons from a light source can e.g. be determined by the distance from the light source to the viewer.

We have implemented the summation of all pixels in the final gathering step using hardware MIP mapping. Currently this can only be done using 8 bit precision. In the next generation of GPUs it is likely that this can be done using floating point precision. Another option would be to calculate the MIP mapping using fragment programs which is done in e.g. [NPG03].

Aliasing is usually a problem when using hemi-cube based methods. We use the hemi-cube for gathering radiance values from textures with large texels. The large texels is a filtering of the radiances which reduces the aliasing problem. Further it is cheap to increase the size of the hemi-cube (See Table 2).

One of the biggest advantages of using photon mapping compared to e.g. radiosity is that it is mesh independent. When using our approach, we group the geometry into surfaces and use local photon maps, and this suggests that our method is less geometry independent than traditional photon mapping. But when calculating the irradiance by using traditional photon mapping with an n-nearest neighbors query, only photons with normals similar to the center of the query are usually used. This can be viewed as an implicit division of the geometry similar to our grouping of the surfaces.

By using our method, shadows and direct illumination is updated in every frame while the indirect illumination is updated progressively. We find this to be a good strategy since our observation is that correct direct illumination and shadows are more important than indirect illumination for the visual impression of a scene.

Our strategy for updating the indirect illumination is in some ways similar to [TPWG02] as the indirect illumination is updated selectively. We based our priorities on invalidated photons in object space while they calculate their priorities in camera space. When using our method it is therefore possible to move the camera quickly without severe artefacts. This is something that is often done in e.g. games. This is possible because the indirect illumination of the entire scene is cached and because the indirect illumination is assumed to be diffusely reflected.

The texture resolution for the indirect illumination (IIM) is fixed in our implementation. Further research should be made to address the problem of dynamically choosing the texture resolution in order to reconstruct the indirect illumination more accurately. One direction for this research could be to apply a filter to the texture in order to find high second order derivatives, as this is probably a good location for increasing the texture resolution. Another direction would be to use methods that depend on distances to other surfaces similar to what is used in irradiance caching ([WRC88]). A hierarchical method similar to [TPWG02] could also be used for subdividing the surfaces although it requires a fine meshing or a constant re-meshing of the scene.

It should be easy to add our methods at specific locations. E.g. in one room, indirect illumination could be enabled and at an outdoor location, caustics could be enabled for a single object. In this way, the designer of e.g. a game can make sure that the application always runs at a sufficient frame rate while adding additional features only where it will not compromise the frame rate.

10. Conclusion

We have introduced a solution for simulating photon mapping for real-time applications. The method uses a combination of CPU and GPU based algorithms and uses features that are currently available in most commodity GPUs.

Our main contributions in this paper are the fast calculation of indirect light and the fast calculation of caustics. These two elements can be implemented individually and added to the remaining light contributions such as direct illumination, shadows and specular reflections to produce a full global illumination solution similar to the strategy of photon mapping.

Both the photon tracing and the direct illumination can handle diffuse and non-diffuse surfaces. However, the reconstruction of the indirect illumination is restricted to diffuse surfaces as the indirect illumination is currently stored in textures.

We have used selective updates in several levels for the indirect illumination and in this way obtained an intelligent caching mechanism while still keeping all the elements of the photon mapping method. The key parameters have been an improved selective photon emission and tracing, together with local selectively updated photon maps. Furthermore we have exploited graphics hardware for fast final gathering using a hemi-cube based method with negligible aliasing artefacts.

Our caustic algorithm can run in real-time without the limitations of previous methods. The method is fast because of a simple selective filtering and the image quality is reasonable even though we have only implemented screen space filtering.

We have proven that our indirect illumination method works well for small scenes but argued that it scales well to larger scenes if the geometric details do not demand too heavy use of detailed texture maps. Furthermore using texture maps we avoid heavy meshing of the scenes.

11. Acknowledgment

We acknowledge Andreas Bærentzen for the help during the development and the implementation of our method.

References

[AMH02] AKENINE-MÖLLER T., HAINES E.: *Real-Time Rendering*. A. K. Peters, Ltd., 2002.

[CG85] COHEN M. F., GREENBERG D. P.: The hemicube: a radiosity solution for complex environments. In *Proceedings of Siggraph* (1985), pp. 31–40.

[CHH02] CARR N. A., HALL J. D., HART J. C.: The ray engine. In *Proceedings of the conference on graphics hardware* (2002), pp. 37–46.

[DBMS02] DMITRIEV K., BRABEC S., MYSZKOWSKI K., SEIDEL H.-P.: Interactive global illumination using selective photon tracing. *Eurographics Workshop on Rendering* (2002), 25–36.

[GTGB84] GORAL C. M., TORRANCE K. E., GREENBERG D. P., BATTAILE B.: Modeling the interaction of light between diffuse surfaces. In *Proceedings of Siggraph* (1984), pp. 213–222.

[HLHS03] HASENFRATZ J.-M., LAPIERRE M., HOLZSCHUCH N., SILLION F.: A survey of real-time soft shadows algorithms (2003). Eurographics. State-of-the-Art Report.

[JCLP04] JACOBSEN B., CHRISTENSEN N. J., LARSEN B. D., PETERSEN K. S.: Boundary correct real-time soft shadows. In *Computer Graphics International* (2004).

[Jen01] JENSEN H. W.: *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, Natick, MA, 2001.

[Kel97] KELLER A.: Instant radiosity. In *Proceedings of Siggraph* (1997), pp. 49–56.

[LC03] LARSEN B. D., CHRISTENSEN N. J.: Optimizing photon mapping using multiple photon maps for irradiance estimates. *WSCG Posters* (2003), 77–80.

[MGA03] MARK W. R., GLANVILLE S., AKELEY K.: Cg: A system for programming graphics hardware in a C-like language. In *Proceeding of Siggraph* (2003), pp. 896–907.

[NC02] NIELSEN K. H., CHRISTENSEN N. J.: Real-time recursive specular reflections on planar and curved surfaces using graphics hardware. *Journal of WSCG*, 3 (2002), 91–98.

[NPG03] NIJASURE M., PATTANAIK S., GOEL V.: Real-time global illumination on GPU. *Submitted for publication* (2003).

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. *Proceedings of the conference on graphics hardware* (2003), 41–50.

[SP89] SILLION F., PUECH C.: A general two-pass method integrating specular and diffuse reflection. In *Proceedings of Siggraph* (1989), pp. 335–344.

[SWH*95] SHIRLEY P., WADE B., HUBBARD P. M., ZARESKI D., WALTER B., GREENBERG D. P.: Global illumination via density estimation. In *Proceedings of Rendering Techniques* (1995), pp. 219–230.

[TPWG02] TOLE P., PELLACINI F., WALTER B., GREENBERG D. P.: Interactive global illumination in dynamic scenes. In *Proceedings of Siggraph* (2002), pp. 537–546.

[WKB*02] WALD I., KOLLIG T., BENTHIN C., KELLER A., SLUSALLEK P.: Interactive global illumination using fast ray tracing. In *Proceedings Eurographics Workshop on Rendering* (2002), pp. 15–24.

[WRC88] WARD G. J., RUBINSTEIN F. M., CLEAR R. D.: A ray tracing solution for diffuse interreflection. In *Proceedings of Siggraph* (1988), ACM Press, pp. 85–92.

[WS03] WAND M., STRASSER W.: Real-time caustics. In *Computer Graphics Forum, Proceedings of Eurographics* (2003), vol. 22(3).