Multi-mesh caching and hardware sampling for progressive and interactive rendering

Gabriel Fournier Bernard Péroche

L.I.R.I.S : Lyon Research Center for Images and Intelligent Information Systems
CNRS / INSA de Lyon / Université Lyon 1 / Université Lyon 2 / Ecole Centrale de Lyon
Bâtiment Nautibus, 8 boulevard Niels Bohr
69622 Villeurbanne Cedex, FRANCE

gabriel.fournier@liris.cnrs.fr bernard.peroche@liris.cnrs.fr

ABSTRACT

We present a framework for progressive and interactive rendering with soft shadows and indirect illumination of a triangulated scene. Our method is a multi-pass algorithm that separates the rendering of each main component of radiance in order to update the image as fast as new samples are computed. Those radiance samples are computed at the vertices of multiple recursively subdivided meshes, allowing fast hardware interpolation between the samples. These radiance samples are computed using irradiance values cached in multiple meshes. These meshes separate the direct irradiance from each light source and the indirect one. Using multiple meshes gives us the ability to better reuse samples and to better adapt the sampling density than if a unique mesh was used. We also propose to quickly compute accurate soft shadows and indirect irradiance using the graphics hardware for visibility determination.

Keywords

global illumination, irradiance caching, progressive rendering, interactive rendering, graphics hardware, area light source

1 INTRODUCTION

Real time realistic rendering on a standard PC, with area light sources and indirect illumination, is still a major challenge in computer graphics. In this paper, we suggest a framework and a few tricks that should bring us closer to this goal. Our approach allows progressive and interactive realistic rendering on a single office PC of a triangulated scene lit by area light sources. We chose to favor interactivity over image quality, but our progressive rendering algorithm makes the image tends towards full quality when the user lingers in the same area. Our method does not need a long preprocessing. It can provide fair quality images in a few seconds, hence it can be useful for image preview while designing a scene.

Our approach is a multi-pass method. We separate the rendering of the main radiance components: direct diffuse, direct specular and indirect diffuse. These radiances are computed at the vertices of multiple progressively refined meshes, using a different mesh for each part of the radiance. To compute these radiances, we use an irradiance sample cache. This cache is also made of multiple progressively refined meshes. We use different meshes to store the direct irradiance of each light source, and the indirect irradiance. This new approach allows us to limit the number of computed samples by reusing already computed ones. Irradiance samples are computed using the hardware for visibility determination.

Conference proceedings ISBN 80-903100-7-9
WSCG'2005, January 31-February 4, 2005
Plzen, Czech Republic.

After a few words on the ideas that led us to our solution (Section 2), we will give a quick overview of our method (Section 3). Then we will describe with more details our multi-mesh caching framework (Section 4), how we propose to sample irradiance to fill our cache (Section 5) and how this cache is rendered through multiple passes (Section 6). We will give some results (Section 7) that we will discuss (Section 8).

2 PREVIOUS WORK

Our approach rests on well-known techniques: irradiance caching, progressive refinement and use of a triangular mesh for hardware interpolations, uncoupling of rendering and lighting computations, storage of illumination samples in an object-space hardware-rendered mesh, and hardware irradiance sampling. All

these techniques will be developed in the next subsections.

Irradiance and radiance caching

Computing high quality indirect irradiance samples for each pixel of an image is very costly. Fortunately, indirect irradiance changes very slowly over a surface. Irradiance caching, introduced by Ward et al. [War88], takes advantage of this property by interpolating indirect irradiance between a few fully computed and cached samples.

Zaninetti et al. extended this method proposing *light vectors* [Zan98]. Instead of caching irradiances, they took into account the BRDF of the objects and stored radiances, allowing glossy objects to be rendered. Noticing that direct, indirect and caustic components of radiance have different frequencies, they sampled and cached each one separately. In their method, rendering is a two pass process. First a seed of samples is generated, then the scene is rendered using those already computed samples to interpolate radiance in between. With this technique, samples are cached in a kd-tree and interpolations are computed using a various number of samples. Those interpolations eventually lead to noisy radiance values. The major problem of this method when trying to implement an interactive renderer, is that radiance samples are view dependent, hence they cannot be re-used from frame to frame.

To overcome this difficulty, Crespin and Péroche [Cre04] extended the *light vectors* cache and used a five dimensional cache where light vectors are computed and cached for many viewing directions. While rendering, light vectors are interpolated according to the viewer's position. However, none of these caching methods is able to provide interactive rendering, and they all require a costly first pass.

Progressive rendering

Progressive rendering algorithms make it possible to bypass a long preprocessing. This comes at a cost: the first rendered images are only coarse approximations, but they may provide useful information to the user who is no more required to wait. The *light vectors* method caches samples in an object space structure. Another solution is to work in the image space. This way, Painter and Sloan [Pai89] proposed to partition the image as a 2D-tree whose leaves are recursively subdivided according to the number of pixels they cover and the variance of the samples they contain.

The idea of working in the image space was taken up by Pighin et al.[Pig97]. They create a triangulation of the image from a set of samples taken around the discontinuities so that these discontinuities can correctly be rendered. The generated triangles are quickly rendered using the graphics hardware that interpolates between the samples. This technique is not interactive but it allows a quick preview of a scene.

The *Tapestry* method proposed by Simmons and Séquin [Sim00] provides interactivity through the use of a 2D and a half mesh. This mesh is projected onto a sphere around the observer: this way, full recomputation of the image can be avoided when the observer moves only a little. Nevertheless, geometric and lighting discontinuities remain fuzzy at the beginning of the rendering process.

Rendering and sampling uncoupling

To reach interactivity, Walter et al. [Wal99] proposed to separate illumination sampling from rendering. The sampling process computes light samples and stores them in a *render cache*, while the rendering process uses those samples to generate the current one. The *render cache* keeps old samples from the previous frame that are reprojected in the current frame by the rendering process. Nonetheless, this image based method suffers from artefacts when the user views some part of the scene that has never been rendered before.

To overcome the *render cache* artefacts, Tole et al. [Tol02] proposed to compute and store shading values in the object space, more precisely at the vertices of a progressively refined mesh. Their *shading cache* is gradually filled while a process renders the image using the graphics hardware. This method does not suffer from reprojection artefacts. Even if the illumination is coarsely computed, the scene geometry and textures are rendered at full quality providing the user much more pleasant images than the *render cache*. This method uses a unique mesh to approximate radiance from different sources, leading to unnecessary sampling and limited re-use of already computed samples. Our method is greatly inspired by this last one, but as explained in the upcoming sections, we use more than one mesh to overcome the *shading cache* limitations.

Dmitriev et al. [Dmi02] also reach interactivity by separating the rendering of hardware computed direct lighting from the rendering of indirect lighting computed by path tracing. Photons are stored at the vertices of a dense mesh of the scene. Photons are packed around a *pilot photon* that has a path close to theirs. When the scene is modified, *pilot photons* are traced again through the scene to detect changed areas that need resampling.

Hardware sampling

Graphics hardware is getting more and more programmable at each new generation, allowing new usages of GPUs. Purcell et al. [Pur02] showed that it is possible to interactively ray trace images using the GPU to compute ray triangles intersections. Global illumination can also be computed with the *photon map* algorithm [Jen96] using the GPU as proposed by Purcell et al. [Pur03]. Those methods use *fragment programs* and *render to texture* functionality of current hardware but are not really faster than the same algorithms implemented on a CPU.

Larsen and Christensen [Lar04] make a more useful usage of both the CPU and the GPU, giving each one some work. Their method separates the rendering of direct and indirect illumination. Direct illumination is computed by the hardware. Indirect illumination is computed using a *photon map*. Photons are traced on the CPU whereas the final gathering step is made on the GPU. This method reaches interactivity but according to the authors results, indirect illumination is very sparsely sampled.

3 OVERVIEW OF OUR METHOD

The framework we propose makes use of the irradiance cache idea. Like Zaninetti and al. [Zan98], we store each irradiance component separately. Like Tole and al. [Tol02], our sample cache is built on the geometrical mesh. This irradiance cache is filled with hardware sampled direct and indirect irradiances. The cached irradiance values are used to progressively and adaptively build and refine a radiance mesh. Final images, displayed to the user, are rendered through multiples passes, mixing radiances with the object colors. (Fig. 1) gives an overview of our framework that will be explained with more details in the next sections.

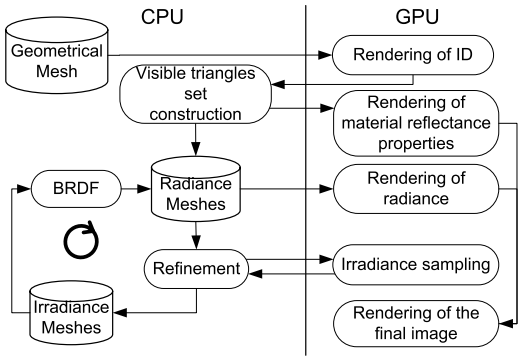

Figure 1: Overview of our framework

4 MULTI-MESH CACHING

Our strategy to reach interactivity is to compute as few radiance samples as possible. With this aim in view, we need to re-use samples from frame to frame and interpolate between samples in the same frame. More than that, we want to re-use part of already computed samples. While computing indirect irradiance, the direct one is needed. Hence, we chose to split direct and indirect irradiance computations and storage in order to be able to use the direct irradiance to compute the indirect one.

Splitting irradiance and radiance

We want to progressively compute and render samples. A recursively subdivided triangular mesh allows us to easily refine the sampled radiance field and to quickly update the image using the graphics hardware. The traditional object space cache approach is a unique irradiance cache that mixes direct and indirect irradiance, limiting their re-use and leading to unnecessary sampling. To overcome those limits, we chose to use more than one cache mesh. Moreover, we want to distinguish irradiance (energy incoming from any directions) from radiance (energy emitted in a particular direction). Irradiance is long to compute but can be reused as long as the scene does not change, while radiance is fast to compute using cached irradiance values (see Fig. 2). We currently use:

- one direct irradiance mesh for each light source
- one indirect diffuse irradiance mesh
- one direct diffuse radiance mesh
- one direct specular radiance mesh

All those meshes are built over the same geometrical mesh. When a triangle is subdivided, it is always split in four, its edges being split in half.

We store in separate meshes the direct irradiance of each light source. This allows us to save computations when the direct irradiances of different light sources do not have the same discontinuities. In a given area, only the meshes whose irradiance contains discontinuities are refined. We use another mesh for indirect irradiance that has far less sharp discontinuities than the direct ones. The irradiance meshes can be seen as illumination maps. The direct ones contain direct shadows, while the indirect one contains color bleedings and indirect shadows. The mesh representation is more suited to store soft shadows as sharp discontinuities lead to deeper subdivision, but it is nevertheless able to handle sharp shadows of point light sources.

What we need to render is the radiance emitted by the seen objects that reaches the eye of the observer. Radiance can be divided in two parts: direct radiance and indirect one. Using a separable BRDF model, direct radiance can also be split in two parts: the diffuse one and the specular one. The diffuse part of direct radiance does not change when the user moves. Thus it can be computed once, cached in the vertices of a

mesh and re-used for many different frames. We use a mesh to progressively compute the specular part of direct radiance for each frame, starting from scratch when the observer moves. Indirect radiance is very long and difficult to compute if all kinds of light paths are taken into account. We currently only take into account indirect diffuse radiance that can be directly computed, on the fly while rendering, from indirect diffuse irradiance, so we don't use another mesh for indirect irradiance.

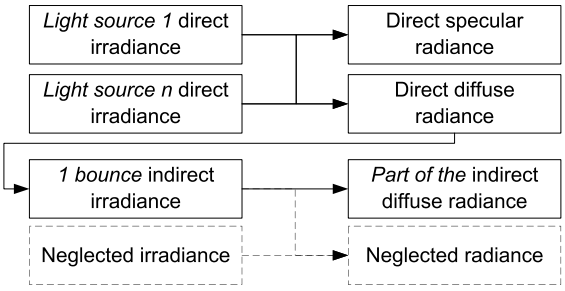

Figure 2: From irradiance to radiance

From irradiance to radiance

We chose the modified Phong BRDF [Laf94] for its simplicity and its energy conservation property, but other separable BRDFs could be used. The radiance at point x incoming from direction $\vec{\omega_r}$ we need to compute and display is

$$L_d(x,\vec{\omega_r}) = \int_\Omega f_d(x,\vec{\omega_r},\vec{\omega_i})L(x',\vec{\omega_i})cos\theta\ d\omega \quad (1)$$

$$L_s(x,\vec{\omega_r}) = \int_\Omega f_s(x,\vec{\omega_r},\vec{\omega_i})L(x',\vec{\omega_i})cos\theta\ d\omega \quad (2)$$

where f_s and f_d are the diffuse and specular components of each object BRDF.

In the direct irradiance mesh, we store for each light source ls of area S the irradiance

$$E_d(x,ls) = \int_{ls} L(x',\vec{\omega_i})cos\theta\ cos\theta'\ \frac{dS}{r^2} \quad (3)$$

The visibility of the light source is taken into account in the computation of $E_d(x,ls)$. Using the modified Phong BRDF, diffuse direct radiance can easily be computed from the irradiance cached for each light in the irradiance mesh:

$$L_{dd}(x,\vec{\omega_r}) = \frac{1}{\pi}\sum_{lights} E_d(x,ls) \quad (4)$$

To compute the specular part of the direct radiance we should integrate:

$$\frac{\text{n+2}}{2\pi}cos^n\alpha\ L(x',\vec{\omega_i})cos\theta \quad (5)$$

over the solid angle sustained by each light source, where n is the specular exponent and α the angle between the perfect specular reflection of the light source center and the outgoing direction $\vec{\omega_r}$. This would be too costly for interactive rendering. We want to re-use already computed samples, so we use the irradiance stored in the irradiance mesh for each light source and multiply it by the specular part of the BRDF evaluated at the center of the light source:

$$L_{ds}(x,\vec{\omega_r}) = \sum_{lights}(\frac{\text{n+2}}{2\pi}cos^n\alpha\ E_d(x,ls)) \quad (6)$$

This simplification comes down to replacing area light sources with point light sources for specular radiance estimation. The resulting errors can be misplaced and wrong shaped specular highlights.

Sampling all the components of indirect irradiance is currently too costly, we only sample a part of diffuse indirect irradiance: light paths that diffusely bounce only once between the light source and the point of interest. Paths with two bounces could easily be added using the one-bounce indirect irradiance being computed; longer paths can often be neglected since in a directly lit scene, their contributions are very small. This method is biased but provides visually satisfying images. Diffuse indirect irradiance is approximated using the diffuse direct radiance:

$$E_i(x) = \int_\Omega L_{dd}(x',\vec{\omega_i})cos\theta\ d\omega \quad (7)$$

To compute the diffuse indirect radiance, we just compute:

$$L_{id}(x) = \frac{1}{\pi}E_i(x) \quad (8)$$

Mesh subdivision

The first image displayed is rendered using radiance and irradiance that are only computed at the vertices of the radiance meshes roots. These roots are in fact the geometrical mesh elements of the scene. The radiance meshes are then progressively refined to take into account radiance discontinuities. Each time a radiance is computed, irradiance values are fetched, for each light source, in the corresponding irradiance meshes. When those values are not available or cannot be confidently interpolated, the irradiance meshes are subdivided. The radiance meshes refinement guides the direct irradiance meshes ones. We will explain in the next subsections how our meshes are subdivided.

4.3.1 Radiance mesh subdivision criteria

Each mesh element is a triangular patch. A radiance patch may be subdivided only if it lies over more than

one pixel. To decide if a mesh element should be subdivided, each of its edges is split into two equals parts. Two radiances are evaluated at its middle: one is computed and the other is interpolated. If the computation and the interpolation give a close result, the edge will not be subdivided any more. When at least one edge of a triangle has been subdivided because it contains a discontinuity, the triangle is split into four triangles. This avoids visible T-vertices, since triangles on both sides of the subdivided edge will be split. A size criterion is required in order not to miss small radiance discontinuities: triangles are subdivided until they cover a small number of pixels. Once triangles have been subdivided enough to meat the size criterion without finding any discontinuities, the unnecessary subdivisions are undone to save memory space.

4.3.2 Noticeable color differences

To compare interpolated and sampled radiances at the middle of an edge, we need a criterion that takes into account the user perception. The maximum unnoticeable difference between the two values depends on the radiance of the rendered pixel on screen, the user visual system, the monitor settings and its environment. Modeling this whole chain is by itself a research area; we wanted something simple. The problem we ran on was that a lot of useless mesh subdivisions occurred in dark areas of the scene. On our monitor, we cannot distinguish a black (*0x000000*) patch from a lighter black (*0x0A0A0A*) one. Our algorithm has to take this fact into account. Supposing the user, the environment and the monitor do not change, we experimentally generate a map that associates a maximum unnoticeable difference value for a set of radiances. To compare interpolated and sampled radiance, we first process these values with the tone mapping algorithm to get an idea of the color values that will be rendered on the screen. Then we compute the difference and compare it with the corresponding value in our map. A real visual model would give far more accurate values but at the expense of a high computation cost. Our solution gives acceptable results at almost no cost.

4.3.3 Priorities

To provide the best quality images in the shortest time, the radiance mesh elements are given subdivision priorities. These priorities depend on the radiance difference between the element vertices and on the element visible size: a high contrast over an element is a good hint for radiance discontinuities and the bigger an element is, the more chance it has to contain discontinuities. We use the visible size of the triangle because we don't want to subdivide hidden triangles. The mesh can be seen as a set of quad-trees, each quad-tree being built over a triangle of the geometrical mesh. The priorities are computed at the leaves of the quad-trees and are spread to their root. The priority of a node is the maximum of its sons priority. This way, given a geometrical mesh triangle, we can quickly find its radiance element leaf that requires to be subdivided first.

To decide which geometrical triangle will be subdivided, we could sort the elements or use a hit and test method as in [Tol02]. Sorting is too slow and the hit and test method requires many tests. Instead, we developed the following algorithm. We start the subdivision process with a big triangle which has a quite high priority. Then we randomly pick a triangle; if the picked triangle has a higher priority than the last subdivided one, we switch to the picked one, otherwise we keep refining the same triangle until its priority falls under the priority of the next randomly picked triangle.

4.3.4 Irradiance mesh subdivision

The subdivision of the direct irradiance meshes is guided by the subdivision of the radiance mesh described before. Each time a direct radiance value needs to be computed, the irradiance of each light source is fetched in the irradiance mesh. If the direct irradiance mesh is not subdivided enough to provide the requested value, it is subdivided at this time. The irradiance mesh subdivision is limited by the subdivision of the radiance one, thus an element will not be indefinitely subdivided.

Diffuse indirect radiance and diffuse indirect irradiance are directly linked (equation 8). As the radiance is computed to be displayed, we use the same view dependent criteria that was used to subdivide the radiance meshes.

5 SAMPLING

Our mesh-based progressive rendering approach is very sensitive to noise. Noisy samples may lead to needless subdivisions of the meshes. To quickly render high quality images, we need to compute high quality irradiance values to fill our irradiance caches. As told earlier, we sample direct irradiance from each light source and indirect irradiance separately. Traditionally, ray tracing was used to collect irradiance. Area light sources were sampled using hundreds of rays to determine their visibility from the point to shade. Indirect irradiance was collected by sampling an hemisphere built over the point to shade. These methods provide good results but are often too slow to be useful in interactive rendering on a single CPU. Furthermore those methods monopolize the CPU and make no use of the GPU. Using the GPU as a SIMD coprocessor to compute ray object intersections is feasible but at a high cost: ray packing and asynchronous

results handling. Our idea is to use the GPU in a more regular way.

Area light sources sampling

We propose to take advantage of the efficient visibility determination capacity of the graphics hardware to obtain a high quality estimation of the irradiance of an area light source at a given point. We render the scene observed from the point to shade, clipping the viewing frustum to a small frustum that fully includes the area light source.

Assuming that the light sources are isotropic, their radiance is the same all over their surface. We need to compute equation (3). Our method uses a fragment program to compute accurate values: $cos\theta$ and the solid angle of the pixel are evaluated for each pixel. To sum the fragment program output values represented and stored as floats in a pixel buffer, we use two pixel buffers to progressively reduce the image until all the values are accumulated in a small enough image (16x16) that can be quickly read back to the CPU.

When the scene is rendered from the point to shade, the whole scene does not need to be sent to the GPU, since only a few triangles in the viewing frustum can occlude the light source. We use a grid to store the scene triangles. Computing the exact intersection of the frustum and the grid could be quite time consuming, so we chose to approximate the frustum with a carefully chosen bounding cone and each grid cell with a bounding sphere. To know if the cone intersects a grid cell, only one test is required: does the grid cell center belongs to the bounding cone? (see Fig. 3). This method is conservative, a few grid cells are being falsely detected as crossed by the frustum, but none are forgotten.

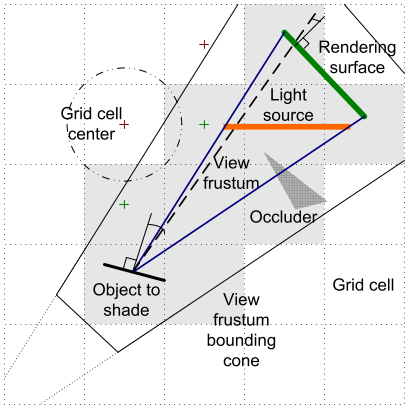

Figure 3: Area light source sampling

Indirect irradiance sampling

Sampling indirect irradiance through an hemisphere using a ray tracing method is far more time consuming than sampling the direct one. More rays are needed and those rays are not as coherent as those sent when sampling direct irradiance over an area light source. This makes CPU SIMD optimizations like those proposed by Wald et al. [Wal01] less efficient. As for sampling direct irradiance, we chose to use the graphics card high visibility determination capacity.

As said earlier, we currently only take into account in our indirect irradiance computations light paths with one diffuse bounce. To gather this indirect irradiance, as Larsen and Christensen [Lar04], we chose to render the scene only once on a plane. Using a field of view of 160 degrees, we forget only **2**% of the incoming radiance. Rendering the whole diffuse radiance mesh would be very costly and useless. If the scene is sampled using a 128x128 image, small triangles in the radiance mesh won't be visible. We chose to send an undersampled version of the radiance mesh to the GPU to increase speed. This coarse version of the direct diffuse radiance mesh has to be built for the whole scene before starting to sample indirect radiance since any part of the scene could indirectly contribute to the indirect radiance of a visible geometrical mesh element. The energy is collected in the image by summing it with two pbuffers as we did for direct irradiance. Again, the value of each pixel has to be weighted with the solid angle covered by the pixel and with the $cos\theta$ term of the irradiance formula. This time, the solid angle of field of view is constant so the weight of each pixel can be precomputed and stored in a texture.

6 MULTI-PASS RENDERING

Generating an image with our method is done through multiple rendering passes. In a first pass, we only render the geometrical mesh using an identifier (a 32 bits address, or an index in an array) for each triangle as color. The generated image is read back to the CPU. This is costly, but this allows us to compute the visible size of each triangle and to build the set of the visible triangles. The occlusion query extension could have been used to avoid the read back, but to get the exact number of visible pixels of each triangle, two rendering passes would have been needed: the first one to initialize the z-buffer, the second one to count visible pixels.

Using the visible triangles set as input, the second pass generates two images containing the material reflectance properties of each triangle. We split in two images the diffuse and specular properties. These material reflectance properties are the diffuse or specular colors of each triangle times their diffuse or specular reflectance coefficient. Those two images can be generated in a single pass using multiple output buffers or packing the two images in a single output texture.

With the number of visible pixels computed in the first pass, we update the priority of each radiance mesh element. This radiance mesh can be rendered in a third pass. Actually, two passes are needed to render the diffuse and the specular direct radiance meshes, and one more pass is needed to render the diffuse indirect irradiance one whose irradiances are transformed on the fly in radiances according to equation (8).

A final pass is needed to merge all the material properties images and the radiance ones. This last pass is very fast, it requires to render only a single quad textured with the five images computed in the preceding passes. According to the number of texture units available on the graphics card, more than one pass may be needed. The fragment program used in this last pass includes a tone mapping algorithm to convert the radiance of each pixel into a displayable color.

This image decomposition may seem costly, but it allows to progressively update the image almost as fast as new samples are computed. When the user keeps looking at the same frame, the mesh subdivision thread works at full speed. Each time a radiance mesh element is subdivided, the four triangles created are rendered on the image corresponding to the subdivided mesh (direct diffuse, direct specular or indirect diffuse). The final pass that recombines all the radiance components and the object reflectance is applied at constant time rate to update the image displayed to the user.

7 RESULTS

We tested our method on a P4 2.5 GHz - 768MB of RAM - Nvidia 256MB QuadroFX3000. The following results are computed on a small scene of 7000 triangles with one area light source. About fifteen seconds are required to compute the coarse direct radiance mesh. During this time a constant indirect radiance value is used while direct radiance is progressively subdivided. Then, direct and indirect radiance meshes are simultaneously refined and the image is progressively updated.

The first rendering pass that requires a read back to the CPU of the rendered triangles identifier is the costliest. Working with a 640x480 image requires 40ms. The material reflectance property and radiance rendering passes are faster: about 12ms each. The final pass that merges the luminance and material reflectance images is also fast: 10ms. Globally, 100ms are required to obtain an image using already computed samples.

On our test scene, the coarse direct radiance mesh used to compute the indirect one contains 9200 samples. To obtain the image (Fig. 4) 26000 direct and 9000 indirect samples are required. The total number of computed samples is linked to the subdivision criteria.

The direct irradiance sampling requires two phases: rendering the objects that might be in the point to area light source frustum and summing the irradiance of each pixel. The current read back limits make the second phase the costliest one (about 70% of the total time). Using a 64x64 image to sample direct irradiance requires about 0.9ms. Sampling the light source with a ray tracer with the same density requires 32ms. With a ray tracer, less rays are needed since non uniform sampling (that provides less banding artifacts) can be used. Ray tracing with 16x16 rays requires 2.6ms, which is the cost of hardware sampling using a 128x128 image. Hardware sampling is a lot faster than ray tracing and it frees the CPU that can be used for other tasks, like mesh subdivision, while samples are computed. Indirect irradiance sampling is a bit longer as all the scene is sent to the GPU. 9ms are required for each sample using a 256x256 pixels image.

8 DISCUSSION AND FUTURE WORK

We think our method has interesting advantages over similar ones. It does not require preprocessing as *photon map* based methods do. Sampling radiances at the vertices of a mesh avoids the costly interpolations of *light vectors* methods or the density estimations of the *photon map*. Our caching method, splitting irradiances and radiances, allows more re-use of already computed values. Our area light source sampling method, that uses different meshes to progressively compute each light source shadow, limits the number of computed samples to a minimum. We chose not to sample direct radiance using traditional interactive GPU based methods (shadow maps or shadow volumes) that might be faster, but would not allow to store the computed radiance in RAM for use in the indirect radiance computations. Our sampling method is not yet able to produce real time soft shadows nor real time indirect illumination, but it is a lot faster than ray tracing based methods. Upcoming GPU extensions might allow us to keep the direct radiance on the GPU board avoiding the GPU to CPU transfer bottleneck.

The strong link between the caches and the geometry can be criticized for its lakes of freedom in the scene representation, but this is the key for interactive speed. The major problem of our solution is its memory cost. Rendering large scene containing very tessellated objects is a problem since we construct our meshes over the geometrical one. A solution we are working on is the construction of the radiance meshes on the object bounding boxes, building cubemaps to render the objects radiances. Other currently missing features we are working on include specular reflections, caustics and dynamic scene handling. We anticipate that our use of different meshes should help us to notice

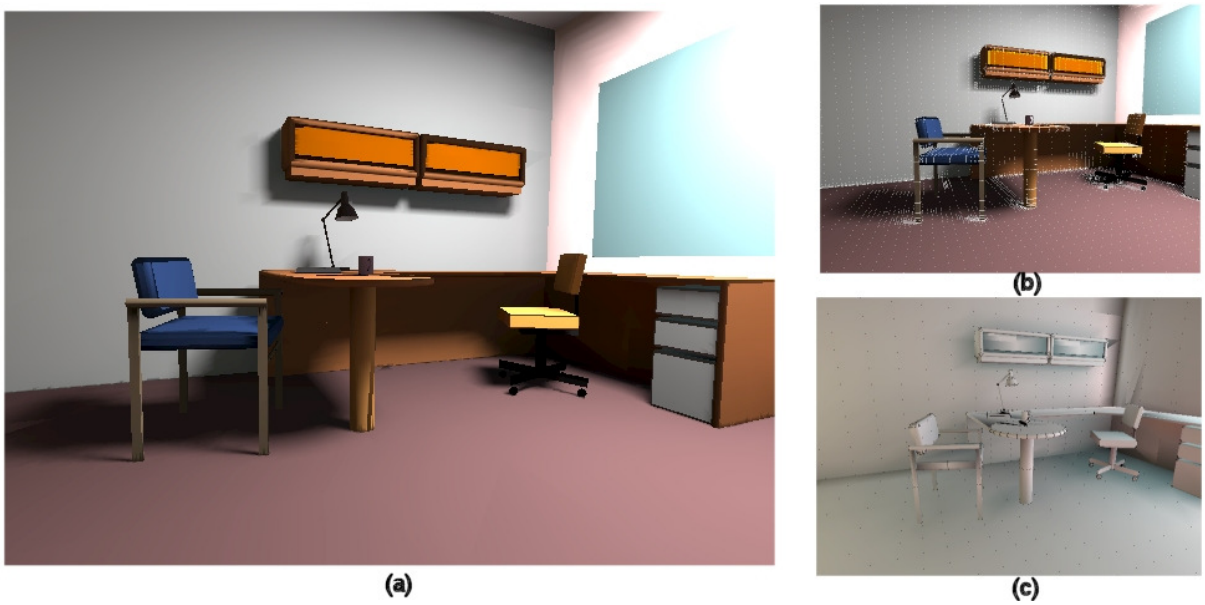

Figure 4: (a) Final image (b) Direct radiance samples (c) Indirect radiance image and samples

changes in dynamic scene irradiance, allowing us to quickly update our irradiance caches. The use of a visual model to guide our mesh subdivision is another research area we want to explore.

9 CONCLUSION

Real time rendering of large scenes with soft shadows, indirect illumination and caustic is not achieved yet. We think our multi-mesh method is an interesting step towards this goal. The fast increase in performance and functionality of GPUs will offer us new possibilities. In our mind, porting to the GPU existing CPU algorithms does not take full advantage of the graphics hardware. GPUs are fast for geometry rasterization and interpolations; its limited programming and memory model are quickly evolving, we have to foresee new algorithms to exploit those upcoming capacities to compute shadows and global illumination.

References

[Cre04] Crespin, R., and Péroche, B. Lights Vectors for a Moving Observer. In *12-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG), Plzen , Czech Republic*, pages 99–96, Plzen , Czech Republic, 2-9 february 2004. University of West Bohemia.

[Dmi02] Dmitriev, K., Brabec, S., Myszkowski, K., and Seidel, H.P. Interactive global illumination using selective photon tracing. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pages 25–36, June 2002.

[Jen96] Jensen,H.W. Global illumination using photon maps. In *Eurographics Rendering Workshop 1996*, pages 21–30, June 1996.

[Laf94] Lafortune, E.P., and Willems, Y.D. Using the Modified Phong BRDF for Physically Based Rendering. Technical Report CW197, Leuven, Belgium, 1994.

[Lar04] Larsen, B.D., and Christensen, N. Simulating photon mapping for real-time applications. In *Eurographics Symposium on Rendering*, jun 2004.

[Pai89] Painter, J., and Sloan, K. Antialiased ray tracing by adaptive progressive refinement. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 281–288, July 1989.

[Pig97] Pighin, F.P., Lischinski, D., and Salesin, D.H. Progressive previewing of ray-traced images using image plane discontinuity meshing. In *Eurographics Rendering Workshop 1997*, pages 115–126, June 1997.

[Pur02] Purcell, T.J., Buck, I., Mark, W.R., and Hanrahan, P. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002.

[Pur03] Purcell, T.J., Donner, C., Cammarano, M., Jensen, H.W., and Hanrahan, P. Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003*, pages 41–50, July 2003.

[Sim00] Simmons, M., and Séquin, C.H. Tapestry: A dynamic mesh-based display representation for interactive rendering. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 329–340, June 2000.

[Tol02] Tole, P., Pellacini, F., Walter, B., and Greenberg, D.P. Interactive global illumination in dynamic scenes. *ACM Transactions on Graphics*, 21(3):537–546, July 2002.

[Wal01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, 20(3):153–164, 2001.

[Wal99] Walter, B., Drettakis, G., and Parker, S.. Interactive rendering using the render cache. In *Eurographics Rendering Workshop 1999*, pages 19–30, June 1999.

[War88] Ward, G.J., Rubinstein, F.M., and Clear, R.D. A ray tracing solution for diffuse interreflection. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, volume 22, pages 85–92, August 1988.

[Zan98] Zaninetti, J., Serpaggi, X., and Péroche, B. A vector approach for global illumination in ray tracing. *Computer Graphics Forum*, 17(3):149–158, 1998.