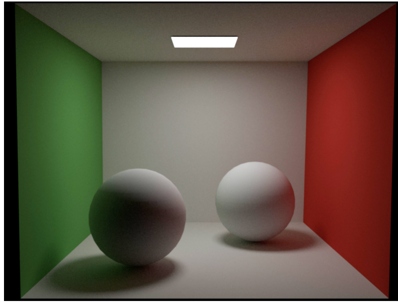
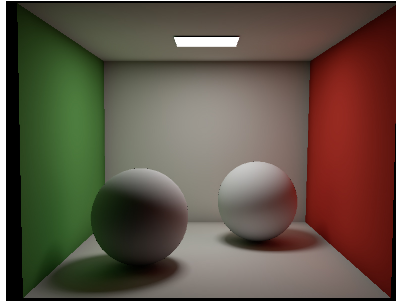
Metropolis Virtual Point Light Rendering

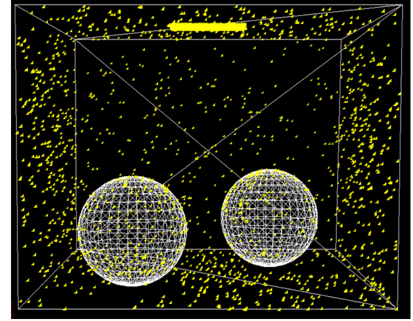
Sebastien Speierer
Semester project, EPFL, RGL



(a) *Path tracer rendering, 256 samples, 270 seconds*



(b) *VPL Rendering 5000 vpls, 10 seconds*



(c) *VPLs distribution*

Figure 1: *The Virtual Point Light technique is a fast rendering methods with results close to the one computed with conventional path tracers.*

Abstract

Solving the rendering equation is a well known problem in computer graphics and there exists plenty of methods to compute physically-based images. However, most of these methods aren't fast enough to allow the user to interact with the scene in real-time.

Virtual Point Lights (VPL) efficiently approximate the global illumation of a given scene and eventually converge to a correct solution. The method proposed in this report combines the rasterization power of modern GPUs and the robustness of Metropolis sampling algorithms to achieve global illumination. In contrast to similar approaches, we propose a fast and scalable system able to render complex scenes and materials such as Microfacet models [Walter et al. 2007].

A second purpose of this project was to implement a state-of-the-art VPL rendering system that would be later ported to the well-know Mitsuba renderer as a preview tool.

Keywords: virtual point light, global illumination, physically-based rendering

1 Introduction

[Keller 1997] proposed a technique for instant rendering called *Instant Radiosity* which, based on quasi-random walk, generates a set of virtual point light sources to describe the incoming radiance field and use it to illuminate the scene. His goal was to exploit graphics hardware to solve the global illumination problem. Several researchers developed extensions of this. [Hašan et al. 2009] introduced a new light type called VSL increasing performance on scenes with glossy materials and complex illumination. A robust importance sampling method has been presented in [Georgiev and Slusallek 2010]. [Segovia et al. 2006] built a bidirectional estimator to generate VPLs in a more efficient manner. Later, they went a step futher, incorporating a Metropolis-Hasting sampler in their method to increase its robustness and performance [Segovia et al. 2007].

Our implementation includes many of these ideas with a specific focus on the Metropolis sampling algorithm. Based on the VPL renderer implemented in Mitsbuba v0.5, we attempt to push it a step forward in terms of flexibility and performance.

The remainder of this paper is organized as follow. Section 2 introduces the basic of the VPL technique and how it uses the GPU. We also give a quick overview of the light transport equation and its implication in our algorithm. In section 4, we describe in details the implementation of our methods and the different issues encounter during the development phase. The mechanisms of the Metropolis-Hasting sampler are explained in section 3. This section also shows results and compares the two samplers in different scene configurations. Other results are presented in section 5 and a conclusion is finally given in section 6.

2 Virtual Point Light method

Like a path tracer, the VPL technique tries to solve the Light Transport problem. In order to increase performance and later allow real-time user interaction with the scene, the use of the GPU is a key point of this method. In this section, we will to describe the differences between a conventional path tracer and A VPL renderer to better understand the functionning of this method.

2.1 The Light Transport Equation

The rendering equation first proposed by [Kajiya 1986] exists in a range of different formulations. Here is a simple recursive formulation that will help us understand our method:

$$L(x' \to x'') = L_e(x', x'') + \int_{M^2} L(x \to x') G(x, x') f(x, x', x'') dA(x) \quad (1)$$

where

- $L(x' \to x'')$: incoming radiance at x'' from x'
- $L_e(x', x'')$: emitted radiance at x' in the direction of x''
- $f(x, x', x'')$: BRDF at x'
- M^2: scene surface
- $G(x, x')$: geometric term between x and x'

In words, we could translate this formula as: *The incoming radiance getting to x'' from x' is the sum of the emitted light at this point and the accumulation of the incoming radiance from every piece of surface in the scene to x', weighted by the material properties and a geometric term.*

By recursion, this equation handles all possible length of light paths in the scene, going from the light sources to the eye.

In the scene, a path is described as a sequence of points. Let P be the space of all possible paths with their origin on the surface of one of the light source in the scene. We define $\bar{\mathbf{x}} \in P$ a path with x_i its ith points and x_k its final point. We also define $P^{\mathbf{x}}$ a subset of P such as

$$P^x = \{\bar{\mathbf{x}} : \bar{\mathbf{x}} \in P \text{ and } x_k = x\}$$

and P_n a subset of P with only paths of length n.

Using this notation, we can reformulate equation 1:

$$L(x' \to x'') = L_e(x', x'') + \sum_{k=1}^{\infty} \int_{M^2} G(x, x') f(x, x', x'') \int_{P_k^x} f(x_{k-1}, x, x') L(\bar{\mathbf{x}}) d\bar{\mathbf{x}} dA(x) \tag{2}$$

With $L(\bar{\mathbf{x}})$ the light carried along the path $\bar{\mathbf{x}}$. Based on equation 1 we can define the integration over the path space as

$$\int_{P_k^x} f(x_{k-1}, x, x') L(\bar{\mathbf{x}}) d\bar{\mathbf{x}} = \underbrace{\int_{M^2} \dots \int_{M^2}}_{k-1 \text{ times}} f(x_{k-1}, x, x') S(x_1, ..., x_k) dA(x_1) ... dA(x_{k-1})$$

with

$$S(x_1, ..., x_k) = L_e(x_1, x_2) \prod_{i=1}^{k-1} G(x_i, x_{i+1}) \prod_{j=1}^{k-2} f(x_j, x_{j+1}, x_{j+2}) \tag{3}$$

Considering a point x'' in the scene and equation 1, we differentiate

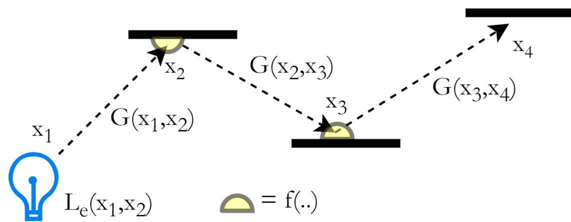


Figure 2: *Illustration of equation 3*

two types of incoming radiance. The first term in equation 1 is called the *direct illumination* while the second term is called the *indirect illumination* as illustrated on Figure 3.

$$L(x' \to x'') = L_e(x', x'') + L_i(x', x'') \tag{4}$$

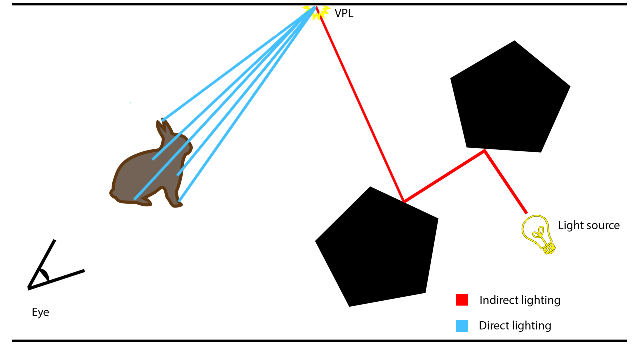


Figure 3: *Complete light path from a light source to the screen*

2.2 Using the GPU

As said before, we will use the to computational power of today's GPUs and the flexibility of the OpenGL 4.0 framework to increase perfomance in solving the Light Transport equation. In fact, GPUs are really good at rendering direct lighting for a given scene. More precisely, it is possible to get a very accurate computation of the direct lighting from a point light in the scene, and this is where VPLs come into play.

The main idea behind the VPL technique is to use a conventional path tracer to compute indirect lighting reaching a point x' in the scene and convert it into a virtual point light source. It will then use the GPU to compute the corresponding indirect lighting term of $L(x' \to x'')$ for each point x'' in the scene seen from the camera.

What's interesting here is that the GPU will compute the indirect lighting contribution for every pixel at the same time thanks to its highly parallel nature.

In order to compute the indirect lighting term in equation 4, we use a Monte Carlo approach to discretize this formula.

$$\begin{aligned} L_i(x', x'') &\approx \sum_{k=1}^{\infty} \int_{M^2} G(x, x') f(x, x', x'') f(x_{k-1}, x, x') \left(\frac{1}{N} \sum_{i=1}^{N} \frac{L(\bar{\mathbf{x}}_{i,k})}{p(\bar{\mathbf{x}}_{i,k})} \right) dA(x) \\ &\approx \frac{1}{N} \frac{1}{M} \sum_{k=1}^{\infty} \sum_{i=1}^{N} \sum_{j=1}^{M} G(x_j, x') f(x_j, x', x'') f(x_{k-1}, x_j, x') \underbrace{\frac{L(\bar{\mathbf{x}}_{i,k})}{p(\bar{\mathbf{x}}_{i,k})} \frac{1}{p(x_j)}}_{\text{path tracer}} \end{aligned}$$

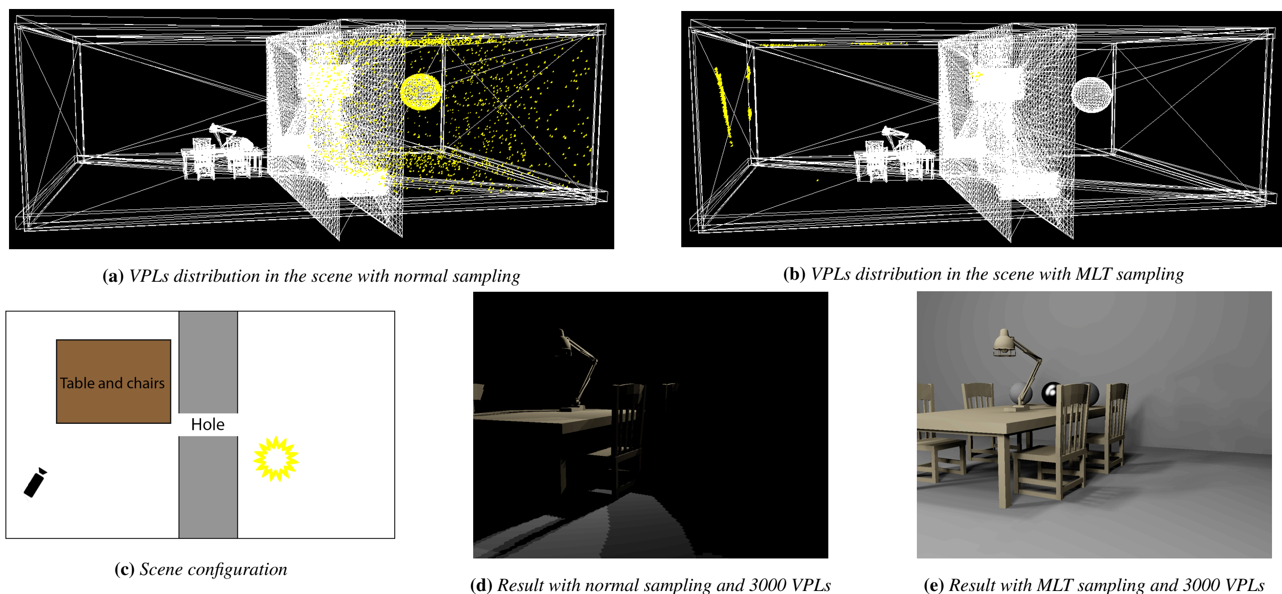
with $p(\bar{\mathbf{x}}_{i,k})$ and $p(x_j)$ the probability of sampling a path and a point respectively.

This method generates one path at a time and accumulates its radiance contribution to the scene to progressively converge to the correct solution. The path and its carried energy is computed using a path tracer. For a given path $\bar{\mathbf{x}}_i$, we create a VPL that will contribute to the point x'' as follow:

$$\delta L_i(x', x'') = G(x_k, x') f(x_k, x', x'') f(x_{k-1}, x_k, x') \underbrace{\frac{L(\bar{\mathbf{x}}_{i,k})}{p(\bar{\mathbf{x}}_{i,k}) p(x_k)}}_{\text{path tracer}} \tag{5}$$

By computing many VPLs, it will eventually converge to the right solution for indirect illumination.

In equation 5, the path tracer takes care of the computation of $\frac{L(\bar{\mathbf{x}}_{i,k})}{p(\bar{\mathbf{x}}_{i,k}) p(x_k)}$. The computation of the three other terms will be done on the GPU as described in sections 4.2 and 4.3.



(a) *VPLs distribution in the scene with normal sampling*

(b) *VPLs distribution in the scene with MLT sampling*

(c) *Scene configuration*

(d) *Result with normal sampling and 3000 VPLs*

(e) *Result with MLT sampling and 3000 VPLs*

Figure 4: *Comparison of the Metropolis Light Transport VPL sampling method and the normal VPL sampling method*

3 Metropolis-Hastings algorithm

The Monte Carlo sampler is good at sampling VPLs in simple scenes. However, it is inefficient in quickly finding the relevant light transport paths in more complex scenes. As described in the example in Figure 4c, the only light paths that will have a contribution to the final rendered image have to go through the hole in the wall. Unfortunately, a conventional Monte Carlo sampler will lose most of its time sampling VPLs in the second room (where the light source is) as you can see on Figure 4a. Only a few VPLs will make their way to the other side of the wall which results in a really dark image as shows Figure 4d. This is due to the fact that the VPLs sampled in the second room don't illuminate the part of the scene view from the camera.

[Veach and Guibas 1997] proposed a extention of the Monte Carlo method inspired by the Metropolis sampling algorithms in computational physics. These algorithms are generally used to approximate distributions with high dimentions, which suits well with our requirement since the light path space has several dimensions per bounces. The Metropolis-Hastings algorithm, also sometimes called Markov Chain Monte Carlo algorithm, samples statistically correlated paths. Each new sampled path $\bar{\mathbf{x}}'$ with this method results from the mutation of the last accepted sampled path $\bar{\mathbf{x}}^{(i-1)}$. Then, with probability α, the newly sampled path is accepted or rejected:

$$\bar{\mathbf{x}}^{(i)} = \begin{cases} \bar{\mathbf{x}}' & \text{with probability } \alpha \\ \bar{\mathbf{x}}^{(i-1)} & \text{otherwise} \end{cases}$$

The acceptance probability depends on the contribution this VPL will bring to the scene and is computed in the following way:

$$\alpha = \frac{I(\bar{\mathbf{x}}')}{I(\bar{\mathbf{x}}^{(i-1)})}$$

where $I(\bar{\mathbf{x}}')$ and $I(\bar{\mathbf{x}}^{(i-1)})$ the overall light contribution in the scene of the current sample and previous sample respectively.

$$I(\bar{\mathbf{x}}) = \int_{\Omega} L(\bar{\mathbf{x}}_k \to x) dA(x)$$

with Ω the set of points in sight from the camera.

In our implementation, we defined two type of mutations:

1. *Large mutations*: The purpose of these mutations is to explore the scene. It constists in redrawing a fully random path from the light transport path space. The new sampled path is independent from the previous sampled one. The probability of choosing this mutation is defined by the constant β. Notice that if $\beta = 1$, then our MLT samplers is exactly the same as a basic Monte Carlo sampler.

2. *Small mutations*: The purpose of these mutations is to exploit good path previously found and perform local exploration. Based on the fact that nearby paths will make similar contributions to the image, the use of small mutations is really beneficial for difficult scene such as 4c.

The *expected values* technique enhance the Metropolis algorithm by accumulating both the current sample and the previous samples samples regardless of whether it has been accepted or not. It weights them according to the acceptance probabily in order to keep the algorithm unbaised. The previous sample has a weight of $(1 - \alpha)$ and the current sample has a weight of α. By also accumulating VPLs that have a low contribution to the final image, this techniques makes the algorithm slightly converge faster.

4 Implementation

This interactive renderer is meant to be used as an real-time preview tool for the well-known offline Mitsuba renderer. Fortunately, the two renderers share a lot of functionalities so we could use the ones already implemented to speedup our development.

4.1 VPL generation

Figure 5 describes the overall pipeline for the VPL algorithm. The first step includes the computation of the indirect illumination along a path in the scene and the creation of a VPL at the end of this path. Mitsuba implements a very efficient path tracer, suitable for these computations.

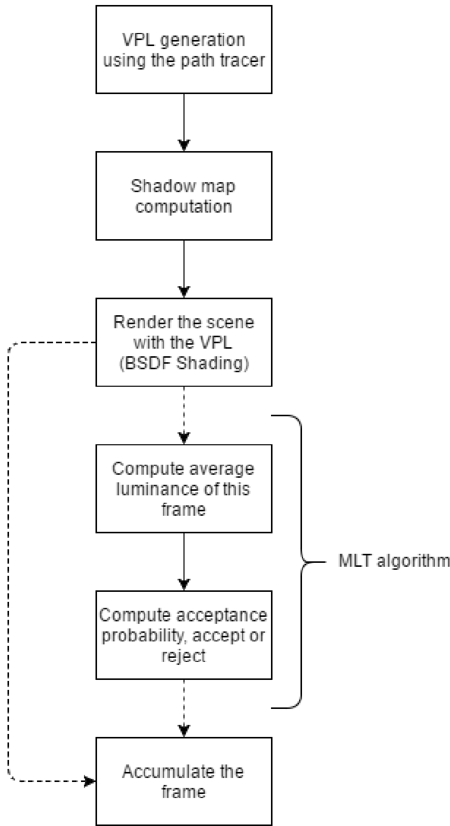


Figure 5: *Overall VPL computation pipeline*

Notice that we use a Monte Carlo approach and sample a VPL at each bounce points along the path. One of these VPLs will be picked randomly. In order to get an unbiased sample, we need to divide the power of this VPL by the probability of sampling this VPL in this set.

For accumulating the light contribution of this VPL, we first need to compute the two remaining terms of eq. 5.

4.2 Shadow maps and geometric term

The $G(..)$ term describes the geometric relationship between the two surfaces, such as visibility and orientation. It can be define as

$$G(x', x'') = V(x', x'') \frac{\cos(\theta'')\cos(\theta')}{||x'' - x'||^2} \tag{6}$$

where θ'' and θ' are the angle of the incident light and the normal of the surface at x'' and x' respectively. $V(x' \to x'')$ is the visibility term which can be defined as

$$V(x', x'') = \begin{cases} 1 & \text{if } x' \text{ and } x'' \text{ are mutually visible} \\ 0 & \text{otherwise} \end{cases}$$

In other words, visibility also means shadows. On a GPU, shadows can be computed using a shadow mapping. The idea here is to first render the scene from the point of view of the VPL in order to compute a *depth image*. We then use this depth image (also called shadow map) to test if a pixel is visible from this light source. Since VPLs are actual point light sources, we need to compute a $360°$ depth image for its shadows. It is done by computing the shadow map on what's called a cube map as decribed in [Gerasimov].

Then, the cosine terms can be easily computed since the GPU is well aware of the surface normals and the incident light directions.

Clamping the distance

In equation 6, we notice that the geometric term is inversely proportional the square of the distance between the two surfaces. A consequence of this observation is that a surface near a VPL will become really bright since its distance to the VPL is close to zero.

$$L \propto \frac{1}{d^2} \Rightarrow \lim_{d \to 0} L = \infty$$

A simple trick here is to clamp d to a minimum distance to avoid these singularities. However, by doing this, our final renderer image won't converge to the correct result anymore. We will lose energy at the corners but it is a worth trade off since it greatly increases the convergence rate of our algorithm.

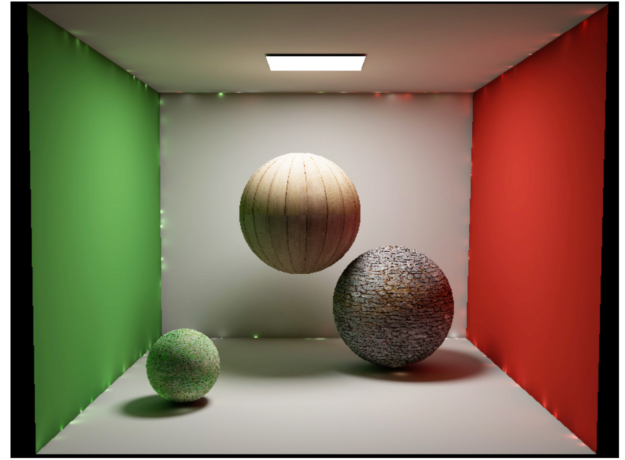


Figure 6: *Clamping prevents singularities at the corners like we see on this image*

4.3 Shader chain

The real world is composed of a vast range of different materials and combination of them. In computer graphics, these are fully described by BSDF (or BSSRDF) functions. In order to approach realistic images, our renderer needs to be able to handle all kind of materials and to combine them. A shader chain code generation system tries to solve this problem giving a flexible framework for BSDF implementation. Each shader has a list of other shaders on which it depends called dependencies. At the initialization step of our system, we generate the code of these shaders, traversing the dependencies tree of the different materials occuring in the scene and recursively binding their code together.

As an example, we could quickly combine a simple shader such as a diffuse material with a texture shader and a bumb mapping shaders.

This corresponds to the $f(x_j, x', x'')$ coefficient in equation 5 which describes the way in which the light scattered by the surface at the VPL position.

This system allows the user to quickly have a preview of his scene even with high specular object such as the specular sphere on Figure 7 or textures like on Figure 6.

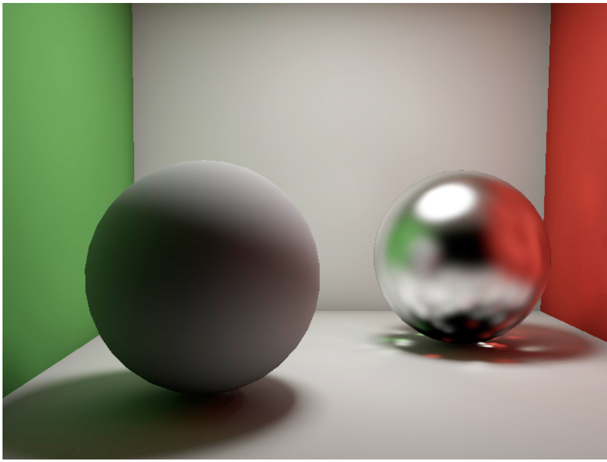


Figure 7: *Example of a high specular material. We can see the reflection of the walls and the second sphere on the specular sphere*

4.4 Metropolis implementation

The PBRT renderer is the practical implementation of the renderer described in [Pharr and Humphreys 2004]. In the third edition, the authors added a chapter specifically about Metropolis sampling techniques in rendering. Thanks to the similarities in our codes, the implementation of their Metropolis sampler suited well with ours system.

The key idea behind their sampler is to represent a path as a vector $\mathbf{X}$ of N real numbers x_n between one and zero. Given this $\mathbf{X}$, the trace function of the path tracer is deterministic as it uses these numbers to sample position and directions in the process. For instance, x_0 and x_1 will define the position of the sampled pixel.

Consequently, since $\mathbf{X}$ totally defines a path in the light space, mutations can be applied on the elements x_n^i to generate a new path.

1. *Large mutations*: The large mutations have the form:

$$x_n' = \mathcal{N}(0, 1)$$

2. *Small mutations*: The small mutations have the form

$$x_n' = \mathcal{N}(x_n^i, \sigma^2)$$

   where σ defines the width of these small mutations.

If $\mathbf{X}'$ is accepted, we override the values of $\mathbf{X}^i$ with the ones of $\mathbf{X}'$.

In [Veach and Guibas 1997], other more specific types of mutations are presented and each of them handle a different lighting problem efficiently such as caustic. Further work could focus on implementing other types of mutations and solve other complex lighting challenges.

5 Results

Figure 8 presents different results obtained with our implementation. Even when the scene contains heavy shapes such as the *armadillo* (made of more than 200000 triangles), our renderer quickly provides good approximations of the overall global illumination in the scene. Comparing the computation times of Figure 8b and Figure 8c, we see that our method can provide more than one order of magnitude in performance gain and still yield similar results. Figure 8d, 8e, 8f present other results where the scene contains multiple light sources.

6 Conclusion

We presented in this report our new state-of-the-art implementation for Virtual Point Light rendering. This extention of the Instant Radiosity algorithm uses modern GPUs power to efficient render global illumination in a physically-based fashion. On top of that, thanks to the use of a Metropolis sampler, our system can handle very difficult visibility scene configuration with almost no performance penalty.

Acknowledgements

To Prof. Wenzel Jakob, for his time and help on this project.

References

GEORGIEV, I., AND SLUSALLEK, P. 2010. Simple and robust iterative importance sampling of virtual point lights. *Proceedings of Eurographics (short papers) 4*.

GERASIMOV, P. S. Omnidirectional shadow mapping. http://http.developer.nvidia.com/GPUGems/gpugems_ch12.html. Accessed: 2016-05-28.

HAŠAN, M., KŘIVÁNEK, J., WALTER, B., AND BALA, K. 2009. Virtual spherical lights for many-light rendering of glossy scenes. In *ACM Transactions on Graphics (TOG)*, vol. 28, ACM, 143.

KAJIYA, J. T. 1986. The rendering equation. In *ACM Siggraph Computer Graphics*, vol. 20, ACM, 143–150.

KELLER, A. 1997. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 49–56.

PHARR, M., AND HUMPHREYS, GREG, W. J. 2004. *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

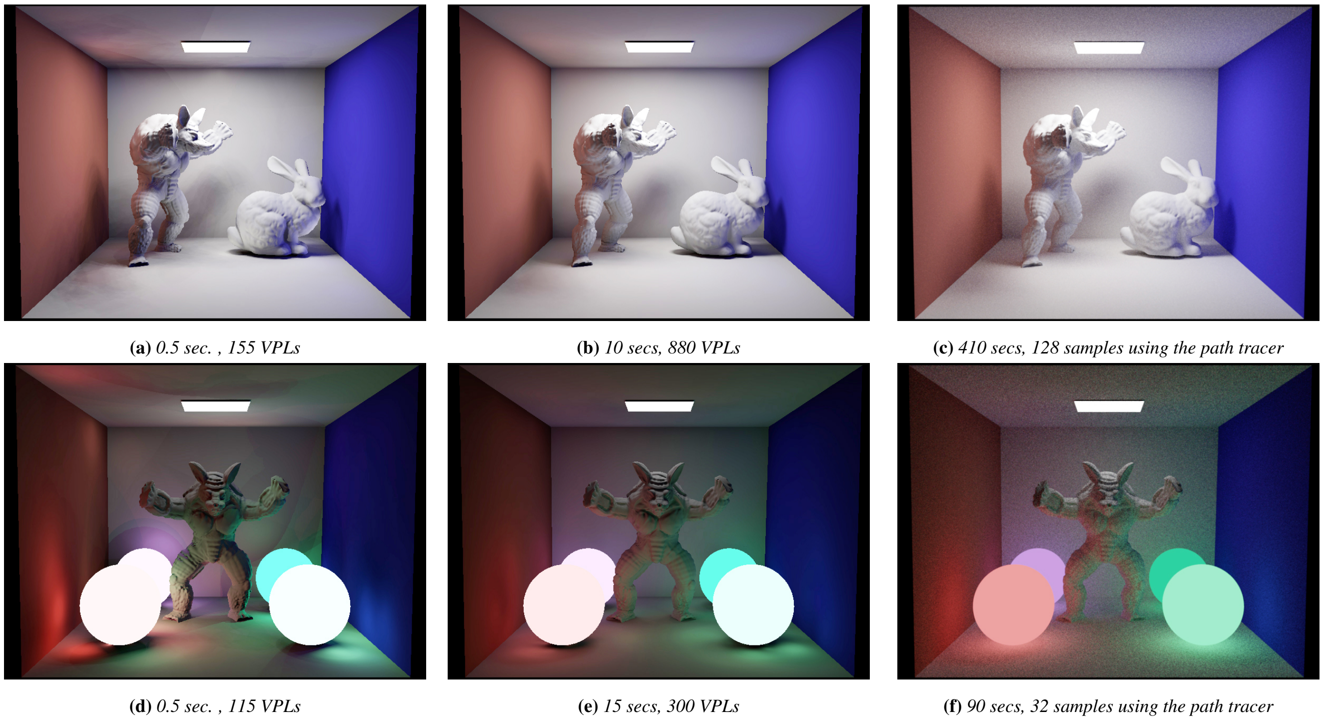
RADAX, I., 2008. Instant radiosity for real-time global illumination.

SEGOVIA, B., IEHL, J. C., MITANCHEY, R., AND PÉROCHE, B. 2006. Bidirectional instant radiosity. In *Rendering Techniques*, 389–397.

SEGOVIA, B., IEHL, J. C., AND PÉROCHE, B. 2007. Metropolis instant radiosity. In *Computer Graphics Forum*, vol. 26, Wiley Online Library, 425–434.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 65–76.

WALTER, B., MARSCHNER, S. R., LI, H., AND TORRANCE, K. E. 2007. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, Eurographics Association, 195–206.



(a) *0.5 sec. , 155 VPLs* **(b)** *10 secs, 880 VPLs* **(c)** *410 secs, 128 samples using the path tracer*

(d) *0.5 sec. , 115 VPLs* **(e)** *15 secs, 300 VPLs* **(f)** *90 secs, 32 samples using the path tracer*

Figure 8: *Results with the monster scenes containing more than 280000 triangles*