GI Next: Global Illumination for Production Rendering on GPUs

E. Catalano R. Yasui-Schöffel K. Dahm N.Binder A. Keller*
NVIDIA

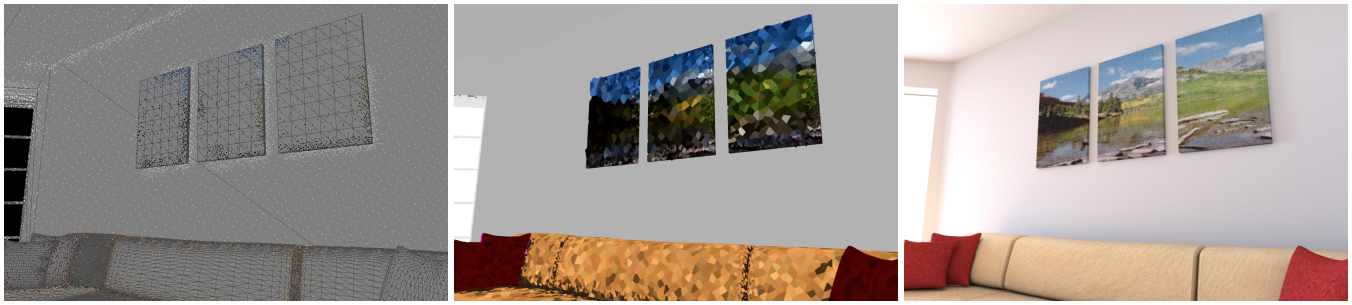

Figure 1: *In order to efficiently compute irradiance on the GPU, only the scene geometry and a point cloud with BRDF parameters (left) need to be transferred to the GPU. During GPU path tracing, BRDF parameters in a hit point are fed by the nearest data point (Voronoi diagram in the middle). Using shader exploration to infer BRDF parameters for the point cloud, global illumination (right) is computed even without access to the custom shader source code. Model courtesy Sandra Pappenguth.*

Abstract

The sheer size of texture data and the complexity of custom shaders in production rendering were the two major hurdles in the way of GPU acceleration. Requiring only tiny modifications of an existing production renderer, we are able to accelerate the computation of global illumination by more than an order of magnitude.

Keywords: Global illumination, production rendering, GPU ray tracing.

Concepts: •**Computing methodologies** → **Ray tracing; Reflectance modeling;**

1 Introduction

Porting a production renderer to take advantage of GPUs is a considerable effort and often requires rewriting the whole engine. In addition, custom shaders may not be accessible in source code and often introduce performance penalties if not specially adapted to the accelerator. However, function calls to the renderer's API from within shaders may be intercepted and thus costly functions in the render core may be accelerated outside the shader code. One such render core API function is the calculation of irradiance [Křivánek and Gautron 2009; Debattista et al. 2006], and it is this part we accelerate on the GPU.

2 Algorithm

In order to compute the irradiance on the GPU, we need the geometry data of the scene and a point cloud. Each point contains information about surface properties, shading properties, and direct light contribution. This data is created before the rendering begins and transferred to the GPU. Note that neither textures nor light sources need to be transferred, because this information is already embedded in the point cloud data.

2.1 Point Cloud Creation

The point cloud is created by placing points on surfaces of the scene geometry, and then sampling properties at these points, such as direct light contribution from all light sources and shading information. The placement process of the points is organized in passes, where each pass refines the result of the previous pass by adding new points. This way, we achieve an adaptation of point density only in locations where high detail requires higher point density. Such high detail regions could be sharp gradients in direct light contribution, abrupt color changes from shading, or varying normals indicating geometric detail. Shading information is extracted by executing the custom shaders in a special manner at each point, a process which we call shader exploration.

2.2 Shader Exploration

The irradiance calculation on the GPU uses a generic shading model, which is parameterized by characteristics extracted from the actual custom shaders. Shader characteristics are inferred by setting special render states before calling the shader, thus allowing to deduce shader internals from the shader result.

As an example, given a point to shade, the state provided to a shader may indicate that lights and environment lighting shall be disabled, and reflection and refraction rays shall not be traced. If the shader calls the render core function, it provides the normalized color white on the front facing side and black on the backside of the surface point under consideration instead of the irradiance. Now calling the shader returns a color that is the diffuse albedo plus the ambient illumination/emission.

In a similar way, more properties are deduced and stored, which include BRDF parameters for diffuse albedo, reflection, refraction, and transmission properties. Even the index of refraction may be extracted this way. Note that the extracted parameters include the

*e-mail: {ecatalano|rschoffel|kdahm|nbinder|akeller}@nvidia.com

SIGGRAPH '16, July 24-28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4282-7/16/07
DOI: http://dx.doi.org/10.1145/2897839.2927452

Figure 2: *Left: Original rendering in mental ray, which took about 7 hours (2 x Intel Xeon E5-2640 v3 2.6 GHz,16 cores HT = 32 threads). Right: Result of the GPU accelerated version of mental ray, which took about 36 minutes using the original custom shaders (2 x NVIDIA Quadro M6000). Model courtesy Sandra Pappenguth.*

evaluation of texture and procedural texture values.

2.3 Accelerated Irradiance Calculation by Approximate Shading

Using the information as baked into the point cloud, the irradiance is computed by tracing rays into the scene like we would have done in the original renderer, however, instead of executing the custom shaders at the hit points, we use one generic BRDF model fed with the parameters of the closest point with matching normal (see Fig. 1). Employing only a single generic BRDF model minimizes execution divergence on the GPU, while in addition being able to benefit from the superior ray tracing performance on GPUs [Parker et al. 2010]. Furthermore, we save on costly shader execution and there is no need to access texture data.

Contrary to irradiance interpolation [Křivánek and Gautron 2009], irradiance is computed as a spherical integral weighted by a cosine, which acts as a smoothing operator. The piecewise constant approximation of the integrand thus further helps reducing the variance. Note that even though shading information is only approximated by the point cloud values and density of points, the computed irradiance contains geometry effects in full detail, like occlusion or color bleeding.

3 Integration into the Rendering Process

In the regular ray tracing rendering process, a custom shader will be executed at each hit point. This custom shader then may call the render core functionality to compute the irradiance at the current hit point. Although the irradiance calculation involves shooting many recursive rays into the hemisphere around the hit point, accelerating single calls would not be efficient on a GPU. Hence many irradiance requests from API calls of custom shaders are collected and processed in a batch, before returning the results to the custom shaders.

In order to avoid “freezing” all custom shaders upon the API call, the rendering process is executed twice. In the first pass, the API calls for irradiance are recorded, a dummy result is returned immediately, and the rendering result of the custom shaders is discarded. Then, the irradiance is computed for the whole batch on the GPU and stored. In the second pass, the renderer is started with exactly the same initial state as the first time in order to generate the exactly same irradiance requests. This time, however, the API returns the stored irradiance results to the custom shaders, which now generate the final rendering result. Obviously the acceleration must compensate for the double invocation of the renderer, which it does, see Fig. 2.

As a requirement, the rendering process must be deterministic and exactly repeatable, which we achieve by using deterministic quasi-Monte Carlo methods [Keller 2013].

References

DEBATTISTA, K., SANTOS, L., AND CHALMERS, A. 2006. Accelerating the irradiance cache through parallel component-based rendering. In *Eurographics Symposium on Parallel Graphics and Visualization*, The Eurographics Association, A. Heirich, B. Raffin, and L. Santos, Eds.

KELLER, A. 2013. Quasi-Monte Carlo image synthesis in a nutshell. In *Monte Carlo and Quasi-Monte Carlo Methods 2012*, J. Dick, F. Kuo, G. Peters, and I. Sloan, Eds. Springer, 203–238.

KŘIVÁNEK, J., AND GAUTRON, P. 2009. *Practical Global Illumination with Irradiance Caching*. Synthesis lectures in computer graphics and animation. Morgan & Claypool.

PARKER, S., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2010 TOG 29*, 4, Article No. 66.