Path Regeneration for Interactive Path Tracing

Jan Novák[1,2] Vlastimil Havran[1] Carsten Dachsbacher[2]

[1]Czech Technical University, Faculty of Electrical Engineering
[2]VISUS, University of Stuttgart

Abstract

Rendering of photo-realistic images at interactive frame rates is currently an extensively researched area of computer graphics. Many of these approaches attempt to utilize the computational power of modern graphics hardware for ray tracing based methods. When using path tracing algorithms the ray paths are highly incoherent, hence we propose an efficient technique that minimizes the divergence in execution flow and ensures full utilization by intelligently regenerating the paths. We analyze the conditions under which our improvements provide the highest speedup, and demonstrate the performance of the overall system by rendering interactive previews of global illumination solutions using (bidirectional) path tracing with progressive refinement.

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

1. Introduction

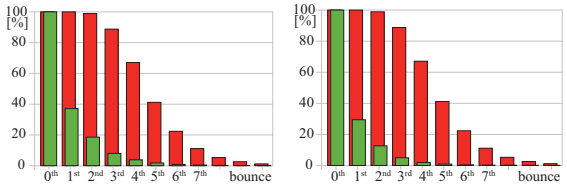
Physically correct computation of light transport is considered to be a hardly solvable problem in real-time on contemporary hardware. Path tracing algorithms [Kaj86, LW93] terminate the paths with Russian roulette to ensure unbiased results, but require a tremendous amount of paths to produce converged results. Porting these algorithms on wide SIMD architectures can achieve notably higher throughput, but the probabilistic termination of paths poses a significant issue for the parallel computation. Unoptimized implementations suffer from high execution flow divergence that results in poor hardware utilization: while some threads still trace paths, others have already been terminated and just occupy the processing units without any contribution to the overall result.

The vast majority of current GPU algorithms based on tracing paths avoids the divergence by tracing *all* paths up to a certain length. This is very inefficient since some of the paths deliver a negligible amount of energy after a few bounces, whereas others may contribute considerably even after exceeding the preset length. Therefore, our method still uses the Russian roulette, but alleviates the poor utilization by restarting terminated threads and computing new paths, leading to higher overall performance. We also show how to limit the length of paths if necessary, such that the amount of introduced bias can be easily controlled.

2. Related Work

While many papers have been devoted to ray tracing on GPUs, only few address efficient implementations of general rendering algorithms in the parallel environment of modern graphics cards. Cassagnabère et al. [CRR04] discuss an implementation of path tracing on the AR350 processor of the RenderDrive product family. Path tracing and distribution ray tracing have been highly optimized on CPUs by Boulos et al. [BEL*06]. Cassagnabère et al. [CRR06] describe software architectures for the mapping path tracing algorithms on several GPUs. Coulthurst et al. [CDDC08] present a path tracing algorithm with high utilization of wide SIMD processors by data buffering on the ClearSpeed CSX architecture. Recently, Budge et al. [BBS*09] suggested a software architecture that allows sharing the resources of CPUs and GPUs and supports path tracing. Sugerman et al. [SFB*09] describe the GRAMPS programming model that handles varying workload automatically by queues. This concept has been verified by Aila and Laine [AL09] for traversing a kd-tree with persistent threads and a pool of rays.

Our technique is complementary to the persistent threads since we focus on maximal utilization on a higher level by intelligently restarting terminated paths with no necessity to enqueue them. We demonstrate the applicability on bidirectional path tracing and present, to our best knowledge, the first efficient (bidirectional) path tracing on the GPU.



(a) Office (b) Conference Room

Figure 1: *The relative number of active (green) and all (red) threads in warps after first few bounces. The difference between the columns expresses the fraction of inactive (idle) threads. The overall utilization of processing units is only* 32% *and* 35% *for the Office and Conference Room.*

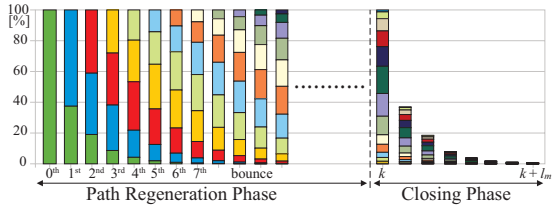


Figure 2: *Regeneration of paths that were terminated by Russian roulette or did not intersect the scene geometry. The color emphasizes different path generations, e.g. green, blue, or red color show the relative number of threads that performed zero, one, or two restarts, respectively.*

3. Path Tracing on SIMD

The parallelization of path tracing follows similar concepts as for ray tracing. In the simplest form, each thread, having been assigned one primary ray, traces the ray until the closest intersection with the scene geometry is found. Then it estimates direct illumination due to one randomly selected light source and stores it in a temporary image buffer. To account for indirect illumination, the algorithm samples the bidirectional scattering distribution function (BSDF) of the hit surface, and with probability derived from material and geometric configuration continues to incrementally construct a path, until it is stochastically terminated by the Russian roulette. At each path vertex, one light source is sampled and the weighted contribution accumulated in the image buffer.

3.1. Sparse Warps

The major problem arising from the Russian roulette is the high divergence in the length of individual paths. At each bounce, a certain fraction of paths is terminated and the corresponding threads become inactive. On GPUs with CUDA architecture, the threads are processed simultaneously in groups of 32, called *warps*. Since the number of active threads in a warp is dropping as the paths are randomly terminated, we call such warps *sparse*. Notice that a warp is considered to be sparse only if it contains at least one active and one inactive thread. Consider for example the Office scene with the average probability of continuing a path equal to 38%. The number of active threads in a warp after i bounces can be estimated as $0.38^i \times 32$, giving 12, 5, and 2 active threads after first, second, and third bounce. The performance of any SIMD architecture scales with the occupancy of the available resources; therefore, *sparse* warps will prevent any path tracing algorithm from exploiting the full GPU power.

Figure 1 shows the ratio between actively working and idle threads for two different scenes. The difference between columns expresses the total number of inactive threads (wasted resources) in *sparse* warps, which drastically reduces the utilization of the GPU.

3.2. Path Regeneration

We address the problem of sparse warps by restarting the terminated paths and tracing additional ones. Once a path P_i is terminated, we generate a new path P_{i+1}, assign it to the inactive thread and continue with tracing the next bounce. The path restart is handled by a routine, which is executed after the Russian roulette. For all terminated paths, the routine stores their contributions and generates new samples that are used to shoot new paths. The rest of the pipeline remains unaffected. This step ensures that all threads that would become inactive continue contributing to image synthesis.

Restarting the new paths in the origin (camera) and tracing additional primary rays is beneficial only in some cases, such as rendering of motion blur or depth-of-field effects. In these situations the routine generates completely new paths for the terminated ones. On the other hand, if the primary rays are used to suppress aliasing artifacts *only*, we not always create a new path, but reuse the primary ray traced by P_i and continue tracing P_{i+1} from the first found intersection. Due to stochastic sampling of BSDFs, the new path will take a different random walk and compute indirect lighting from other parts of the scene than P_i. By sharing the primary rays among the paths, we dedicate the majority of available resources to computation of indirect illumination, the main source of variance.

Since a complete avoidance of tracing multiple primary rays would prevent anti-aliasing, we could enforce creation of an entirely new path after a certain number of restarts has been performed. In our implementation we take different approach: we allow the user to adjust the anti-aliasing by setting the number of samples per pixel, whereas by the length of the *path regeneration phase* T_r the user controls the computation of indirect lighting. For each pixel sample, a single SIMD thread traces (and restarts) paths until T_r bounces are found. Then it stops restarting and continues tracing for another T_c bounces, to which we refer as the *closing phase*. The length of the closing phase T_c, formalized in the next section, should suffice to let the Russian roulette gradually terminate all the remaining paths, as shown in Figure 2. Notice the full utilization of threads during the path regeneration phase.

3.3. Path Length Limitation

In order to avoid introducing bias from forcedly terminating the paths in the closing phase, the algorithm would have to wait for all the paths to be terminated by the Russian roulette. In such case, the length of the closing phase T_c is unpredictable and the algorithm needs to determine the number of remaining paths, e.g. by parallel reduction, at each step. As this would be too expensive, we rather precompute a maximum path length l_m in a way that allows us to control the amount of bias introduced from terminating the path after l_m bounces. Since the termination probability of the Russian roulette is derived from the hemispherical reflectance of the hit material, the average path length is proportional to the average reflectance of the scene ρ_S. Bearing this in mind, we can estimate l_m in the following manner: first, we compute ρ_S as the mean value of material reflectance weighted by the area S_i covered by the respective material i as: $\rho_S = \frac{1}{S}\sum_{i=1}^{N}\rho_i S_i$. The maximum path length l_m is then set to $\log_{\rho_S} c$, where c is the fraction of paths that we admit to terminate forcedly.

By setting the length of the closing phase T_c to l_m, we ensure that the number of forcedly terminated paths is approximately c. Limiting the path length with respect to material properties is a general idea and can be used in another algorithms, i.e. distribution of VPLs for instant radiosity [Kel97] on the GPU.

3.4. Bidirectional Path Tracing

We examined the usability of the path regeneration technique on a basic version of bidirectional path tracing following the original paper by Lafortune and Willems [LW93]. As the main idea is to combine eye subpaths with light subpaths, a straightforward approach would be to precompute one light subpath per image sample and then combine it with all eye subpaths created for the sample. Such implementation introduces a significant coherence into the Monte Carlo estimator, since all eye subpaths generated for one sample get combined with one light subpath only. To break this coupling, we randomly select a different light subpath at each bounce of the eye subpath (Figure 3).

This technique has two major advantages. First, we overcome the problem of combining one particular light subpath with all eye subpaths that are traced for one image sample. Second, we also reduce the deterministic noise caused by connecting vertices of one light subpath with one eye subpath. If the light path happens to carry a lot of energy, it will greatly contribute to each vertex of the eye subpath. Consequently, this eye subpath will tend to gather much more light than another one, which is combined with a low contribution light subpath. By permuting the light subpaths, we allow the energy to be spread among multiple eye subpaths that belong to different image plane samples. It can be shown that the estimator is still unbiased and the Neumann series converges if the contribution of each path is correctly weighted.

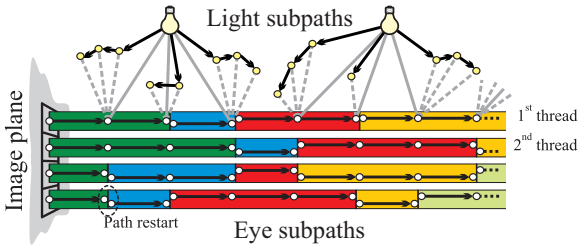


Figure 3: *Path regeneration technique applied to bidirectional path tracing: the combination of eye and light subpaths is shown for the first thread only. Shadow rays towards the lights (gray solid lines) refer to classic path tracing.*

4. Implementation Details

Since obtaining converged results in real-time is currently impossible, we render the final image progressively with successive refinements. This requires interrupting the computation in the path regeneration phase and displaying the intermediate results. When the camera is moving, the results are displayed after only a few iterations. Optionally, we use an edge-preserving bilateral filter to blur the noise in the renderings. Once the position of the camera is fixed, the screen is updated with increasing intervals, which enables interactive frame rates when exploring the scene, while keeping the overhead of interrupting the computation fairly low when waiting for converged results. Figure 4 shows result images of interactive renderings, and more converged solutions. The filtering can be switched on and off at anytime during the progressive refinement. For accelerating the intersection search we use a kd-tree data structure. Random numbers are generated on-the-fly using Mersenne twister random number generator. Converged results are shown in Figure 5.

5. Results

We measured all results using an NVIDIA GeForce GTX 285. Table 1 shows the performance improvements of the optimized path tracing with path regeneration. The unoptimized version (NO) traces one path for each sample up to a maximum length l_m that has been computed with $c = 1\%$. For the path regeneration version (PR) we used regeneration and closing phases with 100 and l_m iterations, respectively. The number of per-pixel samples for both algorithms was adjusted to obtain images with similar quality.

For each tested scene, we also created a brighter and a darker version to examine the performance dependency on ρ_S. For the darker scenes, where the paths are shorter on average, the hardware is still well utilized. However, as the scenes become brighter, the performance of the unoptimized version decreases, since the ratio of active to inactive threads in the sparse warps is lower. The path regeneration technique is by far less prone to introduce any dependency on the materials in the scene, which makes it more robust and stable in terms of throughput. Another advantage of path regeneration

	Initial Scene Configuration						Lower Avg. Reflectance					Higher Avg. Reflectance				
	ρ_S	l_a	l_m	NO	PR	S	ρ_S	l_m	NO	PR	S	ρ_S	l_m	NO	PR	S
Cornell Box	0.32	1.5	5	30.03	48.52	**1.6**	0.16	3	38.55	49.02	**1.3**	0.52	8	23.76	49.16	**2.1**
Office	0.43	1.8	6	10.34	15.37	**1.5**	0.22	4	12.09	15.49	**1.3**	0.71	14	7.81	15.57	**2.0**
Sibenik	0.67	3.0	12	8.99	17.24	**1.9**	0.34	5	12.51	17.41	**1.4**	0.80	22	7.80	17.49	**2.2**
Conference R.	0.44	1.8	6	6.64	9.64	**1.5**	0.23	4	7.77	9.50	**1.2**	0.73	15	5.20	9.66	**1.9**
Golf Balls	0.67	4.0	12	6.79	11.13	**1.6**	0.34	5	8.03	11.12	**1.4**	0.81	23	6.29	11.16	**1.8**

Table 1: *Performance of the unoptimized path tracing (NO) and our version with path regeneration (PR) in 10^6 rays/sec. Column ρ_S shows the average scene reflectance. l_a and l_m report the average and maximum path length, respectively. Column S states the relative speedup of the optimized algorithm (PR/NO). Notice the dependence of S on the average scene reflectance.*

Figure 4: *Interactive rendering of the Sibenik Cathedral and Conference Room. Each triple contains a filtered/original preview image with corresponding frame rates, and rendering results after 1 and 10 seconds of progressive refinement.*

is that it also inherently restarts paths that were terminated for some other reason than the Russian roulette, i.e. those that did not hit the scene geometry.

The upper bound on the saved computation cost due to reusing the primary rays can be derived from the average path length l_a, which equals $\sum_{i=1} i(1-r)r^{i-1} = 1/(1-r)$, where r is the average probability of continuing the path. Since we trace one shadow ray at each bounce, the total number of rays per path is $2l_a$ and the upper bound on the saved computation is $1/(2l_a)$. In our tests, the rendering time was reduced by 9% to 20%, which is slightly less than $1/(2l_a)$, since tracing rays is only a part of the total rendering cost.

6. Conclusion and Future Work

We presented an efficient GPU implementation of (bidirectional) path tracing that fully utilizes processing units achieving performance of 48 millions of incoherent rays per second for simple scenes, and about 13 million rays per second for more complex ones. The algorithm has stable throughput regardless of materials in the scene achieving up to double performance just by regenerating the paths. We would like to continue exploring path re-usability on SIMD architectures in similar spirit as Bekaert et al. [BSH02].

Figure 5: *Converged results rendered with our path tracer.*

Acknowledgement

This work is partially funded by the Deutsche Forschungsgemeinschaft (DFG) as part of the Collaborative Research Centre SFB 627, and by the Czech MŠMT programmes LC-06008 (Center for Computer Graphics), MSM 6840770014, and the Aktion Kontakt OE/CZ fund 2009/6.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on gpus. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009* (2009), ACM, pp. 145–149. 1

[BBS*09] BUDGE B. C., BERNARDIN T., SENGUPTA S., JOY K. I., OWENS J. D.: Out-of-core data management for path tracing on hybrid resources. In *Proc. Eurographics 2009* (2009). 1

[BEL*06] BOULOS S., EDWARDS D., LACEWELL J. D., KNISS J., KAUTZ J., SHIRLEY P., WALD I.: *Interactive Distribution Ray Tracing*. Tech. Rep. UUSCI-2006-022, 2006. 1

[BSH02] BEKAERT P., SBERT M., HALTON J.: Accelerating path tracing by re-using paths. In *EGRW '02: Proc. of the 13th Eurographics workshop on Rendering* (2002), pp. 125–134. 4

[CDDC08] COULTHURST D., DUBLA P., DEBATTISTA K., CHALMERS A.: Parallel path tracing using incoherent path-atom binning. In *proceedings of SCCG 2008* (2008), pp. 103–108. 1

[CRR04] CASSAGNABÈRE C., ROUSSELLE F., RENAUD C.: Path tracing using the AR350 processor. In *GRAPHITE'04* (2004), pp. 23–29. 1

[CRR06] CASSAGNABERE C., ROUSSELLE F., RENAUD C.: Cpu-gpu multithreaded programming model: Application to the path tracing with next event estimation algorithm. In *ISVC06* (2006), pp. II: 265–275. 1

[Kaj86] KAJIYA J. T.: The rendering equation. In *Computer Graphics (Proc. of SIGGRAPH '86)* (1986), pp. 143–150. 1

[Kel97] KELLER A.: Instant radiosity. In *SIGGRAPH '97* (1997), pp. 49–56. 3

[LW93] LAFORTUNE E. P., WILLEMS Y. D.: Bi-directional path tracing. In *Compugraphics '93* (1993), pp. 145–153. 1, 3

[SFB*09] SUGERMAN J., FATAHALIAN K., BOULOS S., AKELEY K., HANRAHAN P.: Gramps: A programming model for graphics pipelines. *ACM Trans. Graph. 28*, 1 (2009), 1–11. 1