

Irradiance Cache for a GPU Ray Tracer

Vladimir Frolov^(1,2), Konstantin Vostryakov⁽²⁾, Alexander Kharlamov⁽²⁾, Vladimir Galaktionov⁽¹⁾,

(1) Keldysh Institute of Applied Mathematics (Russian Academy of Sciences), Moscow, Russia; (2) Nvidia



Figure 1. The presented screenshots were rendered at 1920x1200 resolution on a GTX 560 HW under 5 minutes using Irradiance Caching (IC) technique. We achieved from 5 to 15 times acceleration compare to our naive path tracing implementation.

Abstract

This work proposes a GPU friendly irradiance caching (IC) solution, where performance critical parts of an irradiance cache algorithm are done completely on the GPU. We discuss some practical problems arising in the implementation of GPU irradiance caching, and propose solutions for them. The modified algorithm for the GPU is different from a CPU implementation in 2 ways. The first distinction is a multi-pass construction of irradiance cache followed by a final rendering stage and the second distinction is to insert a large record set at once instead of one by one, as used in traditional approaches. We also consider some details to efficiently implement look-up operations on the GPU.

Keywords: GPU, Irradiance Cache, Global Illumination.

1. INTRODUCTION

For the last decade Graphics Processor Units (GPUs) have made a great advance in performance and have become fully programmable processors. Several commercial GPU photorealistic renderers are available today. Most of them use unbiased path tracing methods in order to minimize intermediate data creation (photon maps, lightcuts, etc). This results in tracing up to ten times more rays than biased alternatives. Moreover, path tracing of complex scenes suffers from highly irregular workload (per ray) and memory access tends to be random. These issues lead to inefficient hardware utilization. On the other hand, although biased approaches have lower complexity they are more difficult to implement on the GPU.

Our paper illustrates the research that we have performed for GPU accelerated biased rendering via irradiance caching and path tracing techniques. The key results of this research are:

- A new GPU friendly IC generation algorithm, performed before final render pass.
- Introduction of a new method for records insertion into octree-based cache for fast irradiance interpolation on the GPU.

Our main contribution is a high quality IC solution that provides from 5 to 15 times acceleration (with an average PSNR of 40 compared with a GPU accelerated path tracing).

2. RELATED WORK

2.1 GPU ray-tracing

Although fast GPU ray tracing for complex scenes is still a challenge we do not focus on ray tracing acceleration in this paper. Aila and Laine's work [Aila and Laine 2009] provides comprehensive performance analysis of ray tracing on the GPU. Our ray tracing implementation has approximately the same performance on diffuse rays (i.e. rays, randomly shot from the single point over the hemisphere) although it's of 1.5x factor slower for coherent frustum tracing. However we have found that path tracing rays after several bounces can be from 2 to 4 times slower (than diffuse rays) – this is the case of poor HW utilization due to random memory access and non-uniform workload.

2.2 CPU Irradiance cache

Irradiance caching decreases the overall cost of indirect illumination computation by performing full hemisphere sampling (or final gathering) only at selected points in the scene, caching the results, and reusing the cached indirect illumination values between those points through interpolation. It was introduced in [Ward et al. 1988]. The algorithm can be summarized as follows:

```
if interpolation is possible then
    reuse cached values through interpolation;
else
    compute new value;
    store it in the cache;
end if;
```

The number of irradiance cache points is usually of 1 - 2 orders of magnitude less than the number of pixels – so the irradiance cache is quite efficient and it can greatly speed up the whole rendering. However, IC is a challenging algorithm, even on a CPU. It has a lot of issues and heuristic approaches that make it practical and suppress its artifacts [Krivanek et al. 2008]. A well-known irradiance cache algorithm [Ward et al. 1988, Krivanek et al. 2008] cannot be implemented on a GPU in a straightforward way because of its serial nature:

- Trace one ray
- Evaluate and insert one record at given “transaction”.

It is difficult to parallelize irradiance cache on multi-core CPUs, although there are several papers available regarding parallel irradiance caching on CPU ([Debattista et al. 2006, Dubla et al. 2009]). These papers focus on solving the problem of sharing

irradiance cache data structure between different CPU threads and different machines (cluster systems).

However, the problem is not only in data sharing between threads and redistributing computational resources (for example between rendering and IC records evaluation), but it also in the fact that IC depends on the records insertion order. For example, irradiance gradients [Krivanek et al. 2008] rely on a serial records insertion. If one places two records in parallel near each other, the gradients and, as a result, validity radiuses of these records will be different compared to serial insertion. The more threads run in parallel, the more serious this problem becomes so we introduce validity clamping heuristics to solve it in our GPU implementation.

PBRT 2.0 [Pharr and Humphreys 2010] has multithreaded implementation of IC. It does the first pass to compute the cache and the second one to render final image. This approach needs to be refined for the case of massive parallelism.

2.3 GPU Irradiance cache

GPU irradiance caching was introduced in [Gauton et al. 2005] and described in details in [Krivanek et al. 2008]. These papers mainly focus on replacing irradiance interpolation via octree lookups with splatting to avoid traversing hierarchical structures on GPUs. The approach used in [Gauton et al. 2005] can be used for primary rays or interactive visualization in computer games, however, it has one serious limitation: only one light bounce can be evaluated either for hemisphere sampling or for final rendering. Thus, it will be hard to have precise photorealistic result with this approach. Besides, it was done mainly for rasterization based engines and cannot be combined directly with a GPU path tracer.

Wang et al. [Wang et al. 2009] presents an efficient approach for global illumination using photon mapping on the GPU. The key aspect of this work is to use irradiance cache with photon mapping and final gathering [Jensen et al. 2002] to quickly compute smooth indirect illumination. Direct lighting is computed using ray tracing and supports hard shadows from point light sources. In this paper irradiance cache point positions are predicted from the geometry discontinuities. Wang's approach to build IC was combined with path tracing in [Frolov et al. 2011] to focus on rendering images with glossy reflections and shadows offline. However, both of these approaches work with geometry term and they use predictive nature without further refinement.

The radiance hints method introduces in [Papaioannou 2011] is a stable (for animation) and a fast solution for diffuse global illumination. The method is based on grid based radiance caching with reflective shadow maps and can handle multiple light bounces. This method works for interactive rendering with view-independent algorithm so it can't control image error that is strictly needed for photorealistic rendering. Besides, using regular grid will not allow one to have high precision with reasonable memory consumption.

3. SUGGESTED APPROACH

Similar to PBRT 2.0 our algorithm consists of 2 main phases. The first phase is "irradiance cache creation" and the second phase – "final rendering". The goal of the first phase is to generate a set of irradiance cache points that will completely cover the space where future samples can occur. This separates computing irradiance cache from using it.

3.1 Creation of irradiance cache

IC creation process consists of multiple passes (20-30 passes, the maximum number is user controlled). It can be summarized in the following pseudo code:

```
procedure Create_IC(ic : out Irradiance_Cache) is
    geomDiscMap : Image;
    irradDiscMap : Image;
    discMap      : Texture2D;
    candidates   : array of IC_Record;
    smallGroup   : array of IC_Record;
    candGroups   : array of (array of IC_Record);
    iterNum      : Integer;
    -- user controlled
    MAX_PASS_NUMBER : Integer := 30;
    MIN_CAND_TRESHOLD : Integer := 100;
begin
    geomDiscMap := CreateGeometryDiscMap();
    discMap     := Build2DMipMapChain(geomDiscMap);
    candidates  := Dithering(discMap);

    ic.Insert(candidates);
    iterNum := 0;
    candidates.resize(MIN_CAND_TRESHOLD+1);

    while (candidates.size() >= MIN_CAND_TRESHOLD and
        iterNum < MAX_PASS_NUMBER) :

        -- screen space stage
        irradDiscMap := CreateIrradianceDiscMap();
        discMap := Build2DMipMapChain(irradDiscMap);
        candidates := Dithering(discMap);

        -- insert candidates except for pixels
        -- for which we already have records
        ic.Insert({candidates} \ {ic.records});

        -- world space stage
        candidates := SelectIfInterpErrorIsLarge();
        candidates := SortWithZCurve(candidates);
        candGroups := GroupRecords(candidates);

        candidates := []
        for group in candGroups :
            smallGroup := SelectSeveralCands(group);
            candidates.append(smallGroup);
        end for;

        ic.Insert(candidates);
        iterNum := iterNum + 1;
    end while;
end Create_IC;
```

The 'Insert' procedure also evaluates irradiance for each records. We will discuss its implementation later.

Each pass consists of 2 independent stages. The first stage works only for visible points. The second stage works for visible and for points that are not directly visible from the eye. During each pass and within each stage new irradiance cache records are inserted into the cache. The very first pass is different from the others and works with geometry discontinuity like [Wang et al. 2009] and [Frolov et al. 2011] do. The important aspect of the irradiance cache generation process is that a large set of points (several hundred or even thousand points) are selected at once, irradiance for these points are computed on the GPU and these points are added to the cache structure in one transaction.

3.1.1 The very first pass (coarse screen space pass)

In the very beginning of the process of irradiance cache creation we have no information about the scene at all. Owing to this fact, the goal of this pass is to create first approximation of irradiance cache that will be used as a starting point for further passes. Our algorithm tries to predict complexity of different screen parts, using geometry discontinuity maps and image processing. It

attempts to place more records in areas where more of them are needed.

First, we trace rays from the eye position and store hit positions and normals in separate full screen textures. A mip-map pyramid for each of these textures is built. Then, each mip-level of surface discontinuity texture map is evaluated according to this formula:

```
surfDisc := k*normDiff + worldPosDiff;
```

where worldPosDiff and normDiff is a maximum difference between positions (and normals accordingly) within neighboring pixels (in an appropriate mip-level). And k is a parameter that depends on the world scale. Having a discontinuity map, we blend all of up-scaled mip levels and perform the dithering algorithm on the resulting image with a binary quantization (i.e. each pixel in resulting image can have a value equal to 1 or 0). The result of this dithering is a binary image - a set of initial points; this initial set is our first approximation of irradiance cache (Fig. 2).

The main idea behind that binary dithering is that it allows us to represent discontinuity maps (both geometry and irradiance) in terms of sparse point set – potential IC records.

Dithering Implementation:

Our dithering implementation is deterministic (but we suppose random or combined solution is also possible). For a given part of screen it decides (based on user defined threshold) whether we need to put irradiance cache record in each 32-d, 16-th, 8-th, 4-th or 2-d pixel.

3.1.2 Screen space stage (1)

The same operation but for irradiance discontinuity is performed in subsequent passes– calculate irradiance by fetching it from the cache, build difference image, create mip-map pyramid, blend up-scaled images, perform dithering and insert newly obtained points into irradiance cache. This procedure is repeated several times. Because screen space solution stops producing new points relatively fast, we disable it after several (3-4) passes and continue creating the cache only with the world space stage.

3.1.3 World space stage (2)

The presented screen space algorithm works on primary visible smooth surfaces. However it cannot be used for indirectly visible surfaces and it tends to miss tiny geometric details. The red triangle on Fig. 3 represents viewers' frustum. Red points are not visible from the camera, however secondary rays can still reach such regions.

Our goal is, to generate a set of irradiance cache points that completely cover the space where rays can hit a surface during path tracing process. We used an idea similar to the clustering approach that had been used in [Gassenbauer et al. 2011].

Rays are traced from the eye and all hit points are saved on each bounce if interpolation has unacceptable error estimation from geometric considerations. All such points are stored in separate buffer during ray tracing using CUDA 'atomicAdd' operation similar to how DirectX10/11 'append buffer' works.

However, a very large set of points is produced and we need to select a subset of the best candidates from it. We believe several approaches can be applied to form clusters; however we used the simple one that can be easily ported to GPU. At first, we sort candidates according to 3D Z-Curve [Morton 1966]. After that we start inserting points into a cluster (around the first point in the sorted array). However we want to keep the bounding box of the current cluster within certain limits. If after inserting a point the bounding box of the current cluster exceeds the maximum bounding box size, we create a new cluster and continue inserting candidates into it.

Next, it is possible to select a single point from each cluster with maximum error however it is not the best approach. Let us consider a cluster that was formed around the corner of the Cornell Box. The corner consists of 3 walls and if our cluster contains points on every wall, we need to select at least one point on each wall, otherwise we lose useful candidates, because we know that radiance difference usually corresponds to rapid changes of the normal field. So, from each cluster we select several candidates with unique normals and thus, deal with corner cases.

We terminate the creation process when the maximum number of passes is reached or when 'candidates.size()' becomes small enough (this parameter is user defined). Due to the stochastic nature of our IC creation process (world space stage uses random 'path tracing style' rays) on some complex scenes there can be a regions that were not covered by IC records and candidates are still produced. However, if we stop IC creation process in that case, it will not introduce a valuable error in the final image because the probability of rays hit such regions tends to be zero.

3.2 Final rendering

After we have irradiance cache computed we do adaptive path tracing as described in [Frolov et al. 2011] with fetching indirect smooth lighting from irradiance cache. Thus, for fast and smooth indirect lighting we use irradiance caching technique and we use path tracing for other effects, such as soft shadows, glossy reflections and refractions, depth of field and motion blur.

4. IMPLEMENTATION AND RESULTS

Our implementation is done using CUDA and C++. All performance critical parts of the algorithm are done in CUDA. However, such things as tree construction and some other algorithms are implemented in C++ on the host.

4.1 Hemisphere sampling

For irradiance computations we use the progressive evaluation algorithm with Monte Carlo path tracing. All irradiance cache records are placed in 'active records list'. For each active record we use a sequence of randomly distributed (but coherent) hemisphere samples – 4096, 16384, 65536 and etc. At first we use 4096 rays for all records in the list. If estimated error for a record is small enough we discard that record from 'list of active records' and process remaining records with 16384 rays (the next value in the sequence). We repeat this process until all records are evaluated or the maximum number of samples per irradiance record is achieved. Using the sequence of pre-generated samples instead of simple random rays is important because we can save rays coherency at least for the first bounce and have valuable speed-up (~ 2-4 times) for ray tracing on GPUs.

To evaluate convergency for a record we use the approach described in [Krivánek et al. 2006] accumulating odd and even partial sums of lighting integral.

We used 'Hammersley' sampling technique described in [Suffern 2007] to cover hemisphere with samples. To have more coherent groups of rays we used stratification (subdivide the hemisphere into sectors and generate $32*k$ rays for each sector where $k \geq 1$). Actually we just need to group 'Hammersley' points in groups of size $32*k$. Initially we do that on the CPU in tangent space. On the GPU we transform directions from tangent to object space to get correct hemisphere sampling.

During the hemisphere sampling, we also calculate initial validity radius for each irradiance cache point using 'sphere split approximation' [Krivánek et al. 2008].

4.2 Insertion records into octree

The insertion process is done on the CPU. Our implementation inserts a set of records in one transaction, and we modified the original insertion algorithm, described in [Krivánek et al. 2008] and [Pharr and Humphreys 2010].

```
procedure Insert (
    self      : inout Irradiance_Cache;
    records   : in array of IC_Record ) is
begin
    EvaluateIrradiance(records);
    self.auxOctree.Insert(records);
    ValidityRadiusClamping(records, self.auxOctree);
    self.mainOctree.Insert(records);
end Insert;
```

The problem with inserting multiple records is that in several cases, we can find a large set of closely-located records, with overlapping validity radiuses. This is a problem, because in those regions octree leafs will contain a large list of points and interpolation becomes slow. This motivated us to develop a special algorithm for decreasing validity radiuses during insertion. Our insertion consists of 3 phases. First, we consider irradiance cache records as points (not as spheres!) and insert them into an auxiliary octree. This octree will be used to speed-up location of k-nearest points (irradiance cache records).

We call the second step ‘validity radius clamping’. It treats irradiance cache records as spheres. The goal of this step is to decrease validity radius for each point. For each irradiance cache record it locates k nearest neighbors (k is 4-7) in ‘different directions’ and if the validity radius of the current point is greater than the distance to the farthest point, the validity radius is clamped to this distance.

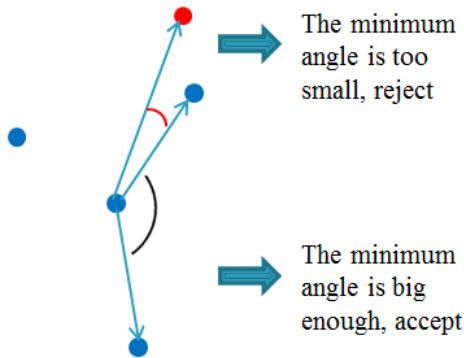


Figure 1: Angle criterion of filtering nearest neighbors.

By ‘different directions’ we mean that while we look for neighboring points we calculate the angle between a new candidate and all the points that we have already found (Fig. 6). If the angle between the direction to a new point and any direction to a point we already found is too small, we do not consider this point, i.e. we do not add it into the nearest neighbors list.

Last we also consider irradiance cache records as spheres. But validity radiuses of these records were clamped by the previous step. The goal of the last step is to insert all points into the final octree that will be used for fetching irradiance from the cache on the GPU. Validity radius clamping is an important part of the algorithm. Table 1 shows performance improvements gained by introducing our validity clamping approach.

Scene	IC1	IC2	look-up acceleration
-------	-----	-----	----------------------

Teapot	29 ms	5.9 ms	4.9 times
Dragon	18 ms	5.0 ms	3.7 times
Conference	25 ms	4.4 ms	5.6 times
Sponza	16 ms	7.1 ms	2.3 times
Cry-Sponza	33 ms	8.3 ms	4.0 times

Table 1. The column marked IC1 presents time (in milliseconds) required to perform one million look-up operations when validity radius clamping is disabled. The column marked IC2 presents time, required to perform one million look-up operations with enabled validity radius clamping. The last column represents acceleration factor. All measurements were done with GTX560 HW.

Thus, inspired by Krivánek’s Neighbor Clamping, we introduce a new validity clamping radius criterion in order to accelerate look-up operation by means of density control.

4.3 Fast Octree Look-Up

We use interpolation formula proposed by Tabellion and Lamorlette in [Tabellion and Lamorlette 2004] and stackless octree look-up as described in [Krivánek et al. 2008]. We have found that the stackless approach is very efficient on GPUs if only several irradiance cache records are stored in octree leaves. The key advantage of the multiple reference octree is the stackless ‘root to leaf’ look-up algorithm. To find all irradiance cache records, which validity radiuses overlaps with a given point, we need to traverse from the root to a leaf and then just iterate through the list of cache points we stored in a leaf.

4.4 Results overview

The results of our renderer are presented in Table 2. We target high quality images at 1920x1200 resolution and we used world space irradiance interpolation. As a result our irradiance cache contains a large record set (100K-200K). Due to the high precision requirements we usually start irradiance evaluation with 4096 Monte-Carlo samples. The ‘Conference Room’ scene has 6 area lights under the ceiling and a significant part of rendering time was taken by soft shadows. Due to a weak indirect component in this scene path tracing converges fast enough and acceleration factor is lower (only 3 times) compared to other scenes.

‘Sponza’ and ‘Crytek-sponza’ scenes (in contrast to Conference Room) have strong indirect illumination and acceleration on these 2 scenes was even higher (14 and 18 times accordingly) than expected. For example, having ~200K records for the last scene, one can’t expect more than $1920 \times 1200 / 200000 = 11$ times acceleration. However we found that on some complex scenes (even disregarding total triangle count), like ‘Crytek-sponza’ naive path tracing is inefficient and the ray tracing performance is far from 60M rays per second. As a result grouping rays to coherent packets, when sampling hemisphere, provides a great advantage for GPU ray tracing performance and for the ‘Sponza’ and ‘Crytek-sponza’ scenes we have super-linear acceleration.

We calculated square error (with ‘The Compressorator’ [Compressorator]) and PSNR (with MatLab) metrics to have a numerical estimation of an image difference. PSNR, for HDR images, is much higher (than for LDR) because absolute value of the signal is higher on HDR images.

4.5 Quality discussion and analysis

For our current implementation we used Monte-Carlo path tracing to evaluate irradiance. We start from 4096 hemisphere samples with 2 path tracing bounces and continue to increase the number of samples with our progressive evaluation algorithm. In contrast to evaluating IC records, path tracing requires on average ~1000

samples per pixel. The produced noise is high frequency and it is filtered by the human eye. However, when we consider irradiance cache, the error will be splashed over the surface resulting in low frequency noise that appear to a human eye as “dirty spots” (Fig. 3). To suppress these artifacts we use more samples per record.

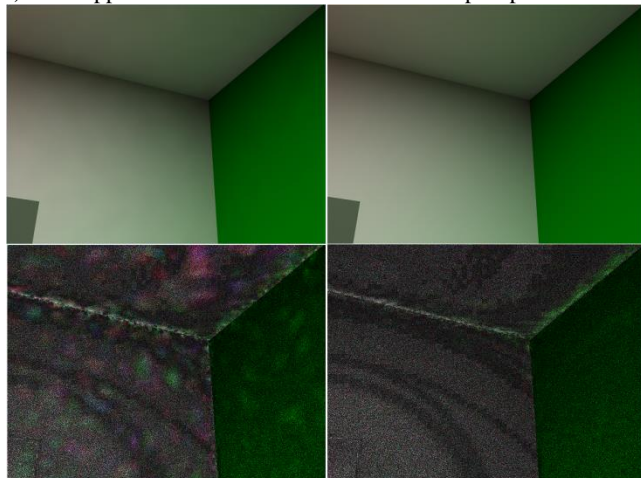


Figure 3: Top row: IC results produced with 1024 and 16384 samples per record. The bottom row: magnified difference (x32) between corresponding IC result and path traced reference.

4.6 Bottleneck analysis

We have measured that during irradiance cache construction ~80% of the time is spent on evaluating records (i.e. hemisphere sampling with Monte-Carlo path tracing) and it takes ~50-90% of the total rendering time. One way to reduce this time is to use fewer samples with one bounce. This will work much faster because on the first bounce we have coherent sets of rays (and the noise is less than for 2 or more bounces). However this will prevent us from computing indirect lighting from multiple diffuse bounces. Another choice is to use photon mapping with final gathering [Krivaneck et al. 2008] instead of Monte-Carlo path tracing. We believe this idea should give us a performance benefit and we'll investigate this in our future research. We also think that using recursive irradiance cache [Krivaneck et al. 2008] is a promising idea; it allows tracing only coherent set of rays to transport light from one level of the cache to another (or even use rasterization).

We perform interpolation in the world space and as a result the IC generation algorithm places a lot of records near tiny geometry details. We suppose screen space IC should be used for primary visible points.

The octree construction (insertion of records) is not a bottleneck in our implementation; it usually takes ~15% of irradiance cache construction time. The remaining 5% of the time was spent on ray tracing during IC construction, building of discontinuity maps (both geometry and irradiance), data transfers between GPU and CPU.

Regarding the final rendering, the irradiance cache look-up operation takes 20% in average of the total rendering time. The ray tracing (incoherent rays) takes 60-70% of this time and the rest is taken by the shading related works and pipeline overhead.

5. LITERATURE

- [1] [Aila and Laine2009] Aila, T. and Laine, S. 2009. *Understanding the efficiency of ray traversal on GPUs*. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana, August 01 - 03, 2009). S. N.

- [2] [Debattista et al. 2006] K. Debattista, L.P. Santos, A. Chalmers, *Accelerating the irradiance cache through parallel component-based rendering*, in: EGPGV2006 - 6th Eurographics Symposium on Parallel Graphics Visualization. Eurographics, May 2006, pp. 27-34.
- [3] [Dubla et al. 2009] Piotr Dubla, Kurt Debattista, Luis Paulo Santos, and Alan Chalmers. *A wait-free shared-memory irradiance caching*. IEEE Computer Graphics and Applications, 2010.
- [4] [Frolov et al. 2011] V. Frolov, A. Kharlamov, A. Ignatenko. “Biased solution of integral illumination via irradiance caching and path tracing on GPUs”. *Programming and Computer Software*, Volume 37, Number 5 (2011), 252-259, DOI: 10.1134/S0361768811050021
- [5] [Gauton et al. 2005] Pascal Gautron, Jaroslav Krivánek, Kadi Bouatouch, and Sumanta Pattanaik. *Radiance cache splatting: A GPU-friendly global illumination algorithm*. In *Proceedings of Eurographics Symposium on Rendering*, June 2005.
- [6] [Gassenbauer et al. 2011] Václav Gassenbauer, Jaroslav Krivánek, Kadi Bouatouch, Christian Bouville, Mickaël Ribardière. *Improving Performance and Accuracy of Local PCA*. 4 NOV 2011 DOI: 10.1111/j.1467-8659.2011.02047.x
- [7] [Jensen et al. 2002] Jensen, H. W., Suykens F., Christensen Per H., Kato T. *A Practical Guide to Global Illumination using Photon Mapping*. SIGGRAPH 2002 Course Note #43. ACM, July 2002. (San Antonio, USA, July 21-26).
- [8] [Krivánek et al. 2006] Jaroslav Krivánek, Kadi Bouatouch, Sumanta Pattanaik, Jiriára. *Making Radiance and Irradiance Caching Practical: Adaptive Caching and Neighbor Clamping*. Eurographics Symposium on Rendering, 2006.
- [9] [Krivaneck et al. 2008] Krivánek, J., Gautron, P., Ward, G., Jensen, H. W., Christensen, P. H., and Tabellion, E. 2008. *Practical global illumination with irradiance caching*. In ACM SIGGRAPH 2008 Classes (Los Angeles, California, August 11 - 15, 2008). SIGGRAPH '08. ACM, New York, NY, 1-20.
- [10] [Manfred and Gunther 2007] Manfred Ernst, Gunther Greiner. *Early Split Clipping for Bounding Volume Hierarchies*, Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing, p.73-78, September 10-12, 2007 doi=10.1109/RT.2007.4342593
- [11] Morton, G. M. (1966), *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*, Technical Report, Ottawa, Canada: IBM Ltd.
- [12] [Papaioannou 2011] G. Papaioannou, *Real-Time Diffuse Global Illumination Using Radiance Hint*. Presented at High Performance Graphics 2011, Vancouver, Canada, Aug. 2011.
- [13] [Pharr and Humphreys 2010] Pharr M., Humphreys G.: *Physically Based Rendering: From Theory to Implementation*, 2nd edition. Morgan Kaufmann, 2004.
- [14] [Suffern 2007] Kevin Suffern. *Ray Tracing from the Ground Up* A. K. Peters, Ltd. Natick, MA, USA©2007 ISBN:1568812728
- [15] [Tabellion and Lamorlette 2004]Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer-generated films. In *Proceedings of SIGGRAPH*, 2004.DOI: 10.1145/1186562.1015748
- [16] [Ward et al. 1988] Ward, G., Rubinstein, F., and Clear, R. 1988. A ray tracing solution for diffuse interreflection. In *SIGGRAPH 1988, Computer Graphics Proceedings*.
- [17] [Wang et al. 2009] Wang R., Zhou K., Pan, M., and Bao, H. 2009. *An efficient GPU-based approach for interactive global illumination*. *ACM Trans. Graph.* 28, 3 (Jul. 2009), 1-8.
- [18] [Ward and Heckbert 1992] Ward, G. J. and Heckbert, P. 1992. Irradiance Gradients. In *Third Eurographics Workshop on Rendering*, 85.98.
- [19] [Compressorator] The Compressorator. A tool for compressing textures and creating mip-map levels. Can visualize image difference and calculate square error.
<http://developer.amd.com/tools/compressorator/pages/default.aspx>

6. ACKNOWLEDGEMENTS

This work was sponsored by the RFFI (Russian Foundation for Fundamental Investigations) grant “MOL_A 12-01 31027”.

Scene	Number of Triangles	Number of IC records	IC creation time	Render pass time	Total time	Naive path tracing time	Acceleration	Square error (png)	PSNR (png)
Teapot	25612	27356	18s	35s	56s	300s	5.3 times	2.5	45
Dragon	871426	71883	52s	36s	88s	490s	5.6 times	3.4	43
Conference	331191	89336	64s*	108s	172s	500s	2.9 times	2.3	47
Sponza	66456	161245	123s	17s	140s	1980s	14.1 times	3.9	41
Cry-Sponza	262267	245369	228s	20s	248s	4632s	18.6 times	5.7	38

Table 2: Test setup. All scenes were rendered in 1920x1200 on GTX560 HW. For path tracing - max samples per pixel was 4000 (however, this number was reached for the 2 last scenes). Image difference and square error were computed with ‘The Compressorator’ tool [Compressorator]. For conference Room we started hemisphere evaluation with 1024 hemisphere samples (instead of usual 4096 samples) (*).

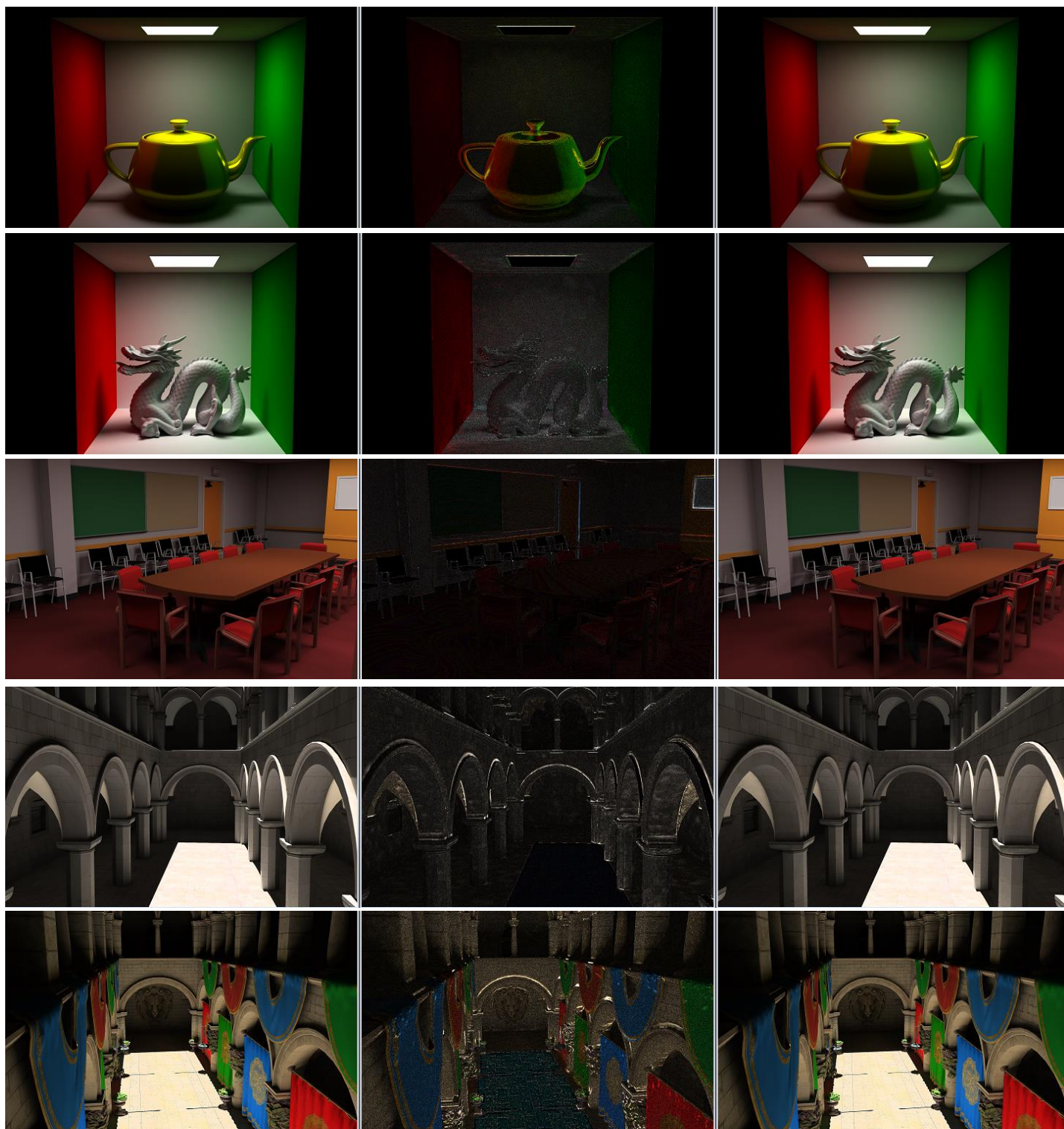


Figure 4: Path tracing compare to our IC implementation. Difference brighter by 1600% (16 times). High Quality images and demo program available at <http://ray-tracing.ru/upload/gc2012/sandbox.zip>