

# Implementing Irradiance Cache in a GPU Realistic Renderer

Vladimir Frolov<sup>(1,2)</sup>, Konstantin Vostryakov<sup>(2)</sup>,  
Alexander Kharlamov<sup>(2)</sup>, Vladimir Galaktionov<sup>(1)</sup>

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

(2) Nvidia

**Abstract.** This work presents an approach to integrating irradiance caching (IC) technique in a complete GPU photorealistic renderer. This work proposes a GPU friendly IC solution, where performance critical parts of an irradiance cache algorithm are done completely on the GPU. The modified algorithm for the GPU is different from a traditional implementation in 2 ways. The first distinction is a predictive nature of our algorithm that allows us to insert a large record set at once instead of inserting records one by one, as in traditional approaches. The second distinction is a new heuristic for validity radius computations. We also consider some low-level details and provide performance analysis of our solution.

**Key words:** Ray Tracing, GPU Global illumination, Irradiance cache, realistic rendering.

## 1 Introduction

For the last decade Graphics Processing 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 due to its simplicity for GPU implementation and scalability. This results in tracing up to ten times more rays than biased alternatives. Moreover, path tracing of complex scenes suffers from highly irregular workload 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 validity radius computation heuristic while inserting records into octree for fast irradiance interpolation on the GPU.

Our main contribution is a high quality IC solution that provides from 4 to 14 times acceleration (with an average PSNR of 40-45 compared to a GPU accelerated path tracing).

## 2 Related Work

The majority of modern papers on ray tracing and global illumination and either irradiance caching are focused on interactive frame rates and fast computations and are not paying enough attention to the quality of their solution. We discuss some of modern papers in the related field further but we would like to underline that we can't provide a quantity comparison of these papers to our work because most of them does not provide a numerical error analysis in contrast to our solution. So, we provide only quantity comparison between proposed IC approach and our own optimized path tracing implementation.

### 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 work [1] provides comprehensive performance analysis of ray tracing on the GPU. Our ray tracing implementation has approximately the same performance on diffuse rays although it's of 1.5x factor slower for coherent frustum tracing. For "Conference Room", "Sibenik" and other commonly used test scenes our ray tracer achieves ~130M rays per sec for primary rays and ~60M rays per sec for diffuse rays on average (measured on a GTX 560 HW). However we found that depending on the scene, path tracing rays after several bounces path tracing rays can be a factor of 2x-4x slower than diffuse rays – this is the case of poor HW utilization due to random memory access and non-uniform workload.

### 2.2 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 [2]. 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 or 2 magnitude order less than the number of pixels – so the irradiance cache is quite efficient and it can greatly speed up the whole rendering. However, irradiance cache is a challenging algorithm, even on a CPU. It has a lot of issues and heuristic approaches that suppress its arti-

facts [3]. A well-known irradiance cache algorithm [2,3] 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”.

### 2.3 Parallel Irradiance cache

It is difficult to parallelize irradiance cache on multi-core CPUs. To understand what the problem is, consider a situation when we have several rendering threads in the simple algorithm discussed above. In this case, each insertion operation will block a part of cache for other threads until the hemisphere sampling is finished. The comprehensive analysis of parallel IC solutions are described at [4] and [5] and a 2 different approaches were introduced.

Mainly, previous researches in parallel IC were focused on the problem of accessing shared memory on SMP systems (or distributed memory on clusters) and achieving efficiency on all processors. Besides the fact that the problem of effective data access and high processors utilization is very important, this is not the only problem when implementing parallel IC. Another challenge is that the IC construction algorithm is based on a variety of heuristics (neighbor clamping and etc) which were used with a serial IC records inserting [3]. When doing parallel implementation and inserting records in batches we have redundant amount of records with strong overlapping of validity areas and because of that poor IC look-up performance. This problem became critical for GPU with the massive number of threads.

PBRT 2.0 [6] also 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. We discuss it further in our paper.

### 2.4 GPU Irradiance cache

GPU irradiance caching was introduced in [7] and described in details in [3]. These papers mainly focus on replacing irradiance interpolation via octree lookups with splatting to avoid traversing hierarchical structures on GPUs. The approach used in [7] 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. [8] 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 [9] 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 [10] to focus on rendering images with glossy reflections and shadows of-

fline. However, both of these approaches work with geometry term and they use predictive nature without further refinement.

The radiance hints method introduces in [11] 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 required 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 and thus we compute the only part of 3D irradiance signal (via computing IC) that is essential for the final image.

The key novelty points of our IC construction are:

1. Usage of 2 different algorithms for directly and indirectly visible surfaces.
2. Image processing algorithms to predict where more IC records should be placed and where we can have only several of them.
3. Z-curve clusterization of irradiance evaluation queries.
4. Adjusting IC records validity radiuses using a new clamping heuristic to get an faster irradiance interpolation.

#### 3.1 Creation of irradiance cache

IC creation process consists of multiple passes (10-20 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;
temp        : 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 := 20;
MIN_CAND_TRESHOLD : Integer := 100;
```

**begin**

```
-- the very first pass
--
geomDiscMap := CreateGeometryDiscMap();
discMap      := Build2DMipMapChain(geomDiscMap);
candidates   := Dithering(discMap);

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

-- multipass construction
--

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

    -- screen space stage
    --
    temp      := CreateIrradianceDiscMap();
    discMap    := Build2DMipMap(temp);
    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 [8] and [10] 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.

### 3.2 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 required.

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, for each mip-level a surface discontinuity texture map is evaluated according to this formula:

$$\text{surfDisc} := k * \text{normDiff} + \text{worldPosDiff};$$

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.**

Here is our dithering implementation in pseudo code:

```
-- in parallel for each screen pixel
```

```
--
d0 := tex2DLod(discontinuity, x, y, 0);
d1 := tex2DLod(discontinuity, x, y, 1);
d2 := tex2DLod(discontinuity, x, y, 2);
...
deterministicSelect := F1(d0,d1,d2,...,x,y);
stochasticSelect    := F2(d0,d1,d2,...,x,y);

if deterministicSelect or stochasticSelect then
    PutRecordInThisPixel(x,y);
end if;
```

For each pixel we read discontinuity value from each mip level of the discontinuity map - (call these values d0,d1,d2,d3,...) and then rely on some functions (F1 and F2) - we decide whether we put record in the target screen pixel or not. We believe the F1 and F2 implementation could be different. We used a set of thresholds for F1 and a stochastic selection with fixed threshold for F2.

#### 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, perform dithering and insert newly obtained points into irradiance cache. This procedure is repeated several times.

#### 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 in Fig. 1 represents viewers' frustum. Red points are not visible from the camera, yet secondary rays can still reach such regions.

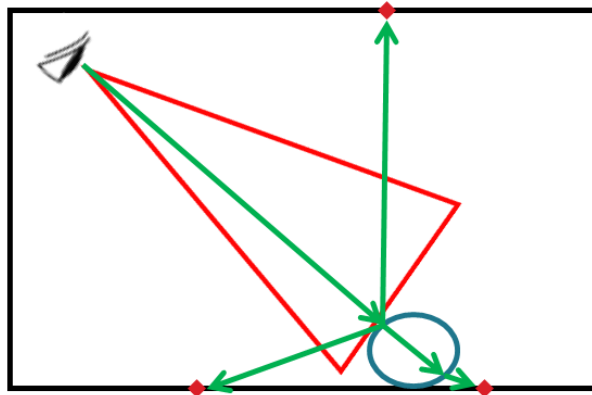


Fig. 1. Rays that are traced from the eye may reach points that are not visible directly.

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 [12].

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’.

However, a very large set of points is produced and we have to select a subset of the best candidates from it. We believe several approaches can be applied to form clusters, we used a simple technique that can be easily ported to GPU. At first, we sort candidates according to 3D Z-Curve [13]. 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. Our motivation to use clustering was to select candidates with maximal interpolation error and this way IC records will be more regularly distributed.

Once all the points are clustered, 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 have 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 significant error in the final image because the probability that rays hit such regions tends to be zero.

### **3.3 Final rendering**

After we have irradiance cache computed we do adaptive path tracing as described in [10] with fetching indirect smooth lighting from the 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.

### **3.4 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.



### 3.5 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 – 1024, 4096, 16384, 65536 and etc. At first we use 1024 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 convergence for a record we use the approach described in [14] accumulating odd and even partial sums of lighting integral. We used ‘Hammersley’ sampling technique described in [15] 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  $k \geq 1$ ). We group ‘Hammersley’ points in groups of size  $32*k$ . We do that once 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 harmonic mean distance [Ward et al 88].

### 3.6 Insertion records in octree

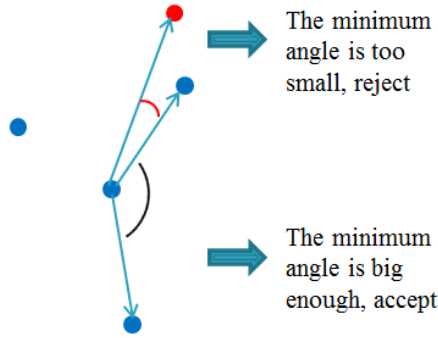
The insertion process is done on the CPU. Our implementation inserts a set of records in one transaction, and we modified the original insertion algorithm, described in [3] and [6].

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



**Fig. 2.** Angular 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. 2). 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.

At 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 time (ms)	IC2 time (ms)	Look-up acceleration
Teapot	30.0 ms	5.8 ms	5.1
Sponza1	59.0 ms	13.0 ms	4.4
Sponza2	44.0 ms	9.1 ms	4.9
Reneissance	20.0 ms	9.6 ms	2.1
Arch-class	53.0 ms	12.0 ms	4.3
Cry-Sponza	45.0 ms	13.0 ms	3.3

**Table 1.** The column marked IC1 time presents time (in milliseconds) required to perform one million look-up operations when validity radius clamping is disabled. The column marked IC2 time 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. The original Krivánek's Neighbor Clamping was performed in all cases - both for IC1 and IC2 columns.

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. To avoid light leaks near tiny geometry details we use original Neighbor Clamping heuristic, extended to the case of simultaneous multiple record insertion.

### 3.7 Fast Octree Look-Up

We use interpolation formula proposed by Tabellion and Lamorlette in [16] and stackless octree look-up as described in [3]. We have found that the stackless approach is very efficient on GPUs if only several irradiance cache records are stored in octree leaves (the timings are presented in table 1).

### 3.8 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. We perform gamma-correction for the final images and a Tone Mapping for several of them (marked in table).

Scene	Triangles number	Rays per sec / Samples per sec;	IC records	IC time	Render pass time
Teapot	25 612	82M / 5.8M**	16994	9.0s	133.7s
Sponza1	66 454	62M / 5.4M	213435	157s	20s
Sponza2	66 454	64M / 5.4M	97789	109s	11s
Renaissance	871 786	64M / 4.3M	306326	167s	10s
Arch-Class	1 010 724	45M / 4.1M	302992	285s	20s
Cry-Sponza	262 267	30M / 2.8M	442332	501s	40s

Scene	Total time (with IC)	Naive path tracing time	Acceleration (times)	Square error (hdr) / (png)	PSNR (png)
Teapot	142.7s	510s	3.6	2.3 / 2.1	46.5
Sponza1	177s	1554s	8.8	1.3 / 2.1	46.4
Sponza2	120s	1695s	14.1	1.6 / 3.6	41.6
Renaissance	177s	1830s	10.3	1.7 / 2.9	43.5
Arch-Class	305s	2001s	6.6	2.1 / 4.3	40.2
Cry-Sponza	541s	3347s	6.2	2.9 / 2.5	45.0

Scene	Avg samples per pixel for Naive Path Tracing	Avg samples per IC record	CPU overhead (octree build)	Tone Mapping
-------	--	---------------------------	-----------------------------	--------------

Teapot	1242	5069	7%	No
Sponza1	3644	4840	19%	No
Sponza2	3972	5398	5.6%	Yes
Reneissance	3416	3156	16%	No
Arch-Class	3562	6432	8%	No
Cry-Sponza	4068	5328	9%	Yes

**Table 2.** Test setup. All scenes were rendered in 1920x1200 on GTX560 HW. Image difference and square error were computed with ‘The Compressorator’ tool [17]. The path tracing accuracy was always set to 5%. Note that path tracing accuracy should not correlate to the IC accuracy except for the cases where the lighting is pure diffuse indirect (i.e. interreflected from diffuse surfaces). The naive path tracing accuracy (as reference method) was set to 5%. Maximum diffuse bounces of path tracing was limited to 2. We used accuracy = 10% for IC records evaluation. The column named ‘Rays per sec/Samples per sec’ show the number of rays per second and samples per second that our renderer performs for naive path tracing. \*\* Our renderer shoots 4 shadow rays per each hit point if only one area light is present in the scene. So the difference between number of rays per second and number of samples per second is higher for this scene.

We also provide zoomed-in side by side comparisons of our IC and path tracing solutions by taking the time we compute image with IC and see what image we will get with path tracing.



**Fig. 3.** Zoomed in comparison (256x128 pixels) of the path tracing and irradiance caching within the same time limit.



**Fig. 4.** Test setup. The difference is magnified 16 times.



**Fig. 5.** IC record positions. Image was rendered in 1024x768.

### 3.9 Bottleneck analysis

For our current implementation we used Monte-Carlo path tracing to evaluate irradiance. We limit number of diffuse light bounces to 2 in naive path tracing and IC. We start from 1024 hemisphere samples and continue to increase the number of samples with our progressive evaluation algorithm. The average samples per IC record is presented in table 2.

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 [3] instead of Monte-Carlo path tracing. We believe this idea should give us a perfor-



mance benefit and we'll investigate this in our future research. We also think that using recursive irradiance cache [3] 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 7-15% 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 on average takes less than 15% of the total rendering time. The ray tracing (incoherent rays) takes 65-75% of this time and the rest is taken by shading and pipeline overhead.

### 3.10 Conclusion

In this work we provided an IC solution for a realistic GPU renderers. We achieved from 4 to 14 times acceleration over our own GPU optimized path tracing with an average PSNR 40-45. Instead of inserting records one by one like in traditional IC approach, we used image processing algorithms to predict IC records distribution and thus inserted large sets of IC records in parallel. We used clustering approach to distribute IC records more regularly and put records in places with maximum error. Also, we introduced clamping heuristic for validity radii in order to solve the problem of overlapping validity areas of IC records. Solving that problem was critical for efficient implementation on massive parallel machines like GPUs when records were inserted in large sets. That way we achieved fast IC look-up.

### 3.11 Acknowledgements

This work was sponsored by the RFFI (Russian Foundation for Fundamental Investigations) grant "MOL\_A 12-01 31027".

## 4 References

1. 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. WARD, G., RUBINSTEIN, F., AND CLEAR, R. 1988. A ray tracing solution for diffuse interrefraction. In SIGGRAPH 1988, Computer Graphics Proceedings.
3. 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. DOI= <http://doi.acm.org/10.1145/1401132.1401213>.

4. 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.
5. Piotr Dubla, Kurt Debattista, Luis Paulo Santos, and Alan Chalmers. A *wait-free shared-memory irradiance* caching. IEEE Computer Graphics and Applications, 2010. accepted for publication.
6. PHARR M., HUMPHREYS G.: Physically Based Rendering: From Theory to Implementation. Morgan Kaufmann, 2010.
7. 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.
8. 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.
9. 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).
10. G. Papaioannou, to be presented at High Performance Graphics 2011, Vancouver, Canada, Aug. 2011.
11. 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
12. Václav Gassenbauer, Jaroslav Krivánek, Kadi Bouatouch, Christian Bouville, Mickaël Ribardiére. Improving Performance and Accuracy of Local PCA. 4 NOV 2011 DOI: 10.1111/j.1467-8659.2011.02047.x
13. Morton, G. M. (1966), *A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*, Technical Report, Ottawa, Canada: IBM Ltd.
14. Jaroslav Krivánek, Kadi Bouatouch, Sumanta Pattanaik, Jiriára. Making Radiance and Irradiance Caching Practical: Adaptive Caching and Neighbor Clamping.
15. Kevin Suffern. Ray Tracing from the Ground Up A. K. Peters, Ltd. Natick, MA, USA©2007 ISBN:1568812728
16. Eric Tabellion and Arnaud Lamorlette. An approximate global illumination system for computer-generated films. In Proceedings of SIGGRAPH, 2004.DOI: 10.1145/1186562.1015748

17. The Compressonator. A tool for compressing textures and creating mip-map levels. Can visualize image difference and calculate square error.  
<http://developer.amd.com/tools/compressonator/pages/default.aspx>

## About the authors

Vladimir Frolov is a post graduate student in Keldysh Institute of Applied Mathematics. He works at NVIDIA as developer technology engineer. His contact email is [vfrolov@graphics.cs.msu.ru](mailto:vfrolov@graphics.cs.msu.ru).

Konstantin Vostryakov is a PhD researcher. He graduated from Keldysh Institute of Applied Mathematics. He works at Nvidia in OptiX team. His contact email is [kvostryakov@nvidia.com](mailto:kvostryakov@nvidia.com).

Alexander Kharlamov is a PhD student at Computational Mathematics and Cybernetics department of Lomonosov Moscow State University. Alexander works at NVIDIA as a developer technology engineer. His contact email is [akharlamov@nvidia.com](mailto:akharlamov@nvidia.com).

Vladimir Galaktionov, PhD, prof. Head of the department at Keldysh Institute of Applied Mathematics RAS. His contact e-mail: [vlgal@gin.keldysh.ru](mailto:vlgal@gin.keldysh.ru).