EUROGRAPHICS 2011 / M. Chen and O. Deussen
(Guest Editors)

Volume 30 (2011), Number 2

Combinatorial Bidirectional Path-Tracing for Efficient Hybrid CPU/GPU Rendering

Anthony Pajot[1], Loïc Barthe[1], Mathias Paulin[1], and Pierre Poulin[2]

[1]IRIT-CNRS, Université de Toulouse, France [2]LIGUM, Dept. I.R.O., Université de Montréal, Canada.

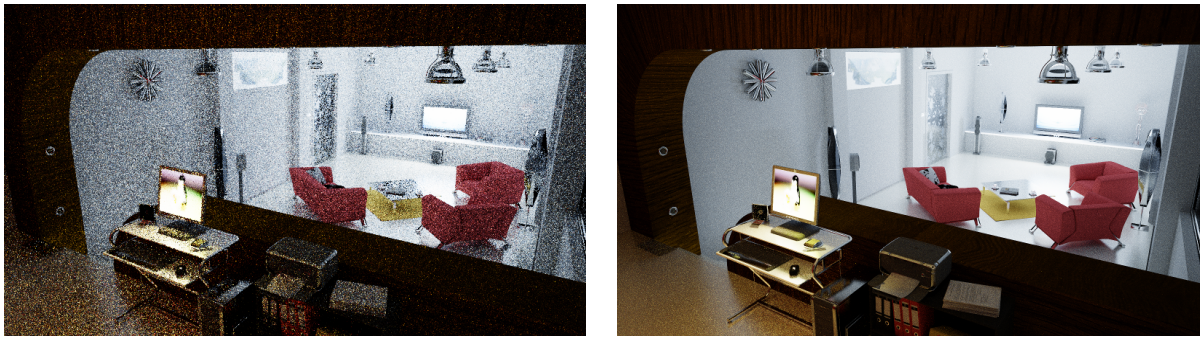


Figure 1: *Images of a scene with a large dataset (*758K *triangles, lots of textures) featuring complex lighting conditions (glossy reflections, caustics, strong indirect lighting, etc.) computed in respectively* 50 *seconds (left) and one hour (right). Standard bidirectional path-tracing requires respectively* 11 *minutes and* 13 *hours to obtain the same results.*

Abstract

This paper presents a reformulation of bidirectional path-tracing that adequately divides the algorithm into processes efficiently executed in parallel on both the CPU and the GPU. We thus benefit from high-level optimization techniques such as double buffering, batch processing, and asyncronous execution, as well as from the exploitation of most of the CPU, GPU, and memory bus capabilities. Our approach, while avoiding pure GPU implementation limitations (such as limited complexity of shaders, light or camera models, and processed scene data sets), is more than ten times faster than standard bidirectional path-tracing implementations, leading to performance suitable for production-oriented rendering engines.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture I.6.8 [Simulation and Modeling]: Type of Simulation—Monte-Carlo

1. Introduction

Global illumination brings a lot of realism to computer-generated images. Therefore, production-oriented rendering engines use it to reach photorealism.

Algorithms to compute global illumination have to meet a certain number of constraints in order to be seamlessly integrated in a production pipeline:

- From an *artist* point of view, the algorithm should have intuitive parameters, and should be able to provide interactive feedback as well as high quality final images.
- From a *scene-design* point of view, it should be able to manage huge datasets as well as complex and flexible shaders, various light models, and various camera models.
- From a *data-management* point of view, it should avoid as much as possible precomputed data. Indeed, it is tedious to keep these data synchronized across artists that work on the same scene, or between the computers of a renderfarm.

- From a *computational* point of view, it must be robust to handle highly dynamic scenes and all-frequency indirect lighting, to give the artists complete freedom on their designs. For automated rendering, it must give predictable and reproducible results in a given time frame. Ideally, it should be easy to use on clusters, to be able to render one image using all the ressources of a renderfarm.

Methods that are used nowadays mostly rely on point clouds or other type of precomputed representations [KFC*10]. As they rely on precomputed data, interactive feedback is not straightforward, as these data should be recomputed each time the scene changes. Even though being predictable and able to handle very large amount of data, precomputed representations still have problems handling highly dynamic or high-frequency indirect lighting. Moreover, production pipelines must be adapted appropriately to keep these data in sync during the production process, and computing a single image on a cluster can be done only once the data have been computed.

To remove all these problems, unbiased methods have been investigated, and path-tracing based algorithms begin to be mature enough to be successfully used in the movie industrie [Faj10]. In addition to being potentially fully automatic (thus user-friendly), unbiasedness makes these methods easy to deploy on clusters, as independent renderings can be simply averaged to compute the final image. As they do not require any precomputed data and do not rely on any interpolation scheme, they also naturally handle highly dynamic scenes. Moreover, they use independent samples, thus precision requirements such as a given number of samples per pixel are easy to formulate. Finally unlike sequential methods, the number of samples computed in a given time can be measured so that the results are predictible and reproducible when the time frame is fixed.

Nevertheless, path-tracing exhibits large variance when high-frequency or strong indirect lighting effects such as caustics are present in a scene, leading to visually unpleasant artefacts in the rendering. To reduce these artefacts, constraints can be added to the indirect lighting, *e.g.* reducing the sharpness of glossy reflections [Faj10], or enlarging lights. Although interactive feedback can be provided for scenes where path-tracing has a very low variance, a large amount of time is needed to obtain a rough preview of the final appearance for scenes with high variance. On a more general point of view, unbiased methods have a larger computational cost than methods based on precomputed data, which is a problem for wide acceptance.

Bidirectional path-tracing (BPT) [VG94] [LW93], has the same advantages as path-tracing, but is much more robust with respect to indirect lighting, providing low variance results even for complex lighting conditions. Even though more computationally efficient than path-tracing, it remains too slow for interactive feedback, and is still slower than methods based on precomputed data. Recently, attempts at making it faster by using GPU as a co-processor have been presented in the rendering community [OMP10], however, the proposed implementation does not allow an efficient collaboration of the CPU and GPU, keeping the most of the processing charge on the CPU while the GPU remains mostly idle.

Contribution: In this paper, we combine correlated sampling and standard BPT to efficiently use both CPU and GPU in a cooperative way (Section 3). The basic principle of BPT is to repeatedly sample an optical path leaving from the camera, and an optical path leaving from the light. Complete paths are then created by linking together each subpath of the camera path with each subpath of the light path. The last vertex of each subpath are called linking vertices, and the segment between the two linking vertices is the linking segment. A complete path created this way contributes to the final image if the linking vertices are mutually visible, and if some of the energy arriving to the light linking vertex is scattered to the camera path. Instead of combining two paths, we combine sets of camera and light paths, computing the values needed for linking on the GPU. As each camera path is combined with each light path, many more linking segments are available, allowing us to use the GPU at its maximum without increasing the cost of sampling the paths (Section 4). We then interleave the CPU and GPU parts in order to obtain an algorithm where both the CPU and GPU are always busy (Section 5). This reformulation reduces the processing time by a factor varying between 12 and 16 compared to standard BPT (Section 6), allowing feedback in less than a minute even for complex scenes, and the computation of high-quality images in one hour, as shown in Figure 1.

2. Related Work

If not considering computational efficiency and GPU use, both biased and unbiased algorithms that do not use precomputed data exist to produce high-quality images.

On the unbiased side, sequential methods based on Markov-Chain Monte-Carlo [VG97, KSKAC02, CTE05, LFCD07] have been used to improve the robustness of standard Monte-Carlo methods for very difficult scenes. Unfortunately, they can be highly dependent on the starting state of the chain, and do not provide feedback as rapidly as standard Monte-Carlo methods, since the time to cover all the screen is typically longer. The gain that these methods bring is most visible on very difficult scenes, but remains quite limited for more common scenes, for which standard BPT is highly efficient.

On the biased side, Hachisuka *et al.* [HOJ08, HJ09] introduced progressive photon mapping and stochastic progressive photon mapping, two consistent algorithms based on photon mapping. Even though robust, efficient, and able to produce high-quality images, being consistent instead of unbiased prevents these algorithms to be directly usable in renderfarms for single image computations. Instead, they need to be specifically adapted to avoid artefacts in the final images.

Using both the CPU and GPU in a cooperative way can

provide a large gain of performance, allowing the methods above to provide high-quality results or rough previews significantly faster. Attempts at isolating parts of algorithms to execute them on GPU are examined in rendering engines, such as in *luxrender* [Lux10], where intersection tests are performed on the GPU. The main problem that face developers is keeping both CPU and GPU busy all the time. In general, the CPU is too slow to provide enough work to the GPU. More generally, it is not easy to adapt the algorithms presented above to efficiently use the GPU to compute intermediate data, without restricting the size of the datasets nor the complexity of the shaders. In fact, sampling, which must be done on CPU as it involves all the dataset and the shaders, would in general require much more time to be computed than the GPU part, leading to a negligible gain.

3. Combinatorial Bidirectional Path-Tracing (CBPT)

3.1. Base Algorithm

In BPT-based algorithms, a camera path $x = (\mathbf{x}_0, \dots, \mathbf{x}_c)$ and a light path $y = (\mathbf{y}_0, \dots, \mathbf{y}_l)$ are sampled. $\mathbf{x}_0, \dots, \mathbf{x}_c$ are called *camera vertices*, $\mathbf{y}_0, \dots, \mathbf{y}_l$ are called *light vertices*. For each vertex $\mathbf{x}_i$ or $\mathbf{y}_j$ located on the surface of an object, the parameters of the bidirectional scattering distribution function (BSDF) are computed using a shader tree. Complete paths are then created by linking subpaths $(\mathbf{x}_0, \dots, \mathbf{x}_i)$ and $(\mathbf{y}_0, \dots, \mathbf{y}_j)$, for all the possible couples (i, j). The number of segments of each complete path is $i + j + 1$, and the linking segment is the segment $(\mathbf{x}_i, \mathbf{y}_j)$. Let function $g_C(x, i)$ give the energy transmitted by x from $\mathbf{x}_i$ to $\mathbf{x}_0$, and $g_L(y, j)$ give the energy transmitted by y from $\mathbf{y}_0$ to $\mathbf{y}_j$. The energy emitted from $\mathbf{y}_0$ which arrives to $\mathbf{x}_0$ *via* the path $z = (\mathbf{y}_0, \dots, \mathbf{y}_j, \mathbf{x}_i, \dots, \mathbf{x}_0)$ is then:

$$\begin{aligned} g_{i,j}(x,y) = & g_L(y,j) G(\mathbf{y}_j, \mathbf{x}_i) g_C(x,i) \times \\ & f_s(\mathbf{y}_{j-1} \to \mathbf{y}_j \to \mathbf{x}_i) \times \\ & V(\mathbf{y}_j, \mathbf{x}_i) \times \\ & f_s(\mathbf{y}_j \to \mathbf{x}_i \to \mathbf{x}_{i-1}) \end{aligned} \tag{1}$$

where f_s is the BSDF, V is the visibility function (1 if unoccluded, 0 otherwise), and G is the geometric term.

We define the *basic contribution* $f_{i,j}(x,y)$ of such a complete path as:

$$f_{i,j}(x,y) = \frac{w_{i,j}(x,y) g_{i,j}(x,y)}{p_{i,j}(x,y)}. \tag{2}$$

$p_{i,j}(x,y)$ is the density probability with which the two subpaths have been sampled, and $w_{i,j}(x,y)$ is the multiple importance sampling (MIS) weight [VG95].

In our implementation, we use the direct BSDF probability density function (PDF) p to sample directions for the camera path, and the adjoint BSDF PDF p^* to sample directions for the light path, and the balance heuristic [VG95] to compute the MIS weights:

$$w_{i,j}(x,y) = \frac{p_{i,j}(x,y)}{\sum_{s,t} p_{s,t}(x,y)} \tag{3}$$

where each (s,t) couple is one of the possible techniques with which z could have been sampled. Computing this weight requires to compute $p(\mathbf{x}_{i-1} \to \mathbf{x}_i \to \mathbf{y}_j)$ and $p^*(\mathbf{y}_j \to \mathbf{x}_i \to \mathbf{x}_{i-1})$ using the BSDF at $\mathbf{x}_i$, and $p(\mathbf{x}_i \to \mathbf{y}_j \to \mathbf{y}_{j-1})$ and $p^*(\mathbf{y}_{j-1} \to \mathbf{y}_j \to \mathbf{x}_i)$ using the BSDF at $\mathbf{y}_j$.

When either i or j are less than 1, the corresponding terms are not based on the BSDF, but instead on the light or camera properties. If $j = -1$, it means that $\mathbf{x}_c$ is on a light, making a complete path by itself.

The data that depend on both $\mathbf{x}_i$ and $\mathbf{y}_j$ has to be computed per linking segment, and is the most time-consuming task when computing the contribution of a complete path. These data can be computed on the GPU very efficiently, in parallel for each linking segment. Unfortunately, producing a sufficient number of linking segments would require to sample and combine a very large number of pairs, leading to very large CPU costs, large memory footprints both on CPU and GPU, and very time consuming CPU-to-GPU memory transfers.

The key idea allowing us to use both CPU and GPU efficiently is to sample populations of N_C camera paths and N_L light paths independently on CPU, and then combine each camera path with each light path. This leads to the combination of $N_C \times N_L$ pairs of paths, and allows us to have largely enough linking segments to benefit from the processing power of GPUs without requiring larger sampling costs. Combining all camera paths with the same light paths introduces a correlation in the estimations, but does not lead to bias in the average estimator.

In practice, we have three kernels which compute, for each linking segment $(\mathbf{x}_i, \mathbf{y}_j)$ in parallel:

- the visibility term $V(\mathbf{x}_i, \mathbf{y}_j)$,
- the shading values involving the BSDF of the camera point: $f_s(\mathbf{y}_j \to \mathbf{x}_i \to \mathbf{x}_{i-1})$, $p(\mathbf{x}_{i-1} \to \mathbf{x}_i \to \mathbf{y}_j)$, and $p^*(\mathbf{y}_j \to \mathbf{x}_i \to \mathbf{x}_{i-1})$, if $\mathbf{x}_i$ has an associated BSDF (*i.e.* it is neither on the camera lens nor on a light),
- the shading values involving the BSDF of the light point: $f_s(\mathbf{y}_{j-1} \to \mathbf{y}_j \to \mathbf{x}_i)$, $p(\mathbf{x}_i \to \mathbf{y}_j \to \mathbf{y}_{j-1})$, and $p^*(\mathbf{y}_{j-1} \to \mathbf{y}_j \to \mathbf{x}_i)$, if $\mathbf{y}_j$ has an associated BSDF.

If $\mathbf{x}_i$ or $\mathbf{y}_j$ does not have an associated BSDF, the probabilities (probability to have sampled the light, probability density to have sampled the point on the light, probability density to have sampled the direction from the camera, *etc.*), and the light emission and importance emission terms are computed on CPU, to keep the flexibility on camera and light models that can be used.

The final contributions of a pair (x,y) can then be split into two parts. The first part is the sum of all the basic contributions that affect the image location intersected by the first segment of x. We denote it as the bidirectional contribution: $f^b(x,y) = \sum_{i>0, j\neq -1} f_{i,j}(x,y) + f_{c,-1}(x,y)$ and we call *bidirectional image* the image obtained by considering only the bidirectional contributions. The second part contains all the contributions obtained by light-tracing, each affecting a different image

location: $\{f_{0,0}(x,y), f_{0,1}(x,y), \ldots, f_{0,l}(x,y)\}$. We call *light-tracing image* the image obtained by adding all the contributions from light-tracing, each multiplied by the number of pixels N_p of the final image. In our implementation, light-tracing does not contribute to direct lighting, as it brings a lot of variance for this type of light transport.

As a result, a step of CBPT consists in:

1. sample a camera population $\{x\}$ of N_C paths, and a light population $\{y\}$ of N_L paths;
2. compute the combination data for these two populations on GPU;
3. compute the contributions of each pair of paths, splatting N_C values to the bidirectional image, and splatting the light-tracing contributions to the light-tracing image.

Note that as is, our algorithm does not directly handle motion blur, but it can be integrated in a straightforward manner by sampling each $(\{x\},\{y\})$ population couple with a specific value of time, *i.e.* all the paths of the two populations have the same time value, and this value is different for each couple of populations.

3.2. Discussion

Setting N_C and N_L: Ideally, we would like to always be perceptually faster than standard BPT. Perceptually faster means computing more camera paths per second, with each camera path being combined with $N_L > 1$ light paths. This leads to a similar or faster coverage of the image, with each camera path bringing a lower-variance estimate than in standard BPT, leading to perceptually faster convergence. N_C and N_L can be computed to ensure faster perceptual convergence, by measuring the time t_b needed by BPT to sample, combine, and splat the contribution for a pair of paths, and the time $t_s(N_C,N_L)$ needed by CBPT to perform one step. As the combination is the most time consuming part of a step, $t_s(N_C,N_L)$ is roughly constant as long as the number of pairs $P = N_C \times N_L$ remains constant. Therefore, for a fixed P, an appropriate N_C value is such that

$$N_C > \frac{t_s(P)}{t_b}. \tag{4}$$

A lower N_C value will lead to lower-variance estimate of each path, larger value will lead to faster coverage, but also more correlation. A side-effect of Equation (4) is that if N_C, computed using this equation, is such that N_L would be < 1, this indicates that the machine on which CBPT is running is not fast enough to bring any advantage over standard BPT *for the chosen P*.

Light-tracing: The discussion above does not take into account light-tracing, and using Equation (4) generally gives N_L values that are small, leading to high-variance caustics. Light-tracing does not really take advantage of the GPU combination system, as each light subpath is combined with only one vertex of a camera path, namely the vertex which lies on the lens of the camera. Moreover, contributions for different camera paths are in general very similar, or even equal when using a pinhole camera, as all the lens vertices are at the exact same location. We therefore choose to compute light-tracing using a standard CPU-based light-tracer.

At each step of CBPT, we sample N_T light paths ($\{y_{lt}\}$) and compute their light-tracing contributions. In general, we choose N_T close to N_C to get approximately the same bidirectional/light-tracing ratio as standard BPT. This leads to the final algorithm for a step of CBPT, presented in Algorithm 1.

Algorithm 1 A complete step of CBPT.

```
sample({x})
sample({y})
upload({x}, {y})
gpu_comp({x}, {y})
combine({x}, {y})
sample({y_lt})
compute_lt({y_lt})
```

Correlated sampling: Correlated sampling can take several forms, such as re-using previous paths in order to improve the sampling efficiency [VG97, CTE05], or re-using a small number of well-behaved random numbers to compute different integrals [KH01]. In our method, the camera and light paths are all sampled independently using different random numbers, as in standard BPT. Therefore, complete paths are sampled in a correlated way, as they are created by linking the subpaths in all possible ways. To avoid visible correlation patterns in the final image while ensuring a proper coverage of the image, the image-space coordinates that are used for each camera path are generated in an array, using a stratified scheme over the entire image, with four samples per pixel. This array of samples is then shuffled. When sampling a camera population, each path uses the samples sequentially in the array, leading to paths that most likely contribute to different parts of the image. Therefore, correlation is present, but as it is spread randomly over the image, no regular patterns appear. This array is regenerated each time all the samples have been used. Adaptive sampling can be used by similarly caching a sufficient number of image coordinates that should be computed according to the sampling scheme, and then shuffling this array of coordinates.

4. Efficient Computation of Combination Data

Our algorithm requires an efficient computation of the combination data on the GPU. In this section, we suppose that for each vertex of the two populations $\{x\}$ and $\{y\}$, we have the position, the BSDF parameters, and the direction to the previous vertex in the path. The size of this data is in $O(N_C + N_L)$. As there are typically few vertices in populations, the GPU memory requirements are very low for the population data. Combining populations exhaustively avoids uploading the $O(N_C \times N_L)$ linking segment array that would otherwise be necessary.

We now give some high- and low-level details on our implementation. Figure 3 shows how the techniques we use are put together.

High-level details: The computation is divided into three main steps: visibility (blue V rectangles in Figure 3), BSDF and PDF computations – called *shading* computations from now on – for camera vertices (green C rectangles), and shading computations for light vertices (red L rectangles). For each step, we divide the work into batches of fixed size, each having an associated memory zone on the CPU-side memory (the batch id where results are downloaded is indicated in the download rectangle). On the GPU-side, we use two buffers of fixed size to store the results of the batches (represented by respectively black and white rectangles inside each task). Using batches allows us to compute results of the current batch while downloading results of the previous batch to the CPU, leading to an increased efficiency. This also avoids the need for any array of size $O(N_C \times N_L)$ on the GPU-side, making the N_C and N_L values bounded only by the CPU-side memory capacity. In practice, this provides more space for the scene's geometry that is needed for the visibility tests.

As is, some shading computations will be done even though the linking vertices are not mutually visible. In fact, for the shading models we use [AS00, WMLT07], introducing an array to only compute the useful shading is much less efficient, as computing on CPU and uploading this array for each batch takes more time than directly computing all the shading values.

Low-level details: We use NVidia's CUDA language for GPU computations. The CPU-side work consists only in synchronization, and is performed in a CUDA-specific thread, thus not interfering with the main computational threads. All the positions, directions, and BSDF data are stored in linear arrays (structure-of-array organisation), that are re-used across populations to avoid memory allocations, and enlarged if needed. Each array is accessible through textures, because each of the values is used many times (once for each linking segment to which a vertex belongs), and generally in coherent ways (subsequent threads are likely to use the same data, or nearby data).

For visibility, we use an adapted version of the radius kd-tree GPU raytracing implementation by Segovia [Seg08], which gives a reasonable throughput and is well suited for individual and incoherent rays that are not stored in an array. The rays are effectively built from the thread index *idx*, by retrieving the camera and light vertices from their indices as (idx/V_L) and $(idx \bmod V_L)$ respectively, where V_L is the number of vertices in the light population.

The same indexing scheme is used for the camera shading computations, which makes a single BSDF processed by consecutive threads, as illustrated by Figure 2. Each thread handles one linking segment. This leads to a very good locality in the accesses to the textures containing the BSDF parameters, as well as a very good code coherency in the BSDF evaluation code. In fact, for most warps, the BSDF parameters are the same across all the threads, the only difference between consecutive threads being the directions. For light shading computations, the indexing is reversed (*i.e.* all the linking segments for one light vertex are processed in consecutive threads), to benefit from the same good properties than for the camera shading. All the results are written in linear arrays indexed by the thread index, leading to coalesced writes.

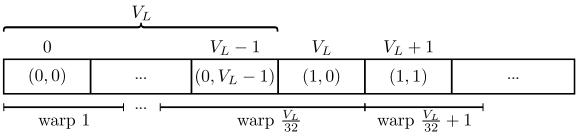


Figure 2: *Threads organisation for the shading of camera vertices. Each vertex is handled by blocks of V_L consecutive threads. At least $(V_L - 2)/32$ warps execute codes with the exact same BSDF parameters, as they all concern the same vertex, leading to high code coherency.*

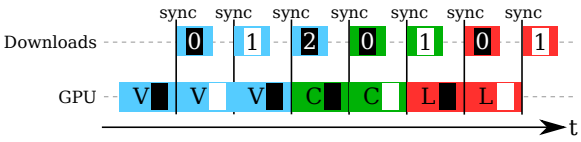


Figure 3: *Temporal execution of our combination system, not temporally to scale for clarity. The meaning of each element is described in the main text.*

5. Implementation of CBPT

Using the combination data computation system described in Section 4, we implement CBPT as described in Algorithm 2. Note that population sampling and combinations are done in parallel on all available CPU cores. The main points to note about Algorithm 2 is that we process two couples of populations at the same time, in an interleaved way. As illustrated by Figure 4, this allows us to perform GPU processing, CPU processing, downloads, and uploads at the same time. As the computation by the GPU of the combination data does not need any upload and is the only process that performs downloads, there is no contention on the memory bus if the GPU is able to perform transfers in both ways at the same time. In Algorithm 2, *combine()* uses the data computed on GPU and downloaded into the CPU memory to compute the $f^b(x,y)$ contribution for each pair of paths, and splats it in a thread-safe way to the final image. As the number of splatted values is small, thread-safety even with a large number of threads does not create a bottleneck. *compute_lt()* computes light-tracing on all available CPU cores.

Timings for each task of a step are reported in Algorithm 2 for a standard scene, and production-oriented parameters. These timings show the efficiency of our asynchronous computation scheme, as the total wall-clock time needed for one

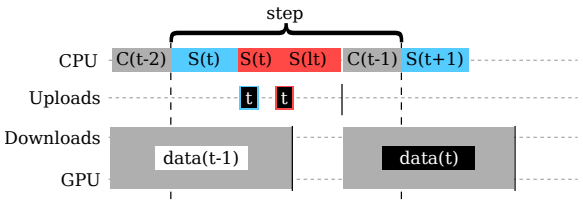


Figure 4: *Temporal execution of CBPT, not temporally to scale for clarity. Exact timings are given in Algorithm 2. The block labelled* C *contains both combine() and compute_lt(). The colors white and black for the rectangles indicate which GPU-side buffer is used to read the population data and store the results.*

loop is 34.5ms, compared to 60.1ms if all computations had been done synchronously. It also shows that GPU work is done "for free", as the complete time to perform a step is equal to the sum of the times needed by each CPU task, ignoring the GPU one.

Algorithm 2 CBPT algorithm, with timings of each noteworthy element using $N_C = 2000$, $N_L = 15$, $N_T = 1500$, in a scene with 758K triangles and 1.5GB of textures. The time spent by the GPU to compute all the results is given in "async time". The total time needed to perform a step is 34.5ms.

```
for t = 0 to ∞ do
    sample({x}_t)                          {time: 13.5ms}
    upload_async({x}_t)
    sample({y}_t)                          {time: 0.1ms}
    upload_async({y}_t)
    sample({y_lt})                         {time: 10.1ms}
    if t > 0 then
        sync_gpu_comp(t − 1)               {time: 0.1ms}
    end if
    sync_upload(t)
    gpu_comp_async(t)                      {async time: 25.7ms}
    if t > 0 then
        combine({x}_{t−1}, {y}_{t−1})      {time: 9.6ms}
        compute_lt({y_lt})                 {time: 1.1ms}
    end if
end for
```

6. Results

We now analyze the computational behavior of the combination system and CBPT. All the measures are done on an Intel i7 920 2.80GHz system, with an NVidia GTX480 GPU, and 16 GB of CPU-side memory. For our tests of CBPT, we use $N_C = 2000$, $N_L = 15$, and $N_T = 1500$ for all the scenes. These settings are not aimed at providing peak GPU performance, but rather at providing a good compromise between throughput of the GPU part and rendering quality. No adaptive sampling is used.

	ring			comp lights		
	GPU	CPU	÷	GPU	CPU	÷
vis	42.6	3.5	12.1	25.4	2.5	10.2
camera	266.7	11.8	22.6	281.7	15.6	18.1
light	266.5	17.3	15.4	280.2	13.0	21.6
	comp monitors			living		
	GPU	CPU	÷	GPU	CPU	÷
vis	25.6	2.9	8.8	32.2	2.1	15.3
camera	275.3	16.2	17.0	256.3	12.5	20.5
light	272.8	15.1	18.1	272.8	15.9	17.6

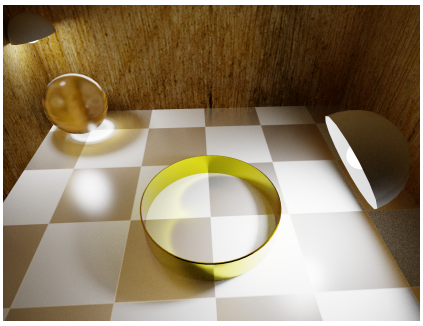
Table 1: *Throughputs for visibility* (vis)*, camera shading (*camera*), and light shading (*light*), when using the system described in Section 4, and when using the* 4 *physical cores of our processor, plus hyper-threading. The "÷" column gives the ratio of throughputs, corresponding to the actual speedups. Visibility is measured in* millions *of visibility tests per second, camera and light shadings are measured in* millions *of computations of* (f_s, p, p^*) *tuples per second (see Section 3 for the components of the tuple). All the measures take all the memory transfers into account.*

We use three different scenes of various complexities, which are presented in Figure 5. We have chosen these challenging scenes for their high lighting complexity:

- The first scene, *ring*, is geometrically simple, but composed of many glossy surfaces. It produces many subtle caustics that typically lead to noticeable noise, for instance on the back wall from the glossy tiles of the floor.
- The *comp* scene, rendered with two different lighting configurations, is much more involved than the *ring* scene. The *lights* version is lit by the ceiling lights, with indirect lighting caused by specular transmission of the light through the glass of the light fixtures. The front room and upper parts of the back room are only indirectly lit. In the *monitors* version, light comes only from the TV and computer monitor. Note the caustics on the wall due to refraction in the twisted pillars made of glass, as well as the caustics beneath the glass table. Nearly all the non-diffuse materials are glossy but not ideal mirrors, leading to very blurry reflections, which is especially visible on the floor.
- The *living* scene is lit by six very small area lights located on the ceiling above the table and the couch. It contains a lot of glossy materials (especially all the wooden objects), of which very few are specular. Note the caustics caused by the shelves on the left, and the completely indirect lighting in the hallway on the right.

6.1. Combination Throughput

Table 1 gives the raw throughputs of visibility and shading values we obtain on CPU and GPU depending on the scene, and the speedup brought by our system. All the measures take all the memory transfers into account. As expected, only visibility thoughputs decrease with the scene's size.



ring, 7.4K triangles (2.5, 3.4, 463K)

comp lights, 758K triangles (3.6, 3.2, 570K)

comp monitors, 758K triangles (3.6, 3.6, 620K)

living, 400K triangles (3.7, 3.4, 620K)

Figure 5: *The three scenes used to test CBPT. We indicate between parentheses the average length in segments of the sampled camera and light paths, as well as the average number of linking edges for each couple of populations in CBPT. Note that the average path lengths for BPT and CBPT are equal, as they use the same code. All the images have been rendered with CBPT. No post-process has been performed except tone-mapping, as our engine produces HDR images. The top-left image has been rendered at a resolution of* 1600×1200 *pixels in* 1 *hour. The three others have been rendered at a resolution of* 1600×900 *pixels, in* 4 *hours. As CBPT is based on standard Monte-Carlo methods, images at a resolution of* 800×450 *for the last three scenes can be obtained with a similar quality in* 1 *hour.*

The shading throughput on CPU is quite sensitive to the type of BSDFs (glossy or purely diffuse) that mostly compose the paths of a certain type, explaining the gap that is present for some scenes between the camera and light shading throughputs. This is mostly visible in *comp lights* because of the glass fixture surrounding the light sources. On the other hand, the GPU throughputs are much less affected by this. Despite the need to transfer the results back to GPU, we achieve a 15-20$\times$ speedup in average compared to CPU for shading only, consistently on all scenes.

The absolute timings in Table 2 give hints about the average time proportions needed by each element of the combination. These timings depend on the number of linking segments that have to be processed for each combination, which depend on the scene.

Figure 6 illustrates the impact of batch size on performance, for visibility and shading computations, on the *ring* scene. This allows us to evaluate the impact of different transfers/computation repartitions, and to find optimal batch sizes for the computer we use.

For the visibility computations, even on this geometrically very simple scene, the transfers are not a limiting factor, as the visibility results are packed in a very compact form. Therefore, using batches does not make any noticeable difference on performance as soon as the batches are large enough. Consequently, the major advantage brought by batches for visibility resides in the control we have on the memory-size requirements on GPU, without much impacting on performance.

For more memory-consuming results such as shading ones, the batch size has a large impact on performance, with the additional benefit of using less memory on the GPU. As a matter of fact, using asynchronism brings a 1.75$\times$ speedup, going from 160 millions to 252 millions of computations per second when transfers are done in parallel. Note that the optimal batch sizes are in practice only machine-dependent, as shading computations efficiency does not depend on the scene, and visibility computation efficiency is almost constant for any batch size larger than very small values.

6.2. CBPT

To quantify the efficiency of CBPT, we count the number of $f_{i,j}(x,y)$ computations performed during a complete CBPT

	ring	comp lights	comp monitors	living
vis	10.9	22.5	24.4	19.3
camera	1.7	2.0	2.2	2.5
light	1.7	2.0	2.2	2.3

Table 2: *Average time needed to complete each step on GPU, for each scene, in milliseconds.*

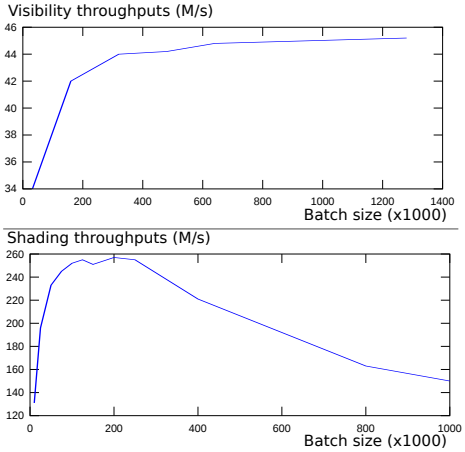


Figure 6: *Top: Visibility throughput, in millions of tests per second, in function of the number of visibility tests to perform in each batch. Bottom: Shading throughput, in million of shading tuples computation per second, in function of the number of shading computations to perform in each batch.*

step, and divide it by the time needed to complete the whole step, including populations sampling and splatting. We call this efficiency measure *basic contributions throughput*. This allows us to have meaningful and consistent results whatever the average path length is in each scene.

Computational efficiency: Table 3 gives the basic contributions throughputs obtained using CBPT, and the speedups compared to standard BPT. We compute these values when using CPU-based light-tracing (in this case $N_T = 1500$), to get actual performance, and when not using it ($N_T = 0$), to get the bidirectional-only basic contributions throughput. The CPU version of BPT uses the same code to sample paths, and the same code to compute the $f_{i,j}(x,y)$ values, except that all shading and visibility values are computed on CPU. Both CBPT and standard BPT uniformly sample the image, and do not use any adaptive sampling scheme.

The impact of light-tracing on throughputs is noticeable (around 20%), but the visual impact of a high variance light-tracing part is much more noticeable than the gain in bidirectional part when setting N_T to a very small value, particularly for very short rendering times. For longer rendering times and scenes where caustics are easily captured by light-tracing, N_T can be set to a smaller value, as it will visually converge faster than the bidirectional part.

	CBPT		BPT
	$N_T = 0$	$N_T = 1500$	
ring	20.9 (17.4×)	15.7 (13.1×)	1.2
comp lights	16.2 (16.9×)	12.7 (13.2×)	0.96
comp monitors	16.3 (14.8×)	13.1 (11.9×)	1.1
living	16.5 (21.7×)	12.5 (16.4×)	0.76

Table 3: *Basic contributions throughput for CBPT and standard BPT, in* millions *of $f_{i,j}(x,y)$ values computed per second, and speedup in parenthesis.*

As shown by timings in Algorithm 2, our reformulation allows us to keep both the CPU and GPU fully loaded, the GPU computation time being masked by the CPU one. The speedup we obtain with "production settings" is consistently greater or equal to 12× on our test scenes. Even if our samples are correlated, the correlation is spread on all the image by our image-sampling process. This effectively avoids the appearance of any noticeable correlation pattern.

Visual comparison with standard BPT: Visually observing noise reduction is made easier when looking at non-converged images, where improvements are clearly visible. Figure 7 presents the images obtained by CBPT and BPT after a few seconds of rendering, and after at least 4 samples per pixel have been computed by CBPT. As images were stored every 10 seconds, it can happen that more than 4 samples per pixel were actually computed, but both BPT and CBPT got the same computation time. The places where the improvements are most visible are on the diffuse walls, where light-space exploration is crucial to get low variance results, and in the glossy reflections. Table 4 gives the actual average number of samples per pixel for the bidirectional part of each image. As expected, the speedups obtained are similar to the ones obtained for the basic contributions throughputs, the little difference coming from the splatting, as BPT needs to splat many more values than CBPT for a same number of pair of paths. The main information of this table is that the images presented in Figure 5 would have required from 50 to 66 hours to be computed using standard BPT, *versus* 4 hours with CBPT.

Memory usage and scalability: Table 5 gives the memory usage both on CPU and GPU of CBPT. As expected, the size of the combination data on CPU and the populations memory size on GPU are related to the average path length. For populations, we use a conservative allocation scheme, reuse memory between populations, and refit memory zones regularly to keep the consumption low. This can lead to a consequent overestimation of the actual memory size needed, but drastically reduces the number of memory allocations, therefore providing a slight speedup. Despite this, memory requirements remain low for all our scenes on CPU (between 100 and 200MB), and very low on GPU (less than 100MB). Table 5 also shows that our method handles scenes much larger than the ones we used. Indeed, the scenes' kd-tree size are kept relatively low even for quite

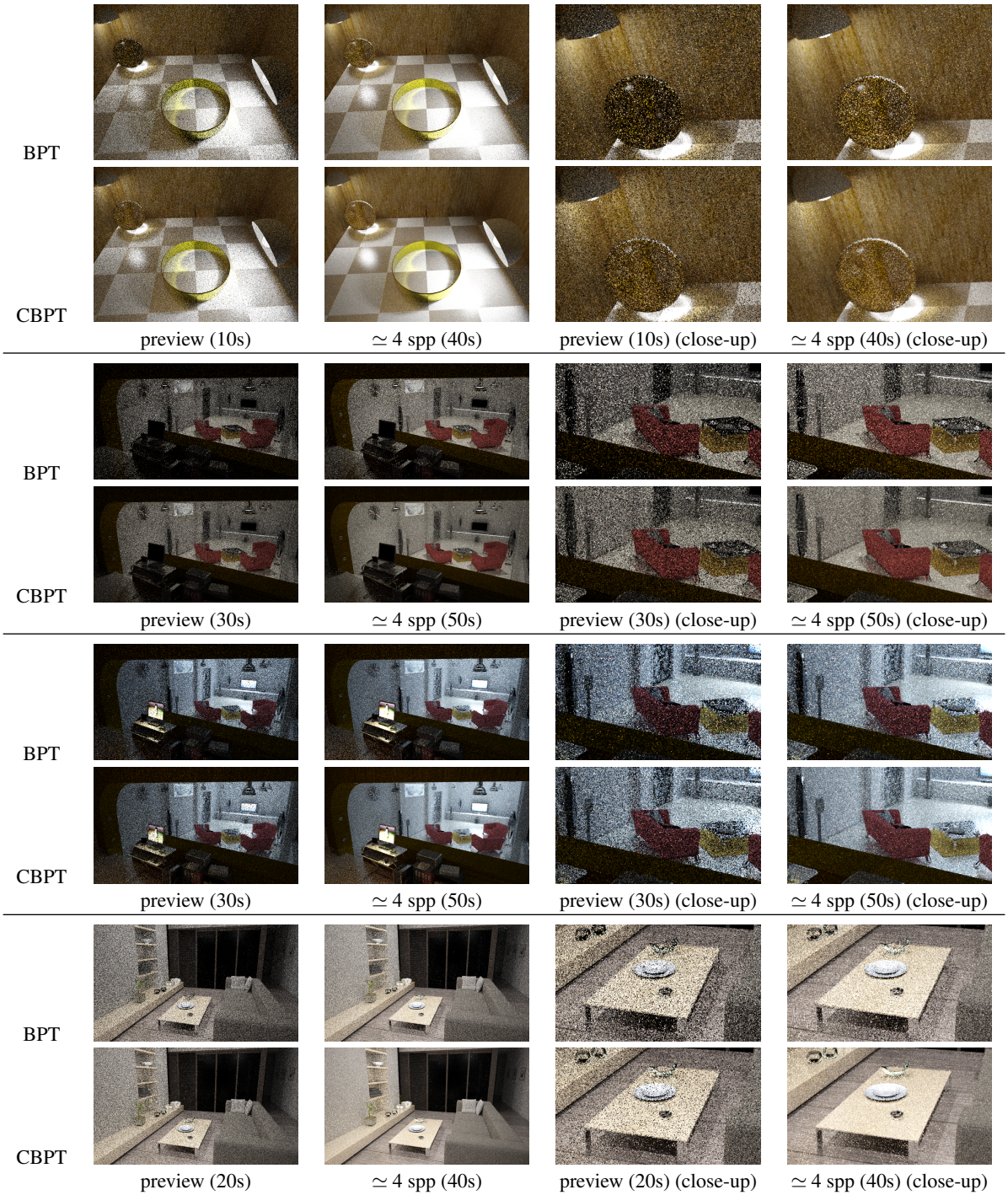


Figure 7: *Results obtained by BPT and CBPT on our test scenes, after approximately* 10 *seconds of actual computations, and after CBPT has computed approximately* 4 *samples per pixel. Images are rendered at* 800×450*, except ring which is rendered at* 800×600*. Note that for all the scenes, mipmaps are lazily built when first accessed, explaining the 30 and 20 seconds of total rendering times for the preview configuration of the comps and living scenes. The time spent building these mipmaps is negligible for the ring scene, but takes* 16 *and* 8 *seconds in the comps and living scenes respectively, and are generally built when sampling the first paths. This also shows that our system can be seamlessly used together with all the usual ways of reducing the peak memory usage, as it does not impact the rendering engine architecture.*

	CBPT		BPT		(x,y)
	prev.	$\simeq 4$ spp	prev.	$\simeq 4$ spp	
ring	1.64	5.15	1.60	5.41	14.3×
comp lights	1.05	3.99	1.38	4.44	13.5×
comp monitors	1.36	4.12	1.61	4.77	12.9×
living	1.62	4.23	1.53	3.86	16.4×

Table 4: ***Overall speedup measurement:*** *Average number of samples computed per-pixel for the bidirectional part of the images of Figure 7. This is equivalent to the average number of camera paths that have contributed to each pixel. The last column gives the ratio between CBPT and BPT of the number of* pairs of paths *contributing to the bidirectional part of each pixel, which is a good measure of the actual speedup brought by CBPT over standard BPT. For standard BPT, each camera path is combined with one light path, therefore the number of pairs of paths per-pixel is equal to the number of camera paths. For CBPT, as each camera path is combined with N_L light paths, the number of pairs is N_L times the number of camera paths per-pixel. In our tests, we use $N_L = 15$.*

	CPU		GPU		
	pops.	comb.	kd-tree	pops.	comb.
ring	73.3	48.0	0.47	3.8	23.5
comp lights	91.2	66.0	56.1	4.8	23.5
comp monitors	95.0	72.3	56.1	4.8	23.5
living	60.8	60.3	58.5	5.0	23.5

Table 5: *Memory usage for populations and combination data on CPU, and memory usage for the kd-tree, the populations data (position, BSDFs parameters, etc.), and all the batch buffers, in MB.*

complex scenes (about 50MB). Therefore, scenes that contain several millions of polygons fits in the GPU memory. Moreover, the memory size of populations is negligible except for idiosyncrasies, as even with participating media, the paths remain short ($10-20$ vertices on average).

7. Conclusion

Bidirectional path-tracing is an unbiased and highly-robust rendering algorithm, but is not well suited for GPU implementation, as it requires a lot of branching. By exhaustively combining populations of paths instead of single paths, we were able to divide the algorithm into two parts, each one being well suited for either the CPU or the GPU. We maintain the CPU, the GPU, and the memory bus between CPU and GPU busy simultaneously by interleaving the steps of CBPT. The GPU part is made efficient by using high-level optimization techniques such as double buffering and asynchronism.

We have shown that CBPT is more than an order of magnitude faster than standard BPT on various test scenes, without affecting the size of the datasets or the flexibility of the underlying rendering engine in terms of shaders, and models of lights and cameras. This makes CBPT very well suited for accelerating image computation in production-oriented engines.

References

[AS00] ASHIKHMIN M., SHIRLEY P.: An anisotropic phong light reflection model. *Journal of Graphics Tools 5* (2000), 25–32.

[CTE05] CLINE D., TALBOT J., EGBERT P.: Energy redistribution path-tracing. In *SIGGRAPH '05* (2005), pp. 1186–1195.

[Faj10] FAJARDO M.: Ray tracing solution in film production rendering. http://www.graphics.cornell.edu/~jaroslav/gicourse2010/, 2010. SIGGRAPH 2010 Course on global illumination in production rendering.

[HJ09] HACHISUKA T., JENSEN H.: Stochastic progressive photon mapping. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers* (2009), ACM, pp. 1–8.

[HOJ08] HACHISUKA T., OGAKI S., JENSEN H.: Progressive photon mapping. *ACM Trans. Graph. 27*, 5 (2008), 1–8.

[KFC*10] KŘIVÁNEK J., FAJARDO M., CHRISTENSEN P. H., TABELLION E., BUNNELL M., LARSSON D., KAPLANYAN A.: Global illumination across industries. http://www.graphics.cornell.edu/~jaroslav/gicourse2010/, 2010. SIGGRAPH 2010 Course on global illumination in production rendering.

[KH01] KELLER A., HEIDRICH W.: Interleaved sampling. In *EGWR'01* (2001), pp. 269–276.

[KSKAC02] KELEMEN C., SZIRMAY-KALOS L., ANTAL G., CSONKA F.: A simple and robust mutation strategy for the Metropolis light transport algorithm. In *Eurographics '02* (2002), pp. 531–540.

[LFCD07] LAI Y.-C., FAN S., CHENNEY S., DYER C.: Photorealistic image rendering with population Monte Carlo energy redistribution. In *EGSR '07* (2007), pp. 287–296.

[Lux10] LUXRENDER: Luxrays. http://www.luxrender.net/wiki/index.php?title=LuxRays, 2010.

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

[OMP10] OMPF: Hybrid bidirectional path-tracer development thread. http://ompf.org/forum/viewtopic.php?f=6&t=1834, 2010.

[Seg08] SEGOVIA B.: Radius-CUDA raytracing kernel. http://bouliiii.blogspot.com/2008/08/real-time-ray-tracing-with-cuda-100.html, 2008.

[VG94] VEACH E., GUIBAS L. J.: Bidirectional estimators for light transport. In *EGWR '94* (1994), pp. 147–162.

[VG95] VEACH E., GUIBAS L. J.: Optimally combining sampling techniques for monte carlo rendering. In *SIGGRAPH '95* (1995), pp. 419–428.

[VG97] VEACH E., GUIBAS L. J.: Metropolis light transport. In *SIGGRAPH '97* (1997), pp. 65–76.

[WMLT07] WALTER B., MARSCHNER S. R., LI H., TORRANCE K. E.: Microfacet models for refraction through rough surfaces. In *EGSR '07* (2007), pp. 195–206.