Interactive Ray Tracing Client

Michal Radziszewski
AGH, Krakow, Poland
mradzisz@student.agh.edu.pl

Witold Alda
AGH, Krakow, Poland
alda@agh.edu.pl

Krzysztof Boryczko
AGH, Krakow, Poland
boryczko@agh.edu.pl

ABSTRACT

In this paper we present an interactive GPU-based, GUI client, working with rendering server employing ray tracing based global illumination. The client is designed to guarantee interactivity (namely 1/60sec response time) no matter how slow the rendering server is. The client dynamically adjusts image resolution to match the server performance and complexity of the rendered scene. When the scene is modified, the image may appear out of focus and noisy, depending on the machine computational power, but usually is readable. With no interrupt from the client, the image is progressively improved with new data from the server. The system exploits hybrid programming model – CPU for the server and GPU for the client.

Keywords: Real-time global illumination, quasi-Monte Carlo ray tracing, hybrid CPU and GPU programming.

1 INTRODUCTION

Many contemporary approaches to ray tracing based global illumination rely on computational power of graphics hardware, eg. [25]. Unfortunately, true, unrestricted, global illumination algorithms, which solve the Rendering Equation [9], are not well suited for GPU architecture. Such implementation is possible, as has been shown numerous times, but is severely restricted when compared with classic multi-core CPU solutions, since GPUs cannot process irregular data structures effectively [7].

Our renderer, based on significantly modified Bidirectional Path Tracing [22] and Photon Mapping [8] with quasi-Monte-Carlo (QMC) approach [12] is designed for flexibility of CPUs. It allows rendering, in full spectrum, of arbitrary scene primitives, arbitrary materials, textures, and more. The only restriction is, in fact, a computer memory size. Such, traditionally CPU based, algorithms are rather difficult to port to GPUs. When, despite all problems, they are ported eventually, performance benefits of GPUs over multicore CPUs are often questionable [7].

This paper presents a different approach to obtain interactivity. Pure ray tracing algorithms are based on point sampling scene primitives, not using scan line rasterization at all. This gives much freedom in the way how samples are chosen, however QMC ray tracing algorithms produce a huge number of samples, which do not fit in raster RGB grid. Converting these data to 3x8bit integer based RGB image at interactive frame rates may be impossible even for multi-core CPUs, especially when dynamic image resolution has to be adjusted to the server rendering speed and scene complexity, with some non-trivial post-processing added. As we will show, conversion of ray tracing output to a displayable image and many post-processing effects can be expressed purely by rasterization operations, in which GPUs excel. The main idea behind the presented approach is therefore the usage of the best suitable processor for a given algorithm, instead of porting everything to GPUs.

2 RELATED WORK

The concept of ray tracing is not new [27]. Because it can produce much better image than hardware rasterization, for several years there has been a lot of research dedicated to run it in real time, despite its high computational cost [15]. Ray tracing based global illumination is even more expensive. However, for some time now real time global illumination algorithms are being developed also [23].

Just after the appearance of first programmable DirectX 9 class graphics processors there were first attempts to use it for ray tracing [17]. Nowadays, vast majority of contemporary real time global illumination algorithms are based on computational power of modern GPUs, e.g. [11, 25]. Unfortunately, they still put restrictions, often quite severe, on scene content (limited range of material and geometry representation), scene size, and illumination phenomena which are possible to capture.

However, this is not the only way to obtain interactivity – nowadays multi-CPU Intel workstations can perform interactive ray tracing [16], yet true global illumination is still unachievable. Interactivity can also be obtained using clusters of machines with CPU rendering [2].

On the other hand, approach presented here is substantially different from those above – placing abso-

lutely no restrictions on scene and illumination effects. It uses GPU just to display and postprocess image made from CPU ray traced point samples, in resolution dynamically adjusted for real time performance.

3 REQUIRED SERVER OUTPUT

In general the server may run any point sampling algorithm, but in this project we rely on QMC ray tracing. The visualization client assumes the specific format of the server's output. In the following subsections we describe in detail the conditions which should be fulfilled to make the client work properly. We also show how to convert Photon Mapping to meet these assumptions.

The server should provide stream of color values scattered uniformly at random locations in the screen space. The uniformity of sampling ensures acceptable image quality even at low sampling rates, which is typical due to high computational cost of ray tracing.

Additionally, the output stream should be generated roughly uniformly in time. Otherwise the client might fail to maintain interactive refresh rates.

3.1 Bidirectional Algorithms

Some most advanced ray tracing algorithms trace rays in both directions – from the camera towards lights (camera rays), and in the opposite one (light rays). Such approaches produce two kind of samples, which must be processed differently in order to produce displayable images [22].

The client accepts two input streams. The format of samples is identical in both streams: $([u,v],[x,y,z,w])$, where $[u,v]$ are screen space coordinates, in $[0,1]^2$ range, or, perhaps, with slight overscan to avoid post-process filtering edge artifacts, x,y,z is sample color value in CIE standard [6], and w is sample weight.

The two streams differ only in interpretation of sample density. The pixels of image from camera rays are evaluated by averaging local samples using any suitable filter – sum of weighted samples is divided by sum of weights. On the other hand, pixels of light image are formed using a suitable density estimation technique – samples are filtered and summed, but not divided by sum of weights. Therefore, a sample density affects only quality of camera image, while it affects both quality and brightness of light image. The final, displayable, image is a sum of both camera and light images, the latter divided by a number of traced paths.

Obviously, not all ray tracing algorithms need both – camera and light – output streams. For example, Path Tracing [9] and Photon Mapping [8] produce camera samples only, while Particle Tracing [1] needs only light image. Therefore, the visualization client employs an obvious optimization – it skips processing of a stream given that no samples were generated into it.

3.2 Coherent vs. Non-Coherent Rays

For some time now it is often claimed that it is benefical to trace camera rays in a coherent way, because it can significantly accelerate rendering [24, 2]. This is true, but only for primary rays (sent directly from camera or light source). Unfortunately, rays, which are scattered through the scene, do not follow any coherent pattern and caching does not help much. Since true global illumination algorithms typically trace paths of several rays, these algorithms do not benefit much from coherent ray tracing.

What is more, coherent ray tracing tends to provide new image data in tiles, which make progressive improvement of image quality difficult. On the other hand, we have chosen to spread even primary rays as evenly as possible, using carefully designed Niederreiter-Xing QMC sequence [13] as the source of pseudorandom numbers. Therefore, it can be expected that very few traced rays provide reasonable estimate of colour of the entire image, and subsequently traced rays improve image quality evenly.

3.3 Full Spectral Rendering

Having in mind further processing, it may be useful to output full spectral images [5, 18]. However, full spectral representation requires huge amount of memory. For example, full HD spectral image in 16bit floating precision and with 3nm wavelength sampling from 400nm to 700nm needs as much as $1920 \times 1080 \times 100 \times 2B \approx 400MB$, while RGB one requires $1920 \times 1080 \times 3 \times 2B \approx 12MB$.

The standard CIE XYZ space seems to be the best option instead, since an RGB space, which depends on a particular display hardware, is not a plausible choice. For this reason our client accepts CIE XYZ color samples. The presented server natively generates full spectral data and converts it internally from full spectrum to the three component color space.

3.4 One-pass Photon Mapping

Original Photon Mapping [8] is a two pass technique. This obviously violates the requirement of steady sample stream – during photon tracing there are no samples generated, causing high latency before image starts to appear. We have found that Photon Mapping actually can be done in one pass, with only minor loses in efficiency compared to the original approach. The new algorithm uses a linear function of number of image samples (n) to estimate minimal necessary photon count in photon map to obtain image with quality determined by n. Therefore, the photon map is no more static structure – new photons are added while new image samples are rendered.

Immediately two issues have to be solved – synchronization of read and write accesses to the photon map

structure in parallel photon mapping and balancing kd-tree. Synchronization can be performed with simple read-write locks (classic readers-writers problem).

On the other hand, kd-tree balancing requires significant algorithm modification. We have chosen to balance the scene space instead of photons. The original algorithm starts with bounding box of all photons (unknown in our approach) and in each iteration places splitting plane at a position such that half of the photons remains on the one side of the plane. Otherwise, our algorithm starts with bounding box of the entire scene, and in each iteration it splits it in half across dimension in which the box is the longest. Splitting stops when all nodes contain less photons than a certain threshold (5-6 seems to be optimal) or a maximum recursion depth is reached. Adding new photons require just splitting of some of the nodes, where there happens to be too many photons.

The idea is somehow similar to Irradiance Caching algorithm [26]. Similarly as in this method, our approach starts with empty structure and fills it through rendering. However, Irradiance Caching calculates irradiance samples when they are needed by camera rays, while our modified Photon Mapping traces photons in a view independent manner.

Strictly speaking, the new approach does not generate batches of samples in roughly uniform time. Due to kd-tree lookup computational complexity as well as linear dependence between number of photons in kd-tree and number of samples computed, the average time to calculate nth sample is the order of $\mathcal{O}(\log n)$, where n is the sample number. Logarithm, however, changes slowly, and the client is designed to adjust to slow changes of rendering speed by modifying size of batch of samples.

4 CLIENT AND SERVER ALGO-RITHMS

Finally, a GPU task is to convert point samples into a raster image. The conversion is done with resolution dynamically adjusted to the number and variance of point samples. In the image, a color conversion from XYZ to RGB space of current monitor, together with gamut mapping, tone mapping, gamma correction and other post-processing effects are performed.

As a target platform we have chosen a GPU compatible with OpenGL 3.x [19] and GLSL 1.5 [10]. Major part of algorithm is coded as a GLSL shader, which suits our needs very well. Recent technologies, such as Nvidia CUDA, ATI Stream, or currently being developed OpenCL are not necessary for this kind of algorithm.

The rendering task is split into two processes (or threads in one process, if a single application is used as a client and server) running in parallel: a server wrapper process and visualization process. The rendering

process may be further split into independent threads, if multicore CPUs or multiple CPU machines are used.

4.1 Server Wrapper Process

Ray tracing can produce virtually unlimited number of samples, being limited only theoretically by machine numerical precision (our implementation can generate as many as 2^{64} samples before sample locations eventually start overlap). Therefore, ray tracing process is reset only immediately after user input, which modifies the scene. Otherwise, it runs indefinitely, progressively improving image quality.

The server wrapper runs on a separate thread, processing commands. The wrapper recognizes three commands: *term*, *abort*, and *render*. The *term* command causes wrapper to exit its command loop, and is used to terminate the application. The *abort* commad aborts current rendering, and is used to reset server to the new user input (for example, camera position change).

The *render* command orders server to perform rendering. The rendering is aborted when either *abort* or *term* command is issued. Maximum time to abort rendering is a time necessary to generate just one sample. Any algorithm capable of generating the specified output (see Section 3) can be used. In our server implementation, rendering is performed in parallel on multicore CPUs.

The wrapper allows registering asynchronous *finish* event. This event is generated when rendering is finished (either a prespecified number of samples was generated or *abort* was issued). The event can be used to synchronize client with server. Apart from sending asynchronous messages, the wrapper can be queried synchronically for already rendered samples. Since this query just copies the data to the provided buffer, server blocking due to synchronization takes little time.

4.2 Client Process

Client is responsible for visualizing samples generated by server, and additionally it processes GUI window system messages. Client stores its internal data in the four screen-aligned textures, in the IEEE 32bit floating point format. A 4-channel $[X, Y, Z, W]$ texture and a single component variance $[Var]$ texture are stored for camera and light input streams. Therefore, client stores 40 bytes of data per screen pixel, apart from standard integer front and back buffers. The details of client main loop are presented in Figure 1.

When all GUI messages are processed, client rasterizes new samples, generated by the server, into its internal textures. This task is performed by the render-to-texture feature of Framebuffer Object (FBO). The client sets an empty vertex program, which only passes through data, and a geometry program which is equivalent to rendering textured point sprites fixed functionality. The input is a stream of two elements – two
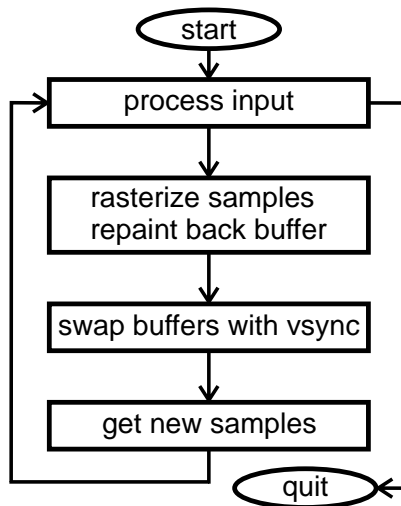
```
        ┌─────────┐
        │  start  │
        └─────────┘
             │
             ▼
    ┌─────────────────┐ ◄─┐
 ┌─►│  process input  │   │
 │  └─────────────────┘   │
 │        │               │
 │        ▼               │
 │  ┌─────────────────┐   │
 │  │ rasterize samples│  │
 │  │repaint back buffer│ │
 │  └─────────────────┘   │
 │        │               │
 │        ▼               │
 │  ┌─────────────────┐   │
 │  │swap buffers with │  │
 │  │     vsync        │  │
 │  └─────────────────┘   │
 │        │               │
 │        ▼               │
 │  ┌─────────────────┐   │
 │  │ get new samples │   │
 │  └─────────────────┘   │
 │        │               │
 └────────┤               │
          ▼               │
     ┌─────────┐ ◄────────┘
     │  quit   │
     └─────────┘
```

Figure 1: Main loop of visualization client process.

component screen position (u,v) and four component color (x,y,z,w). Input is placed in Vertex Buffer Object (VBO), and is then rendered with GL 'render points' command. Points are rendered in blending mode set to perform addition, ensuring that all samples add up instead of overwriting previous texture content.

Additional input is a monochromatic HDR filter texture, used to draw point sprites. The texture is normalized (all the texel values add up to one) and the texture border value is set to zero. The filter texture is applied without rescaling and with bilinear filtering, thus preserving filter normalization, which is crucial for algorithm correctness. We have found that 5x5 texel windowed Gaussian blur gives good results.

The rendering is performed in two passes. First, color textures are updated. In the second pass, using already up-to-date color textures, variance textures are updated. In both passes, the same samples are rendered. The variance is updated using the formula $V_j = V_{j-1} + \sum_i (Y_i - \overline{Y}_j)^2$, for jth batch of i samples. The formula does not give the best theoretically possible results, since the mean $\overline{Y}$ is approximated using only already evaluated samples. The alternative formula $V_j = Y2_j - \overline{Y}_j^2$, $Y2_j = Y2_{j-1} + \sum_i Y_i^2$, which requires storing sum of squares ($Y2$) instead of variance, should be avoided due to poor numerical stability (even negative variance results are possible). In both formulas the division by $n-1$ factor, where n is the total number of samples in a given stream, is omitted. This division is performed when variance data is read from its texture.

The sample rasterization algorithm works as follows:

1. The content of client sample buffer (pairs $[u,v]$, $[x,y,z,w]$) is loaded into VBO, interpreted as 2D point coordinates and 4D color. There is one buffer for both streams. Samples which come from light stream are encoded with negative weights.

Stream separation is performed further in the fragment program.

2. Monochromatic float texture with filter image is selected and point draw command is issued. The texture is used as a texture sprite for emulated point sprites. Fragment program performs multiplication of 'color' attribute by the texture value $[X,Y,Z,|W|]$. The output is saved to the color texture of camera stream if $W \geq 0$ or light stream otherwise.

3. After rasterization, textures are detached from FBO, GPU MIP-map build command is issued.

4. Texture LoDs (used by 'repaint back buffer' processing) for both streams are evaluated as $LoD_i = \log_4(P/S_i)$, where P is number of pixels on the screen and S_i is the number of samples from ith stream computed so forth.

5. Second draw is issued, with variance textures as output this time. The variance is evaluated only for luminance (Y) component, since three component variance typically do not help much and substantially complicates algorithm. Variance output for each stream is $(Y_{avg} - Y)^2$, where Y_{avg} is read from previously generated color texture, and Y is luminance of currently processed sample, multiplied by filter texture.

6. Similarly to color textures, variance textures are detached from FBO, GPU MIP-map build command is issued.

In order to repaint back buffer, client draws a screen-sized quad, using the four textures as an input. The screen is filled with custom fragment program. The program accepts following control parameters: level of detail (LoD) for both streams, light image weight (Lw), image brightness (B), contrast (C), gamma (G), color profile matrix (P), and variance masking strength (Vm). Level of detail (LoD) is already evaluated during rasterization. Now, the LoD values are used by fragment program to blur texture data if not enough samples are computed. Light image weight is got from the server along with samples, and its value is equal to the number of paths traced from light sources. This parameter is used to scale light image texture appropriately, such that the texture can be summed with camera image texture.

Image brightness, contrast, gamma and color profile are set by the user, and their values adjust the image appearance. Additionally, the visualization client is able to add a glare effect as an additional post-process, implemented as a convolution with a HDR glare texture, generated according to [20]. However, sufficiently large glare filters are far beyond computational power of contemporary GPUs for real-time screen refresh rate. Since

these parameters are defined only for client, and do not affect server rendering at all, their values can be modified freely without resetting the server rendering process.

Variance of samples is estimated only for luminance (CIE Y channel), using the standard variance estimator ($V \approx \frac{1}{N-1}\sum(E(Y)-Y_i)^2$, where N is the number of samples, Y_i are luminance values, and $E(Y)$ is the luminance value estimated from samples computed so far. The client is able to increase blurriness according to the local changes in estimated variance, hence slightly masking noise produced by stochastic ray tracing. The noise to blurriness ratio can be controlled by Vm parameter.

The blurriness is created by low pass filter or bilateral filtering [14] guided by variance estimation, which potentially can be much better in preserving image features than a simple low pass filter. However, bilateral filtering works correctly only if noise is less intense than image features. When image is heavily undersampled, this assumption may not be satisfied, and a low pass filter remains the only viable option. For example, in Figure 3, the two leftmost images cannot be enhanced by bilateral filtering. On the other hand, this technique does a good job improving the quality of middle image from Figure 5.

Unfortunately, the noise masking feature can hide only the random error which is the result of variance. It cannot hide (in fact, it cannot even detect) other kind of error resulting from bias. The variance is the only source of error in Bidirectional Path Tracing, while Photon Mapping error is dominated by bias.

The algorithm processes its input as follows:

1. The program reads data from both variance maps, using requested *LoD*s through hardware MIP-mapping.

2. *LoD*s for both streams are evaluated according to initial *LoD*s, the variance and Vm, for ith stream: $LoD'_i \leftarrow LoD_i + Vm\log_4([Var])$.

3. $[X,Y,Z,W]$ textures of both streams are sampled, this time using just evaluated LoD' and custom filtering technique (hardware MIP-mapping produces very poor results, see section 4.3 for more detailed discussion).

4. Texture samples for both streams are normalized, i.e. $[X,Y,Z,W] \rightarrow [X/W, Y/W, Z/W, 1]$ (if $W = 0$, then sample is considered to be $[0,0,0,1]$). Then, light texture sample, divided by Lw, is added to camera texture sample, producing single result for further processing.

5. Optionally, glare effect is applied here. Our glare texture is generated to be applied in XYZ color space instead of RGB one.

6. Tone mapping of luminance (Y) is performed, using very simple yet effective procedure: $Y' \leftarrow 1 - exp(-(B*Y)^C)$, while X and Z components are scaled by Y/Y' ratio. If $Y = 0$ it means that image is black at that point and $X'Y'Z' \leftarrow (0,0,0)$ is used.

7. Resulting $X'Y'Z'$ is multiplied by matrix P, and a basic gamut mapping is performed. We do not use elaborated algorithms here – simple desaturation of out-of-gamut colors, just to keep mapped luminance unmodified, works reasonably well. Now output is in *RGB* format, normalized to $[0,1]$ range.

8. Finally, gamma correction using G is performed.

Next, client swaps front and back buffers, in synchronization with screen refresh period. This guarantees constant frame rate (typically 60Hz for common LCDs).[1] Finally, client reads new samples from the server. The reading is performed with synchronization, blocking the server for a moment. However, client does not display samples immediately, blocking server just for copying this portion of data to its internal buffer for later processing.

4.3 MIP-mapping Issues

Images produced by rasterizing ray traced samples are created as screen-sized textures. Should enough samples be generated, these images could be used immediately without any resampling. Unfortunately, contemporary CPUs are far too slow to generate at least *#screen_pixels* of such samples in, say, 1/30sec, which is required for real time performance. Therefore, some kind of blurring texture data, according to fraction of necessary samples generated and the local sample variance, have to be performed.

While MIP-mapping is reasonably good in filtering out texture details which would otherwise cause aliasing, it cannot be used reliably to blur the texture image. Blurring by using *LoD* bias parameter of texture sampling function produces extremely conspicuous and distracting square pattern, with severe bilinear filtering artifacts (see Figure 2 for details). This is not surprising, since a GPU uses box filter to generate MIP-maps and linear interpolation between texels to evaluate texture value at sampled point. Moreover, MIP-mapping with polynomial reconstruction instead of linear one fails as well. We have used custom texture sampling with Catmull-Rom spline interpolation for this purpose.

---

[1] GPU class must be properly selected for a monitor resolution. If GPU is too poor, interactivity is not obtained. We found that best contemporary single processor GPU (Nvidia GTX 285, at the time of testing) is enough for refresh rate of 30Hz in full HD. Such issue, however, does not slow down the server – the same number of samples is still rendered in the same amount of time, they are just displayed more rarely, in larger batches.

Visually good results can be obtained by using Gaussian blur:

$$I(u,v) = \frac{\sum_i \sum_j T_{ij} g_{ij}(u,v)}{\sum_i \sum_j g_{ij}(u,v)}.$$

The I is texture sample, u,v is the sample position, T are texel values, and $g_{ij} = exp(-\sigma d_{ij}^2)$ is the filter kernel, with σ controlling blurriness, and d_{ij} is the distance between the u,v position and texel T_{ij}. Unfortunately, direct implementation of Gaussian blur requires sampling an entire texture for evaluation of any texture sample, which is far beyond computational capabilities of contemporary GPUs. The weight of Gaussian filter, however, quickly drops to zero with increasing distance from evaluated sample. Truncating the filter to a fixed size window containing limited number of samples is a commonly used practice.

The simple truncation is not always optimal, since quality of truncated Gaussian filter depends strongly on the σ parameter – to obtain similar quality with different sigmas, an $\mathcal{O}(\sigma^{-1})$ number of texels have to be summed. That is, if a Gaussian filter is truncated too much, it starts to resemble a box filter. In our case, σ varies substantially, and therefore more advanced technique should be used. We may notice that decreasing a resolution of the original image twice, and increasing σ four times, approximates the original filter on the original image. Eventually, the following algorithm is employed: initial MIP-map level is set to zero, and while σ is smaller than a threshold t, the σ is multiplied by four, and MIP-map level is increased by one.

The threshold t and number of summed texels have been adjusted empirically to balance the blur quality and computational cost. First we have found that truncation range R of roughly 2.5 is a maximum value which ensures reasonable performance. For such truncation, setting $t \approx 1$ is reasonable. Additionally, it is better to use a product of g and smooth windowing function w instead of original g if truncation is used. The $w = 1 - smoothstep(0, R, d)^E$, where E controls how quickly w drops to zero with distance, works quite well. The value $E = 8$ yields good results.

What is more, the transition between MIP-map levels is noticeable and decreases image quality. This is especially distracting if σ varies across the image, which is the case because blur is adjusted to the locally estimated variance. Therefore, similarly as in trilinear filtering, the Gaussian blur is performed on two most appropriate MIP-map levels, and the results are linearly interpolated, avoiding sudden pops when MIP-map level changes. Therefore, truncation to range 2.5 cause blurring to use $2[(2 \cdot 2.5)^2] = 50$ texture fetches on average, which is costly, yet acceptable on contemporary GPUs.

The sophisticated filtering scheme is used only for $[X,Y,Z,W]$ textures. Variance $[Var]$ textures, not being displayed directly, do not have to be sampled with any-

thing more complicated that basic MIP-mapping. This saves some computational power of a GPU, yet does not produce noticeable visual artifacts.

5 RESULTS

The quality of rendered images obviously mostly depends on the rendering algorithm used. We have tested the visualization client in cooperation with Path Tracing (Figure 3) and Photon Mapping (Figure 4). Both figures present initial image rendered after 1/30sec and show the speed of image quality improvement. All the tests were performed on Intel Core i7 CPU and Nvidia 9800 GT GPU, in 512x512 resolution.

The client is responsible merely for visualization and postprocessing, assuming that it is provided with stream of point samples, scattered roughly evenly through entire image. The only algorithm for image quality improvement is noise reduction based on variance analysis. The error due to variance (seen as high frequency noise) is much more prominent in results of Path Tracing than in Photon Mapping, so the noise reduction has been tested on the first algorithm. The results are presented in Figure 5.

When multiple processors are used in the same application, good load balancing is important. While it is well known how to load balance ray tracing work between multiple CPUs, in our application it is impossible to balance loads between visualization client and ray tracing server. The subtasks performed by CPUs and GPU are substantially different and suited for different architectures of these two processors, so work cannot be moved to the less busy unit as needed. In fact, on contemporary machines rendering server is always at full load, and GPU can be not fully utilized, especially when low resolution images are displayed. However, it is good to have some reserve in GPU power to ensure real time client response.

6 CONCLUSION

We have presented an interactive GUI visualization client for displaying ray traced images online, written mainly in GLSL. Apart from visualization, the client can hide noise of input data by means of variance analysis. Additionally, the client can apply glare effect as a postprocessing technique, which is performed quite efficiently on GPU.

The client is able to obtain interactivity regardless of the ray tracing speed. However, the price to pay is blurriness of images rendered at interactive rate. Nevertheless, the image quality improves quickly with time whenever rendered scene is not changed.

Our approach scales well with increasing number of CPU cores for ray-tracing, as well as with increasing number of shader processors on a GPU. Moreover, the program never reads results from the GPU, so it does not cause synchronization bottlenecks, and should
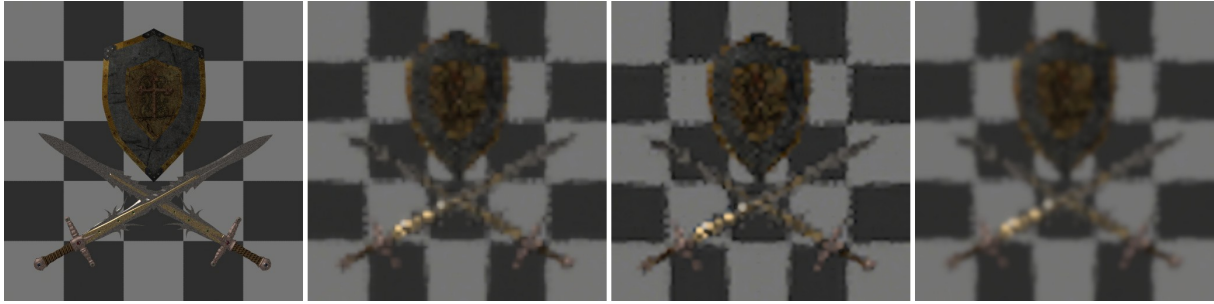
Figure 2: Comparison of MIP-mapping and custom filtering based blur quality. From left: reference image, hardware mipmapping, custom reconstruction based on Catmull-Rom polynomials, windowed Gaussian blur.
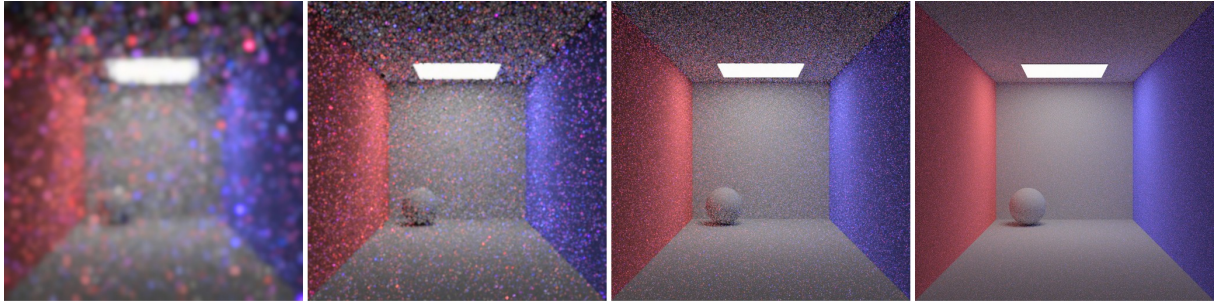

Figure 3: Results of Path Tracing (from left: after 1/30sec, 1/3sec, 3sec, 30sec). The Path Tracing error appears as noise, blur in the first two images is caused by undersampling (far less than 1 sample per pixel were evaluated).
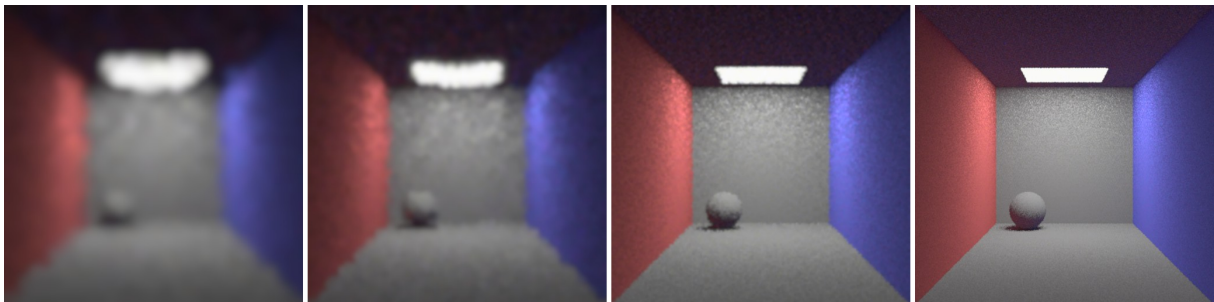

Figure 4: Results of Photon Mapping (from left: after 1/30sec, 1/3sec, 3sec, 30sec). Photon Mapping does not produce much noise, but due to overhead caused by photon tracing and final gathering, less image samples than with Path Tracing were computed, which cause some blurriness.

be friendly with multi-GPU technologies like SLI or Crossfire.

Additionally, we have modified the Photon Mapping algorithm to be a one-pass technique, with the photon map being updated interactively during the whole rendering process. This enables using Photon Mapping with the presented visualization client, which then could ensure progressive image quality improvement, without any latencies resulting from construction of photon map structure.

Our visualization client has a lot of potential for future upgrades. The adaptive filtering technique [21] seems to be good approach to significantly reduce image noise on the side of the visualization client. Moreover the client can be extended to support frameless rendering [3, 4]. This very interesting and promising technique can improve image quality substantially us-

ing samples from previous frames, provided that subsequent images do not differ too much.

In future we plan to introduce to our client stereo capability, using OpenGL quad-buffered stereo technology. Ray tracing algorithms can easily be converted to render images from two cameras at once, and a lot of them can do this even more efficiently than rendering two images sequentially (for example, Photon Mapping can employ one photon map for both cameras, and similarly, Bidirectional Path Tracing can generate one light subpath for two camera subpaths). Unfortunately, stereo rendering doubles the load on the GPU shaders, as well as on the GPU memory. However, it seems that interactive stereo can be obtained by slight decrease of custom texture filtering quality.
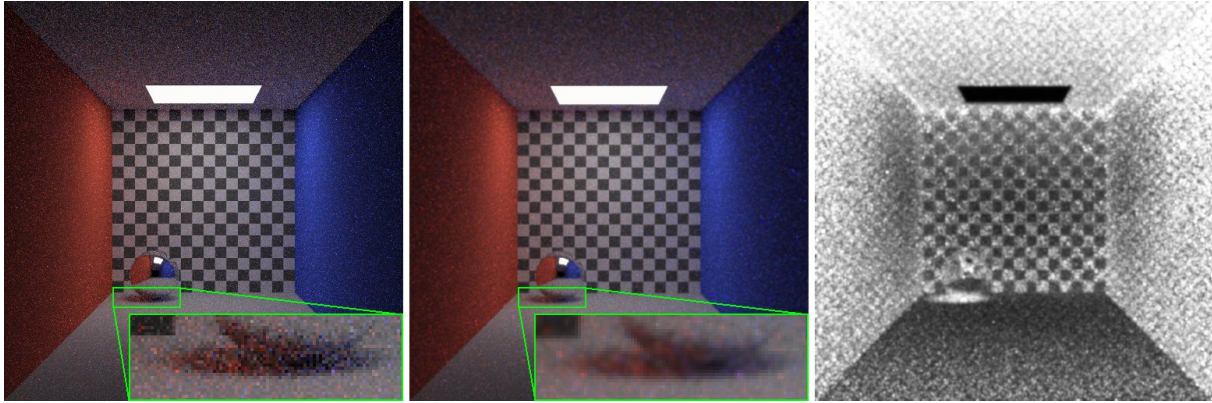
Figure 5: Noise reduction based on variance analysis of Path Tracing image (from left: no noise reduction, with noise reduction, variance image). The difference is noticeable especially in shadowed area beneath the sphere and on the indirectly illuminated ceiling.

ACKNOWLEDGEMENTS

REFERENCES

[1] James Arvo and David Kirk. Particle Transport and Image Synthesis. In *SIGGRAPH 1990 Proceedings*, pages 63–66, New York, NY, USA, 1990.

[2] Carsten Benthin. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University, Saarbrücken, Germany, 2006.

[3] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen Scher Zagier. Frameless rendering: Double buffering considered harmful. In *SIGGRAPH 1994 Proceedings*, volume 28, pages 175–176, New York, NY, USA, 1994.

[4] Abhinav Dayal, Cliff Woolley, Benjamin Watson, and David Luebke. Adaptive Frameless Rendering. In *Rendering Techniques 2005*, pages 265–275, 2005.

[5] Kate Devlin, Alan Chalmers, Alexander Wilkie, and Werner Purgathofer. Tone reproduction and physically based spectral rendering. In *State of the Art Reports, Eurographics 2002*, pages 101–123, September 2002.

[6] Bruce Fraser, Chris Murphy, and Fred Bunting. *Real World Color Management, second edition*. Peachpit Press, Berkeley, CA, USA, 2005.

[7] Anwar Ghuloum. The Problem(s) with GPGPU. http://blogs.intel.com/research/2007/10/the_problem_with_gpgpu.php, 2007.

[8] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[9] James T. Kajiya. The rendering equation. In *SIGGRAPH 1986 Proceedings*, pages 143–150, New York, NY, USA, 1986.

[10] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL Shading Language, version 1.50*, 2009.

[11] Morgan McGuire and David Luebke. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Perf. Graphics*, New York, NY, USA, 2009.

[12] Harald Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, Philadelphia, USA, 1992.

[13] Harald Niederreiter and Chaoping Xing. Low-discrepancy sequences and global function fields with many rational places. *Finite Fields and Their Applications*, 2(3):241–273, jul 1996.

[14] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, and Frédo Durand. A gentle introduction to bilateral filtering and its applications. Siggraph 2008 course notes, 2008.

[15] Steven Parker, William Martin, Peter-Pike Sloan, Peter Shirley, Brian Smits, and Charles Hansen. Interactive Ray Tracing. In *Symposium on Interactive 3D Graphics*, pages 119–126, 1999.

[16] Daniel Pohl. Light It Up! Quake Wars Gets Ray Traced. *Intel Visual Adrenaline*, 2:34–39, 2009.

[17] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, 2002.

[18] Michal Radziszewski, Krzysztof Boryczko, and Witold Alda. An Improved Technique for Full Spectral Rendering. *Journal of WSCG*, 17(1):9–16, 2009.

[19] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification, version 3.2*, 2009.

[20] Greg Spencer, Peter Shirley, Kurt Zimmerman, and Donald P. Greenberg. Physically-based glare effects for digital images. In *SIGGRAPH 1995 Proceedings*, pages 325–334, New York, NY, USA, 1995. ACM.

[21] Frank Suykens and Yves D. Willems. Adaptive Filtering for Progressive Monte Carlo Image Rendering. In *Proceedings of the 8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG) 2000*, pages 220–227, 2000.

[22] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, Stanford, CA, USA, 1997.

[23] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Interactive Global Illumination Using Fast Ray Tracing. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 15–24, June 2002.

[24] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, pages 153–164, 2001.

[25] Rui Wang, Rui Wang, Kun Zhou, Minghao Pan, and Hujun Bao. An efficient gpu-based approach for interactive global illumination. *ACM Transactions on Graphics*, 28(3):1–8, 2009.

[26] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. In *SIGGRAPH 1988 Proceedings*, pages 85–92, New York, NY, USA, 1988.

[27] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, 1980.