Implementing Classical Ray Tracing on GPU – a Case Study of GPU Programming

Andrew V. Adinetz, Sergey B. Berezin
Department of Computational Mathematics and Cybernetics,
Moscow State University, Moscow, Russia
{adinetz, s_berezin}@cs.msu.su

Abstract

General purpose GPU programming has become a topic of intensive research recently. In this context, GPUs are typically used as general-purpose stream processors. Although is usually suites most of existing algorithms, it is far from ideal for ray tracing as it generates extensive memory bandwidth.

We take an alternative approach and instead implement the entire algorithm in a single pass, as in classical ray tracing. This allows reducing memory bandwidth and increasing the performance of the algorithm. The results are demonstrated on several test scenes.

Keywords*: Ray Tracing, Interactive Graphics, GPGPU, GPU Ray Tracing*

1. INTRODUCTION

As GPU evolved to support programmable features, general purpose GPU programming became a field of intensive scientific research. The number of projects using GPU for various general purpose computations has been rapidly increasing in recent years. A site exists [1] where general purpose GPU news are regularly published. GPU computations has been applied to different domains, ranging from rendering-related to those which exhibit no direct connection to rendering problems.

The first class of problems includes rendering of translucent objects [2], classical radiosity [3], rendering caustics [4], [5], and, of course, GPU ray tracing. The second class of problems includes fluid dynamics simulation [6], [7], matrix multiplication [8], image and video processing [9], FFT [10], neural networking [11] etc. See [1] for more domains and solutions. Most of these solutions use the streaming programming approach [12] and envision a GPU as a streaming processor.

In this approach, the algorithm is usually broken into small parts – *kernels*, having no internal state, which take *streams* and also place streams into their output. A stream in this approach is a (possibly endless) sequence of uniform structures, resembling a file. At each step, a kernel takes a chunk from each of the input streams, processes it, and (possibly) puts one data chunk in each of its output streams.

This paradigm can be easily mapped to modern GPU architecture. Kernels are easily mapped to *pixel shaders*. A pixel shader is a GPU program that is run for each of the display image pixels independently. Independence of pixel thus allows executing the shaders for different pixels in parallel. Typically, modern graphics processors have from 8 to 48 pixel pipelines.

Streams are mapped to textures. Textures also allow random access to some kind of data, which can also be used to access static data during kernel execution.

"Conditional" kernels, that is, kernels, which can output to a stream depending on a certain condition on input data, are also mapped to shaders. However, an additional pass is performed after executing such a shader. This pass determines whether the input actually exists or whether it doesn't. This can usually be deduced from the output itself. Depending on whether the logical output exists, it writes the appropriate depth value. During the following shader execution, the shader is executed only for pixels which have the correct depth value, thus reducing the rendering time considerably because of "early depth kill". This write to the depth buffer cannot be performed in the main shader since it is usually also executed conditionally, and writing to the depth buffer in a shader prevents "early depth kill". In order to define whether or not there are pixels for which to execute the shader (i.e. the condition is "true"), occlusion queries are used [13].

As this framework seemed very natural on first GPUs, which were able to execute only short and simple pixel shaders, it has been applied to solving many general purpose computation tasks on GPU, including ray tracing. The first GPU ray tracing solution has been implemented in the context of this framework. It was envisioned by Purcell et. al in [14] and implemented for the first time in [15] for the photon mapping algorithm. The entire algorithm, however, could run on GPU only very slowly due to the limitations of graphics processors of that days (GeForce FX 5900 was used). Later, several more GPU ray tracing works appeared, which were all implemented using streaming programming paradigm, which compared using different acceleration structures for ray tracing.

Although this implementation seems quite natural and works on a wide range of modern graphics processors, it has been proved to have significant drawbacks.

First of all, it creates prohibitively extensive memory bandwidth by reading and writing intermediate variables to textures at every pass. Being justified for the previous-generation graphics cards, which had few registers and were able to execute only short shaders, it seems unfeasible on modern graphics processors, which have 32 registers and can run shaders of up to 512 instructions [16].

Second, it requires a very large number of rendering passes. As most of the passes are dependent and require an occlusion query operation, this, theoretically, can create a bottleneck. In practice, however, this is not a problem since the graphics memory bus seems to saturate much faster.

Extensive memory bandwidth created by reading and writing to textures makes developers to use different tricks on order to reduce the bandwidth. This, in turn, leads to cutting the

number and precision of intermediate variables, which either results in computational overhead or leads to loss of precision. All this leads to significant underuse of GPU computational possibilities in application to ray tracing. As a result, the GPU ray tracers perform much slower compared to their CPU counterparts.

On the other hand, modern graphics processors, which completely support OpenGL 2.0 feature set [17], provide efficient support for complex flow control instructions, such branching and looping with early break instruction. In order to be efficient, GPU ray tracing has to make use of all these features of modern graphics processors.

In this paper, we take a completely different approach to implementing ray tracing on a graphics processor. In contrast to streaming programming approach, which splits the algorithm into a number of passes, we implement the classical ray tracing algorithm in a single pass. Our implementation supports reflections and simple Phong shading [18]. It also runs faster than any of the existing GPU grid ray tracers and has a performance roughly comparable to that of an optimized CPU ray tracer. Our approach also supports moving objects, which is connected with a number of difficulties in other approaches.

The rest of the paper is organized as follows. Section 2 provides a brief overview of the classical ray tracing algorithm various subdivision structures used in it. Section 3 describes our approach and its implementation. Section 4 discusses some algorithmic issues connected with implementing classical ray tracing on GPU. The performance of our GPU ray tracer is demonstrated in section 5. Section 6 provides the discussion of the performance and outlines possible future work in this field.

2. CLASSICAL RAY TRACING ALGORITHM

Although ray tracing has been used for different purposes long before emergence of computer graphics and even computing, it had not been before 1980 that it was used for rendering purposes.

Whitted et. al [19] was the first to use ray tracing to render physically accurate images. Although using ray tracing for rendering had been proposed before, those proposals required a prohibitively large (especially on that-time computers) amount of computations because of the need to intersect every ray with every possible object (i.e., triangle) in the scene. What Whitted proposed were BSP trees. They allowed significantly reducing the amount of computation needed, since a ray had to intersect significantly less objects. Research has shown that spatial subdivision structures give acceleration by more than a factor of 100 on scenes complex enough.

The classical algorithm is widely known, however, we repeat it here. For every screen pixel, a ray is traced, and the first ray-object intersection is found. If the object intersected does not exhibit reflective or refractive properties, shadow rays are cast towards point light sources to define the color of the point. If the primary ray encounters a purely reflective surface, it is reflected and the entire procedure is repeated. For the refractive surface, two rays are spanned: the reflective ray and the refractive ray. In order for the algorithm to terminate in a finite time, either the maximum ray tracing depth is set or there exists a minimum intensity at which no further tracing is performed.

In 1985, Fujimoto et. al [20] proposed to use uniform grid subdivision structure for ray tracing. For the uniform grid, an efficient traversal algorithm exists. However, due to its uniformity, it demonstrates performance decrease on scenes with non-uniform object distribution.

As uniform grid, unlike BSP tree, does not require any kind of recursion or stack to be traversed, it is the primary candidate for GPU ray tracing implementation. It has been used in first GPU ray tracers. Although it has been proved to be not the fastest one in subsequent implementations, it is used in our approach.

3. OUR IMPLEMENTATION

In our approach, single pass uniform grid ray tracer is used. Single pass means that the entire ray tracing is done in a single pixel shader.

In order to provide support for moving objects, two-level spatial subdivision is used. At both subdivision levels, the uniform grid is used as the subdivision structure. The first level subdivision (also further referred to as “scene subdivision”) is a uniform grid, which is constructed for the bounding boxes of the scene *instances.* An instance is defined as an object pointer and a transformation matrix. The object pointer indicates which object is instantiated (hence the name instance), and the transformation matrix places the object in the scene. As multiple instances can reference a single object, memory space required to store geometry can be saved for the scene which contain a large number of the same objects.

Each object has an associated bounding box and its own subdivision structure (also referred to as “second-level subdivision”). This subdivision structure is a grid which is constructed for the triangles of the object. The entire approach to handling moving objects is similar to one used in some computer games, with uniform grids used instead of BSP trees.

As the number of various scene objects can be high, we are unable to hand-tune the size of the uniform grid for each of the scene objects. Therefore, we typically use a simple formula to compute the size of the grid by default. If hand-tuned grid size is used in some of the test examples, it is stated explicitly.

Although some traversal overhead is generated by two-level subdivision structure, it is completely offset by the quality of the grids created even without any manual tuning.

The ray tracing algorithm proceeds as follows. For each screen pixel, a view ray is computed, which is cast into the scene. Due to some limitations of modern graphics processors, we were able to implement only one procedure for ray casting, so the same ray tracing procedure is used for both primary and shadow rays.

The ray is first intersected with the scene bounding box. The data of the scene bounding box as well as the scene grid coefficients (see below) are stored in the shader uniform variables. If the ray does not intersect the scene bounding box, it is not traced further and no intersection is reported. If this is a primary ray, the pixel is simply colored with background color.

If the ray intersects the bounding box, its segment is clipped by the scene bounding box and proceeds to grid traversal. Grid traversal is first initialized, and at each traversal step, the ray either proceeds to the next voxel or terminates the traversal. When a ray encounters an empty voxel, it simply proceeds further. If, on the contrary, it encounters a non-empty voxel, it is successively intersected with each object of the grid. For the scene-level subdivision, these are object instances. For each instance intersection, the ray is

transformed to the local coordinate system of the object. Then it is tested against the object bounding box. In case of intersection, the ray segment is again clipped and proceeds to second-level grid traversal. This traversal is performed in the same way as the first-level subdivision. However, instead of instance intersections, the ray is tested against triangles.

The details of grid traversal are described below, in section 4.

The triangle intersection test is, in fact, a modified barycentric test [21]. First, the intersection point ray parameter is calculated. It is tested against the current segment. If it does not belong to the current ray segment, no intersection is report. Otherwise, the intersection point and barycentric coordinates are computed, and the latter are tested against the [0, 1] segment. The intersection takes place only if all barycentric coordinates are between 0 and 1.

The details of ray-triangle intersection are elaborated in section 4.

All attributes of the intersection point are computed only after the intersection point has been computed. This is performed for the sake of efficiency.

Figure 1: A Hind24 scene rendered with our GPU ray tracer. At ATI X1800 XL, it renders at 6 fps.

4. SOME ALGORITHMIC ISSUES

As modern graphics processors are significantly different from CPUs, the algorithm must be designed in a different way than for a central processor. Here we discuss some of the issues of algorithm design and implementations to achieve more effective performance on GPU.

First of all, general problems, such as addressing and texture allocation, are discussed. Then the discussion proceeds to some particular problems, such as implementing grid traversal and triangle intersection. The results are given in section 6.

4.1. General Issues

In order to perform ray on a scene using the graphics processor, the scene data must be first loaded to the graphics memory. The only way to access graphics memory in a pixel shader running on a modern graphics processor is to access a texture. Fortunately, there are no explicit limitations on a number of texture accesses.

The scene data, therefore, has to be written into the texture. It is unfeasible, however, to use a single texture to store scene data. First of all, the size of the texture is limited. The maximum extent of a 2D texture is 4096. As 1D textures have no more than 4096 elements, storing scenes in such textures is completely unfeasible. Although it is possible to use a 3D texture to store the entire scene, it is again unfeasible due to two reasons. First, there will be significant problems with addressing, as (at least) 2 floating point coordinates are needed. Second, different scene data require different levels of precision, and it is unfeasible to store all of them with the same precision. For example, geometry data require at least 32 bit floats to store in order not to exhibit “holes” in geometry. 2D addresses, however, can be stored with 16-bit precision (halves), as it is shown below.

2D textures may seem of insufficient capacity. However, as modern hardware is optimized for 2D texture access and 2D texture addressing has less overhead than 3D texture one, 2D textures are used, as their capacity is enough to store scenes of moderate complexity.

Most of existing GPU ray tracing approaches use 2D textures as virtual 1D arrays. They store addresses as single-precision floating point values and translate them into 2D values for actual texture fetches. While that simplifies programming, it also creates address translation overhead. As address translation may involve operations which cannot be easily vectorized, and the number of address translations per ray is large enough, this overhead tends to be significant.

In order to overcome this, we use 3D addressing for grids and 2D addressing for other data. This eliminates address translation overhead. However, it introduces some other obstacles. Each address requires now 2 floating point values to store, which may require more memory and more bandwidth. However, we found out this overhead to be not of great importance. Address data are often stored in separate textures (as indices of triangle vertices), and they are stored with 16-bit precision, which means that no memory or bandwidth overhead is created.

Texture allocation is another problem. As the scene now contains multiple objects, and the number of textures available is limited, a single texture must be shared between similar data from multiple objects. Therefore, texture allocation must be performed. However, various types of data require various allocation types. For example, 3D grids require allocation of contiguous 3D data blocks, while triangle data may be allocated in multiple blocks containing a certain number of triangles each. Currently allocation is performed by using custom allocation code for different types of textures; however, we are now switching to performing it in a separate library.

4.2. Grid Traversal

Using uniform grids is a common approach in GPU ray tracing. From the first sight, grids seem to be the most GPU-friendly subdivision structure. Modern graphics processors provide support for 3D textures, which is the natural way to store grid data.

In our approach, we store grid data in a 3D texture. First of all, as grid texture must be shared between multiple objects, it must be allocated. A simple approach for 3D texture allocation is used. The entire 256 x 256 x 256 texture is split into blocks of size 16 x 16 x 16. For each block, a flag is maintained whether it is currently allocated or not. If a grid data block of a specific size needs to be allocated, the number of 16x16x16 blocks required to contain is computed. Then the contiguous amount of these blocks is allocated for the grid data.

For the grid traversal itself, we use a variation of 3DDDA algorithm [20]. As modern graphics hardware does not

provide implementation for fast dynamic indexing of vector components, no leading direction is selected. The ray direction is used as a leading one instead. Though it results in some computational overhead, it can be efficiently implemented in GPU using fast vector operations.

The algorithm is described in detail in the pseudocode provided in figure 2.

```
vec3 signDir, steps, voxSeg, nextTs, vc;
float vstart, vend, gridt;
initTraversal(grid, ray, start, end) {
 gridt = end;
 signDir = sign(k) / l;
 steps = abs(l / k);
 vc = ceil((r0g + kg * start) / l) * l;
 nextTs = (vc + step(0, signDir) – r) / k;
 vstart = start;
 vend = min(nextTs.x, nextTs.y, nextTs.z);
}

nextVoxel() {
 t = vend;
 mask = step(nextTs, t);
 vc += mask * signDir;
 nextTs += steps * mask;
 vstart = t;
 vend = min(nextTs.x, nextTs.y, nextTs.z);
}

bool finished() {
 return vend >= gridt;
}

bool traverseGrid(grid, ray, segment) {
 s = clip(grid.box, ray, segment);
 if(!empty(s)) {
  initTraversal(grid, ray, s);
  do {
   v = gridVoxelAt(vc);
   if(v.nObjects != 0) {
    if(intersectTrigs(ray, vstart, vend, v.addr)) {
     update t;
     return true;
    }
   } //end of if()
  } while(!finished())
 } else
  return false;
}
```

Figure 2: The pseudo code of our grid traversal algorithm.

Fortunately, we were able to design a grid traversal approach which results in encoding the entire grid data in only 2 3D vector coefficients. These grid coefficients are computed using the following formulae:

$$\boldsymbol{a} = \boldsymbol{NL} / \Delta l,$$

$$\boldsymbol{b} = \boldsymbol{c_0} - \boldsymbol{g_0 NL} / \Delta l.$$

The coefficients in the formulae have the following meaning:

$\boldsymbol{L}$ – the size of the grid in world coordinates.

$\boldsymbol{N}$ – the number of voxels along each dimension

$\boldsymbol{c_0}$ – the address of the texture portion allocated to the grid

Δl – the "size" of a single 3D texel in texture coordinate. Since the size of the entire texture in texture coordinates is 1, Δl would be usually 1/256, that is, inverted number of "voxels" along a single grid texture dimension.

$\boldsymbol{g_0}$ – the origin of the grid in the world coordinates.

These coefficients are applied to ray coordinates in the current coordinate system (which is either the world coordinate system for the scene grid or the local coordinate system for the object grid). The transformed ray data are computed using the following formulae:

$$\boldsymbol{r_{0g}} = \boldsymbol{a r_0} + \boldsymbol{b},$$

$$\boldsymbol{k_g} = \boldsymbol{a k}.$$

Then the algorithm described in the pseudocode in figure 2 is applied for ray-grid traversal. Whenever a non-empty voxel is encountered, all objects contained in it are successively intersected.

This algorithm is, of course, applied on the 2-level basis. First, it is applied for scene grid traversal, and then for the object grid traversal. The actual code for grid traversal is a bit different for two levels, however. Current graphics processor available supports only 4 nesting loops, which is not enough to use the same algorithms at both levels (as 1 loop is required at each level for both grid traversal and successive object intersections, and 1 loop is required for a high-level ray tracer, totaling to 5 nested loops). For the object-level, as the cost of ray-triangle intersection are relatively low, a common 2-loop approach is used. For the grid traversal, however, another approach is used. Both grid traversal and scene object intersection is done in one loop. At each iteration, the current action (further traverse or intersect an object) is selected and then performed. It requires an additional if statement. Although it may result in some overhead, we've found out that it is very little, and that this technique can be applied.

4.3. Triangle Intersection

For the triangle intersection itself, the barycentric test is used [21]. Originally, the projection plane (one of the coordinate planes) has been chosen for the triangle before the actual ray tracing (that is, at the preprocessing stage). During the intersection, the ray would have been projected to that plane and the coordinates of the ray projection would be used for computation of barycentric coordinates.

This, however, is impossible for graphics hardware, as it provides no support for dynamic indexing. Therefore, a full set of ray coordinates needs be used for the barycentric test computations. As such, the test can benefit from vector instructions, such as dot products, available on modern graphics hardware.

Since modern GPUs are designed to work with 4D vectors, it is useful to write the entire test in terms of 4D vectors (ATI cards have separate units for performing 3D vector / scalar instructions; however, implementing the test in 3D vectors resulted in a performance lost even on ATI cards). The 3D vectors are extended to 4D as follows. A positional vector (such as ray origin or an intersection point) gets the 4^{th} coordinate (further referred to as w) equal to 1, while the directional vector (a normal or a ray direction) gets the 4^{th} coordinate equal to 0. The plane with the equation $Ax + By +$

$Cz + D = 0$ is written in the form of (A, B, C, D). As the equation is specified only up to the constant factor, we use various normalizations for the plane equation discussed below.

The triangle is actually represented as a set of 4 planes. Actually, 3 planes are enough, but the 4^{th} is still stored. Each plane is stored in a separate texel of a 4-component 32-bit precision floating point texture. The first plane represents the triangle plane. It is normalized so that $A^2 + B^2 + C^2 = 1$. 3 other planes are orthogonal to the triangle plane and actually form the sides of the triangles in intersection with the triangle plane. The normals of these planes are directed inside the triangle and they are normalized so that $A^2 + B^2 + C^2 = h^2$, where h is the height of the triangle to the respective triangle side.

In this setting, the barycentric coordinate of a specific point lying in the triangle plane can be computed by a simple dot product of the triangle side 4D vector and the intersection point 4D vector. The t-value of the intersection point is simply a negated relation of dot products of the triangle plane vector and the ray origin and direction, respectively.

The entire ray-triangle intersection code is presented in figure 3.

```
float t = planeTime(pl, r);
 if(s.start <= t && t <= vv.z) {
  vec4 ttc = vc.xyxy + vec4(TEX_INCR_TRIGS, 0.0, 2.0 *
TEX_INCR_TRIGS, 0.0);
 vec4 p = pointAt(r, t);
 Plane pa = planeAt(ttc.xy);
 float la = dist(pa, p);
 Plane pb = planeAt(ttc.zw);
 float lb = dist(pb, p);
 float ifl = step(4.0, dot(vec4(float(1.0), float(la >= 0.0),
float(lb >= 0.0), float(la + lb <= 1.004)), vec4(1.0, 1.0, 1.0,
1.0)));
}
```

Figure 3: Ray-triangle intersection pseudocode.

Note it is provided here with a number of optimizations. The entire set of variables updated after each triangle intersection is packed in a single 4D vector variable for the sake of efficiency. A complicated expression is used to determine whether or not the ray intersects the triangle. Originally, we used a simple sequential "and" expression. However, we replaced it with the one above for the sake of efficiency. For the same reason, the entire portion of code after computation of the plane intersection t-value and testing it against the segment is placed inside a conditional expression. While this expression is simply ignored on NVIDIA graphics cards, it yields about 10% additional performance on ATI graphics boards since the latter provide a special execution unit for conditional and loop statements.

Another question is choosing a data layout for storing triangles. One way is to use a common approach and store separate triangle data for each voxel. This could bring performance benefits; however, it is likely to increase the storage requirements greatly as a single triangle may be shared between multiple voxels (and it is usually so for a fine enough subdivision). We have therefore chosen to store triangle data in a separate texture, in which every triangle is stored only once. Another texture holds lists of references to the triangles. The lists are contiguous for each voxel, and the list of references of each voxel occupies only a single line, not stranding to the next line. This allows us to switch to the next triangle (or the next object) by simply incrementing the x coordinate of the texture address. This also allows reducing the overhead of additional reference since the addresses are stored in a 2-component form. As 16-bit floating point values are used to store addresses, only little amount of memory is required to store addresses, no more than normally on a PC for storing pointers. On the other hand the storage overhead for triangle data is greatly reduced. The same approach is used for storing pointers to the instances of the scene subdivision.

5. RESULTS

The GPU ray tracing approach has been implemented in Visual C# 2.0 and .NET 2.0 language using Microsoft Visual Studio 2005. OpenGL 2.0 [17] has been used as a graphics API. The shader has been written in GLSL v. 1.10 [22]. GLSL has been preferred to other GPU shader languages (HLSL and Cg) due to its more direct support for the features available in modern graphics hardware (such as break instructions).

Although C# is typically considered not a fast language for scientific programming, it proved to be fast enough for our purposes. Moreover, GPU performance is far more critical in our applications, so, C# overhead is insignificant.

The performance of our approach has been measured on a number of test scenes. As our ray tracing approach provides support for moving objects, moving objects has also been tested. Since ray tracing itself is actually a major bottleneck in our approach, moving various objects does not affect significantly the performance of our application.

The parameters of our test scenes are summarized in table 1. The performance is given in table 2.

Table 1. Parameters of the scenes used for testing.

scene	#trigs	#insts	#lights
Chair_wood	33404	11	1
PalmPC	13532	7	1
Balls-1	31200	10	2
Knife	1676	1	1
SR71A	4446	1	1
Palace	33000	36	1

Table 2. Performance of our ray tracing system on a number of test scenes.

Scene	Performance (fps)
Chair_wood	12.8
PalmPC	6.3
Balls-1	5.8
Knife	16
SR71A	7.9
Palace	10.1

Figures 4, 5, 6 and 7 present some of the images generated. All tests were run on a PCI-E ATI X1800 XL graphics card with no clocking. Images were rendered at 512x512 resolution.

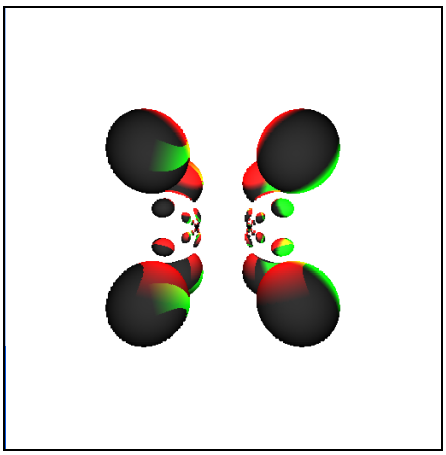

Figure 4: The Balls-1 scene image rendered using our approach. Note the interreflections between the two reflecting spheres. As the spheres are moved, the interreflections are fully recomputed interactively as the scene is rendered with ray tracing.

Figure 5: The wooden chair scene rendered using our approach. Note the shadow cast from a distant light source by the back of the chair.

6. DISCUSSION AND POSSIBLE FUTURE WORK

We have implemented a GPU ray tracing system which outperforms existing GPU ray tracers and demonstrates performance which is roughly comparable with highly optimized CPU ray tracers. Our system is also able to support moving objects, a feature which previous GPU ray tracers lack Although it is interactive, it still does not provides real-time ray tracing on a single computer.

This case study has shown that modern GPUs are suitable not only for streaming programming with short shaders, but also for writing much more complex programs which perform complex computations, such as entire ray tracing, in a single pass.

Figure 6: The palace scene.

Although complex branching can create some overhead, single-pass implementation pays off in terms of memory bandwidth. Unlike streaming programming, it does not generate a large amount of graphics memory bandwidth, but is rather computationally-bound. This can be viewed as a drawback; however, it can also be viewed as the advantage of the approach. Typically, increasing the computational power by just adding more pixel shader processors is much simpler than increasing memory bandwidth. Therefore, our approach is likely to benefit significantly from the ATI X1900 graphics board, which provides 3 times more shader processors. Our approach is also likely to benefit from the CrossFire solution, as adaptive load balancing between 2 graphics cards will most likely double the performance.

There is still room for improvement, however. Approaches which perform single ray casting instead of entire ray tracing in a single pass needs to be investigated. Although memory bandwidth is slightly increased in such approaches, they are more flexible since they allow more complicated traversal algorithms. They would also allow separate code for tracing shadow rays.

The GPU instruction set resembles that of and SSE processor, such as Pentium 4. Therefore, tracing 4 rays simultaneously is also worth investigating, which is likely to be performed in our future work.

Finally, for scenes small enough, a hybrid approach can be used. It may involve using GPU ray tracing for reflections computations and common graphics pipeline for other features (shadows). This may prove much faster in scenes with not very large number of triangles.

7. ACKNOWLEDGEMENTS

We are thankful to the Department of Computational Mathematics and Cybernetics and to the Student Microsoft Technology Lab at CMC MSU for the equipment provided.

8. REFERENCES

[1] General Purpose GPU Computations. http://www.gpgpu.org

[2] Hendrik P.A. Lensch, Michael Goesele, Philippe Bekaert, Jan Kautz, Marcus A. Magnor, Jochen Lang, Hans-Peter Seidel. *Interactive Rendering of Translucent Objects. Proceedings of Pacific Graphics, 2002, pp. 214 – 224, October 2002*.

[3] Nathan A. Carr, Jesse D. Hall, John C. Hart. *GPU Algorithms for Radiosity and Subsurface Scattering. Proc. Graphics Hardware, July 2003*.

[4] Chris Trendall, A. James Stewart. *General Calculations using Graphics Hardware, with Application to Interactive Caustics. 11th Eurographics Workshop on Rendering, June 2000, pp. 287 – 298*.

[5] Tim Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, Pat Hanrahan. *Photon Mapping on Programmable Graphics Hardware. Graphics Hardware, 2003*.

[6] Wei Li, Xiaoming Wei, Arie Kaufman. *Implementing Lattice Boltzmann Computation on Graphics Hardware. To appear in Visual Computer, (Heidelberg, Germany), 2003*.

[7] Mark J. Harris. *Fast Fluid Dynamics Simulation on the GPU. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, Chapter 38. Addison-Wesley, 2004*.

[8] Jesse D. Hall, Nathan A. Carr, John C. Hart. *Cache and Bandwidth Aware Matrix Multiplication on the GPU Tech Report UIUCDCS-R-2003-2328, University of Illinois Dept. of Computer Science, 2003*.

[9] Pete Warden. *GPU Image Processing in Apple's Motion. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, 2005*.

[10] Thilaka Sumanaweera, Donald Liu. *Medical Image Reconstruction with the FFT. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, 2005*.

[11] Kyoung-Su Oh, Keechul Jung. *GPU implementation of neural networks, Pattern Recognition, Vol. 37, No. 6, 2004, pp. 1311-1314*.

[12] John Owens. *Streaming Architectures and Technology Trends. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, 2005*.

[13] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. *A Survey of General-Purpose Computation on Graphics Hardware. Eurographics 2005, State of the Art Reports, August 2005, pp. 21-51*.

[14] Timothy J. Purcell, Ian Buck, William R. Mark, Pat Hanrahan. *Ray tracing on programmable graphics hardware. Proceedings of the 29th annual conference on Computer graphics and interactive techniques, Pp. 703 - 712, 2002*.

[15] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. *Photon Mapping on Programmable Graphics Hardware. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, pp. 41-50, 2003*.

[16] Emmett Kilgariff, Randima Fernando. *The GeForce 6 Series GPU Architecture. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, Addison-Wesley, 2005*.

[17] The OpenGL Graphics System: A Specification (Version 2.0). http://www.opengl.org

[18] Bui-Tuong Phong. *Illumination for computer generated images, Communications of ACM Vol. 18, Issue 6, 1975, Pp. 311-317*.

[19] Turner Whitted. *An improved illumination model for shaded display. Communications of the ACM. Vol. 23, Issue 6, 1980, Pp. 343 – 349*.

[20] A. Fujimoto, K. Iwata. *Accelerated Ray Tracing, Proc. CG Tokyo'85, Pp. 41-65*.

[21] Tomas Möller, Ben Trumbore. *Fast, minimum storage ray-triangle intersection. Journal of graphics tools, Vol. 2(1), Pp. 21-28, 1997*.

[22] John Kessenich, Dave Baldwin, Randi Rost. *OpenGL Shading Language. http://www.opengl.org* .

About the Authors

Andrew V. Adinetz is a student at the Department of Computational Mathematics and Cybernetics at Moscow State University. His research field includes interactive ray tracing and global illumination, GPGPU programming and mobile technologies. His phone number is +79262833921. His contact e-mail is adinetz@cs.msu.su.

Sergey B. Berezin, Ph. D., is an assistant professor at the Department of Computational Mathematics and Cybernetics at Moscow State University. His research interests include hardware accelerated rendering, scientific visualization and practical and theoretical aspects of Microsoft .NET platform. E-mail: s_berezin@cs.msu.su.