Physically Accurate Rendering with Coherent Ray Tracing

Andrew Adinetz**, Boris Barladian*, Vladimir Galaktionov*, Lev Shapiro*, Alexey Voloboy*
*Keldysh Institute for Applied Mathematics RAS, Moscow
**Moscow State University

Abstract

As the processing power of modern CPUs increases, coherent ray tracing becomes more and more popular, as it allows significantly accelerating ray tracing using SIMD instructions. It turns out, however, that as ray tracing is accelerated, other parts of physically accurate rendering algorithms tend to become bottlenecks.

In this paper, we introduce a coherent physically accurate rendering approach, which allows taking advantage of SIMD capabilities of modern CPUs at every stage of rendering computations. We demonstrate coherent algorithms for lighting and material computations as well as for anti-aliasing and tone mapping. The comparison performed on a number of test scenes demonstrates significant acceleration compared to common non-coherent approach.

Keywords: *SSE, interactive ray tracing, tone mapping, antialiasing, texturing, BRDF, photorealistic rendering.*

1. Introduction

Coherent ray tracing, that is, tracing a number of rays simultaneously, has been a subject of scientific research in recent years. As modern commodity CPUs (central processing units) appear to support various kinds of SIMD (Single Instruction Multiple Data) extensions, which allow performing arithmetic operations on multiple floating point numbers simultaneously, tracing several rays in parallel becomes quite natural. Various CPUs offer various SIMD extensions, such as Intel SSE (Streaming SIMD Extension) [1], AMD 3DNow! [2] and Motorola AltiVec [3]. Of all these, only Intel SSE became common for coherent ray tracing implementation. Currently, there exist multiple SSE coherent ray tracing projects. One of the most widely known is [4] by Slusallek, Wald et al., which uses SSE to perform interactive ray tracing. Another example is the VirtualRay project [5], which uses SSE for interactive ray tracing of scenes consisting entirely of spheres. In fact, due to its wide availability, SSE became a de-facto standard for implementing coherent ray tracing. Hence both SSE ray tracing and coherent ray tracing will be used as synonyms across the paper.

Computer Graphics department of Keldysh Institute for Applied Mathematics RAS, has a long experience of creating systems for physically accurate rendering [6], [7], [8]. It has created a number of applications which support various aspects of photorealistic rendering. Our scientific research results were used by Integra Inc. to create commercial products [9]. As SSE becomes widely available, it becomes highly desirable to be supported for both interactive and offline rendering.

It is important to note that ray tracing is an essential, though not the only part of photorealistic rendering. The latter also includes modeling of physically accurate surface properties and physically accurate lighting with complex light sources. To render scene features with subpixel size correctly, rendering algorithm must include antialiasing, and tone mapping is necessary to obtain final images from lighting simulation results which usually are of high dynamic range.

In this paper, we present an approach which benefits from SSE support and coherency not only in the process of ray tracing, but also for shading, tone mapping, lighting etc. Using SSE instructions gives nearly 4-times acceleration compared to non-SSE implementation.

The paper is organized as follows. Section 2 briefly describes the architecture and design of the Inspirer2 rendering system and outlines the main parts of the coherent ray tracing solution. Section 3 describes the coherent ray tracer. Section 4 deals with surface materials and BRDFs (bidirectional reflection distribution functions) in coherent ray tracing approach. Section 5 describes the support of multiple types of light sources for coherent ray tracing. Section 6 is about coherent antialiasing and tone mapping. The performance of the system is demonstrated on a number of test scenes and the results are reported in section 7. Section 8 is devoted to the results discussion and possible future work.

2. Outline of the Solution

The physically accurate rendering with coherent ray tracing is implemented on the basis of Inspirer2 rendering system (formerly Fly) [10].

This system has been designed to provide support for both interactive and offline rendering modes. Interactive mode implementation was based on OpenGL. In interactive mode, the system, while providing real-time framerates (25 – 30 fps), aims at supporting the best level of physical accuracy possible at such frame rates. In interactive mode, the system is able to provide physically accurate shadows from point light sources and BRDF support for surface material specification. It is also able to approximate reflections using environment maps.

In offline rendering mode, it provides physically accurate rendering using bi-directional ray tracing. It supports both point and surface light sources with goniodiagrams in order to create realistic lighting. It also provides support for materials with complex BRDFs and textures. Rendering is able of calculating multiple order reflections. For the global illumination computation, an algorithm based on forward Monte-Carlo ray tracing (MCRT) is used. The results of MCRT are stored in illumination maps [11] and used in both rendering modes.

In fact, both interactive and offline rendering may benefit from the SSE ray tracing. Offline rendering is likely to perform 2 – 3 times faster since SSE is able to trace 4 rays in parallel. As to interactive rendering, SSE ray tracing may be used in a hybrid approach to provide physically accurate reflections and refractions over an OpenGL rendered image.

The Inspirer2 architecture is presented in fig.1. The design of the architecture has been driven by the necessity of preserving physical attributes of the scene.

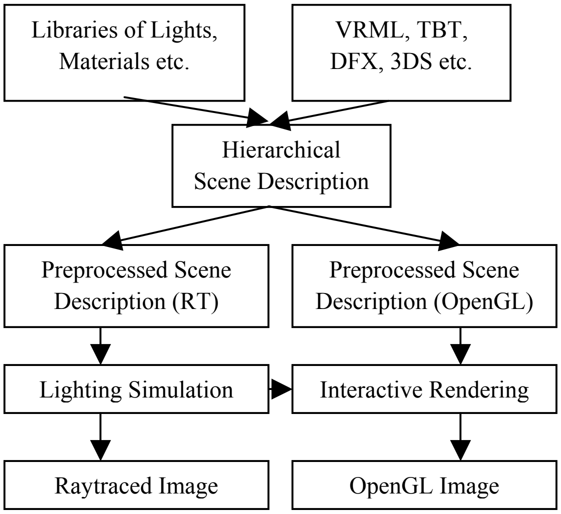
Libraries of Lights, Materials etc.
VRML, TBT, DFX, 3DS etc.
Hierarchical Scene Description
Preprocessed Scene Description (RT)
Preprocessed Scene Description (OpenGL)
Lighting Simulation
Interactive Rendering
Raytraced Image
OpenGL Image

Figure 1: Overview of the system architecture. Generic Hierarchical scene description with physical attributes is converted to two Preprocessed Scene Description – one for OpenGL engine and another for ray tracing engine (including MCRT).

As it is possible to build several rendering databases simultaneously for a single scene data base, it becomes easier to attach a new rendering engine, including one based on SSE, to the existing browser application.

It has been decided, however, that a new scene rendering database need not be created to support SSE rendering as there exists already a database for Monte-Carlo ray tracing. Instead, this database has been slightly modified, where necessary, to better support coherent ray tracing. Such optimizations, however, has not been numerous due to the fact that MCRT database has been optimized already for the existing ray tracer.

The coherent ray tracing, as well as other components of the solution, has been implemented in C++ language. No assembler has been used. In order to use SSE functionality, classes have been designed, which presented a high-level wrapper over the SSE intrinsics provided by the Microsoft/Intel C++ compiler [12]. As modern compilers tend to be able to optimize high-level code efficiently, this is not likely to cause a large loss of performance, while considerably improving the maintainability of the code written.

3. Coherent Ray Tracer

Ray tracing is typically considered the most time-consuming part of any physically accurate algorithm. Whitted estimated the time spent on ray tracing as 95% of the total rendering time [13]. For physically accurate rendering, the relative amount of time spent on ray tracing is less; however, it is still about 65 – 75%, according to our estimates [14]. This makes the ray tracing process a primary candidate for SSE optimization.

A common approach to SSE ray tracing optimization is tracing 4 rays in parallel as SSE performs operations on 4 32-bit floating point numbers simultaneously.

Conceptually, the algorithm is not changed significantly. The BSP tree space subdivision technique is used for ray tracing speedup. So the algorithm consists of both ray traversal phase and object (or triangle) intersection phase. As support for moving objects is highly desirable, a two-level hierarchy is used, similar to [15]. The scene consists of objects. Each object is a triangle mesh, which has its own BSP tree (further referred to as second-level BSP tree or object BSP tree) and a bounding box. The object is placed into the scene using the transformation matrix. The entire scene consists of the set of the object bounding boxes, the scene BSP tree (further referred to as first-level BSP tree) and the scene bounding box. As only object bounding boxes are included into the scene tree, the subdivision may seem to be not as efficient; however, this pays off by providing support for moving objects. This two-level subdivision may even improve the subdivision structure in sparse scenes, where it allows better handling of empty spaces between objects.

As up to 4 rays are traced simultaneously, they may take different code paths in the ray tracing algorithm. As such, means for temporarily blocking some of the rays is necessary. In our approach, the mask of currently active rays is used for this. Typically, it is an SSE value (that is, a quadruple of 32-bit floating point values), which contains either 0x00000000 or 0xffffffff (bitwise notation) in each of the four positions. Using the mask allows to block some of the rays, if different rays take different paths in conditional or loop instructions. Masks are used for blocking rays which do not traverse the current object, which do not traverse the current node of the BSP tree or the first intersection of which has already been found. Masking is a widely used technique in SSE programming. In this paper, it is used in almost all components of our solution. Therefore, a special type has been designed for SSE mask (QBool type), for the sake of code maintainability.

The coherent ray tracing algorithm proceeds as follows. First, all the rays are tested for intersection with the scene axis-aligned bounding box (AABB). If all the rays miss AABB, the algorithm immediately reports no intersection. If some of the rays intersect the AABB, the mask of currently active rays is updated, and non-intersecting rays are excluded.

Then the algorithm proceeds to BSP traversal. As the rays which have different direction signs can have different BSP traversal order, the entire group is split into subgroups with the same direction signs. Although this reduces the efficiency of SSE ray tracing, the actual splits occur rather rarely. Moreover, it can be shown that rays which have a common intersection point, for instance, primary rays for a pinhole camera or shadow rays for a point light source, always have the same traversal order.

After splitting the rays into groups, the algorithm sets the mask for currently active ray and proceeds to BSP tree traversal. As the hierarchy is, in fact, two-level, the same occurs both for the entire scene and for each of the objects which is being tested for intersection. For each non-leaf node, the BSP traversal algorithm proceeds as follows. If all of the rays go only to right or only to left subnode, the algorithm just updates the current node address and proceeds further with this subnode. If some of the rays intersect both subnodes, then the far one is pushed to the stack, the mask of active rays is updated and the algorithm proceeds with the near subnode of the current node. Note that due to splitting rays into groups (see above) the situation where the rays traverse two nodes in opposite order is impossible. It is possible, however, that some of the rays traverse both nodes and some traverse only one of them. In this case, rays which do not traverse the current node are blocked; they are activated again only for the other node traversal.

When the algorithm reaches a leaf-node, it proceeds to the ray-object intersection. For the first level subdivision, objects are actually scene objects, and the ray is transformed and proceeds with every object in much the same way it does with the entire scene. For the second level subdivision, objects are actually triangles which need be intersected.

Ray-triangle intersection actually uses the modified projection barycentric test implemented in SSE as described in [4]. First, it is checked whether the intersection has been cached. In case of a cache hit, the intersection data (that is, the t value at the intersection point) are simply taken from the intersection cache and tested against the current ray segment. If this test succeeds, then the ray-triangle intersection is reported.

In case of a cache miss, the entire intersection procedure is performed. First, the t value at the ray-plane intersection point is computed for currently active rays. If t is negative for all active rays, then the procedure returns immediately reporting no intersection. Otherwise, the active rays mask is corrected and a pair of coordinates of ray-plane intersection point is computed. Actually, the coordinates computed correspond to the axes of the plane to which the triangle projection is the largest. Barycentric coordinates of this point are computed and tested for the intersection. If the intersection occurs, then the t value is checked against the current ray segment. If it belongs to the current ray segment, the intersection is reported. The barycentric coordinates are then written to the shading context (a global structure which contains shading-related data). The same is performed in case of a cache hit, if the intersection t value belongs to the current ray segment.

If the intersection t value does not fall into the current ray segment, only the cache data are updated. The intersection t value and the intersection point coordinates are written to the plane cache and no intersection is reported.

After testing all the objects in the node, those rays which reported intersection are deactivated, as they need not traverse any further. If all the objects in the first-level leaf node have been tested, and intersections have been found for a number of rays in this node, these intersections are actually considered *first intersections* of those rays. After the first intersection has been found, all intersection data are put into the shade context.

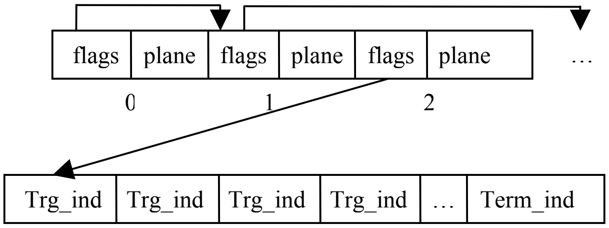

Figure 2. The BSP and triangles data layout used in our SSE ray tracer.

Our framework, in fact, requires having 2 intersection methods. One of them finds only the first intersection of the ray. The algorithm how it works has been described above. The other one finds *all intersections* of the current ray quadruple. Actually, it proceeds in much the same way as the first intersection algorithm, those rays which have intersected the object, are not immediately disabled but rather continue further – until all intersections for them has been found.

The layout of the scene subdivision and triangle data (shown in figure 2) is optimized with respect to cache coherency. Therefore, both children of a BSP tree node are stored in one memory chunk.

The BSP is constructed using the parameterizable algorithm described in [16]. As object BSP trees are typically constructed only once, at the preprocessing stage, more time is devoted to their construction, which results in faster ray traversal. On the contrary the first level BSP tree is constructed often due to object movement. As BSP subdivision algorithm typically requires superlinear time to run, less time can be devoted to BSP tree construction. Therefore, a fast algorithm is used here, which constructs a less efficient BSP tree, though it does it faster.

The BSP ray tracer has been run separately for a number of test scenes. All of the tests are performed on the basis of the same number of pixel. Figure 3 shows a picture rendered with SSE ray tracer described here.

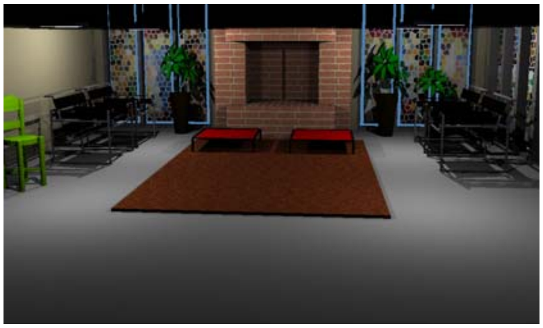

Figure 3. A room scene rendered with SSE ray tracer.

4. Surface Materials and BRDFs

As coherent ray tracing gives about 3.5 times speedup compared to ordinary ray tracing algorithms, other parts of the physically accurate rendering algorithms can actually become bottlenecks. As it has been mentioned in the beginning, the portion of time spent in ray tracing is about 70% total rendering time. SSE ray tracing accelerates the ray tracing itself about 3 times on the average, therefore its portion of rendering time is reduced to less than a half. In order to reduce the whole rendering time in the most effective way, other components do also need SSE acceleration.

In fact, this problem has been encountered by other SSE ray tracing projects. In [17], after the authors of the project implemented ray tracing on SSE, shading actually became a bottleneck. According to their meaning, even simple Phong shading can increase rendering time considerably in such a situation, let alone complex shading algorithms involving textures and BDFs.

Therefore, the need for implementing coherent material and BDF processing is clear. Since all previous our developments were oriented on delivering physically accurate images rather than creating visually pleasing effects, no general procedural shading is considered.

A physically accurate material we use typically consists of the following components:

1. A simple bundle of Phong-like material attributes
2. Reflection and refraction-related attributes
3. Material textures
4. Material BDFs

The first two items are rather straightforward to implement in SSE. In fact, they involve simple vector and color operations, which can be easily implemented in SSE. Lighting computations require tracing shadow, reflection and refraction rays, but this can be easily accomplished by the means of efficient SSE ray tracer described above. Texturing and BDF support not as straightforward to implement in SSE, however. The aspects of their implementation are described and discussed separately in two subsequent sections.

4.1. Coherent Texturing.

In our framework, texture is an image applied to an object to modify its visual appearance. Figure 4 shows an example of an image with texture rendered using our framework.

Texture itself belongs to the material, and each face has a material applied to it. Texture mapping is done by the means of 2D texture coordinates, which are provided for each vertex belonging to a textured triangle.

Thus, the texture coordinates at the intersection points need to be interpolated first of all. This is easily accomplished for 4 rays simultaneously using the barycentric coordinates calculated during the intersection point. The texture coordinates for the triangle vertices are loaded separately for different intersection triangles and are accumulated using a mask. The interpolation is performed using SSE instructions.

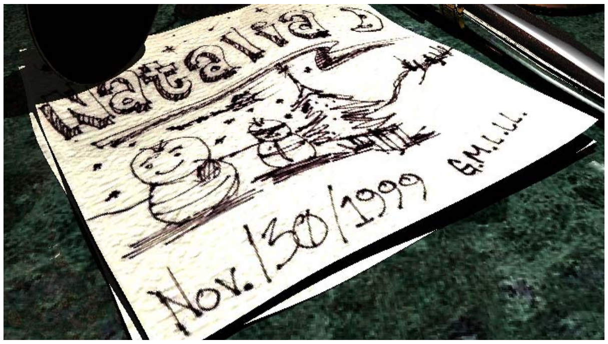

Figure 4. An example of a texturized Glass1 scene rendered with our coherent physically accurate algorithm.

The second stage is the interpolation of the texture value itself. Prior to color loading, the exact coordinates at which to take texture values must be determined. As the texture are 2D and use mipmapping, tri-linear filtering needs to be performed. Therefore, 3 coordinates are to be determined for each of the active quadruple rays. Calculation of first two texture coordinates, which correspond to the *x* and *y* position of the texel, are straightforward since they involve only division and number conversion. To provide the wider applicability of our solution, we refrain from using SSE 2; therefore, we model integer numbers by the means of floating-point ones, and the conversion actually takes place only when the texture is actually sampled.

The computation of the mipmap coordinate is more complicated as it involves computing logarithm of the distance to the viewpoint (the t value of the intersection point). As the nearest integer to the logarithm is needed rather than the value of the logarithm itself, it has been decided that successive division by 2 (that is, multiplication by 0.5 for the sake of efficiency) be used instead. As the size of the texture is not very large (typically not more than 1024 x 1024), the number of mipmap levels is not very large too (not more than 11), so the number of iterations is typically small. In fact, the loop is likely to terminate in the same number of iterations for all active the rays of the quadruple since they hit the surface near to one another. On the other hand the computation of the approximate value of the logarithm by the means of the Taylor series, for example, may turn out to be rather complicated and unstable due to its slow convergence. Moreover, it would involve normalizing the ray coordinate of the intersection point by some other value for the series to converge.

Finally, the inverted tone mapping needs to be performed since the initial texture image is stored with only 8 bits of precision. This is accomplished in much the same way as the final tone mapping of the rendered image, which is discussed in section 6.

Separate tests for only texturing performance were not fulfilled. The reason is that it can vary greatly for different surroundings and for different textures, as texture access, contrary to BDF interpolation (see below), exhibits less coherence. We believe, however, that due to many vectorizable operations, the acceleration of about 2.5 times is achievable.

4.2. Coherent BDFs

Support for complex material properties is crucial for physically accurate rendering. Most of the objects of everyday occurrence, such as car paint, wood, plastic and clothes exhibit complex optical properties which cannot be explained using Phong model [18] or other simplified material models. In such cases, a more general model of surface scattering needs to be used.

In our framework, we use BDFs based on various physical data. These BDFs can be either measured in a special setting [19] or calculated based on the material microstructure, as for clothes [20]. Tabulating seems to be the only practical way of representing such BDFs. The framework for BDF tabulation and computations is as follows.

The BDF is parameterized using angles describing direction of illumination, observation direction and sample orientation. Depending on the number of the angles used for parameterization, the BDF is said to be 3- or 4-dimensional. The 3D are often referred to as isotropic BDFs and 4D are called anisotropic BDFs.

The BDFs have distinct features and are of high dimension, so they can't be tabulated uniformly for memory space reasons. As they are tabulated in a non-uniform fashion, binary search is to be applied for the computation of the BDF cell in which to interpolate.

The entire algorithm for BDF computation thus proceeds as follows. First, the BDF angles of the rays are calculated. This is done using inverse trigonometric functions. Then binary search is performed to define the BDF interpolation cell. Finally, the value of the BDF is interpolated inside this cell for the given ray directions.

The algorithms have been proposed to implement all of the above mentioned in SSE. Interpolation is rather straightforward to implement in SSE. For inverse trigonometric functions, an approximation has been used. Finally, the binary search algorithm has been modified to handle 4 values simultaneously. The detailed description of our approach is given in [21].

Explicit BDF performance measurements have been performed. The dimensions of the anisotropic BDF were 17 x 7 x 17 x 13. The tests were run on the Intel Centrino notebook with 1800 MHz Mobile Pentium-IV processor and 512 MB of 433 MHz RAM. The timings are given in the table 1.

#calls	100000	200000	400000
non-SSE (sec.)	0.137	0.248	0.495
SSE (sec.)	0.040	0.078	0.156
Acceleration	3.43	3.17	3.17

Table 1. Comparative timings for anisotropic BDF evaluations with and without SSE.

The acceleration achieved is about 3.2 on the average. This is less than 4 due to the fact that evaluation of the tabulated BDF is rather a complicated procedure, which involves lots of branching in binary searches and the like algorithms.

It is also important to note that only one material is processed at a time. If the rays from the same quadruple hit two or more different materials, they are processed in turn, with the rays not hitting the current material being blocked. This allows simplifying the resulting code, as it works with only one material at a time.

5. Light Sources

In order to have the most efficient physically accurate coherent rendering, lighting should also be done using SSE instructions. The term "Lighting" denotes here the process of computing the incoming light intensity at the given point rather than visibility determination. As the visibility determination can be performed efficiently using SSE shadow ray caster, it is lighting computations which need acceleration.

We have several types of light sources in our framework. These can be subdivided into point lights and surface lights. Surface lights are actually processed using Monte-Carlo approach, that is, a number of points is randomly generated on the light source and, based on these points, the intensity of the light source is determined. In determining the intensity, each of these points is treated similarly to a point light source, and the intensity is evaluated using one of the approaches described below.

The other group includes various point light sources. They vary from simple ones, such as omnidirectional or spot light sources, to complex light sources with goniodiagrams.

Figure 5 demonstrates rendering with HDRI lighting in our framework.

For the simple light sources, implementing coherent lighting is rather straightforward, although some issues exist. As in the case of materials, the algorithm works with only one light source at a time. If by some reason (for example, the triangle is back-facing with respect to the light) no lighting computations need be done for some of the rays, they are simply blocked.

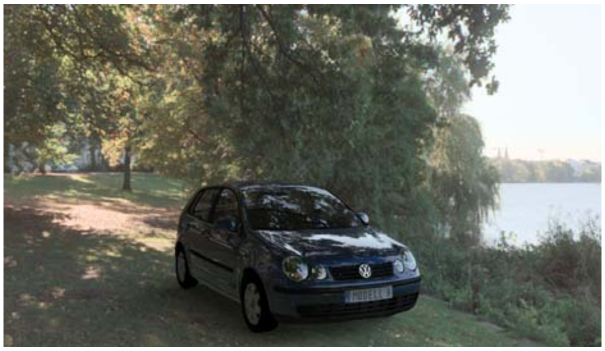

Figure 5. An example of an Inspirer2-rendered image with HDR panorama.

As only simple computations are performed for most of the light sources, the same computations are now performed in SSE for a quadruple of rays. The only light sources needing change are spot light sources. In order to compute the falloff, the cosine of the angle needs be computed. We have found, however, that replacing it with a rough approximation

$$\cos\varphi \approx 1 - \frac{x^2}{2}$$

tends to work well as the falloff itself is important rather than the exact shape of the curve.

The situation is more complicated with point light sources having goniodiagrams. The goniodiagram is a common industrial format to represent the outgoing light intensity of light source in various directions. Its support is crucial in our framework which aims at physically accurate rendering. The intensity of the goniodiagram light source is tabulated in a 2-dimensional non-uniform table, very much like that of the BDF. In order to evaluate it for the specified direction, the following computations have to be performed. First, the spherical coordinates of the ray direction need to be computed. Second, the exact cell the current ray direction belongs to has to be determined. Finally, the interpolation of the light intensity needs to be performed inside the given cell. These steps correspond exactly to what is done for BDF interpolation. In fact, both algorithms share a number of common functions used for both BDF and goniodiagram evaluation.

The performance testing has been done for various kinds of light sources. Both SSE and non-SSE rendering have sufficient accuracy, so images rendered with these two approaches are virtually indistinguishable. SSE approach, however, performs more than 3.5 times faster than a non-SSE one.

The tests were run on the Pentium 4 2.8 GHz computer with 1 GB 433MHz memory. The results are summarized in table 2. All times are given in seconds.

Type of Light	non-SSE	SSE	Acceleration
Omnidirectional	1.137	0.157	7.24
Spot	0.816	0.211	3.87
Parallel	0.444	0.103	4.31
Direct	0.936	0.150	6.24
Linear	6.696	1.149	5.83
Circular	27.936	4.828	5.79
Rectangular	145.252	24.375	5.96

Goniodiagram	2.573	0.588	4.38

Table 2. Comparative timings for non-SSE and SSE lighting for different light sources.

For the linear light source, it has been subdivided into 7 point light sources for rendering. For the rectangular light source, it has been subdivided into 7x3 = 21 light sources for rendering.

As it can bee seen, for most of the light sources the acceleration achieved exceeds 4. Actually, only spot light sources yield less acceleration. For the non-point light sources, the acceleration is, on the average, greater than for point ones, for which it varies greatly. It can also be seen that the goniodiagram light source has less acceleration compared to an omnidirectional one, for example. This is due to the fact that goniodiagrams require more complex algorithm to evaluate.

6. Antialiasing and Tone Mapping

Physically accurate rendering cannot do without tone mapping and antialiasing. While the former is required to map the high dynamic range image obtained during rendering to the limited dynamic range of the monitor, the latter allows to get sufficiently accurate images of scenes with low-size details. Moreover, antialiasing is needed just to get visually pleasing images without jagged borders.

As these two are rather independent procedures, they are discussed separately in the following subsections.

6.1. Tone Mapping

The tone mapping algorithm is performed as follows. First of all, a lower-sized copy of the image with high dynamic range values (that is, with floating-point values) is computed. This copy is used to compute the logarithmic average of the intensity. As the image itself may appear not at precise, it gives the general impression about the light distribution in the final image, and thus, about the logarithmic average of the final image. As the final image is updated iteratively in our antialiasing algorithm (see below), its logarithmic average cannot be used since it changes continuously.

The tone mapping method used is in fact the one described in [22]. The only difficulty with implementing it in SSE is the power function, needed to compute x^y. Since the number of iterations needed to obtain sufficient precision depends greatly on the range, the range is desirable to be reduced. The expression itself can be reformulated:

$$x^y = Max^y(x)\cdot\left(\frac{x}{Max(x)}\right)^y = Max^y(x)\cdot e^{y\ln\left(\frac{x}{Max(x)}\right)}$$

where Max(x) is the maximum value of the color obtained from the low-sized image pre-rendered. As this variable affects only the precision of computations, this is sufficient. The domain of the logarithm is thus reduced to [0, 1] and the domain of the exponent to [-∞, 0]. For the exponent approximation, a hybrid approach is used. In the [-3.8, 0] range, Pade approximation [23] is used

$$e^x = \frac{x^4+20x^3+840x+180x^2+1680}{x^4-20x^3-840x+180x^2+1680}$$

In that range, it gives sufficient precision (about 1%). In [-14, -3.8], however, Pade approximation works poorly, so the table lookup with interpolation is used. A 1024-entry uniform table is sufficient, providing about 1% precision in the entire range. As arguments for the exponent typically do not fall outside the [-14, 0] range in our applications, this approach works well.

A similar hybrid approach is used for logarithm computations. For values greater than 0.17, Pade approximation is used, while for values between 0 and 0.17 the interpolation lookup table is used instead. This was found to provide sufficient precision and is also easy to implement in SSE for both logarithm and exponent computations.

6.2. Antialiasing

Since the coherent physically accurate rendering algorithm is required to work in an interactive setting, it has to exhibit convergence and progressiveness. That is, while the image is still (neither camera nor scene objects move), it must be updated iteratively and the quality has to increase. Alternatively, in an offline setting, the image is rendered progressively and the current image is displayed. When the user is satisfied (or when the precision objective has been reached), the rendering is terminated.

Antialiasing is one of the ways of improving the quality of rendered image. As such, it must possess progressiveness and adaptivity. Our framework also requires precision control, as it is required in many industrial applications. These considerations governed the design of antialiasing algorithm used.

The algorithm is based on the ability to generate a sequence of coherent portions of 4 rays which eventually cover the entire screen with any required density and which can be generated on different levels of the hierarchy. In fact, the algorithm starts with a sparse uniform grid of *superpixels*, with the size of a superpixel being greater than the size of a pixel. The span of the ray quadruple generated is governed by a so-called *coherence radius R*, which depends on the current superpixel (or subpixel) size. When the *R* is decreased, the level of details at which the current rendering takes place is increased.

For the generation of samples in the screen plane, the *2-dimensional Halton sequence* [24] is used with base 2^k along x-dimension and base 3^n along y-dimension. The number s = 2^k3^n is called *span* in our algorithm. Due to the quasi-periodicity of the Halton sequence, the sequence samples with indices j, j + s, j + 2s, j + 3s are located nearly to one another and can thus be traced simultaneously as they are coherent. The span thus defines the number of rays in a single portion. In order not to trace the same rays twice, the j index ranges from 0 to s – 1. The span thus allows us to control the number of rays generated.

Typically, the coherence radius is inversely proportional to the span. More precisely,

$$R = \frac{x_{res}y_{res}}{\pi s}$$

If sample accuracy is not yet sufficient (see below), a ray is traced. For four coherently generated samples, 4 coherent rays are traced simultaneously. Each ray is traced to the end (i.e. the entire ray stack produced by the ray is traced) and the color calculated is returned. The color is then tone mapped and written to the screen matrix.

In order to control accuracy, a simple heuristic is used [25]. There are, in fact, 2 copies of the screen matrix. For each copy, 2 arrays are stored. The first one is the arrays of pixel colors. The second one is the number of samples taken at that pixel. Independently of the current coherence radius and span, the size of the screen matrix is always the same as the resolution of the image.

The resulting pixel color is computed based on both copies of the screen matrix. The difference of the estimates given by these two matrices is used as a measure of accuracy. If for the current coherent group of 4 rays the accuracy is acceptable for all rays, then these rays are not traced.

As the accuracy tends to be unacceptable in those regions where aliasing takes place, this algorithm efficiently deals with antialiasing. Moreover, as the size of initial superpixels can be set to more than one pixel (sizes up to 16 x 24 have been used), in those regions of the image where lighting changes slowly (on walls, for example), interpolation may be used to further reduce rendering time. To be eligible for interpolation, the superpixel has to have the difference between the values at its corners less than the desired accuracy.

It has been found out that an adaptive algorithm, even with antialiasing turned on, can even be faster than classical algorithm due to superpixel interpolation and adaptability, thus providing a reasonable speed – quality tradeoff.

7. Results

The algorithms discussed above have been implemented in C++ language in Visual Studio 2003 development environment. No assembler has been used for the reasons of code maintainability. SSE instructions were accessed via intrinsics, which, in turn, have been wrapped into classes which provide common functionality.

The performance of the ray tracer has been tested on a number of test scenes. The tests have been performed on a dual 933 MHz Pentium III – machine with 1 GB of 133MHz memory. For 1 CPU tests, one of the processors has been disabled. All times are given in seconds. Acceleration gives the ration of time spent by non-SSE renderer in 1 CPU setting to that of an SSE renderer in 1 CPU setting. All images were rendered at 1024 x 768 resolution. The rendering times are given in table 3 and the scene characteristics in table 4. Table 5 compares performance of our approach on single-CPU and dual-CPU machines. As the number of CPUs double, the performance increases approximately 1.9 times.

Scene	non-SSE	SSE (1 CPU)	Acceleration
Car	489.88	82.11	5.97
SPDemo	43.41	2.63	16.51
Glass1	52.66	6.18	8.52
Room2	35.12	4.77	7.36

Table 3. The comparative timings of rendering a 1024 x 768 image with and without using SSE instructions.

Scene	№ trigs	№ lights	depth
Car	233000	4	2
SPDemo	988	3	2
Glass1	44794	2	2
Room2	12000	4	1

Table 4. The characteristics of the scenes used for testing.

Scene	1 CPU	2 CPU	Acceleration
Polo	82.11	41.06	2.00
SPDemo	2.63	1.39	1.89
Glass1	6.18	3.20	1.93
Room2	4.77	2.49	1.91

Table 5. Comparative timings of rendering 1024x768 image with SSE with 1 and 2 CPUs.

The Car test scene contains measured tabulated BRDF, transparent and refractive objects. SPDemo and Glass1 scenes exhibit high reflective complexity. In addition, Glass1 scene is heavily textured. Ray tracing depth 0 corresponds to tracing only camera rays, depth 1 means one level of reflection etc.

8. Discussion

We have presented a physically accurate coherent rendering algorithm which is more than 6 times faster than common ones. The acceleration achieved is mainly due to the use of SSE instructions, which gives a speedup of about a factor of 4. The remaining speedup is due to more careful selection of algorithms and data structures. It is also partly due to more time spent on code optimization.

The resulting rendering times do not seem interactive, although they are rather small. It should be noted, however, that images were rendered at a resolution 1024x768 with 3 light sources. Reducing the resolution to 512x512 will decrease the rendering time roughly 3 times (as it almost linearly depends on the image resolution). Cutting the number of the light sources in the scene will also decrease the rendering time.

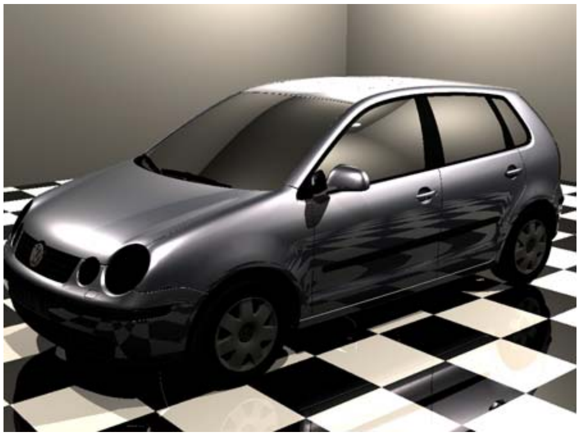

Figure 7. The Car scene rendered using our approach.

Other direction of SSE ray tracing application is acceleration of global illumination calculation. In order to render higher-order indirect illumination the illumination maps technique is used now. As i-maps are calculated by Monte-Carlo ray tracing method they can also benefit from SSE optimizations. However, the rays cast for the illumination map computation, are far less coherent than those cast during ordinary ray tracing. Therefore, coherent algorithms for illumination mapping need to be developed.

Currently, only RGB colors are supported using SSE. It would be interesting, however, to investigate spectral color support. As spectral colors are harder to compute (typically, one spectral color object contains 20 to 40 intensities measured for different wavelengths), this approach does not seem to be interactive. However, it would allow for faster rendering of spectral-based effects, which is required in some areas of industrial rendering. For non-SSE rendering, our framework currently supports spectral BDFs and materials.

The version of the paper with color illustrations can be found at http://www.keldysh.ru/pages/cgraph/publications/cgd_publ.htm

Acknowledgements

This work has been supported by the Russian President "Leading Scientific Schools" grant № RI-112/001/278, RFBR grant № 05-01-00345 and by the Integra Inc. (Tokyo, Japan).

References

[1] IA-32 Intel Architecture Optimization Reference Manual, p. 440 http://www.intel.com/design/pentium4/manuals/24896612.pdf

[2] AMD 3DNow! extensions http://www.amd.com/us-en/Processors/SellAMDProducts/0,,30_177_4458_4513^1413^2137,00.html

[3] PrPMC800: MPC7410 Processor PMC with AltiVec Technology http://www.motorola.com/content/0,,5626,00.html

[4] Ingo Wald, Carsten Benthin, Markus Wagner, Philipp Slusallek: Interactive Rendering with Coherent Ray Tracing. Proc. of *Eurographics* 2001, vol. 20, № 3, pp. 153 – 164.

[5] Virtual Ray Interactive Sphere Ray Tracing Engine, http://www.virtualray.ru/

[6] Andrei Khodulev, Edward Kopylov: Physically Accurate Lighting Simulation in Computer Graphics Software. *Proc. GraphiCon'96: The 6-th International conference on Computer Graphics and Visualization*, St. Petersburg, Russia, July 1-5, 1996. Vol.2, pp.111-119.

[7] Konstantin V. Kolchin, Andrei B. Khodulev: Device-Independent Rendering in Display Color Space. *Proc. Graphicon'98*: *The 8-th International Conference on Computer Graphics and Visualization*, Moscow, Russia, September 7-11, 1998, pp.162-163.

[8] A.G. Voloboi, V. A. Galaktionov, K.A. Dmitriev, and E.A. Kopylov: Bidirectional Ray Tracing for the Integration of Illumination by the Quasi-Monte Carlo Method. *Programming and Computer Software*, Vol. 30, No. 5, 2004, pp. 258-265.

[9] Products Page of Integra Inc. http://www.integra.jp/eng/products/index.htm

[10] A. Ignatenko, B. Barladian, K. Dmitriev, S. Ershov, V. Galaktionov, I. Valiev, A. Voloboy: A Real-Time 3D Rendering System with BRDF Materials and Natural Lighting. *Proc. Graphicon'2004*: *The 14-th International Conference on Computer Graphics and its Applications*, Moscow, Russia, pp. 159-162.

[11] E.Kopylov, A.Khodulev, V.Volevich: The Comparison of Illumination Maps Technique in Computer Graphics Software. *Proc. GraphiCon'98: The 8-th International Conference on Computer Graphics and Visualization*, Moscow, Russia, September 7-11, 1998, pp.146-153.

[12] Intel C++ Compiler Product Page http://www.intel.com/cd/software/products/asmo-na/eng/compilers/220009.htm

[13] Turner Whitted: An Improved Illumination Model for Shaded Display. *Communications of ACM*, Vol. 23, № 6, June 1980, pp. 343-349.

[14] Волобой А.Г., Метод компактного хранения октарного дерева в задаче трассировки лучей. «Программирование», № 1, 1992, стр. 21-27.

[15] Ingo Wald, Carsten Benthin, Philipp Slusallek: OpenRT – A Scalable and Flexible Engine for Interactive 3D Graphics. http://graphics.cs.uni-sb.de/%7Ewald/Publications/2002_OpenRT/2002_OpenRT.pdf

[16] V. Havran: Heuristic Ray Shooting Algorithms. Dissertation Thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, 2000.

[17] Carsten Benthin, Ingo Wald, Philipp Slusallek: A Scalable Approach to Interactive Global Illumination. Proceedings of *Eurographics 2003, Computer Graphics Forum, v.22, №3,* pp. 621 – 630.

[18] B. Phong: "Illumination for Computer Generated Pictures". Communications of *the ACM,* vol. 18, № 6, 1975, pp. 311 – 317.

[19] Letunov A. A., Barladian B. H., Zueva E. Yu., Veshnevetc V. P., Soldatov S. A.: CCD-based device for BDF measurements in computer graphics. *Proc. GraphiCon'99: The 9th International Conference on Computer Graphics and Computer Vision,* Moscow, Russia, 1999, pp. 129-135.

[20] Vladimir Volevich, Andrei Khodulev, Edward Kopylov, Olga Karpenko: An Approach to Cloth Synthesis and Visualization. *Proc. GraphiCon'97: The 7th International Conference on Computer Graphics and Visualization*, Moscow, Russia, 1997, pp. 45-49.

[21] Адинец А.В., Барладян Б.Х., Волобой А.Г., Галактионов В.А., Копылов Э.А., Шапиро Л.З., Когерентная трассировка лучей для сцен, содержащих объекты со сложными светорассеивающими свойствами. Препринт ИПМ им. М.В. Келдыша РАН № 107, 2005.

[22] B. Kh. Barladian, A.G. Voloboi, V. A. Galaktionov, and E.A. Kopylov "An Effective Tone Mapping Operator for High Dynamic Range Images" *Programming and Computer Software*, Vol. 30, No. 5, 2004, pp. 266-272.

[23] An article on Pade approximants in MathWorld online encyclopedia. http://mathworld.wolfram.com/PadeApproximant.html

[24] H. Niederreiter. Random Number Generation and Quasi-Monte Carlo Methods. Chapter 4, SIAM, Pennsylvania, 1992.

[25] V.Volevich, K.Myszkowski, A.Khodulev, E.Kopylov: Using the Visual Differences Predictor to Improve Performance of Progressive Global Illumination Computations. *ACM Transactions on Graphics*, 2000, v.19, № 2, pp.122-161.

About Authors

Andrew V. Adinetz, five course student of the Moscow State University. E-mail: adi_@mail.ru.

Boris H. Barladian, PhD, senior researcher of the Keldysh Institute for Applied Mathematics RAS.
E-mail: obb@gin.keldysh.ru.

Vladimir A. Galaktionov, PhD, head of department of the Keldysh Institute for Applied Mathematics RAS.
E-mail: vlgal@gin.keldysh.ru.

Lev Z. Shapiro, PhD, senior researcher of the Keldysh Institute for Applied Mathematics RAS.

Alexey G. Voloboy, PhD, senior researcher of the Keldysh Institute for Applied Mathematics RAS.
E-mail: voloboy@gin.keldysh.ru.