ISSN 0361-7688, Programming and Computer Software, 2008, Vol. 34, No. 5, pp. 294–303. © Pleiades Publishing, Ltd., 2008.
Original Russian Text © B.Kh. Barladyan, A.G. Voloboi, K.A. Vostryakov, V.A. Galaktionov, L.Z. Shapiro, 2008, published in Programmirovanie, 2008, Vol. 34, No. 5.

The Use of Coherent Ray Tracing for Physically Accurate Rendering

B. Kh. Barladyan, A. G. Voloboi, K. A. Vostryakov, V. A. Galaktionov, and L. Z. Shapiro

Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, Miusskaya pl. 4, Moscow, 125047 Russia
e-mail: voloboy@gin.keldysh.ru

Received November 30, 2007

Abstract—As the power of modern microprocessors increases, the coherent ray tracing becomes increasingly popular in computer graphics because the use of SIMD instructions considerably speeds up this operation. However, after speeding up ray tracing, it turns out that other algorithms for physically accurate rendering, such as the calculation of illumination or application of texture, etc., become a bottleneck in improving the performance. In this paper, a coherent physically accurate rendering algorithm is proposed that makes use of SIMD instructions of modern processors at each stage of the image generation. Coherent algorithms for the calculation of illumination and materials, for antialiasing, and for tone mapping are presented. The comparison of the execution time of coherent and incoherent algorithms using benchmark scenes showed that the former are considerably faster.

DOI: 10.1134/S036176880805006X

1. INTRODUCTION

Coherent ray tracing (that is, tracing several coherent rays simultaneously by the same processor) has recently become a subject of research. Modern microprocessors support SIMD (Single Instruction Multiple Data) extensions, which make it possible to perform arithmetic and logic operations simultaneously on several floating-point numbers. Different processors have different SIMD extensions; for example, these are Intel SSE (Streaming SIMD Extension) [1], AMD 3DNow! [2], and Motorola AltiVec [3]. Among all these extensions, Intel SSE is most widely used in coherent ray tracing. Since recently, AMD processors also support SSE, and Apple computers now also use Intel processors. Thus, SSE has become a de facto standard for SIMD extensions for personal computers. For that reason, the terms coherent, SIMD, and SSE ray tracing will be used as synonyms in this paper.

Presently, there are several projects using SSE ray tracing. The most well known one is the project developed in the Max Planck Institute [4] that uses SSE for coherent ray tracing. Among the other projects is Manta, which is the interactive ray tracer with the open source code created for rendering huge models on supercomputers with shared memory and multicore workstations [5].

The group of researchers working in the Computer Graphics Department of the Keldysh Institute of Applied Mathematics (Russian Academy of Sciences) has rich experience in developing physically accurate rendering systems [6–8]. This group developed many applications that support various aspects of photorealistic rendering. The scientific results of these studies were used in a commercial product (see [9]). The use of SSE, which is now widespread, makes image generation considerably faster; for some scenes, it provides interactive rendering. The computational effort involved in ray tracing constitute a considerable part of the total effort of generating photorealistic images, but it does not cover all the computational cost. Rendering also includes simulation of light dispersing properties of surfaces and of physically accurate illumination created by various light sources. To obtain a high-quality image, the aliasing effect must be eliminated and tone mapping algorithms must be used to transform the radiometric magnitudes used to perform the simulation to the color values shown by a display.

In this paper, we present an approach to using the SSE instructions for speeding up the computations by a factor of three or four times at all the rendering stages—illumination calculation, treating complex materials and light sources, transformation of physical magnitudes to display colors, etc.

The paper is organized as follows. In Section 2, we briefly describe the architecture and design of the Inspirer2 rendering system, which was used as a basis for coherent rendering. This section also outlines the main parts of the system for coherent ray tracing. Section 3 is devoted to coherent ray tracing. In Sections 4 and 5, we discuss the coherent treatment of complex surface materials defined by the bidirectional reflection distribution function (BRDF) in the general case, and describe the set of types of light sources for coherent treatment. Section 6 deals with the tone mapping operator. In Section 7, we describe an adaptive algorithm for eliminating the aliasing effect in the four-ray SSE trac-

Fig. 1. Image of a car generated using the image-based lighting technique.

ing. In Sections 8 and 9, we present estimates of the system performance for certain scenes, and discuss conclusions and the trends for future research.

2. ARCHITECTURE OF THE BASE RENDERING SYSTEM

We implemented physically accurate rendering using coherent ray tracing on the basis of the rendering system Inspirer2 (earlier called Fly) [10].

This system supports both the interactive rendering and the generation of high-quality images in the noninteractive mode. The interactive mode was implemented using OpenGL. In this mode, the system can visualize various scenes at a rate close to real time (up to 20–60 images per second). The main goal of the system development was to give as high level of physical accuracy as is possible at such an image generation rate. In the interactive mode, the system is able to generate physically accurate shadows from point light sources, surface materials specified by their bidirectional reflection distribution function (BRDF), and approximate reflection using environment maps.

In the high-quality image generation mode, the system provides for illumination modeling and physically accurate rendering by using bidirectional ray tracing. In this mode, the point, linear, and surface light sources described by goniograms can be correctly modeled, which enables one to define almost any realistic illumination. We developed capabilities for specifying a natural illumination directly using the geographic location and time parameters or calculating the illumination defined by a high dynamic range (HDR) panorama [11]. An example of a scene image illuminated by a HDR panorama is shown in Fig. 1. Materials with complex properties may also be specified by a BRDF in the most common tabular form. The tabular specification of the BRDF enables one to use the data measured using a spectrophotometer [12]. To take into account the specular reflection and refraction, backward ray tracing is used. To calculate the global illumination, the forward ray-tracing algorithm based on the Monte Carlo method is used. The resulting global illumination data are stored in illumination maps [13] and are used in both rendering modes.

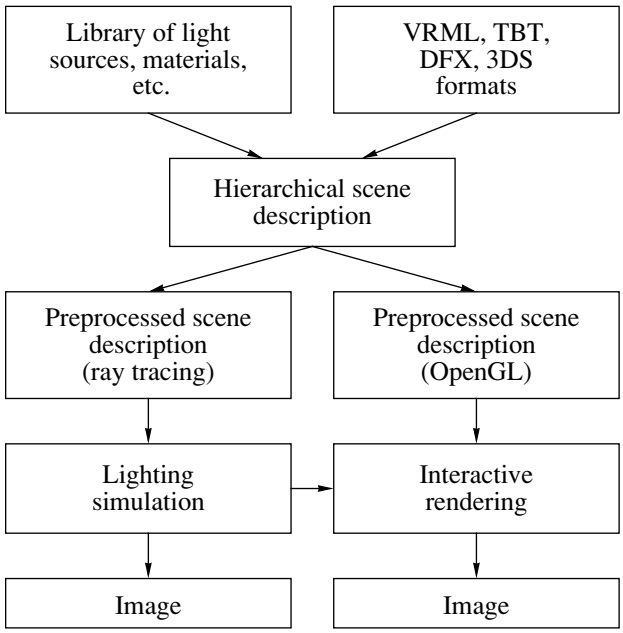

Fig. 2. Architecture of Inspirer2: A hierarchical description of the scene physical attributes is transformed into two preprocessed scene descriptions (one of them for OpenGL rendering, and the other for Monte Carlo ray tracing).

The use of SSE ray tracing provides benefits both for the interactive and for the noninteractive modes. The high-quality noninteractive rendering can be accelerated by a factor of two or three because the SSE instructions make it possible to trace 4 coherent rays simultaneously. In the interactive mode, SSE ray tracing may be used in the framework of a hybrid approach for superimposing physically accurate reflections and refractions on an OpenGL image.

The general architecture of Inspirer2 is presented in Fig. 2. On the one hand, it satisfies the requirement that a scene must be specified using physical quantities; on the other hand, it can use different internal representations for different rendering modes. Such an architecture facilitates integration of a new coherent tracing algorithm by adding a new internal representation.

The coherent ray tracing, as well as the other components of the system, was implemented in C++; Assembler was not used. In order to use SSE instructions, wrapper classes over SSE intrinsic functions (special functions that are translated by a compiler into an SSE code) that are supported by Microsoft/Intel C++ compilers. Since modern compilers can efficiently optimize a high-level code, this does not result in a significant loss of performance; at the same time, the code is easier to maintain.

3. COHERENT RAY TRACING

Ray tracing is usually considered to be the most time consuming part of any physically accurate rendering algorithm. In [14], the time needed for ray tracing is estimated as 95% of the total rendering time. However, according to our estimates, the relative tracing time for physically accurate modeling is only about 65–75%. This makes ray tracing the primary candidate for SSE optimization.

SSE instructions are performed on four 32-bit floating-point numbers simultaneously. Therefore, SSE ray tracing makes it possible to trace four rays simultaneously. The algorithm does not differ significantly from the corresponding one-ray tracing algorithm. To accelerate tracing, space is represented in the form of a BSP tree (binary space partition). Thus, the ray-tracing algorithm consists of the tree traversal phase and the phase of finding the intersections of the ray with the scene objects belonging to a selected subspace. To support interactive of scene objects, two-level tracing [16] is used. Under this approach, each object (represented by a triangular mesh) has a specific BSP tree (which is called the second-level BSP tree or the BSP tree of the object) and a specific bounding box (an axes aligned rectangular parallelepiped). Every object is placed into the scene together with its transformation matrix. The scene consists of a set of objects framed into boxes. The BSP tree of the scene (which is called the first-level BSP tree below) arranges all those complex scene objects in space. Since only the bounding boxes of objects are used in the first-level BSP tree, the partition is not always as efficient as in the case when the BSP tree is constructed using the triangular mesh of the entire scene; this is the cost of the possibility to have moving objects in the scene. In very sparse scenes, such a two-level tree can even slightly improve the performance because the empty space between the objects is handled more efficiently.

Due to the use of SSE, up to four rays can be traced simultaneously; however, they can take different paths in the tree in the ray-tracing algorithm. This fact implies that some rays of these four must be temporarily blocked. For that purpose, we use the active ray mask. Usually, the mask is an SSE variable (four 32-bit floating point numbers) containing either 0×00000000 or $0 \times \mathrm{ffffffff}$ (in binary representation) in each of the four positions. The mask blocks the rays that do not pass through the current branch of the BSP tree and the rays for which the first intersection has already been found. Masking is a common technique in SSE programming; it reduces branching and makes the algorithm more streaming. In the project described in this paper, masking is used in all the components of the rendering algorithm.

The coherent ray-tracing algorithm proceeds as follows. First, all the rays are checked for the intersection with the scene bounding box. If all the rays miss this box, the algorithm immediately reports that there are no intersections. If some of the rays intersect the scene box, the ray mask is updated, and the rays that do not intersect the scene box are removed. If two rays have different signs in the direction vector, they can have different traversal order. In this case, the group of rays is divided into subgroups with the same direction sign (coherent subgroups). This slightly reduces the efficiency of SSE tracing; however, such cases are rare. Moreover, it can be shown that the rays that have a common origin, for example, the rays emanating from a camera or shadow rays from a point light source, always have the same traversal order.

After dividing the rays into subgroups, the algorithm sets the mask for the current rays and starts traversing the BSP tree. Since we have a two-level hierarchy, the procedure used for the entire scene is repeated for each object that is tested for intersection. For every non-leaf node, the BSP traversal is performed as follows. If all the rays go to the same subnode (to the right or to the left one), the algorithm updates the current node address and proceeds to this subnode. If some of the rays go through both subnodes, then the farthest subnode is pushed into the stack, the mask of active rays is updated, and the algorithm goes to the nearest subnode. Note that, due to dividing the rays into groups (see above), the rays cannot traverse nodes in a different order. However, it is possible that some rays have to traverse both subnodes although actually they pass only through one of them. In this case, such rays are blocked by the mask; they are activated again after the node has been traversed. When a leaf node is reached, the ray intersects the corresponding object. For the first-level tree, the objects are actually scene objects; therefore, the ray is transformed to the object's system of coordinates, and the algorithm proceeds in much the same way as for the scene as a whole. For the second-level tree, objects are actually triangles.

To find the intersection of a ray with a triangle, the modified projection barycentric algorithm (test) is used that is implemented using SSE as described in [4]. After checking all the objects in a node, the rays for which no intersection is found are deactivated because they do not need to be further traced in the tree. If all the objects in a first-level leaf node have been tested and some intersections have been found, they are considered the *first intersections* of those rays.

Actually, the proposed scheme uses two intersection methods. The first one, which finds only the first ray intersections, was just described. The other one finds *all the intersections* up to the first opaque object. The second method works like the first one; however, if a ray hits a transparent object, it is not masked but is traced further until it hits the first opaque object.

The arrangement of the scene tree and the triangle data in memory is optimized with regard for processor cache. Both child nodes are stored one after the other in the same cache line, which reduces data load time from the main memory.

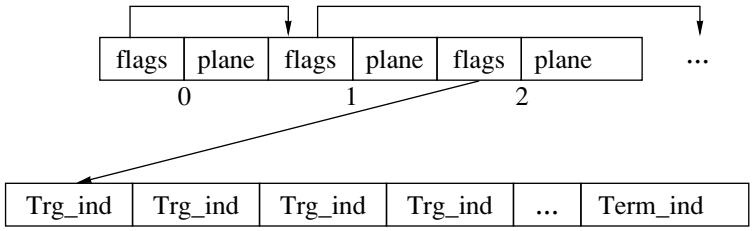

Fig. 3. Arrangement of the scene tree and triangle indexes in memory.

The BSP tree is constructed using the algorithm described in [17]. Since the construction of the BSP tree is a time-consuming task, a two-level approach is used that makes it possible to modify the tree as an object location changes rather than to construct it from scratch. As a result, dynamic scenes can be treated interactively. However, as it has already been mentioned above, the BSP trees thus constructed are slightly less efficient than the simple one-level BSP trees.

Figure 4 shows a scene rendered using SSE ray tracing.

4. OPTICAL PROPERTIES OF MATERIALS AND BRDF

Since the coherent ray tracing improves the performance of ray tracing by a factor of two to three compared with conventional ray tracing algorithms, the other parts of the physically accurate rendering algorithm become a bottleneck for the total performance. We have already mentioned that the time spent on ray tracing is about 70% of the total rendering time. Therefore, the total image generation time decreases less than by a factor of two. We conclude that the other parts of the rendering algorithm should be also accelerated.

Fig. 4. A scene rendered using SSE ray tracing.

Other developers of the coherent ray tracing faced the same problem. For example, in [18], the authors note that after the SSE tracing has been implemented, shading became the bottleneck. It was found that even the simple Phong model can considerably slow down the rendering let alone the use of more complicated BRDFs.

In the framework of the project described in this paper, the main goal was to generate physically accurate images rather than to create visually plausible effects. Therefore, we had to implement coherent treatment of materials and BRDFs.

In order to accurately model the optical properties of materials, our implementation uses the following components to describe materials:

1. a simple set of the material's attributes represented by the Phong model;

2. refraction and reflection-related attributes;

3. textures;

4. the material's BRDF.

The first two components can be implemented straightforwardly using SSE; indeed, they only involve simple vector and color operations. Illumination computations require tracing shadowed, reflected, and refracted rays, which can easily be implemented using the efficient SSE ray tracing described above.

However, texturing and BRDF support are not easy to implement using SSE. These issues are discussed in the following subsections.

4.1. Coherent Texturing

Texture (two-dimensional color matrix) is imposed on the surface of a scene object to change its appearance. An example of using texture is illustrated in Fig. 5.

In the framework of our system, textures are assigned to materials, and materials are assigned to the triangles constituting objects. The texture mapping is performed using texture coordinates specified for each triangle vertex with a textured material assigned to it. To compute the texture coordinates at the points where the quadruple of rays hits a triangle, the barycentric coordinates obtained while finding the intersection of rays with the triangle are used. Using the barycentric coordinates, the texture coordinates can be interpo-

Fig. 5. A scene with textures rendered using the proposed coherent algorithm.

Fig. 6. Rendering of an optically complex material described by an BRDF using the proposed approach.

lated. This can easily be done using SSE for four points simultaneously if they belong to the same triangle. If the points belong to different triangles, interpolation is performed using several passes with masking inactive points.

Then, the texture values are interpolated using trilinear filtering. Computation of a mipmap level (a prefiltered image with various filter sizes multiple of two) is more complicated because it involves the base 2 logarithm of the distance from the point to the camera. Since we need only the nearest integer value of the logarithm, we may use its binary representation (fraction and exponent). The logarithm of a floating-point number can be represented by the sum of the logarithm of the fraction and the exponent, which can be obtained using bitwise operations. The fraction is in the range from 0.5 to 1, and its logarithm can easily be approximated by a polynomial. The accuracy of such an approximation depends only on the accuracy of the polynomial on this interval, where the logarithm has no singularities.

The mipmap level is chosen in such a way that the adjacent rays almost always hit different pixels; therefore, the data coherence, which is necessary for the efficiency of SSE, is lost. However, we can use SIMD instructions for the bilinear interpolation of the RGBA components, where the data are always coherent. Hence, we perform bilinear interpolation at the two nearest mipmap levels, and then perform linear interpolation between them.

4.2. Coherent BRDF Implementation

A support of complex material properties is crucial for physically accurate rendering. Most objects of everyday occurrence, such as car paint, wood, plastic and fabrics exhibit complex optical properties that cannot be described using simple heuristic models, such as the Phong model [18]. In such cases, a more general surface scattering model must be used.

In the project discussed in this paper, we use BRDFs based on various physical data. These BRDFs can be either measured using special equipment [19] or simulated. A tabular representation seems to be the only practical way of representing such BRDFs.

BRDFs can be parameterized using an angle description of the directions of illumination and observation. Depending on the number of the angles used for the parameterization, BRDFs can be 3- or 4-dimensional. 3-dimensional BRDFs are often said to be isotropic and 4-dimensional ones are said to be anisotropic.

Since BRDFs can be singular and can be multidimensional, they cannot be tabulated in a regular fashion due to high memory requirements. For that reason, we use binary search to find the cell in which interpolation will be performed.

Let us now describe the BRDF computation algorithm. First, the incident and observation angles of the rays are calculated using the inverse trigonometric functions. Then, binary search is performed to define the interpolation cell. Finally, the value of the BRDF is found using the interpolation inside this cell for the given ray directions.

In order to efficiently implement this algorithm using SSE, we developed algorithms for the coherent binary search of four values simultaneously and for the approximation of inverse trigonometric functions. Interpolation within a cell is straightforwardly implemented using SSE. A detailed description of the proposed approach can be found in [20]. Figure 6 shows an example of the SSE rendering of a scene containing a material described by a BRDF.

Table 1 presents the results of comparing the SSE implementation of the BRDF with an implementation that does not employ SSE for an anisotropic BRDF of the size $17 \times 7 \times 17 \times 13$. The comparison was performed on an Intel Centrino 1800MHz Mobile Pentium-IV computer with 512 Mb of 433 MHz memory.

The acceleration was about 3.2 times on the average, which is less than 4 because the computation of BRDFs include a large number of branchings thus reducing the efficiency of SSE. It is important that only one material

can be processed at the same time. If a quadruple of rays hit different materials, then they are processed one after another. The rays that are irrelevant for a particular computation are masked.

5. LIGHT SOURCES

In order for the physically accurate coherent rendering to be efficient, illumination computations should be implemented using SSE. The term *illumination computations* here denotes the computation of the incident light intensity at the given point without regard for the visibility of the light source. The computation of visibility can be performed efficiently using SSE tracing for shadow rays.

We considered several types of light sources that can be subdivided into point and surface sources. In order to compute the illumination created by surface light sources, certain points on their surface must be generated, the intensity of light created by each of those points must be determined, and then integrated over the surface. Surface light sources produce natural soft shadows (half-shadows).

The other group includes various point light sources. They vary from simple ones, such as omnidirectional or spotlight sources, to complex light sources described by their goniograms. For simple light sources, the coherent illumination computation is implemented straightforwardly. As for materials, the algorithm treats a single light source at a time. If the illumination for a ray need not be calculated for a certain reason (for example, the light source is on the other side of the triangle being processes), the ray is masked.

For most light sources, fairly simple computations are performed; with SSE, the same computations are performed on quadruples of rays. The situation is more complicated with point light sources described by their goniograms. A goniogram is an industrial format for representing the emitted intensity of light sources. The support of such light sources is crucial for physically accurate rendering. The intensity of a light source described by a goniogram is tabulated in a two-dimensional nonuniform table, and, in that respect, it is very similar to the BRDF. In order to evaluate the intensity for the specified direction, its spherical coordinates must be first found. Then, the cell containing the given direction must be determined. Finally, the intensity must be interpolated inside this cell. These steps are the same as those used for the computation of BRDFs. Actually, both algorithms share a number of functions.

The performance of the proposed coherent implementation was measured for various types of light sources. Both implementations (using SSE and without SSE) are very accurate so that the images produced by them are almost identical. However, the SSE implementation is by a factor 3.5 faster.

Table 2 summarizes the efficiency of the SSE implementation for several types of light sources. The comparison was performed on a 2.8 GHz Pentium-IV computer with 433 MHz memory of 1 Gb. The results are given in seconds.

Table 1. Comparison of the performance of the SSE implementation for an anisotropic BRDF with the implementation that does not employ SSE

Number of calls	100000	200000	400000
Without SSE (s)	0.137	0.248	0.495
With SSE (s)	0.040	0.078	0.156
Acceleration	3.43	3.17	3.17

Table 2. Comparison of the performance of the SSE implementation of light sources with the implementation that does not employ SSE

Type of light source	Without SSE	SSE	Acceleration
Omnidirectional	1.137	0.157	7.24
Spot	0.816	0.211	3.87
Parallel	0.444	0.103	4.31
Directed	0.936	0.150	6.24
Linear	6.696	1.149	5.83
Circular	27.936	4.828	5.79
Rectangular	145.25	24.375	5.96
With a goniogram	2.573	0.588	4.38

It is seen that the acceleration was more than fourfold for most light sources. The acceleration was less than four times only for the spotlight. The source described by its goniogram also had a less speedup than, e.g., the omnidirectional source because the goniogram requires binary search, which causes incoherence of the data.

6. TONE MAPPING OPERATOR

Physically accurate rendering is impossible without tone mapping. This is because the images obtained as a result of modeling are represented in terms of units in the range $[0, \infty)$ while display devices have limited dynamic ranges. Hence, we face the problem of constructing a contracting mapping preserving important details.

The tone mapping algorithm is as follows. First, a low-resolution copy of the image with a high dynamic range is produced. This copy is used to compute the logarithmic average of the intensity, which yields a general impression about the intensity distribution in the final image.

We used the method described in [21]. Here, we briefly describe the approach to its SSE implementation, while a detailed description of the algorithm can be found in [21].

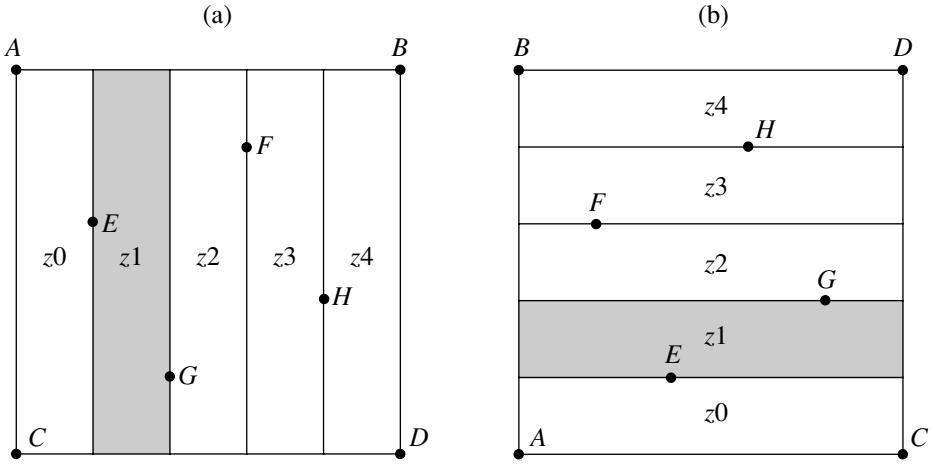

Fig. 7. Geometric discontinuities in image pixels: (horizontal discontinuity in the region $z1$ between the points E and G; (b) the vertical version rotated by 90° counterclockwise.

The main difficulty in the implementation of this algorithm is the evaluation of the power function x^y, which can be represented in terms of the logarithm and the exponential function:

$$x^y = 2^{y\log_2 x}.$$

An efficient SSE implementation of the logarithm was described in Section 4.1. A similar idea is also used for evaluating the exponential function. First, it is reduced to base 2, and then represented in the form $2^x = 2^{\lfloor x\rfloor}2^{x-\lfloor x\rfloor}$, where $\lfloor x\rfloor$ is the integral part of x. Here, the first factor is computed using the binary shift, and the second one is approximated by a polynomial. A significant acceleration of the evaluation of the power, logarithmic, and exponential functions can be achieved only by using SSE2 instructions, which are able to perform operations on four integer numbers simultaneously while keeping them in the SSE registers.

7. ANTIALIASING ALGORITHM FOR TRACING FOUR RAYS SIMULTANEOUSLY

In this paper, we present only the main ideas of the new adaptive antialiasing algorithm based on SSE tracing of four rays simultaneously. A more detailed description of this algorithm can be found in [22].

Each time when a region with a sharp variation of the image intensity is found, it is desired to trace four new rays rather than trace these rays one-by-one. The proposed algorithm uses the SSE mask of the difference of colors of close rays as an index in the discontinuity table. This helps determine the regions of the sharp change of image intensity very quickly. The proposed algorithm makes it possible to trace 1.5–2 rays per pixel on the average with the quality similar to that obtained by tracing 25 rays per pixel.

The screen is subdivided into square regions (tiles) of the size 64 × 64 pixels. First, rasterization is performed for the current tile using a graphic processor. This can save us tracing many additional rays that could be needed for determining geometric discontinuities. Thus, the visibility map is constructed represented in the form of a matrix of the triangles' indices that are visible from the camera through the pixels. The resolution of this matrix is several times higher that the tile size (by a factor of four for ordinary quality and by a factor of six for high quality). Next, the algorithm checks, for each pixel, if it contains a discontinuity; that is, the algorithm checks if the pixel contains values of the visibility map corresponding to objects with very different normals or distances from the camera. This yields a discontinuity matrix of the size equal to the tile size. At this stage, we only determine the pixels containing geometric discontinuities but do not find the location of the discontinuity within the pixel.

The proposed adaptive algorithm has three levels. The next level is applied when a discontinuity at the preceding level is detected.

At the first level, a regular sample is used. The rays are traced in quadruples through the pixels' vertices. The color values for each ray are stored in a matrix. Then, the next pixel in the tile is chosen, and the color values corresponding to its vertex rays are extracted from this matrix (the points $ABCD$ in Fig. 7a). The algorithm determines the type of the discontinuity (horizontal or vertical). A horizontal discontinuity is detected if there is a high gradient between the points A and B or C and D. If the detection is ambiguous, i.e., if there is a great variation between the points A and B and between the points B and D, then the algorithm assumes that there is a horizontal discontinuity only. If a discontinuity is detected in the visibility map and the differences between the vertex values are small, we also assume that there is a horizontal discontinuity.

If the algorithm detects a vertical discontinuity, the points $ABCD$ are rotated by 90° counterclockwise.

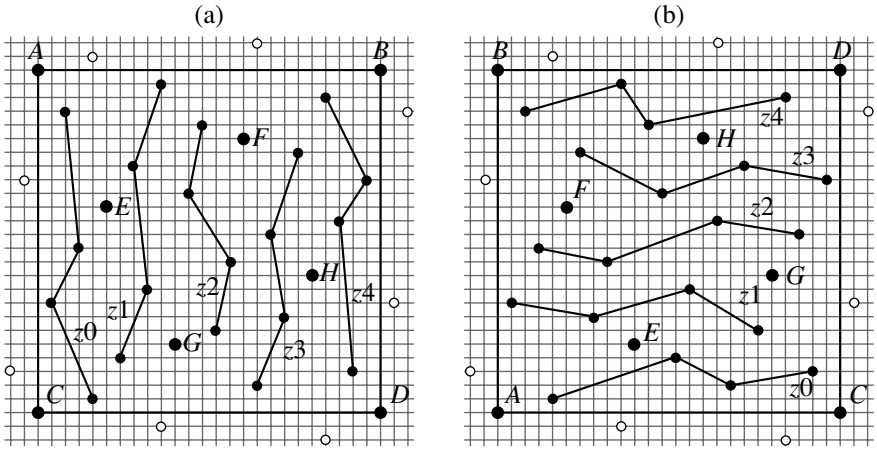

Fig. 8. The points used at the third level of the discontinuity detection algorithm: (a) horizontal version; (b) vertical version.

Such a rotation makes the algorithm linear independently of the discontinuity type (see Fig. 7b).

If discontinuity is detected neither in the visibility map nor by comparing the vertex values, then and only then the algorithm sets the color of the pixel to the average of its vertex values and proceeds to the next pixel. Otherwise, the second level of the algorithm is applied to this pixel.

Inside this pixel, we shoot four rays through the points E, F, G, and H as shown in Fig. 7. To determine discontinuities between the points obtained at the first and at the second levels of the algorithm, we perform three quads of comparisons between the R, G, and B color components using SSE instructions. Each SSE comparison produces a 4-bit mask that is used as an index in the discontinuity table (this table is created in advance). The entries in this table contain flags defining the discontinuity regions $z0$, $z1$, $z2$, $z3$, and $z4$ (Fig. 7). The values obtained from the three tables (three quads of comparisons) are logically added (disjunction). The resulting flags indicate the discontinuity regions in which additional rays should be shot.

At the third level, four rays are shot in a zigzag fashion (see Fig. 8). Each zigzag covers one of the discontinuity regions. At most, five zigzags per pixel are shot. Note that the points in each zigzag are spaced at 1/20 of the pixel side apart along the high gradient of the intensity (see Fig. 8). In other words, in the case of a large horizontal (for example) discontinuity, each of the vertical lines spaced at 1/20 of the pixel side apart contains one zigzag point. Moreover, independently of the case (horizontal or vertical discontinuity), the closest points in the adjacent pixels are located at the same places. This property ensures a relative uniformity of the point arrangement even in pixels with different orientations.

A zigzag can be traced or interpolated using the values of the sample obtained at the preceding level of the algorithm. The decision on what operation to perform (tracing or interpolation) is made on the basis of the flags obtained from the discontinuity tables. Interpolation is performed by the points belonging to the current discontinuity region because each zigzag is processed independently of the others. For example, the points of the zigzag $z2$ are interpolated using the points F and G only, while E and H are not used for the interpolation because the regions $z1$ and $z3$ can contain discontinuities. Each point of the zigzag $z0$ is interpolated by two of the three points A, C, and E: the upper point (see Fig. 8a) is interpolated by the points A and E; and the three other points by E and C. The zigzag $z1$ is interpolated by the points E and G; the zigzag $z2$ by G and F; the zigzag $z3$ by H and F; and the points of $z4$ are interpolated by two of the three points H, B, and D.

The location of all the points is prescribed in advance; therefore, in order to calculate the color of a particular pixel, the weight of each of these points in the sum used to obtain the color of the pixel can be determined in advance. As a result, the algorithm becomes much faster. If a rectangular filter is used (a common situation), then the color of every pixel can be immediately placed in the frame buffer upon processing—there is no need to store information about subpixel colors.

Comparisons show that, for typical scenes, the proposed adaptive algorithm with 1.5–2 rays per pixel gives the quality comparable with that given by 25 rays per pixel in the regular sample.

8. RESULTS

The algorithms described above were implemented in C++ using the Visual Studio 2003 development environment. No assembler was used for SSE instructions. SSE instructions were invoked through intrinsics, which, in turn, were wrapped by special classes. Such an approach provides good code maintainability at the cost of a minor loss of performance.

Table 3 presents the results of the comparison of the rendering rate with SSE and without SSE for three scenes Car, Glass, and Room shown in Figs. 6, 5, and 4,

Table 3. Comparison of rendering with SSE and without SSE for 1024 ± 768 images

Scene	Without SSE	SSE	Acceleration
Car	9.0	2.5	3.6
Glass	15.6	6.0	2.6
Room	21.6	6.1	3.5

Table 4

Scene	Number of triangles	Number of light sources	Tracing depth
Car	141512	3	3
Glass	44928	2	2
Room	11788	9	2

respectively. The experiments were performed on a dual 2.13 Athlon MP computer with memory of 2 Gb. Only one processor was used (the other was disabled). Images of the size 1024 × 768 pixels were generated. The characteristics of the rendered scenes are given in Table 4, where the tracing depth denotes the maximum height of the ray tree; that is, the depth 0 corresponds to rays from the camera, the depth 1 corresponds to one ray reflection, etc.

The scene Car is descriped by a measured tabulated BRDF, transparent and refracting objects. The scene Glass contains many textured surfaces.

9. CONCLUSIONS

In this paper, physically accurate coherent rendering algorithms are presented that accelerate the image generation rate more than by a factor of 3. The acceleration is achieved mainly due to the use of SSE instructions, which can give a speedup up to four times in the case of complete coherence. A careful selection of algorithms and data structures, as well as considerable effort aimed at code optimization, also contributed to the speedup.

The resulting rendering rate is not an interactive one but is close to it. If the image resolution is reduced and the number of light sources is limited, an interactive rendering rate can be achieved.

Another direction of application of the SSE ray tracing is the computation of global illumination. To render the indirect illumination, we currently use illumination maps (see [13]). To compute the illumination maps, Monte Carlo ray tracing is used. However, the rays in this method are much less coherent; therefore, the direct use of SSE is complicated.

Currently, only the RGB color model is used in the SSE implementation. It would be interesting to try the spectral color representation as well. Usually, a spectral representation involves 20–40 light intensities measured for various wavelengths; therefore, such an approach is expected to be far from interactive. However, it can accelerate the rendering of various spectral effects required in industrial visualization. The implementation of Inspirer2 without SSE already supports spectral BRDFs and optical material properties. Calculations for several dozens of spectral values can fit into the SIMD scheme, which will give an acceleration of up to a factor of four.

A version of this paper with color illustrations is available at http://www.keldysh.ru/pages/cgraph/publications/cgd_publ.htm.

ACKNOWLEDGMENTS

This work was supported by the Russian Foundation for Basic Research, project no. 05-01-00345, and by INTEGRA Inc. (Japan).

REFERENCES

1. IA-32 Intel Architecture Optimization Reference Manual, 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. Wald, I., Benthin, C., Wagner, M., and Slusallek, Ph., Interactive Rendering with Coherent Ray Tracing, *Proc. of Eurographics, 2001*, Vol. 20, no. 3, pp. 153–164.
5. Stephens, A., Boulos, S., Bigler, J., Wald, I., and Parker, S., An Application of Scalable Massive Model Interaction Using Shared-Memory Systems, *Eurographics Symposium on Parallel Graphics and Visualization*, 2006.
6. Khodulev, A. and Kopylov, E. Physically Accurate Lighting Simulation in Computer Graphics Software, in *Proc. 6th Int. Conf. on Computer Graphics and Visualization GraphiCon'96*, St. Petersburg, 1996, Vol. 2, pp. 111–119.
7. Bayakovsky, Yu. and Galaktionov, V., On Some Fundamental Problems in Computer Graphics, *Inf. Teknol. Vychisl. Sist.*, 2004, no. 4, pp. 3–24.
8. Voloboi, A.G., Galaktionov, V.A., Dmitriev, K.A., and Kopylov, E.A., Bidirectional Ray Tracing for the Integration of Illumination by the Quasi-Monte Carlo Method, *Programmirovanie*, 2004, no. 5, pp. 25–34.
9. Voloboi, A.G. and Galaktionov, V.A., Computer Graphics in Automated Design, *Inf. Techn. Design Manufacturing*, 2006, no. 1, pp. 64-73.
10. Ignatenko, A., Barladian, B., Dmitriev, K., Ershov, S., Galaktionov, V., Valiev, I., and Voloboy, A. A Real-Time 3D Rendering System with BRDF Materials and Natural Lighting, in *Proc. 14th Int. Conf. on Computer Graphics and Vision, GraphiCon-2004*, Moscow, 2004, pp. 159–162.
11. Voloboi, A.G., Galaktionov, V.A., Kopylov, E.A., and Shapiro, L.Z., Simulation of Natural Daylight Illumina-

tion Determined by a High Dynamic Range Image, *Programmirovanie,* 2006, no. 5, pp. 62–80.

12. Voloboi, A.G., Galaktionov, V.A., Ershov, S.V., Letunov, A.A., and Potemin, I.S., Hybrid Hardware/Software System for Measuring Light-Dispersion Properties of Surfaces, *"Inf. Tekhnol. v Proektirovanii i Proizvodstve"*, No. 4, pp. 24–39.
13. Kopylov, E., Khodulev, A. and Volevich, V., The Comparison of Illumination Map Techniques in Computer Graphics Software, in *Proc. of 8th Int. Conf. on Computer Graphics and Visualization*, Moscow, 1998, pp. 146–153.
14. Whitted, T., An Improved Illumination Model for Shaded Display, *Commun. ACM*, 1980, vol. 23, no. 6, pp. 343–349.
15. Voloboi, A.G., A Method for Compact Storage of Octal Tree in the Ray Tracing Problem, *Programmirovanie,* 1992, no. 1, pp. 21–27.
16. Wald, I., Benthin, C., and Slusallek, Ph., A Scalable and Flexible Engine for Interactive 3D Graphics, *Technical Report of Computer Graphics Group, Saarland Univ.*, 2002, TR-2002-1; http://graphics.cs.uni-sb.de/%7Ewald /Publications/2002_OpenRT/2002_OpenRT.pdf.
17. Havran, V., Heuristic Ray Shooting Algorithms, *Dissertation Thesis,* Prague: Czech Technical Univ., 2000.
18. Benthin, C., Wald, I., and Slusallek, Ph., Scalable Approach to Interactive Global Illumination, in *Proc. Of Eurographics 2003*, "Comput. Graph. Forum", vol. 22, no. 3, pp. 621–630.
19. Phong, B., Illumination for Computer Generated Pictures, *Commun. ACM*, 1975, vol. 18, no. 6, pp. 311–317.
20. Adinets, A.V., Barladian, B.,Kh., Voloboi, A.G., Galaktionov, V.A., Kopylov, E.A., and Shapiro, L.Z., Coherent Ray Tracing for Scenes Containing Objects with Complex Light-Dispersing Properties, *Preprint of Keldysh Inst. of Applied Mathematics, Russ. Acad. Sci.*, Moscow, 2005, 107.
21. Barladian, B.Kh., Voloboi, A.G., Galaktionov, V.A., and Kopylov, E.A., An Effective Tone Mapping Operator for High Dynamic Range Images, *Programmirovanie*, 2004, no. 5, pp. 35–42.
22. Vostryakov, K.A. and Voloboi, A.G., Antialiasing Algo- 1 rithm for the Four-Ray SSE Ray Tracing, in *Proc. Of the 17th Int. Conf. on Computer Graphics and Computer Vision, GraphiCon'2007*, Moscow, 2007.

SPELL: 1. antialiasing, 2. goniogram