Coherent Ray Tracing of Complex BRDF Objects

A.V. Adinetz[1], B.H. Barladian[2], L.S. Shapiro[2], A.G. Voloboy[2]

[1]Moscow State University, Faculty of Computational Mathematics and Cybernetics

[2]Keldysh Institute of Applied Mathematics, Russian Academy of Sciences

Abstract

An approach for providing BRDF support for coherent ray tracing is described. Unlike earlier approaches, this approach is able to handle real-world measured BRDF represented as 3D or 4D grid of samples. The presented SSE implementation speedups BRDF support in 3-4 times comparing with traditional ray tracing BRDF computations.

Keywords: *SSE, interactive ray tracing, BRDF, photorealistic rendering.*

1. Introduction

The amount of computations involved, and consequently, the amount of time required for the algorithm to run is one of the main problems of realistic image synthesis. In order to compute the pixel color using backward ray tracing several rays have to be traced: the ray from the camera to the scene, the rays from the hit point to the light source, reflected and refracted rays etc. Given the image size of 1280 by 1024 pixels, the number of rays can be in the range of 10 to 20 millions. When a single ray is traced, it is tested against all scene objects, and the nearest intersection point is selected, as it is the visible point. In our estimate a rendering system (without global illumination computations) spends 65 – 75% amount of time performing ray tracing. According to T. Whitted, this amount may even reach 95% [1].

Almost all existing realistic ray tracing-based rendering systems employ some kind of spatial hierarchical structure for acceleration of ray tracing. The entire scene is subdivided into subregions, and the list of objects belonging to each of the subregions is constructed. The goal of the subdivision is to accelerate the process of ray tracing. When a ray intersects a subregion, it is tested against all of the objects in that subregion. If one of them is intersected, the algorithm terminates for this particular ray and the intersection is reported. If there are no intersections, the ray proceeds to the next subregions, and so on. Spatial subdivision methods decrease the ray tracing time substantially (often by several orders of magnitude).

Coherent ray tracing is a complement to spatial subdivision structures. In this approach, several rays are traced together in a bundle. The rays being traced are usually chosen in such a way that they require the same (or located near to one another) data in order maximize the acceleration provided by the use of SIMD instructions. This "similarity" among the rays traced simultaneously is called "ray coherency", or just "coherency", for short, hence the term "coherent ray tracing". In order to achieve coherence for primary rays, we place rays from nearby pixels in the same bundle. Reflection or shadow rays spawned by these primary rays are also traced together, providing coherency for these rays too. If properly implemented, coherent ray tracing gives a significant speed-up compared with classical single-ray approach [2]. The speed-up factor is usually equal to the number of SIMD data elements processed simultaneously.

Coherent ray tracing has been developing since the SIMD support in commodity processors became available, namely SSE (Streaming SIMD extensions) in Pentium III and Pentium 4. This extension is now supported also by AMD processors, and it makes SSE a kind of de-facto standard for implementing coherent ray tracing. Coherent ray tracing is often called SSE ray tracing, as is sometimes done in this very paper. Since SSE operates with 4 single-precision floating-point values simultaneously, its use in ray tracing typically gives acceleration by a factor of 4 compared to optimized C++ implementation.

BRDF (Bidirectional reflection distribution function) is the most general way to represent material properties for physically accurate rendering. BRDF is a very useful feature for realistic rendering. BRDF describes the energy transfer between arbitrary incoming and outgoing directions. The transfer factor may actually depend on wavelength and polarization of incoming light. But for simple cases, however, RGB color model with no polarization is sufficient. There are multiple ways to represent BRDFs (Phong and Blinn models [3-4]). More complex BRDFs may be represented by the set of samples for the pairs of incoming and outgoing directions.

BRDF has already been used in our work ([5-6]). BRDF's we use come from real-world data. They are obtained by either measuring the BRDF samples ([7]) or by deriving BRDF from the microstructure of the material. BRDF obtained in such ways exhibits complex structure which is hard to approximate by a particular BRDF model. Therefore, the only way to represent BRDFs suiting our purposes is tabular representation of BRDF.

Tabular BRDF computations may become a bottleneck when other components are rather fast due to SSE ray tracing. Therefore, BRDF computations using SSE instructions would provide significant speedup.

2. General overview

The angles of BRDF parameterization are σ, ψ, θ and φ (see fig. 1). Based on the number of angles the BRDF really depends on, the BRDFs may be classified as anisotropic (depends on 3 angles) and isotropic (depends on 4 angles).

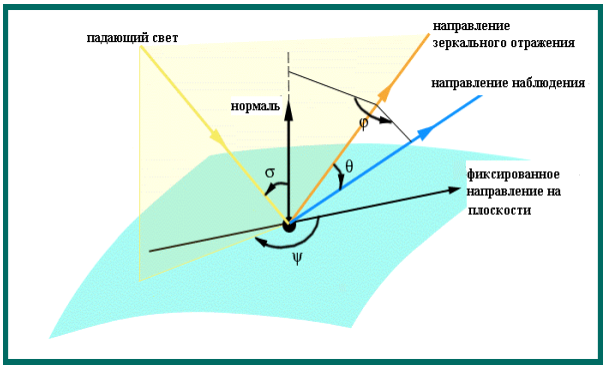

Fig. 1. BRDF parameterization.

The actual BRDF value is computed by n-linear interpolation. One should notice that although there may be uniform sampling along certain directions, some directions do not allow this kind of sampling. This is true for θ

direction. Typical BRDF has a maximum at θ = 0 and rapidly decreases with the increase of θ. Therefore, one needs to place more samples near θ = 0 and few samples near θ = π. The BRDF table grid thus becomes non-uniform, which requires performing binary search in order to define proper interpolation cell.

It should be noticed that the BRDF values are stored with the angle grid, not the cosine or sine grid. This is done in order to perform n-linear interpolation with these values. Therefore, for each incoming and outgoing ray angles are to be computed. This requires using inverse trigonometric functions which usually take some 250-300 processor cycles to compute [8].

When trying to implement all this on SSE for 4 rays in parallel, the complexity of the task increases. Things so familiar for non-SSE (indexing, binary search) become not so simple when trying to implement them in SSE. Therefore, following problems arise for SSE BRDF support:

- Calculation of inverse trigonometric functions
- Performing SSE binary search
- Indexing and interpolation in SSE case

3. Calculation of inverse trigonometric functions in SSE

For non-SSE case, computing inverse trigonometric functions is very slow. Evaluating inverse trigonometric function must be performing once per dimension, slowing down the entire process. So, this is the first thing to accelerate.

Notice that the angles need not be computed exactly since they are used only for interpolation. Therefore, it is feasible to use an approximation for inverse trigonometric functions.

After considering various approximations, we have finally decided to use one from [9]. It requires only moderate amount of computation. The actual approximation has the form

$$\arcsin x \approx \frac{\pi}{2} - \sqrt{1-x}\left(\frac{\pi}{2} + a_1 x + a_2 x^2 + a_3 x^3\right)$$

It computes approximations for positive x only. For negative x, we compute arcsin(x) = -arcsin(-x) since arcsine is an odd function. Having computed arcsin, we can compute arccosine as

$$\arccos x = \tfrac{\pi}{2} - \arcsin x$$

The precision of this approach is enough for our purposes. The precision plot is given in fig. 2 and the performance measurements are given in table one.

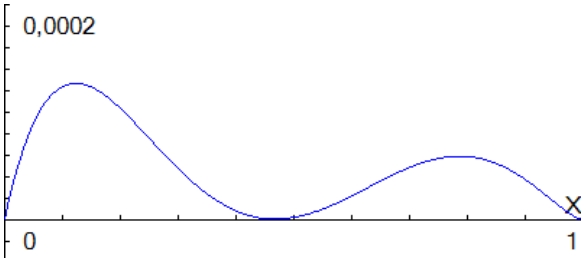

Fig. 2. Precision plot for acos(x) approximation

# calls, millions	10	20	40
non-SSE(sec.)	1.479	2.954	5.908
SSE (sec.)	0.084	0.156	0.319
Speedup	17.6	18.93	18.52

Table 1. Computational performance of inverse trigonometric functions (using SSE and without SSE).

We have obtained significant performance gain by computing inverse trigonometric functions approximately in SSE. For the same number of values to compute, SSE approximation gives more than 17 times acceleration over original approach.

4. SSE binary search

Another item to be done in SSE is binary search. Binary search in non-SSE is a classical algorithm. However, it is not so for SSE case.

The main issue in implementing SSE binary search is how to deal with branching, i.e. when different elements of the quadruple give different comparison results. In order to deal with it, a constant-size stack is introduced. Whenever the comparison gives different results, we select some of them as the current search value and push current search state to stack. When search for the current value is finished, the indices obtained are written to the output and we check whether the stack is empty. If the stack is empty, we finish the search. Otherwise, we pop the state from the stack and continue. The state stored in the stack consists of the current mask and current limit. The mask is used to select the values which are currently active. The initial value of the mask is passed to the procedure as a parameter. Here is the pseudocode of the SSE binary search algorithm:

```
sse_int sse_binarysearch(array<float> arr, sse_float val,
int mask) {
 i = 0, j = arr.length();
 sse_int res;
 while(true) {
  if(i == j) {
    res.set(i, mask);
    if(stack is empty)
      return res;
    st.pop(i, j, mask);
    continue;
  }
  k = (i + j) >> 1;
  cmp = val < sse_float(arr[k]);
  if(all_true(cmp, mask))
    j = k;
  else if(all_false(cmp, mask))
    i = k;
  else {
   push(false_mask(cmp, mask), i, k);
   j = k;
   mask = true_mask(cmp, mask);
  }
 }
}
```

For the sake of performance, we use several specialized versions of SSE binary search algorithm. Namely, there is a version which computes interpolation weights and evaluates 1d grid functions. This allows avoiding some extra SSE indexing operations.

# calls, millions	10	20	40
non-SSE(s)	0.9	1.78	3.55
SSE (s)	0.41	0.79	1.61
Speedup	2.23	2.25	2.20

Table 2. Performance measurements for binary search using SSE and without SSE.

The results of the performance measurements are given in table 2. The achieved speedup is 2.5. This is quite enough for the purposes of the fast BRDF interpolation.

5. SSE indexing and interpolation

Having retrieved the indices for BRDF computations, now we need to linearly interpolate between them in order to get the BRDF value for the specified incoming and outgoing directions. Prior to interpolation, we need to extract the BRDF values between which to interpolate. And that assumes using indexing operators.

Indexing in SSE case is not so simple because one needs to extract 4 values from different memory locations simultaneously. The naive way is to read 4 different values from memory and place them in the respective elements of an SSE register. However, this approach is inefficient for the case of coherent rays since we would often read the same value several times which would incur additional penalties in creating SSE values. This is further aggravated by the fact that we need to do this 8 (for 3D BRDF) or 16 (for 4D BRDF) times per BRDF computation for a single ray. This may become a real bottleneck.

In order to accelerate the process, we modify the indexing algorithm a little. We take the 0-th element of the SSE quadruple and check whether it equals to the other components (taking the current mask into account). If all the active elements equal to the 0-th element, we perform indexing and interpolation only once, thus considerably reducing the number of operations required and avoiding setting operations. In case of incoherency, we successively check the elements beginning from 0-th. For each element, we define those elements of the quadruple which are equal to it, and then perform indexing and interpolation for these elements. Then we exclude these elements from the active mask and do the same thing for the elements which have not yet been processed.

In order to accelerate index comparison, we store them as floating-point values. This also accelerates computation of single array index from 3 or 4 BRDF indices since multiplication operations are performed simultaneously.

Due to performing 4 operations simultaneously, the acceleration must be near to 4 for coherent cases, but will drop down significantly (almost by the number of the groups of coherent rays in the quadruple) for the case when rays are incoherent. So, the average acceleration is about 2.5. It must be also kept in mind that 3D or 4D interpolation for non-SSE case involves significant number of operations and the compiler may optimize even the code for the single ray to take benefit from using SSE instructions which will accelerate non-SSE case.

6. Results

We have implemented SSE BRDF computations for backward ray tracing rendering. SSE BRDF support was implemented in C++, under Microsoft Visual Studio 2003. Microsoft compiler has been used with all optimization options turned on. We have not used any assembler code or inline assembly at all and we have used SSE intrinsic functions only in basic classes which support operations with quadruples of integers and single-precision floating-point values.

We have measured the time of BRDF computations separately. The timings were performed for both 3D and 4D BRDFs separately. The results are given below (Tables 3 and 4).

# calls, x 10^5	1	2	4
non-SSE(sec.)	0.085	0.177	0.350
SSE (sec.)	0.024	0.050	0.101
Speedup	3.54	3.54	3.46

Table 3. The results for 4D BRDF performance test.

# calls, x 10^5	1	2	4
non-SSE(sec.)	0.137	0.248	0.495
SSE (sec.)	0.040	0.078	0.156
Speedup	3.43	3.17	3.17

Table 4. The results for 4D BRDF performance test.

The actual results depend on the characteristics of the PC and the dimensionality of the BRDF. For performance measurements, the BRDFs with the following dimensionalities have been used: (7 along sigma, 8 along phi, 13 along theta) for 3D BRDF and (17 along psi, 7 along sigma, 17 along phi, 13 along theta) for 4D BRDF. All timings were performed on a Mobile Pentium-IV 1800 MHz Intel Centrino notebook with 512 MB of 433 MHz RAM.

Fig. 3. An example of 4D BRDF rendering using SSE BRDF support.

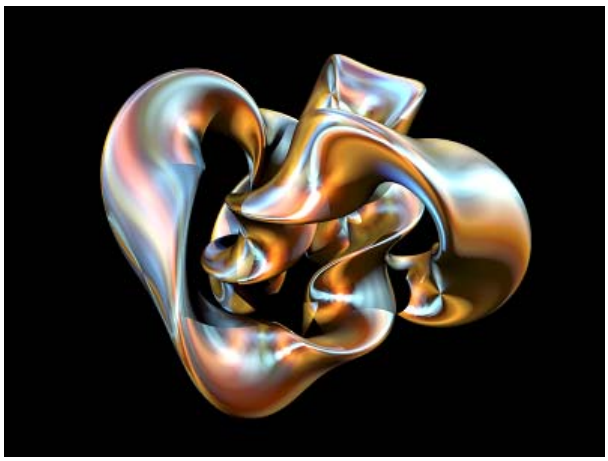

Fig. 4. Another example of 4D BRDF rendering using SSE BRDF support.

7. Conclusion

We have implemented SSE BRDF support for ray tracing and have achieved speedup by a factor of 3 – 3.5. The performance increase may be even higher if we use Intel C++ compiler. It would be interesting to consider the use of integer instructions available in SSE 2 and some new instructions provided by SSE 3.

We have already implemented BRDF support for RGB color model only for the case of backward ray tracing. But our framework supports complex global illumination algorithms and spectral color models. All this is based on ray tracing, and additional performance may be achieved if these algorithms are implemented on the top of coherent ray tracing. The speedup for forward ray tracing, however, would be not as big as for backward ray tracing since ray coherence is lower. New structures and techniques are to be used in order to provide greater speedup for global illumination algorithms. Spectral color model, on the contrary, will benefit more since it uses much more color components (about 30) and allows using SSE instructions not only for working with color quadruples, but also for working with single color.

The work was supported by RFBR, grant № 05-01-00345-a, and Integra Inc. (Tokyo, Japan). The version of the paper with color illustrations can be found at http://www.keldysh.ru/pages/cgraph/publications/cgd_publ.htm

REFERENCES

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

[2] I. Wald, C. Benthin, M. Wagner, Ph. Slusallek. Interactive Rendering with Coherent Ray Tracing. Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001) Vol. 20, № 3, pp. 153-164.

[3] B. Phong, “Illumination for Computer Generated Pictures”, Communications of the ACM, Vol. 18, № 6, pp. 311-317, 1975.

[4] J. F. Blinn. Models of light reflections for computer synthetized pictures. In *Computer Graphics (SIGGRAPH’77 Proceedings)*, pp. 192–198, 1977.

[5] A.Khodulev, E.Kopylov, Physically accurate lighting simulation in computer graphics software. Proc. GraphiCon’96 - The 6-th International Conference on Computer Graphics and Visualization, St.Petersburg, 1996.

[6] А.Г. Волобой, В.А. Галактионов, К.А. Дмитриев, Э.А. Копылов. Двунаправленная трассировка лучей для интегрирования освещенности методом квази- Монте Карло. "Программирование", № 5, 2004, с. 25-34.

[7] Letunov A.A., Barladian B.H., Zueva E.Yu., Veshnevetc V.P., Soldatov S.A. CCD-based device for BDF measurements in computer graphics. The 9-th International Conference on Computer Graphics and Vision, Moscow, Russia, Aug 26 - Sep 1, 1999.

[8] IA-32 Intel Architecture Optimization Reference Manual, p. 440. ftp://download.intel.com/design/Pentium4/manuals/24896611.pdf

[9] Mathematical Handbook for Scientists and Engineers, Second Edition. McGraw-Hill Book Company, 1968.

Authors:

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

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