Optimizing Cache Behavior of Ray-Driven Volume Rendering Using Space-Filling Curves

A thesis presented

by

Oleg Mishchenko

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Master of Science

in

Computer Science

Stony Brook University

May 2006

Stony Brook University

The Graduate School

Oleg Mishchenko

We, the thesis committee for the above candidate for the

Master of Science in Computer Science degree

hereby recommend acceptance of this thesis.

Klaus Mueller, Associate Professor, Department of Computer Science

---

Tzi-cker Chiueh, Professor, Department of Computer Science

Hong Qin, Associate Professor, Department of Computer Science

This thesis is accepted by the Graduate School

Dean of the Graduate School

Abstract of the Thesis

Optimizing Cache Behavior of Ray-Driven Volume Rendering Using Space-Filling Curves

by

Oleg Mishchenko

Master of Science

in

Computer Science

Stony Brook University

2006

We study the cache behavior of space-filling curves in the context of volume rendering with raycasting. An efficient bit-oriented algorithm for 3D Hilbert curve traversal is also proposed. We find that data organization along space-filling curves provides an effective means to preserve cache coherency and to reduce expensive cache misses. A surprising result of our study is that the Z curve has better characteristics in conjunction with smaller cache sizes than the theoretically superior Hilbert curve. We tested a wide range of ray traversal models, such as random ray direction changes associated with scattering, bent rays resulting from refractions, and the cache-related effects of interpolation filter size.

Contents

Abstract **iii**

1 Introduction and Related Work **1**

1.1 Background 1

1.2 Sampling and Filters 5

2 Space-Filling Curves **7**

2.1 3D Hilbert Curve Traversal Algorithm 8

3 Results **13**

3.1 Comparison with other methods 15

3.2 Conclusions 16

Bibliography **19**

List of Figures

1.1 *Filters and space-filling curves* 6

2.1 *3D second-order Hilbert curve* 8

2.2 *Measure of coherence for 2D Hilbert and Z curves.* 9

2.3 *Measure of coherence for 3D Hilbert and Z curves* 9

2.4 *Hilbert curve index generation algorithm transformations* 10

2.5 *Source code for Hilbert curve traversal algorithm* 12

3.1 *Average number of cache misses per filter application* 14

3.2 *The test volumes* 16

3.3 *Results for linear ray traversal.* 17

3.4 *Results for non-linear ray traversal* 18

Chapter 1

Introduction and Related Work

1.1 Background

Direct volume rendering is concerned with the visualization of volumetric datasets, described as 3D or 4D arrays of scalar or vector (RGBA) values. These types of datasets are being ubiquitously produced by medical scanners, scientific simulations, and engineering design. They even occur in mainstream entertainment applications, for example in the modeling of amorphous effects, such as smoke, gas, or fluid. A number of paradigms have been described for the volume rendering of regular grids, with the most popular being raycasting [15], splatting [33], shear-warp [11], and cell projection [27]. These methods have received significant attention and, due to this widespread effort, have been well developed at this point. While all of these algorithms have their strengths and weaknesses, ray-based aproaches provide the highest flexibility to conveniently realize non-linear light propagation effects, as occuring in refracting and diffracting media. The modeling of global scattering and shadows are likewise best handled with ray-based approaches. Unfortunately, these rendering scenarios also give the most irregular memory access patterns, which poses significant challenges to cache management. Furthermore, sampling (interpolating) the volume in 3D coordinates creates large leaps in data addressing if the memory access is not inherently 3D, and costly cache missses are a direct consequence. These effects are even exaggerated when the (3D) interpolation filters are of higher quality and therefore larger [19].

Recent work on GPU-accelerated volume rendering convincingly advocates a single-loop one-pass raycasting approach, that is, a single fragment program steps each ray across the volume, front-to-back [29]. This scheme only requires one control primitive to be rasterized which then triggers all rays to be spawned. The rays can be non-linear, supporting refraction, diffraction, and scattering effects, as their positions are updated independently. The single-loop approach minimizes the number of passes and hence the switch time requitred and the control polygon rendering effort. Interpolation of sample values is achieved via indexing a 3D texture. However, with GPU data access being localized into 2D texture addressing, such a 3D texture is typically realized as a 2D sheet of 2D textures, either implicitly or explicitly. Consequently, a 3D texture access will incur a significant jump in the addressing space, possibly acompanied by a costly cache fault. While there may also be cache faults in 2D texture access, these may be buffered by the L2 cache, but frequent L1 cache faults seem certain. Hence, to cope with these data communication problems, we desire a data access and organization scheme which is more coherent in the presence of these high-dimensional access patterns, incurring a lesser amount of cache misses.

A space-filling curve is a data parameterization scheme that can provide the desired high level of data access coherency, even in three and higher dimensions. With space-filling curves, the data are stored in memory in the traversal sequence implied by the curves' space traversal pattern. In this thesis, we will show that storing the data in such a pattern lowers the number of cache faults in raycasting scenarios significantly. The use of space-filling curves is certainly not new, and they have given rise to impressive results in various applications, ranging from data clustering in high-dimensional spaces [20] to computer graphics [16] to volume slicing [23]. Nevertheless, our work makes a number of unique and novel contributions:

- It extends the theory of space-filling curves to actual volume rendering via raycasting and raytracing, with a GPU-based implementation in mind.
- It compares two popular space-filling curves, Hilbert and Z-curve (also known as Lebesgue curve), in terms of cache performance in this scenario.
- It addresses issues, such as cache size, rendering effects, and interpolation filter

size.

- It presents a fast indexing scheme to convert spatial coordinates into Hilbert curve coordinates (an indexing scheme for the z-curve was presented in [23]).

Our experiments indicate that well known clustering properties of the curves give much better performance than the typical sequential layout (referred to as scanline layout). This should not come as a surprise. With space-filling curves, the acceleration is due to a better locality and access coherence of the neigborhood being interpolated. Filters used for interpolation always need access to several close memory locations. The curve coherence properties make such locations closer on the average, thus improving cache behavior. As mentioned above, we use two well known space filling curves - the Lebesgue curve, also known as Z-curve, and the Hilbert curve. The Z-curve has two very attractive properties - it is highly coherent and has very simple and efficient index computing algorithm. However, the Hilbert curve has higher coherence than the Z-curve.

Our index calculation algorithm, presented here, for the Hilbert curve, is simple and efficient, and, though still slower than that for the Z-curve, allows us to get better overall performance. Just like the Z-curve scheme in [23], it uses bitwise operators, which are currently not available on GPUs. But we believe that soon this situation will change and hope that this work will provide one more strong motivation for this.

While we use the GPU as an example platform, the study presented in this thesis is general enough for any architecture, be it CPU, GPU, or a processor in a mobile device, where the size of cache and memory is an issue. With this in mind, the thesis is organized as follows. First, in Section 2, we provide background and also report on related work. Then, in Section 3, we give a review of the special data access issues pertinent to ray-based volume rendering. Section 4, describes our fast Hilbert curve traversal algorithm and compares it with the Z-curve of [23]. Finally, in Sections 5 and 6, we present results, a discussion of these, and conclusions.

Caches usually get smaller the closer they are to the ALU. Usually a hierarchy is present composed of L1 cache, L2 cache, main memory and disk. This is even

true for the most recent GPUs, which is composed of L1 cache, L2 cache, texture cache, main (CPU) memory, and disk. Govindaraju [8] reports that in the current generation of GPUs the L1 cache is local to a particular fragent processoir, while the L2 cache is shared among multiple fragment processors. Small register banks are also available to store reusable data. For GPUs, the exact sizes of these resources are a well-kept secret of the hardware manufacturers, but frameworks like GPUBench [9] have come online that provide facilities to bechmark the hardware to find the sweet spots in the balance of overhead incurred by data managment and cache faults. In fact, a number of papers have been written that optimize the subtask granularity of various applications, such as sorting [7], and matrix multiplications [6] (but this list is not exhaustive), for GPU execution. We take a more general look at cache size in the context of space-filling curves, but with GPU-accelerated ray-based volume rendering applications in mind. While GPUs make 2D data access (in form of 2D textures) the pattern of choice, it is not clear if they actually use 2D memory, or just a clever caching and indexing scheme. But in any case, it has been shown that 2D texture accesses are faster than 1D accesses. Thus, we will store our curves in a 2D texture and use the 2D addressing, but we will make this texure as narrow as possible to keep the sequential access pattern encoding the space-filling traversal patterns undisturbed. But as mentioned before, the project′s scope extends well beyond GPU applications, where traditional 1D addressing might be used.

With the growing popularity of GPUs almost all volume rendering paradigms, as listed in the introduction, have been ported to the GPU platform, in numerous incarnations. The unique strength of graphics hardware to accelerate texture mapping has given rise to a more novel scheme called slice-based volume rendering [24], but raycasting [28], splatting [22], and cell projection [25] have also been successfully accelerated on the GPU. Volume datasets can be massive - depending on precision a 512^3 dataset can exceed the size of texture cache, requiring fetches from main memory. But we have also noticed in a GPU-accelerated Computed Tomography application [34] that the much smaller L1/L2 caches play a significant role in performance. There we found that the data needed to be subdivided into smaller blocks of 128^3 to achieve a linear scale in performance. This application still was able to use

the shared L2 cache, which is typically in the order of MB on mainstream CPUs, since the task was of a large granularity. An application like raycasting, especially with non-linear rays, has a more local nature, and a closer look at the local L1 cache is desirable. These tend to be on the order of KB for CPUs and we expect that this is also true for GPUs (the exact details remain unknown in consumer circles). At the same time, smaller devices, such as graphics units for mobile PDA-type units probably feature reduced caches as well. Therefore, we have paid a close look at smaller cache sizes in our study.

We layout the data in memory according to the space filling curve indexing (see Figure 2). Since 2D textures are the preferred data organization on GPUs, we pack the (1D) curve through the 3D data into a 2D texture. However, in order to retain the continuous 1D traversal properties, it seems most appropriate to create a 2D texture as narrow and as tall as possible, as mentioned above.

1.2 Sampling and Filters

In ray-based volume rendering, rays are cast into the volume and the volume rendering integral is calculated. Generally speaking, the calculation is an opacity-weighted color accumulation (called compositing) during the ray's volume traversal. Both color and opacity are based on the interpolated scalar density and are looked up by mapping this density into a transfer function. Then the shading calculations are performed. This requires a gradient, which is typically interpolated as well from a gradient volume, but it could also be obtained by interpolation with a derivative filter, or by central-differencing newly interpolated neighboring samples. Note that the latter will make the footprint of interpolation even larger. If a precomputed gradient volume is used, the density and the 3-component gradient are typically stored in the RGBA channels, respectively.

To obtain a ray sample, both density and (x,y,z) gradient, the value of the 3D function at this particular point, are reconstructed from the grid values. This is done with a specific filter. The proper reconstruction of a discrete grid into a continuous function is a very important step, as careless filter selection can cause a number of

artifacts in the image [19]. Filters in use, in order of reconstruction quality, are the trilinear and the cubic function. The trilinear filter uses a 2^3 neighborhood, while the cubic filter uses a 5^3 neighborhood. As is explained in [19], larger filters improve the reconstruction quality further and should be used in accuracy-critical scenarios. The filter (kernel) size determines the extent of the 3D neighborhood requested from the cache (we call this addressed neighborhood the samples footprint). Favorable cache organization schemes must aim to be sufficiently coherent, causing only few cache misses.

We are interested in the data access cost of a particular filter. We have just mentioned that in order to use the filter in the interpolatrion of the volume, a number of memory references is necessary. If the difference between memory addresses is small, it is likely that after the first memory access the next voxels will be already in cache. This is, of course, desirable. This difference can be used as a simple and useful metric for evaluating the filters with regard to different data layouts. Figure 1.1 illustrates this idea.

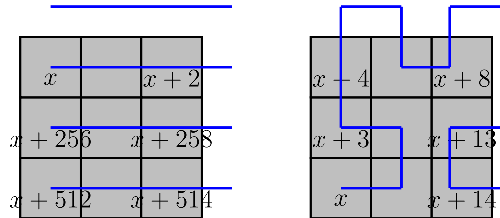

Figure 1.1: *Filters and space-filling curves*

When applying kernel to data with a scanline layout, the difference between memory addresses can be very large. When using space filling curve, the distance on the average is significantly smaller. Here 3x3 kernel is applied to a 256x256 image.

Chapter 2

Space-Filling Curves

A space-filling curve is a mapping from N dimensions to one, but in this thesis we deal only with the 3D case. Historically, the curves were found first for the 2D case (that is where the popular name plane-filling curves originates). 3D and higher dimension versions were discovered later. Peano was the first to find the original curve in 1890. Later Hilbert introduced a different curve, which is known as Hilbert curve or Peano-Hilbert curve. The important properties distinguishing these curves from arbitrary N-to-1 dimension mappings are self-similarity and good clustering behavior. The first property means that at a different level of detail, the curve fills the space with the same pattern. The second property means that points close in N dimensions remain close in 1D. The latter is the most important property, and it explains the popularity of space-filling curves in computer science. When locality preservation is necessary, space-filling curves provide an efficient solution. Applications of space-filling curves include databases [14] and cache-oblivious matrix multiplication [3]. In graphics, some of the applications of space-filling curves have been image compression [18] and digital halftoning [30]. Ma and McCool used Hilbert curve in their block hashing algorithm for accelerating hardware-based photon mapping [16].

Having found the coherence of different curves, we cannot, however, simply select the one with the highest coherence. Important question is how fast we can calculate the indexes on the curve, i.e. calculate the mapping from the given 3D coordinates to 1D coordinate on the curve. Lebesgue curve has a very simple and efficient

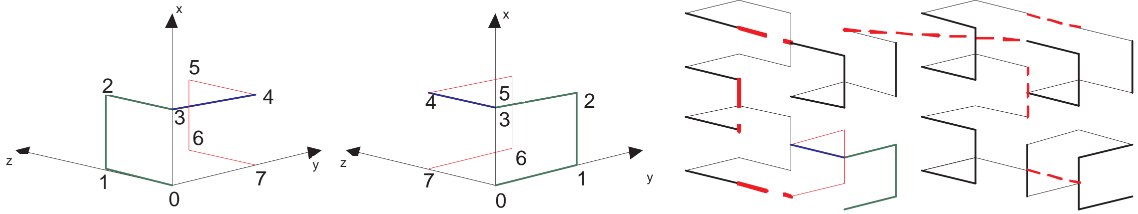

Figure 2.1: *3D second-order Hilbert curve*

Hilbert curve of order two is created with 8 order one curves, appropriately rotated and/or translated. Left: Hilbert curve of order one. Middle: one of the segments is created by swapping y and z coordinates of the original curve. Right: 2nd order Hilbert curve. Red dashed lines are used to show how the order one segments are connected. Note that the layout of the segments corresponds to the layout of order one curve. The process is recursive, i.e. 3rd order curve is created from 8 order 2 curves, etc.

algorithm, described in a paper by Pascucci and Frank [23]. The index is calculated by interleaving the bits of the corresponding coordinates. Because of the ease and speed of the algorithm, they have proposed its implementation in hardware.

2.1 3D Hilbert Curve Traversal Algorithm

Compared to the Z-curve, the Hilbert curve index calculation is more complicated. There are a number of algorithms for this task, for example [4]. We extend the approach for the 2D case, shown in [12], to 3D. A 2D Hilbert curve of order n is created from 4 appropriately rotated and/or translated curves of order n-1. The 3D Hilbert curve is then created from 8 curves of order n-1, also appropriately rotated and/or translated. The idea of our algorithm is that such rotations/translations can be calculated with only bitwise operations applied to corresponding coordinates. Note that, according to [2], there are more that 1,500 different Hilbert curves of order 3, and we are using only one of them. In Figure 2.1, we illustrate all eight curves of order one. To create an order 2 curve we merge the eight curves into one. Eight curves of order one are located according to the layout of the original order one curve.

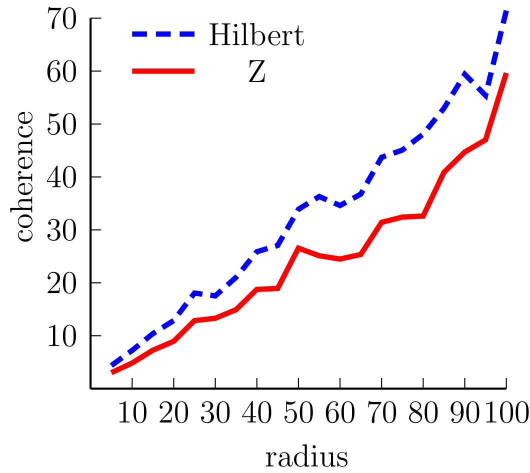

Figure 2.2: *Measure of coherence for 2D Hilbert and Z curves.*

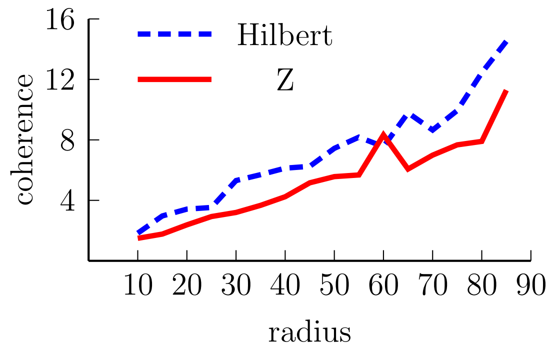

Figure 2.3: *Measure of coherence for 3D Hilbert and Z curves*

Given 3 coordinates x y z, we should get an index s, corresponding to the position of the point on the curve. Because the curve is self similar, most significant bits of n-order curve coordinates correspond to one of the 8 possible positions of n-1 curve. You can see it in Figure 2.1. The number determines 3 most significant bits of the index. Before we apply this procedure recursively, the curve of order n-1 should be appropriately rotated/translated/flipped, to make its layout correspond to 1st order curve layout. The appropriate transformations are shown in Figure 2.4, and the algorithm source code is shown in Figure 2.5. In our experiments the algorithm on the average was 2,5 times slower than Z curve traversal algorithm.

xyz	*Transformation*	*Index*
0 0 0	change x, y	0(000)
0 1 0	change z, y; complement them	7(111)
1 0 0	complement z, x	3(011)
1 1 0	complement z, x	4(100)
0 0 1	change x, y	1(001)
0 1 1	change x, y; complement them	6(110)
1 0 1	change x, y	2(010)
1 1 1	change x, y; complement them	5(101)

Figure 2.4: *Hilbert curve index generation algorithm transformations*
Left column shows the most significant bits of the x, y, z coordinates of Hilbert curve. Center column shows transformation necessary before applying next recursive algorithm step. Third column shows the bits that are added to the index at each step of the algorithm. For convenience decimal representations of these bits are also shown.

To better understand how algorithm works, consider the following example. We have volume 4x4x4, with indexes range from 0 to 3. We would like to get index on the Hilbert curve for the coordinates (3 1 2). The coordinates in binary are (11 01 10). Algorithm starts with the most significant bits of the coordinates. Looking at the Figure 2.4, we see that bits (1 0 1) correspond to vertex 2, with index (010). These bits (010) become the most significant bits in the calculated index. Next step is performing corresponding transformation of the curve of order n-1 (in our case 1). Changing x and y means getting (1 1 0) from the bits (1 1 0) (the least significant bits of the original coordinates). This gives us next three bits to the index (100) (found in the fourth row of the transformation table) We have reached the level of the curve 1, thus finish the procedure. The index is (010100), which is 20 in decimal. You can verify looking at the figure that this is indeed correct.

```
//long long s - curve index, short x,y,z - coordinates
//in 3D, int order - curve order
int transform[8] = {0, 1, 7, 6, 3, 2, 4, 5};
```

```
unsigned long long s = 0;
unsigned short xi, yi, zi, temp;
int offset = sizeof(short)*8 - order;
for(int i = sizeof(short)*8 - 1 - offset; i >= 0; i--) {
    xi = ((x)>>i)&1;
    yi = ((y)>>i)&1;
    zi = ((z)>>i)&1;
    //change y and z
    if(xi == 0 && yi == 0 && zi == 0) {
        temp = z;
        z = y;
        y = temp;
    }
    //change x and y
    else if(xi == 0 && yi == 0 && zi == 1) {
        temp = x;
        x = y;
        y = temp;
    }
    //change x and y
    else if(xi == 1 && yi == 0 && zi == 1) {
        temp = x;
        x = y;
        y = temp;
    }
    //complement z and x
    else if(xi == 1 && yi == 0 && zi == 0) {
        x = (x)^(-1);
        z = (z)^(-1);
    }
    //complement z and x
    else if(xi == 1 && yi == 1 && zi == 0) {
        x = (x)^(-1);
```

```
        z = (z)^(-1);
    }
    //change x and y and complement them
    else if(xi == 1 && yi == 1 && zi == 1) {
        temp = (x)^(-1);
        x = (y)^(-1);
        y = temp;
    }
    //change x and y and complement them
    else if(xi == 0 && yi == 1 && zi == 1) {
        temp = (x)^(-1);
        x = (y)^(-1);
        y = temp;
    }
    //xi == 0, yi == 1, zi == 0
    //change z and y and complement them
    else {
        temp = (z)^(-1);
        z = (y)^(-1);
        y = temp;
    }
    int index = (xi<<2)+(yi<<1)+zi;
    s = (s<<3) + transform[index];
}
return s;
```

Figure 2.5: *Source code for Hilbert curve traversal algorithm*

Source code for the Hilbert curve traversal algorithm. Note how transformations correspond to the ones shown in the previous figure.

Chapter 3

Results

For testing we used a purely software-based ray caster augmented by a cache simulator. Since bitwise operations are presently not supported on GPUs, we resorted to measuring cache performance by rendering a few characteristic volumes and counting the number of cache misses. This makes our analysis platform independent. We wanted to keep our results free of other hardware-dependent effects which are always subject to change, as is definitely true for GPUs. We sought to keep our tests as general as possible, testing a good collection of different ray-based traversal scenarions (see below). Our tests are targeted to provide insights into pure cache performance as well as into the prospects of space-filling curves for application in ray-based volume rendering. The cost of a cache fault itself is likely dependent on the underlying hardware - users and engineers may just use our graphs and numbers and calculate their personal costs by multiplying these with their cost tables. Even if the situation more complex, our results still have good value in assessing overall trends. In our rendering memory application, normals were calculated on the fly, with central differencing. This gives an enlarged footprint of the interpolation operations. We used Hilbert, Z and scanline layouts with a variety of cache sizes and associativities. Cache line size was 4 in all experiments. LRU replacement policy was used when cache associativity was 2 and 4. To avoid any kind of incorrect results due to volume orientation (for example, rays are parallel to one of the sides of the volume), we rendered the images with seventeen camera positions, and then averaged the total number of cache misses.

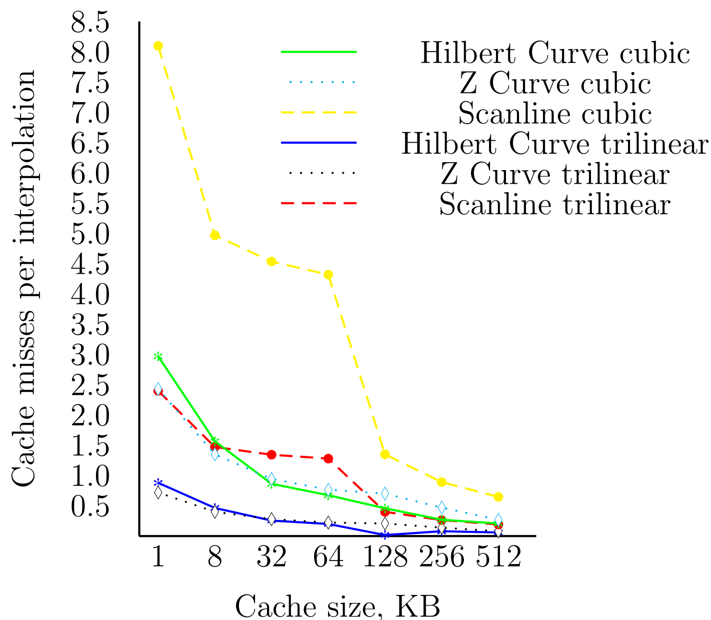

Figure 3.1: *Average number of cache misses per filter application*

In Figure 3.1 we show the results obtained for the UNC brain dataset. As expected, the scanline layout performs much worse than the others (for this reason, we did not include the scanline access method in remaining figures). We plot the number of average cache misses per interpolation as a function of cache size and interpolation filter used. With all of the volumes tested, the number of cache misses with scanline was always typically 2-3 times larger than with space-filling curves. This ratio starts to improve with larger cache sizes (around 512 kB, the size of a typical L2 cache), but there still remains a considerable difference. Note that this graph does not consider the cost of a cache miss, which is typically much greater than the cost of an arithmetic operation, given the persisting mismatch of memory and arithmetic bandwidth (and this gap grows with the level of the memory hierarchy). In some profiled GPU tests, we have experienced that the time spent on waiting for texture data typically consumed 20% of the computations and more. These and other performance numbers given in GPUBench [9] help to put the plots given in Figure 3.1 into perspective.

In Figure 3.3 we give detailed comparisons with different cache associativities for the three test volumes: UNC brain, engine and vessel aneurysm. The UNC brain and engine datasets are rather dense, while the vessels dataset is very sparse, i.e. the

number of voxels with value 0 is much larger. Because of this, it is not surprising to see a larger number of cache misses with the vessels dataset, in case of both Hilbert and Z curves. A more interesting observation is that the Z curve has fewer cache misses than the Hilbert curve, in case of smaller caches. This is to some extent counterintuitive, because the Z curve has worse coherence (although the difference is smaller for smaller circle radii, see Figure 2.2 and Figure 2.3).

In the above experiments the rays didn't change their directions while traversing. Some applications, however, may require non-linear ray traversal, when rendering refractions or external field effects. In other applications, ray reflections are necessary. We modeled these situations in two ways. In the first case the rays were 'bent' at a small angle during traversal. In the second case we implemented a 'random' ray traversal - after several steps each ray received an arbitrary new direction. The results are shown in Figure 3.4. Here the Z curve continues to perform better with small cache sizes, but only slightly worse with 512 and 256 KB caches. Notice that with ray 'bending', the total number of cache misses is the smallest in all of the ray traversal strategies, while for the 'random' ray traversal, the behavior of the curves is almost the same, though the total number of cache misses is larger and there is only modest reduction at larger cache sizes.

3.1 Comparison with other methods

We were not sending rays in packets, nor did we implement any other optimizations. The reason for this is that different optimization strategies usually work well only for a limited number of cases, and it is often possible to find a specific case when the given optimization does not perform well. Finally, implementation quality is also an issue. It is worth mentioning that our method is an example of blocking, also utilized with octrees and spatial subdivisions with scheduled ray queues. With blocking, the memory is subdivided in a number of blocks such that data in each block is kept close in terms of their memory addresses. This helps utilize data coherence. In meta-tiling [17], each block is itself located according to a pattern of a 'next-level' block. As noted in [35], the self-similarity of space-filling curves is a

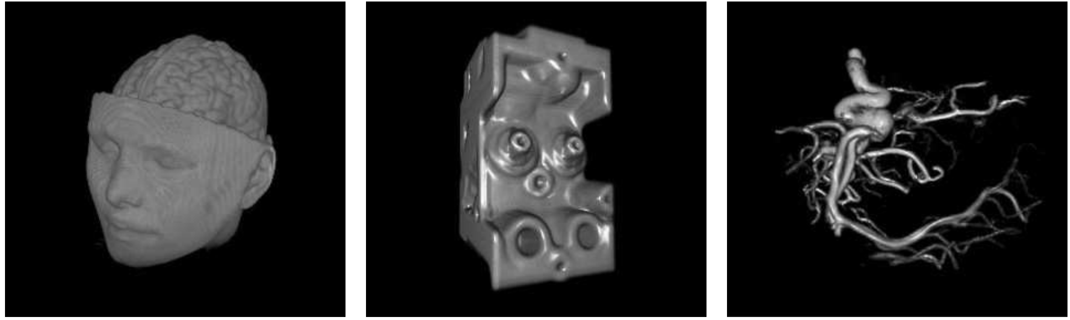

Figure 3.2: *The test volumes*
UNCBrain (128x128x72), Engine (128x128x64), Vessels (128x128x128)

form of meta-tiling. Four (in 2D) of eight (in 3D) 1st order curves make a 2nd order curve (next level block), four (eight in 3D) 2nd order curves make a 3rd order curve (next level block), and so on.

3.2 Conclusions

In this thesis, we have studied the cache behavior of space filling curves in the context of volume rendering. An efficient algorithm for 3D Hilbert curve traversal was proposed. Our experiments show that the better clustering behavior of Hilbert curve actually gives better cache behavior only with relatively large caches, when applied in the special task scenario of ray-based volume rendering. For smaller cache sizes the Z curve is a better match, and it also offers a more effcient indexing scheme. In that regards, we have found our study quite useful in that it points out a discrepancy of theory and practice which can be useful to hardware designers. A future research challenge is to use space-filling curves 'underneath' other memory-hierarchy optimizing schemes. That is, if a special datastructure is used, such as an octree, will it be possible to get some performance improvement with a non-scanline layout of the leaves? Another interesting topic is to use the curves for non-raycasting approaches, such as splatting.

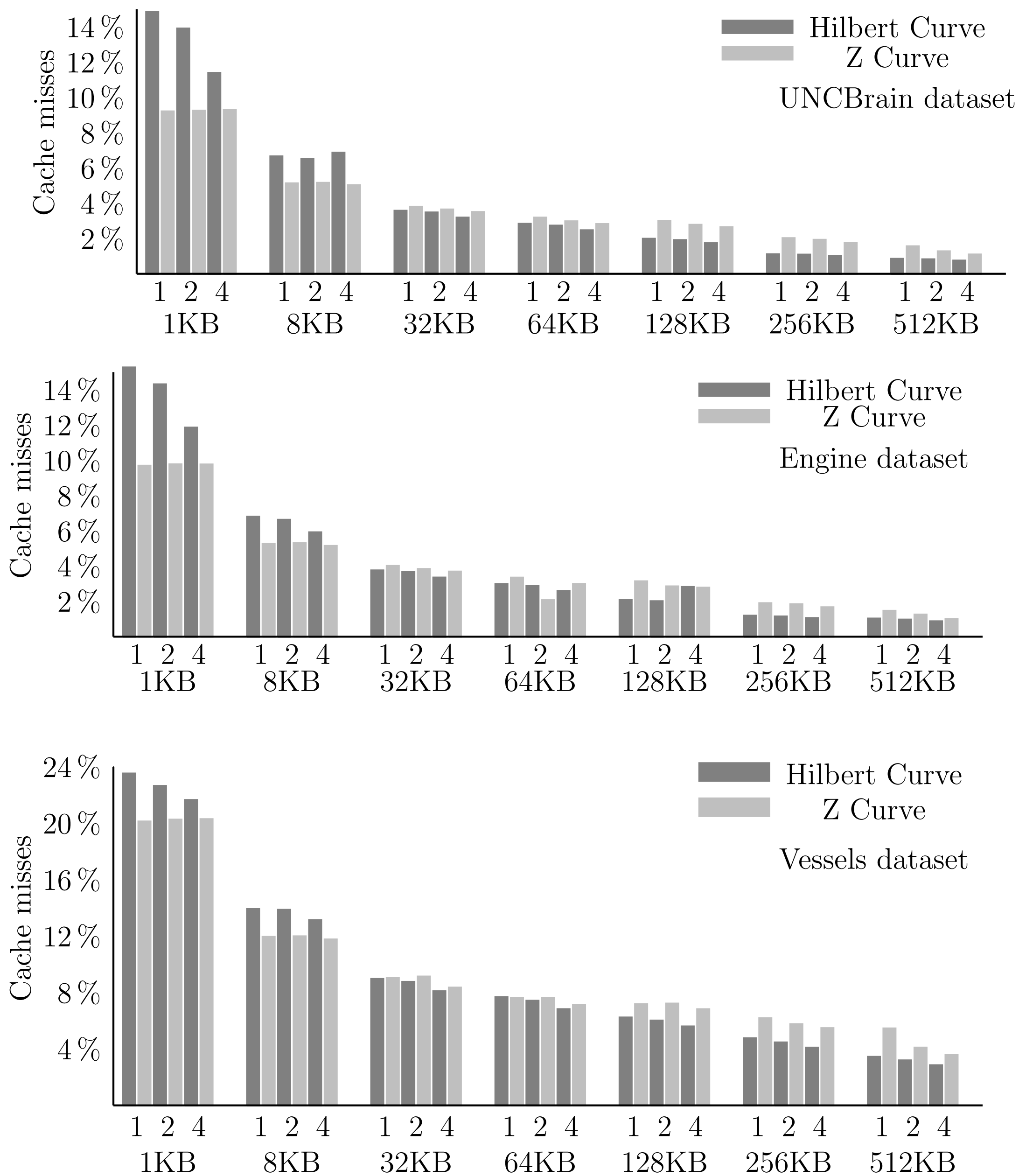

Figure 3.3: *Results for linear ray traversal.*

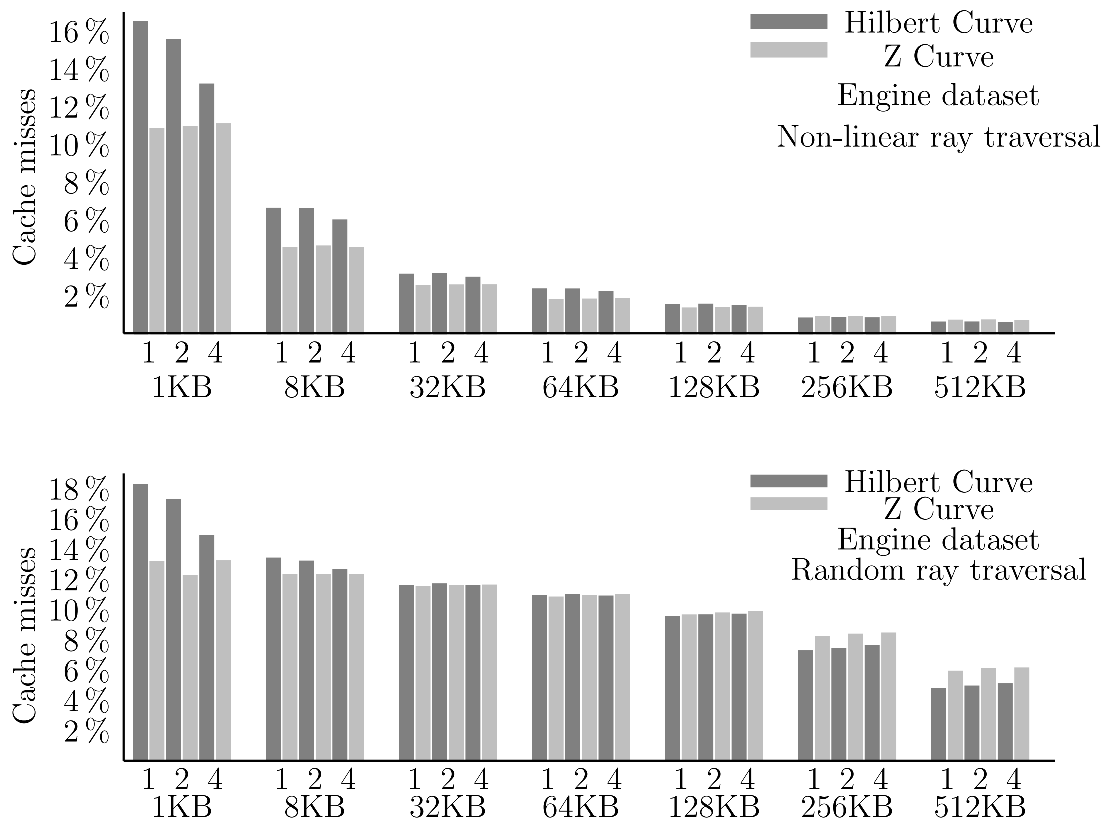

Figure 3.4: *Results for non-linear ray traversal*

Bibliography

[1] M.J. Aftosmis, M.J. Berger and S.M. Murman (2004), *Applications of space-filling curves to Cartesian methods for CFD*, 42nd AIAA Aerospace Sciences Meeting and Exhibit.

[2] Jochen Alber and Rolf Niedermeier (1998), *On Multi-Dimensional Hilbert Indexings*, Proc. of the Fourth Annual International Computing and Combinatorics Conference, pp 329-338.

[3] Michael Bader and Christoph Zenger (2005), *A Cache Oblivious Algorithm for Matrix Multiplication Based on Peano Space Filling Curve*, PPAM 2005 - Sixth International Conference on Parallel Processing and Applied Mathematics.

[4] G. Breinholt and C. Schierz (1998), *Algorithm 781: Generating Hilbert's Space-Filling Curve by Recursion*, ACM Transactions on Mathematical Software, volume 24, number 2, pp 184-189.

[5] J. M. Buhmann, D. W. Fellner, M. Held, J. Ketterer, J. Puzicha (1998), *Dithered Color Quantization*, CGforum, Editors N. Ferreira and M. Göbel, Volume 17, Number 3, September 1998, pp C219-C231, (Proc. Eurographics'98), http://diglib.eg.org/EG/CGF/volume17/issue3/ColQuant98.

[6] K. Fatahalian, J. Sugerman, P. Hanrahan, *Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication.*

[7] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming C. Lin, Dinesh Manocha 2004, *Fast Computation of Database Operations using Graphics Processors*, SIGMOD Conference 2004, pp 215-226.

[8] N.K. Govindaraju, N. Raghuvanshi, M. Henson, D. Tuft, D. Manocha (2005), *A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors. Tech. Rep. TR05-016*, University of North Carolina.

[9] GPUBench, http://graphics.stanford.edu/projects/gpubench/.

[10] Martin Kraus and Thomas Ertl (2001), *Cell-projection of cyclic meshes*, VIS '01: Proceedings of the conference on Visualization '01, pp 215–222.

[11] Philippe Lacroute and Marc Levoy (1994), *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*, Computer Graphics, volume 28, Annual Conference Series, pp 451-458.

[12] , Warren M. Lam and Jerome M. Shapiro (1994), *A Class of Fast Algorithms for the Peano-Hilbert Space-Filling Curve*, Proc. ICIP 94, volume 1, pp 638-641.

[13] Jonathan K. Lawder and Peter J. H. King (2001), *Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve*, Lecture Notes in Computer Science, Volume 30, Number 1, pp 19-24.

[14] J. K. Lawder and P. J. H. King (2000), *Using Space-Filling Curves for Multi-dimensional Indexing*, Lecture Notes in Computer Science, volume 1832, pp 20-??.

[15] Marc Levoy (1990), *Efficient ray tracing of volume data*, ACM Trans. Graph., volume 9, number 3, pp 245-261.

[16] Vincent C. H. Ma and Michael D. McCool (2002), *Low Latency Photon Mapping Using Block Hashing*, SIGGRAPH/Eurographics Graphics Hardware Workshop, editors Thomas Ertl and Wolfgang Heidrich and Michael Doggett, pp 89-98.

[17] Joel McCormack and Robert McNamara (2000), *Tiled polygon traversal using half-plane edge functions*, HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pp 15-21.

[18] B. Moghaddam, K.J. Hintz and C.V. Stewart (1991) *Space-filling curves for image compression*, Proc. SPIE Vol. 1471, p. 414-421, Automatic Object Recognition, Firooz A. Sadjadi; Ed., pp 414-421.

[19] Torsten Möller, Raghu Machiraju, Klaus Mueller, Roni Yagel (1997), *Evaluation and Design of Filters Using a Taylor Series Expansion*, IEEE Trans. Vis. Comput. Graph. , pp 184-199.

[20] Bongki Moon, H.v. Jagadish, Christos Faloutsos, Joel H. Saltz (2001), *Analysis of the Clustering Properties of the Hilbert Space-Filling Curve*, IEEE Transactions on Knowledge and Data Engineering, Volume 13, pp 124-141.

[21] Michael Meissner, Jian Huang, Dirk Bartz, Klaus Mueller, Roger Crawfis (2000), *A practical evaluation of popular volume rendering algorithms*, VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization, pp 81-90.

[22] Neophytos Neophytou and Klaus Mueller (2005), *GPU Accelerated Image Aligned Splatting*, Volume Graphics, pp 197-205.

[23] Valerio Pascucci and Randall J. Frank (2001), *Global Static Indexing for Real-time Exploration of Very Large Regular Grids*, Proceedings of Super Computing 2001, (Online proceedings), http://www.sc2001.org/techpaper.shtml.

[24] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, T. Ertl (2000), *Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization*, HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, pp 109-118.

[25] Stefan Röttger and Thomas Ertl (2002), *A two-step approach for interactive pre-integrated volume rendering of unstructured grids*, VolVis 2002, pp 23-28.

[26] Hans-Peter Seidel (1993), *Polar Forms for Geometrically Continuous Spline Curves of Arbitrary Degree*, Journal TOG, Volume 12, Number 1, January 1993, pp 1-34.

[27] P. Shirley and A. A. Tuchman (1990), *Polygonal Approximation to Direct Scalar Volume Rendering*, Proceedings San Diego Workshop on Volume Visualization, Computer Graphics, volume 24, pp 63-70.

[28] Jens Krüger and Rüdiger Westermann (2003), *Acceleration Techniques for GPU-based Volume Rendering*, IEEE Visualization 2003, pp 287-292.

[29] Simon Stegmaier, Magnus Strengert, Thomas Klein, Thomas Ertl (2005), *A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting*, Volume Graphics 2005, pp 187-195.

[30] Velho, L. and de Miranda Gomes, J. (1991), Digital Halftoning With Space Filling Curves, SIGGRAPH, Volume 91, pp 81-90.

[31] Douglas Voorhies (1991), *Space-Filling Curves and a Measure of Coherence*, Graphics Gems 2, pp 26-30.

[32] Lujin Wang, Ye Zhao, Klaus Mueller, Arie E. Kaufman (2005), *The Magic Volume Lens: An Interactive Focus+Context Technique for Volume Rendering*, IEEE Visualization, pp 47.

[33] Lee Alan Westover (1991), *Splatting: a parallel, feed-forward volume rendering algorithm*, PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, University of North Carolina at Chapel Hill.

[34] Fang Xu and Klaus Mueller (2006), *A Comparative Study of Popular Interpolation and Integration Methods for Use in Computed Tomography*, IEEE 2006 International Symposium on Biomedical Imaging (ISBI '06).

[35] Wai Min Yee (2004), *Cache Design for a Hardware Accelerated Sparse Texture Storage System*.