Faculty of Mathematics, Physics and Informatics
Comenius University

Master's Thesis

2005 **Anton Vaško**

Anton Vaško

SIMD Optimization in Volume Rendering

Master's Thesis

supervised by

Doc. Dr. techn. Ing. Miloš Šrámek

Department of Computer Graphics and Image Processing
Faculty of Mathematics, Physics and Informatics
Comenius University
Bratislava

I honestly declare that I wrote this Master's Thesis independently only with the help of listed literature.

Bratislava, 29th April 2005 Anton Vaško

I would like to thank my consultant Doc. Dr. techn. Ing. Miloš Šrámek for his valuable advices, comments and suggestions.

Contents

List of Figures

List of Tables

Chapter 1

Summary

1.1 Aim

Information (data) in the contemporary world is often more valuable than gold. The most common 3D data (volume data) is from computed tomography or magnetic resonance. Such data (if we are capable to process and read them quickly enough) can help doctors by diagnosing without invasive surgery. We can thus see that it is very important to do quick and precise visualisation of 3D data.

Problem, arising when we want to visualize medical 3D data, is its size (512 x 512 x 512 x 16bit = 256 MB !) and expensive calculations during rendering. Therefore, the interactive visualization of 3D data was at the beginning impossible.

The aim of this work is to investigate whether the SIMD capabilities, that are commonly available on the most of currently widespread processors, can decrease the volume rendering times.

According to the Moor's law the processor's speed is doubled every 18 months. The processor's power is thus rapidly growing from year to year. But along with processor's power grows also our demand for quicker and more accurate visualisation of 3D data that are also permanently growing in size. Nowadays, we can observe, that also GPUs' power grows very fast. The incredible power of current GPUs lies mostly in their vertex and pixel shaders. Volume rendering can be also implemented with use of the recent GPUs. In this work we would like to find out whether SIMD optimized volume rendering can beat GPU accelerated volume rendering. So we will answer the following question:

"To do or not to do the SIMD optimization in volume rendering?"

1.2 Abstract

We will implement software and hardware optimized volume rendering algorithm. Therefore, Chapter 2 includes some theoretical background in the area of volume rendering and description of several volume rendering algorithms, together with some software optimization techniques. In this case, under the term 'hardware optimization' we do not mean volume rendering using the modern GPU, but optimization using the CPU extensions (3DNow!, SSE). Chapter 3 describes the basic principles of SIMD optimization that are later in Chapter 4 applied to volume rendering . Chapter 5 discusses results obtained using this implementation. Here we finally answer the question put forward in this Master's Thesis and sketch the possible future work.

Chapter 2

Background

This chapter introduces the basic concepts of volume rendering and gives a brief overview of some important software volume rendering algorithms.

2.1 Basic Concepts

In computer graphics we often work with 2D images. 2D image (picture of mostly square or rectangle shape) is 2D array consisting of small elements called pixels (for picture elements). 3D arrays of voxels (volume elements) like a cube or a box are a 3D analogy of 2D images. They are also called volume data. But like the 2D image has not necessarily to be of a rectangular shape, the volume data could be treated more generally.
We can think of the volume data as sampled scalar functions of three spatial dimensions. This kind of data typically arises e.g. in medical imaging (Computer Tomography – CT, Magnetic Resonance Imaging – MRI, Emission-Computed Tomography – ECT ...). To visualize it we can choose from two basic techniques:

1. **Surface Rendering**
   With this technique we try to fit geometric primitives (triangles) to the sampled data. The main disadvantages of this approach are (see [3], p.29):

   - the model is separated from the original data and thus a great deal of information is lost,
   - a large number of polygons ($\approx 10^6$) is necessary to approximate a surface of even medium complexity, and
   - it is hard to simulate weakly defined surfaces and fuzzy phenomena.

   The great advantage of surface rendering is the native support for triangle meshes in all current graphics accelerators (GPU).

   The alternative to surface rendering is

2. **Volume Rendering**
   With this technique if we want to obtain a 2D image from the 3D volume data we first shade each sample and then project it onto the image plane.

The standard volume rendering pipeline is shown in Figure 2.1(from [1], p. 30).

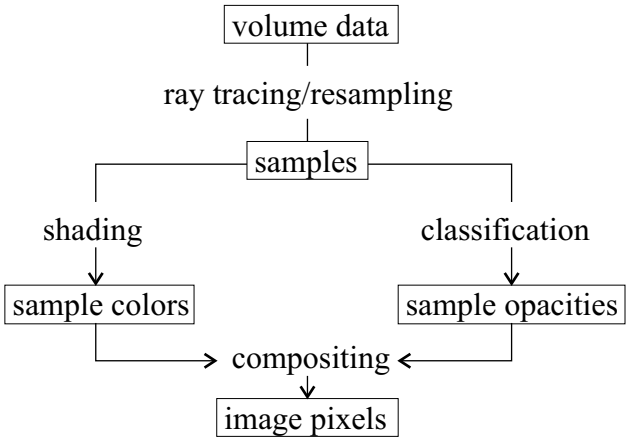

Figure 2.1: *Standard volume rendering pipeline.*

As for shading, Phong[1] shading model is used the most, but it is not the rule. The classification is responsible for visualizing the desired feature of the volume data. Depending on the classification function the volume rendering techniques can be split into:

binary - each sample is assigned a binary value which expresses whether the sample does or does not represent the object of interest. This approach causes loosing of small or poor defined surfaces.

probabilistic - instead of binary decision we assign to each sample its probability or transparency.

Another criterion for classifying the volume rendering algorithms is the domain the algorithm works in. We distinguish

image-order algorithms - they iterate through the image pixels,

object-order algorithms - they iterate through the voxels of the object being rendered, and

hybrid-order algorithms - they combine both the image-order and object-order approaches.

However, the basic two approaches used today are software and hardware volume rendering.

[1]For more details about Phong shading see e.g. [9]

2.2 Software Volume Rendering

Software volume rendering is the approach where the available graphics card (Graphics Processing Unit - GPU) is used only to display the calculated frame. The entire calculations themselves are done on processor (Central Processing Unit - CPU). In this section, there are chronologically arranged and described some of the important milestones in software volume rendering algorithms.

2.2.1 Brute Force

In May 1988 the article Display of Surfaces From Volume Data was published in IEEE Computer Graphics and Applications ([1]). Its author, Marc Levoy, explains there the brute force method of ray casting.

From the observer eyepoint we cast a ray through each pixel of the image plane into the volume data and resample the volume data at K evenly spaced locations $\mathbf{x_i}$ along the ray. Then we calculate a vector of sample colors c_i and opacities α_i by trilinear interpolation from the eight neighbouring (surrounding) voxels closest to the sample location $\mathbf{x_i}$. As the 0-th component of the vector we add fully opaque ($\alpha_0 = 1$) background color. Finally, to obtain the color $\mathbf{C}$ of the pixel the ray was cast through, we use the **over** operator [2]:

$$\begin{aligned} \mathbf{C} &= \sum_{k=0}^{K} c_k \prod_{j=0}^{k-1} 1 - \alpha_j \\ &= c_0 + c_1(1-\alpha_0) + c_2(1-\alpha_0)(1-\alpha_1) + \cdots + c_K(1-\alpha_0)\cdots(1-\alpha_{K-1}) \\ &= c_0 \text{ over } c_1 \text{ over } c_2 \ \cdots \text{ over } c_K \end{aligned}$$

i

The skeleton of the algorithm is

```
for(y=0;y<ImageHeight;y++)
  for(x=0;x<ImageWidth;x++){
    Ray r = CastRayFromEyeThroughImage[x,y];
    Pixels[x,y] = ResampleAndCompositeColorAlongRay(r);
  }
```

Notice that the data are processed from background. Algorithms of this type are called **back-to-front** algorithms. It can be also easily seen from the pseudocode that ray casters belong inherently to image-order algorithms.

2.2.2 Efficient Ray Tracing

In 1990 Marc Levoy published another article - Efficient Ray Tracing of Volume Data [5]. Here we can find the front-to-back (data closer to the eye are processed sooner) ray caster together with two optimization.

The algorithm works as follows: we cast ray through each pixel on the image plane. First, we calculate the vector of colors and opacities by resampling the volume data at evenly

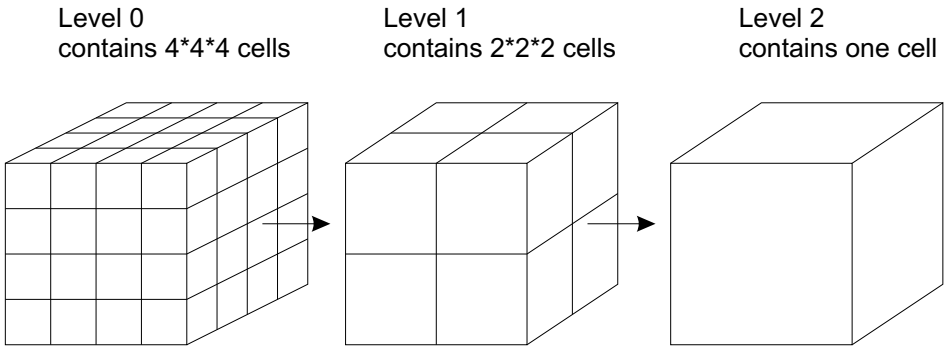

Figure 2.2: *Hierarchical enumeration of object space for cube of size 5^3.*

spaced locations along the ray by trilinear interpolation from the colors and opacities in the eight voxels surrounding each sample location[2]. The final pixel color $\mathbf{C}(u, v)$ of ray $\mathbf{u}$ is obtained by compositing the color and opacity at each sample location from front to back using the **under** operator. Specifically, the color $\mathbf{C}_{out}(\mathbf{u}, \mathbf{U})$ and opacity $\alpha_{out}(\mathbf{u}, \mathbf{U})$ of ray $\mathbf{u}$ after processing the sample $\mathbf{U}$ are related to the color $\mathbf{C}_{in}(\mathbf{u}, \mathbf{U})$ and opacity $\alpha_{in}(\mathbf{u}, \mathbf{U})$ of the ray before processing the sample and the color $\mathbf{C}(\mathbf{U})$ and opacity $\alpha(\mathbf{U})$ of the sample by transparency formula

$$\begin{aligned}\hat{\mathbf{C}}_{out}(\mathbf{u}, \mathbf{U}) &= \hat{\mathbf{C}}_{in}(\mathbf{u}, \mathbf{U}) + \hat{\mathbf{C}}(\mathbf{U})(1 - \alpha_{in}(\mathbf{u}, \mathbf{U})) \\ \alpha_{out}(\mathbf{u}, \mathbf{U}) &= \alpha_{in}(\mathbf{u}, \mathbf{U}) + \alpha(\mathbf{U})(1 - \alpha_{in}(\mathbf{u}, \mathbf{U}))\end{aligned}$$

where $\hat{\mathbf{C}}_{in}(\mathbf{u}, \mathbf{U}) = \mathbf{C}_{in}(\mathbf{u}, \mathbf{U})\alpha_{in}(\mathbf{u}, \mathbf{U})$, $\hat{\mathbf{C}}_{out}(\mathbf{u}, \mathbf{U}) = \mathbf{C}_{out}(\mathbf{u}, \mathbf{U})\alpha_{out}(\mathbf{u}, \mathbf{U})$ and $\hat{\mathbf{C}}(\mathbf{U}) = \mathbf{C}(\mathbf{U})\alpha(\mathbf{U})$.

The first optimization is a *hierarchical spatial enumeration.* The idea is similar to 3D mipmapping and/or hierarchical octree encoding. We can represent this enumeration by a pyramid of volumes V_i. The pyramid is constructed recursively as follows: Every cell in the base volume V_0 (level 0) contains a zero value if all eight voxels lying at its vertices have opacity equal to zero. Every cell in any volume V_i, $(i > 0)$ contains a zero if all eight cells on level $i - 1$ that form its octants contain zero (see Fig. 2.2). Thanks to this representation, we can now quickly advance across the empty region space and do resampling and compositing only when it is necessary.

The second optimization is *adaptive ray termination* of ray casting. While compositing the final pixel color $\mathbf{C}(u, v)$ the opacity $\alpha_{in}(\mathbf{u}, \mathbf{U})$ increases monotonically along the ray whereas the contribution to the pixel color decreases. Hence, no significant color change occurs after exceeding opacity of $1 - \varepsilon$ for small $\varepsilon > 0$. By setting ε to nonzero value we can stop the ray casting earlier, but we can introduce image artifacts. Therefore it is up to us to find a satisfactory compromise.

[2]Interpolation of colors and opacities introduces artifacts. Today, interpolation of densities and gradients is preferred, and opacity and colors are assigned only to these interpolated values.

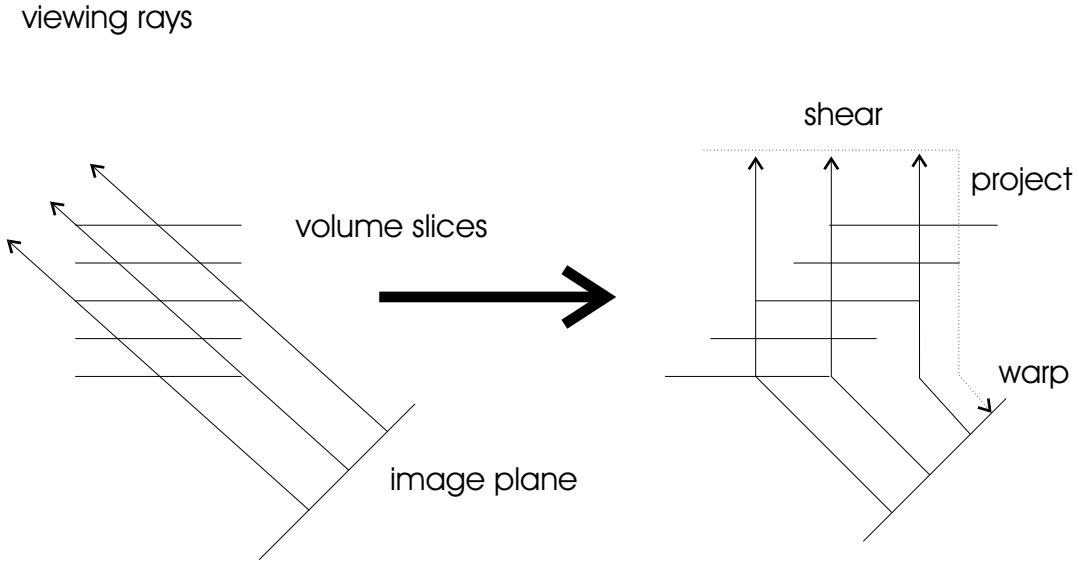

Figure 2.3: *For parallel projection to transform volume into sheared object space it is sufficient to translate each slice.*

2.2.3 Shear-Warp Factorization

On the SIGGRAPH'94, Philippe Lacroute together with Marc Levoy introduced a new family of fast volume rendering algorithms [6]. They combined the advantages of image order algorithms (early ray termination, hierarchical spatial enumeration) with the advantages of object order algorithms (traversing the volume in the storage order). At that time, the algorithms based on shear-warp factorization of the viewing transformation belonged to the fastest known.

Actually, volume rendering is mapping from 3D (volume data) to 2D (image plane). Image-order algorithms iterate over image—they know image coordinates of just the rendered pixel and have to do complicated volume addressing arithmetic together with resampling and compositing. On the other hand, object-order algorithms iterate over volume data (addressing arithmetic is thus very simple) but have to compute convolution of the voxels with the view-dependent filter.

This problem can be solved by choosing a suitable intermediate coordinate system with a simple mapping from the object coordinate system and efficient projection to 2D image. These conditions are met in the sheared object space. For parallel projection, to transform into the sheared object space it is sufficient to shear the volume parallel to the set of slices that is most perpendicular to the viewing direction. For perspective projection after shear we also have to scale and translate each slice (Figures 2.3 and 2.4) (see [4], p. 30-31). In the sheared object space we can project the voxel slices onto an image efficiently.

A simple object-order volume rendering algorithm based on the shear-warp factorization works as follows:

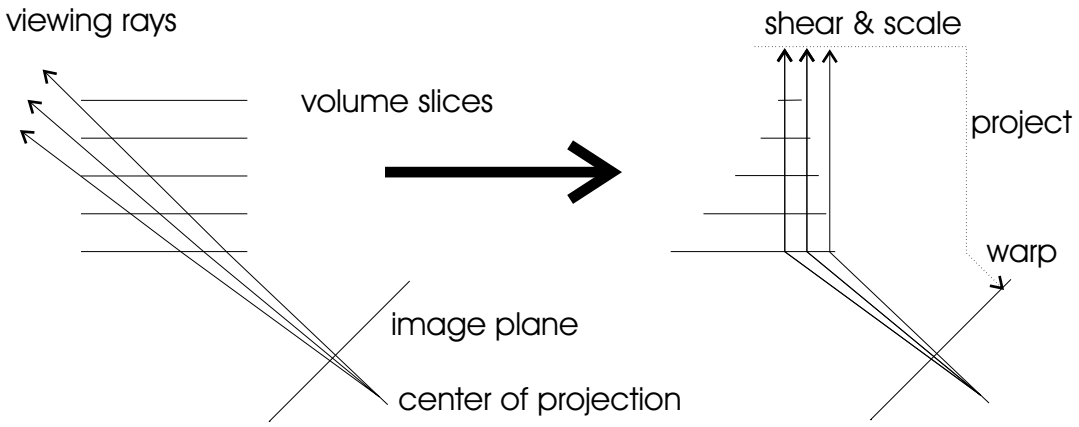

Figure 2.4: *For perspective projection to transform volume into sheared object yet we have to do scaling after translation of each slice.*

1. Transform the volume data to the sheared object space and resample each voxel slice.
2. Composite the resampled slice in front-to-back order using the over operator onto a distorted 2D intermediate image in sheared object space.
3. Transform the distorted intermediate image into final image by warping.

Thanks to the factorization properties, we can now iterate through the volume data in the storage order (thus only once) with very simple addressing arithmetics and 2D resampling filter.

Together with the shear-warp factorization the authors also introduced three optimized shear-warp volume rendering algorithms.

The first two are both for parallel and perspective projection rendering. To skip the runs of transparent voxels (thus taking the advantage of object coherence) they create in a preprocessing step view-independent run-length encoded volume. Also, to skip the long runs of opaque pixels (and so taking the advantage of the image coherence) they are remembering information about opaque pixels during rendering.

Although the run-length encoded volume (precalculated in the preprocessing step) is view-independent, it is transfer function-dependent. This preprocessing step is computationally very expensive. Therefore, the transfer function cannot be changed interactively. To solve this problem, the third algorithm introduces another volume data structure—instead of the run-length encoding they use the min-max octree which is independent of the chosen transfer function. Specifically, in preprocessing step, while loading volume into memory, one can precalculate the min-max octree, because it is opacity transfer function- and view-independent. Then, just before rendering, a summed-area table is computed. The octree and the summed-area table help during rendering to determine all non-transparent voxels.

The disadvantage of this solution is the common problem of the object-order algorithms: if the major viewing axis changes then the volume data must be accessed against the stride

and performance degrades. Therefore it is recommended to use only small range of view angles and for animation rendering to switch to the first or second algorithm.
The main disadvantage of algorithms based on the shear-warp factorization is that they use only bilinear interpolation within slices which deteriorates quality of the rendered imaging by introduction of reconstruction artifacts.

2.2.4 Multiresolution Min-Max Octrees

Lacroute & Levoy have brought the compass of interactive volume rendering closer to the standard desktop PC. The next step (and it seems that the last substantial step in the algorithms based on shear-warp factorization) to enable the interactive volume rendering on a standard PC is multiresolution. Unfortunately, the min-max octree in the original L&L algorithm is resolution-dependent and its computation is expensive, so interactive switching between different resolutions is practically impossible. So F. Dong, M. Krokos and G. Clapworthy designed another structure for capturing min and max parameter values, namely a multiresolution min-max octree [7].
Multiresolution data is stored in a pyramid form with a hierarchical data structure. At a particular dataset resolution, any voxel corresponds to a cell of 2x2x2 voxels at the higher resolution. To encode volume dataset in a multiresolution representation, the ADS (*averaging and differencing scheme*) is used (see Section 4.1 in [7]).
Multiresolution is an excellent tool for trading quality for speed. It is up to the user, whether he or she chooses better quality or higher rendering speed. Moreover, if the user is working with a large dataset and the PC is not keeping up the rendering, he or she can select the lower resolution and after choosing the right transfer function and view direction the higher resolution for the final image can be used.

2.2.5 Parallel Ray Casting

In [9], additional techniques aimed at acceleration of volumetric ray casting were introduced.
First, the volume is bricked into smaller blocks. Blocks are then sorted in front-to-back order according to the current viewing direction. The ordered blocks are grouped into levels in such a way that for every level (group of blocks) the following condition holds: There does not exist any ray from a block that intersects another block from the same group. In other words, if a ray belongs to block A and intersects block B, then blocks A and B cannot belong to the same level (see Fig. 2.5). This condition ensures that all blocks on the same level are independent and thus can be processed in parallel.
The ray-casting itself begins with initializing rays. To quickly skip empty space (transparent voxels), hierarchical octree according to the current classification is built and the *Entry Point Buffer (EPB)* is calculated (EPB is simply a buffer that holds, for each pixel, the distance from the camera to the appropriate non-transparent voxel—if such exists). After rays are initialized from EPB, they travel through the volume. Because the front-to-back

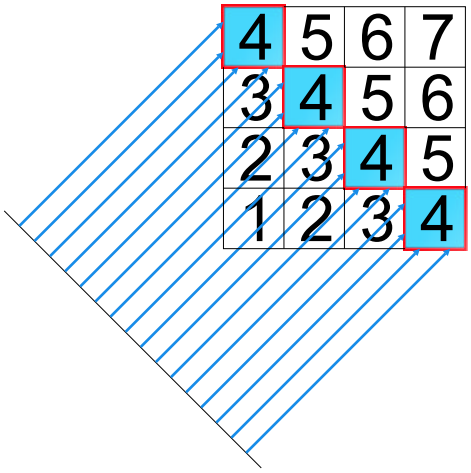

Figure 2.5: *The 2D version of blocks grouping. 16 blocks are divided into 7 levels.*

accumulation is used, the *alpha* value increases monotonically from initially zero (fully transparent) to one (fully opaque) and early ray termination can be used for speed up (see Section 2.2.2).
Although this algorithm was primarily designed for MIMD computers, its ideas are general and give good results on computers with one CPU, too. Perhaps the only thing that can be hold against this approach, is that for performance reasons parallel projection is used.

2.3 Hardware Volume Rendering

Interesting trends in computer technology can be noticed nowadays. The current CPUs are developed slower than current GPUs. Whereas the CPU development is in a moderate slump (the maximal CPU frequency is about 3.4 GHz and is not rising extremely) current GPUs are witnessing big revolution. (Something similar can be observed also by comparing CPU and RAM, where the gap in frequency between new processors and memories is even more visible.) Therefore, the number of GPUs which can handle *3D textures* is permanently growing. And especially this GPU feature offers a new way to visualize volume data.
There are several algorithms that visualize volume using the GPU support (see e.g. [11] or [12]). The basic idea is that the volume is stored as a single 3D texture. Then, set of planes which are parallel to the image plane with suitable texture coordinates is rendered back-to-front. The final frame is composed through alpha blending (see Fig. 2.6).
Although the hardware assisted volume rendering can be very fast it can suffer regarding quality and is binded tightly to graphic accelerator. As this work deals with pure software volume rendering we will not be concerned with greater details (see e.g. [13]).

Figure 2.6: *3D Textures.*

2.4 Conclusion

As we could see, there are many ways to visualize volume data. The hardware volume rendering reaches very interesting frame rates when working with datasets that fit into GPU memory. However, we are concerned with software methods for volume rendering as they are universal (they run practically on every CPU) and do not need a most recent GPU with 3D texture support.

In Section 2.2 we introduced couple of software rendering algorithms. They all have some advantages and disadvantages. The Brute Force approach (see Section 2.2.1) is totally unsuitable because current CPUs cannot handle volume data of typical sizes (256∗256∗256) within a reasonable time. The shear-warp factorization of the viewing transformation (see Section 2.2.3) was very popular in the past. The reason is quite simple. At that time, the CPU and RAM have run roughly at the same speed, so it was worth to precalculate everything possible and store it in the memory, because later querying for the precalculated data was faster rather then calculate it on the fly. But nowadays, especially querying for the precalculated data is the bottleneck of rendering algorithms, because the memory is not fast enough to feed the CPU with data (the buses are slow) and the caches are to small to hold everything precalculated. The locality concept (see Section 4.3) is significantly violated and the results are poor. Another reason is that shear-warp does only bilinear interpolation within slices. The Multiresolution Min-Max Octrees technique (Section 2.2.4 is also based on the shear-warp transform and suffers from the same problems. Therefore, we have decided to implement and further optimize the Stefan Bruckner's approach (Section 2.2.5). It is a simple ray casting, but suitable for parallel computations, which is giving interesting results on current desktop computers.

Chapter 3

SIMD

This chapter describes various SIMD technologies. The first subsection deals shortly with SIMD technologies (theoretically) which are available today on the overwhelming majority of desktop processors. In the second subsection we give practical examples demonstrating contribution of SIMD technologies to the 'classical' SISD computers.

3.1 Theoretical SIMD

A widely used classification of parallel systems, due to Michael J. Flynn [23], is based on the number of simultaneous instruction and data streams seen by the processor during program execution . In the case we have only one data stream we are talking about SISD (Single Instruction stream, Single Data stream) and MISD (Multiple Instruction stream, Single Data stream). If we have more than one data stream, we are talking about SIMD (Single Instruction stream, Multiple Data stream) and MIMD (Multiple Instruction stream, Multiple Data stream).
Apart from modern processors' features like out-of-order execution or multiple decode and fetch units, the common desktop computers and notebooks are inherently SISD[1]. However, they bear some SIMD features. To illustrate the difference between SISD and SIMD, consider some binary operation $*$ (e.g. plus or logical and). A SISD processor is capable to perform only one single instruction on one single data stream, $z = x * y$, whereas one single instruction of a SIMD processor works on more (typically from two to eight) data streams in parallel, see Fig. 3.1. This leads to significant speedup without increasing processor's core speed. The most widespread processors today are 32-bit ones from Intel [21] and AMD [20]. Therefore, we will concentrate on these two CPU families. They support SIMD in the form of extended instruction set and new registers. The technologies standing behind are known as MMX, SSE, SSE2, SSE3 and 3DNow!.

[1] except for the newest Intel processors with the HyperThreading technology

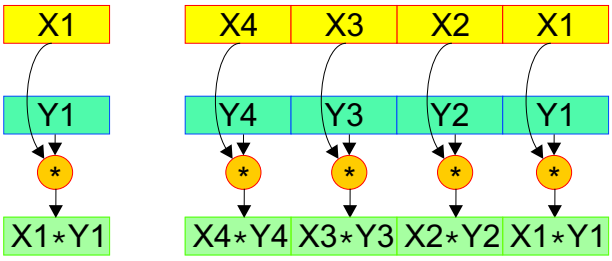

Figure 3.1: *Comparison of SISD (left) and SIMD (right) model.*

3.1.1 MMX

MMX(**M**ulti**M**edia e**X**tension) can be thought of as a pioneer in SIMD. This technology was first used in Intel Pentium processors.

MMX offers 8 logically new registers called *mm*0–*mm*7. They are 64 bits wide and physically mirrored on the FPU stack (Fig. 3.2). Together with the new registers Intel introduced four new integer data types. One MMX register can hold 8 packed bytes, 4 words, 2 doublewords or 1 quadword (Fig. 3.3). 57 new instructions extend the old IA instruction set to exploit the capabilities of the new registers. The new instruction types are

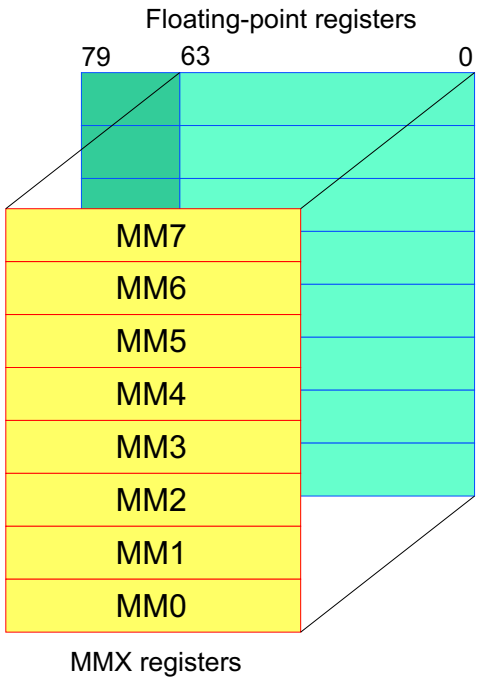

Figure 3.2: *MMX registers are in fact just lower parts of 80-bit FP registers.*

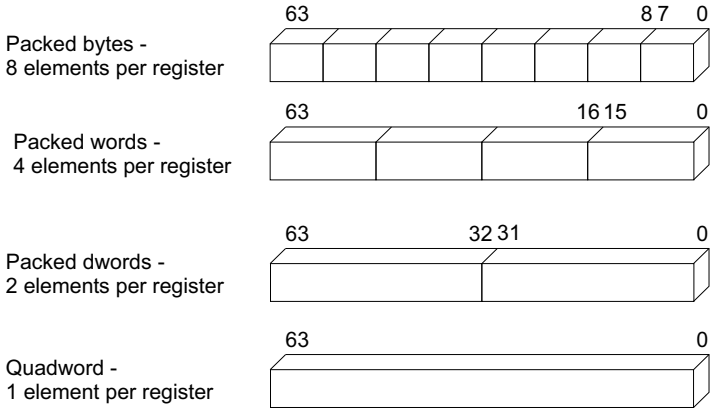

Figure 3.3: *MMX data types.*

packed arithmetic - `padd, psub, pmul, pmadd, ...`

data manipulation - `movq, pack, punpck, psr, psl, ...`

logical - `pand, pandn, por, pxor, ...`

saturating arithmetic - `padds, psubs, ...`

3.1.2 SSE

SSE (**S**treaming **S**IMD **E**xtension) is descendant of MMX. It is available on all Intel Pentium III (and higher) processors and on some AMD processors. SSE provides 8 physically new 128-bit registers (called $xmm0$ - $xmm7$) and 70 new instructions. While the whole MMX technology was integer(or fixed-point) based, SSE is inherently floating-point based. It is evident from the new instructions' constitution–up to 50 of them are designed to work with floating-point data, 8 instructions control cacheability and the remaining 12 enhance MMX instruction set. The instructions can be divided into the following groups:

- *Arithmetic instructions* - `addps,subps, mulps, divps, sqrtps, ...`
- *Logical instructions* - `andps, andnps, orps, xorps, ...`
- *Reciprocal instructions* - `rcpps, rsqrtps, ...`
- *Conversion instructions* - `cvtpi2ps, cvtps2pi, ...`
- *Comparison instructions* - `maxps, minps, cmpeqps, cmpltps, ...`
- *Shuffle instructions* - `shufps, unpcklhps, unpckhps, ...`
- *Data movement* - `movaps, movhps, movhlps, ...`

- *Cacheability control instructions* - `prefetch, sfence, movntq, ...`
- *State management instructions* - `ldmxcsr, stmxcsr, ...`
- *Additional SIMD integer instructions* - `pshufw, pminsw, ...`

3.1.3 SSE2

SSE2 is Intel's second **S**treaming **S**IMD **E**xtension. It was first introduced in Pentium IV processors. Whereas SSE worked on 128-bit registers as on four single-precision floating-point registers in parallel, SSE2 can work on the same 128-bit register as on two double-precision floating-point numbers. So, SSE2 is focused mainly for applications that require double-precision calculations. Although the effective speedup decreased from four to 'only' two when using SSE2, the real speedup can be even more than two times if we consider that the old FP unit suffers much longer latencies than the streamlined SSE2 unit. SSE2 also extends MMX by using 128-bit registers instead of 64-bit ones. A brief list of new instructions follows.

Double-precision:

- *Arithmetic instructions* - `addpd,subpd, mulpd, divpd, sqrtpd, ...`
- *Logical instructions* - `andpd, andnpd, orpd, xorpd, ...`
- *Conversion instructions* - `cvtpi2pd, cvtps2dq, ...`
- *Comparison instructions* - `maxpd, minpd, cmpeqpd, cmpltpd, ...`
- *Shuffle instructions* - `shufpd, unpckhpd, unpcklpd, ...`
- *Data movement* - `movapd, movhpd, movmskpd, ...`

'128-bit MMX':

- *Arithmetic instructions* - `paddq, psubq, pmuludq, ...`
- *Data manipulation* - `movq2dq, punpckhqdq, psrldq, pslldq, ...`
- *Logical instructions* - `pand, pandn, por, pxor, ...`
- *Shuffle instructions* - `pshufhw, pshuflw, pshufd, ...`

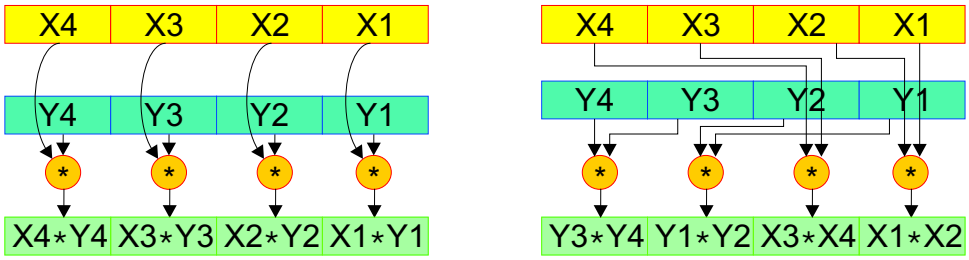

Figure 3.4: *Vertical (left) and horizontal (right) computation model.*

3.1.4 SSE3

SSE3 is the last extension of the SSE's instruction set. It is privilege only of the most recent Intel Pentium 4 processors. It enhances SSE and SSE2 with 9 new SIMD floating-point instructions. Contrary to the SSE or SSE2 sets, which prefer homogeneous arithmetic operations on parallel data elements (see Fig. 3.1), SSE3 instructions can perform asymmetric arithmetic operations and arithmetic operations on horizontal data elements (so-called horizontal computation model, see Fig. 3.4).
The 9 new SIMD instructions can be divided into:

- *Asymmetric arithmetic instructions* - `addsubps, addsubpd`
- *Horizontal arithmetic instructions* - `haddps,hsubps,haddpd,hsubpd`
- *Data manipulation* - `movsldup, movshdup, movddup`

Figure 3.5 demonstrates *asymmetric arithmetic instruction* `addsubps`.
Horizontal arithmetic instructions facilitate calculation of numerous algebraic formulas without swizzling the data between SOA and AOS type representation (Section 3.2). For example, instruction `haddps` (substitute + for * in Fig. 3.4) is similar to the 3DNow! accumulation instruction `pfacc` (see later in this chapter).

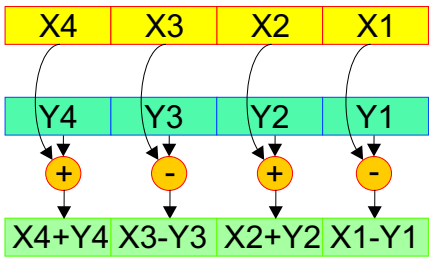

Figure 3.5: *SSE3 asymmetric arithmetic instruction addsubps.*

Data manipulation instructions help with common problem of data swizzling[2] when trying to use SIMD instructions. They help to replicate data elements already at the loading time. For example,

```
movsldup xmm0, [a]   ; a[1] a[1] a[0] a[0]
```

loads and duplicates two values `a[0]` and `a[1]` (assuming that `a` is a single floating-point array pointer).

3.1.5 3DNow!

3DNow! is AMD's SIMD technology. It is a mixed MMX and SSE. 3DNow! uses 8 logically new 64-bit registers which are mirrored on floating-point registers. Unlike MMX, 17 of 21 new instructions operate on these registers as on two 32-bit single-precision floating-point numbers. New groups of instructions, that enrich the original MMX instruction set, are:

- *Arithmetic instructions* - `pfadd, pfacc, pfsub, pfmul, ...`
- *Reciprocal instructions* - `pfrcp, pfrsqrt, pfrcpit1, ...`
- *Conversion instructions* - `pi2fd, pf2id`
- *Comparison instructions* - `pfmax, pfmin, pfcmpgt, ...`
- *Cacheability control instructions* - `prefetch, sfence ...`
- *State management instructions* - `femms`
- *Additional SIMD integer instructions* - `pavgsub, pmulhrw`

3.2 Practical SIMD

As we have seen, SIMD optimization is especially suitable for algorithms that are processing a lot of data. They often contain conditional assignments and long but quite simple loops. Exactly these parts can be speeded up by removing branches and processing more data in parallel with the help of SIMD. However, SIMD increases algorithm data throughput, and therefore along with SIMD optimization also cache consideration comes into play. Although current compilers are very smart and can do various optimizations, they still often do not optimize for SIMD (or the optimization is very poor). Therefore, all these optimizations have to be written manually by programmers and mostly in assembler.
The natural data structure often used in 3D geometry computations is AOS (array of structures) [14]:

[2]Data swizzling is the operation which rearranges the data from AOS into SOA form, see Section 3.2

```
struct Vector3f_AOS{
    float x,y,z,w;
}
Vector3f_AOS *verts;
```

To process data stored in this form, we have to use the horizontal computation model (e.g. SSE3 or some 3DNow! instructions). However, still the most widespread instruction set extensions are SSE, SSE2 and 3DNow! that work more efficiently if the data are stored in another form—SOA (structure of arrays):

```
struct Vector3f_SOA{
    float x[4], float y[4], float z[4], float w[4];
}
```

In order to achieve the full utilization of the SIMD registers when the instructions are designed for vertical computation, we have to prepare the data in this 'vertical-friendly' form. Rearranging the data from AOS into SOA form is the operation referred to as "swizzling", whereas the reverse operation (used after computation before storing results again in the AOS form) is referred to as "deswizzling".
Now, let us look at the advantages and disadvantages of some of the above-mentioned SIMD technologies.

3.2.1 MMX

MMX was the first SIMD technology, that spread around and was used. Because the registers are physically mirrored on FPU stack, the MMX- and floating-point- code cannot be mixed together. Switching between using MMX and FPU code (by the `emms` instruction) costs about 50 cycles, one must therefore carefully consider reorganizing the code or use fixed-point arithmetic. The instructions are prefixed with `p` (packed) and suffixed with `b`, `w` or `d` according to the data type you are using (8 bytes, 4 words or 2 doublewords). There are two modes in MMX arithmetic. In the saturation mode, the result never overflows or underflows, however, it is clipped to maximal or minimal value respectively. This mode is especially useful in many applications, e.g. when doing color accumulation. The wraparound mode causes truncation of the result after overflow or underflow. Typically MMX (and SIMD in general) is used in wide range of applications, e.g. image and speech processing, MPEG video, games, etc. The following very simple example shows how to speed up the standard C function `memset` using MMX:

```
;=============================================================
; void memsetMMX(void* dest, int value, int dwSize);
;
memsetMMX:

    emms                ; clean up MMX and FP state
    ;edx = dest
    ;mm0 contains [value | value]
    ;eax is counter
    ...

align 16
.loop:                  ; eight times unrolled loop
    movntq [edx],mm0    ; movntq writes into memory
    movntq [edx+8],mm0  ;  without cache pollution
    movntq [edx+16],mm0
    movntq [edx+24],mm0
    movntq [edx+32],mm0
    movntq [edx+40],mm0
    movntq [edx+48],mm0
    movntq [edx+56],mm0

    add      edx,64     ; advance dest pointer
    dec     eax         ; advance counter
    jnz      .loop      ; and loop while not end

    sfence              ; due to movntq's
    emms                ; clean up MMX and FP state
    ret                 ; return
;
; end of memsetMMX
;=============================================================
```

3.2.2 SSE

As we have mentioned earlier, SSE introduced 8 physically new registers $xmm0$ - $xmm7$. So, together with MMX registers, SSE places additional 16 general-purpose registers at programmer's disposal. The algorithms now can use packed, single-precision, floating-point and integer (both MMX and SSE) instructions respectively. Since the registers are 128 bits wide, the SSE instructions can operate on four single-precision floating-point numbers in parallel (these instructions are suffixed `ps` for packed single). We can also use scalar single version of these instructions (suffixed `ss`) which operate only on the last 32 bits of

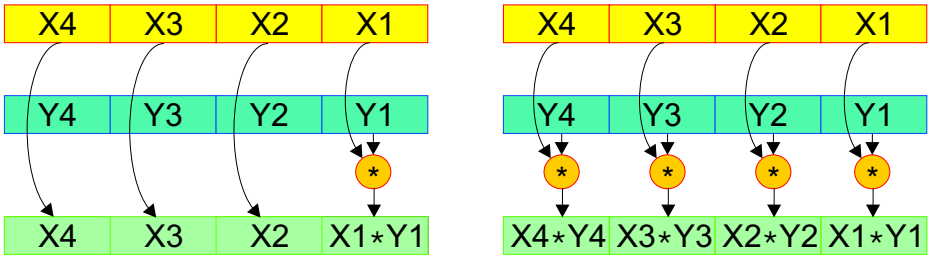

Figure 3.6: *Difference between scalar (left) and packed (right) SSE instructions.*

xmm register (see Fig. 3.6).
To demonstrate the power of SSE consider simple matrix-vector transformation often used in computer graphic:

$$(x', y', z', w')^T = \begin{pmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

The code performing this transformation follows:

```
;============================================================
; xformVectorSSE(const float* m, float* vector);
;
xformVectorSSE:
    ;edx contains pointer to vector
    ;eax contains pointer to matrix m
    ...
; load matrix
    movaps xmm0,[eax]           ; [m3  m2  m1  m0]
    movaps xmm1,[eax+16]        ; [m7  m6  m5  m4]
    movaps xmm2,[eax+32]        ; [m11 m10 m9  m8]
    movaps xmm3,[eax+48]        ; [m15 m14 m13 m12]
; load vector
    movaps xmm4,[edx]           ; [w z y x]
    movaps xmm5,xmm4            ; [w z y x]
    movaps xmm6,xmm4            ; [w z y x]
    movaps xmm7,xmm5            ; [w z y x]
; swizzle the data - prepare vectors in the apposite form
    shufps xmm4,xmm4,0          ; [x x x x]
    shufps xmm5,xmm5,01010101b  ; [y y y y]
    shufps xmm6,xmm6,10101010b  ; [z z z z]
    shufps xmm7,xmm7,11111111b  ; [w w w w]
```

```
; multiply - very nice
    mulps xmm4,xmm0  ; [x*m3  x*m2  x*m1  x*m0]
    mulps xmm5,xmm1  ; [y*m7  y*m6  y*m5  y*m4]
    mulps xmm6,xmm2  ; [z*m11 z*m10 z*m9  z*m8]
    mulps xmm7,xmm3  ; [w*m15 w*m14 w*m13 w*m12]
; and accumulate into result
    addps xmm4,xmm5  ;
    addps xmm6,xmm7  ;
    addps xmm4,xmm6  ; [w' z' y' x']
; store result and return
    movaps [edx],xmm4
    ret
;
; end of xformVectorSSE
;============================================================
```

3.2.3 3DNow!

Since 3DNow! uses the MMX registers, they are called $mm0$ - $mm7$, too. The instructions are prefixed with `pf` (packed float) and operate on two single-precision floating-point numbers in parallel. 3DNow! extended MMX by 21 new instructions that are very similar to SSE instructions. Therefore, we would like to emphasize only two of them, which we consider especially interesting. Firstly, since 3DNow! is only extended MMX, we have to carefully switch between using FPU and MMX registers. To minimize penalty for the switch, we can use `femms` instead of `emms`, which is executed in two cycles only. Secondly, 3DNow! contains one very useful instruction - `pfacc`. It accumulates the high and low part of the destination and source registers. We are sure that when optimizing for both 3DNow! and SSE one will often miss similar instruction in the SSE instruction set.

The following example demonstrates that—due to the width of the registers—optimizing for 3DNow! is a little bit harder and less readable than for SSE (matrix and vector are the same as in the last example):

```
;============================================================
; xformVector3DNow(const float* m, float* vector);
;
xformVector3DNow:
    ;edx contains pointer to vector
    ;eax contains pointer to matrix m
    ...
    femms               ;clear MMX state

    movq mm0, [edx]    ; y | x
    movd mm1, [edx+8]  ; 0 | z
```

```
    movq mm2, mm0      ; y | x
    movq mm3, [eax]    ; a1 | a0
    punpckldq mm0, mm0 ; x | x
    movq mm4, [eax+16] ; a5 | a4
    pfmul mm3, mm0     ; x*a1 | x*a0
    punpckhdq mm2, mm2 ; y | y
    pfmul mm4, mm2     ; y*a5 | y*a4
    movq mm5, [eax+8]  ; a3 | a2
    movq mm7, [eax+24] ; a7 | a6
    movq mm6, mm1      ; w | z
    pfmul mm5, mm0     ; x*a3 | x*a2
    movq mm0, [eax+32] ; a9 | a8
    punpckldq mm1, mm1 ; z | z
    pfmul mm7, mm2     ; y*a7 | y*a6
    movq mm2, [eax+40] ; a11 | a10
    pfmul mm0, mm1     ; z*a9 | z*a8
    pfadd mm3, mm4     ; x*a1+y*a5 | x*a0+y*a4
    movq mm4, [eax+48] ; a13 | a12
    pfmul mm2, mm1     ; z*a11 | z*a10
    pfadd mm5, mm7     ; x*a3+y*a7  | x*a2+y*a6
    movq mm1, [eax+56] ; a15 | a14
    puncpckhdq mm6,mm6 ; w | w
    pfadd mm3, mm0     ; x*a1+y*a5+z*a9 | x*a0+y*a4+z*a8
    pfmul mm4,mm6      ; a13*w | a12*w
    pfmul mm1,mm6      ; a15*w | a14*w
    pfadd mm5, mm2     ; x*a3+y*a7+z*a11 | x*a2+y*a6+z*a10

    ;x*a1+y*a5+z*a9+w*a13 | x*a0+y*a4+z*a8+w*a12 = [y' x']
    pfadd mm3, mm4     ; [y' | x']
    movq [edx], mm3    ; store [y' | x']

    ;x*a3+y*a7+z*a11+w*a15 | x*a2+y*a6+z*a10+w*a14 = [w' z']
    pfadd mm5, mm1     ; [w' | z']
    movq [edx+8], mm5  ; store [w' | z']

    femms              ; clear MMX state
    ret
;
; end of xformVector3DNow
;============================================================
```

3.3 Conclusion

Current processors support various SIMD technologies. We find the SSE and 3DNow! to be the most important ones. SSE is present on all nowadays manufactured Intel desktop processors and on some AMD processors, too. 3DNow! is implemented in all current 32-bit AMD processors. Both are floating-point based and support the older MMX technology, too. Therefore, we decided to optimize the algorithm in the next section for these two technologies.

Chapter 4

SIMD Optimization in Volume Rendering

This chapter describes SIMD optimization of ray-casting used in volume rendering. The first section specifies software requirements and summarizes the decisions we have made when more choices were possible. The next sections introduce some of the general optimizations techniques that have been used. Section 4.3 emphasizes the importance of the bricking technique. The fourth section shows how the SIMD optimization principles apply to volumetric ray-casting. Experimental results obtained with different optimized and non-optimized algorithms are summarized in Chapter 5.

4.1 Software Specification

Our task is to write a simple program that should interactively visualize volume data, and optimize it (with respect to SIMD), for the two chosen families of processors. This program will be used to analyze suitability of the volume rendering problem for SIMD optimization. It will also help us to answer the main question put forward in this Master's Thesis, namely: *"To do or not to do the SIMD optimization in volume rendering?"*.

The GUI of the program will offer only a minimal functionality. Various parameters such as CPU optimization type or brick size can be set to observe their impact on the final rendering time.

The program should be platform independent and as we decided in Section 3.3 it will be written in three versions–non optimized, optimized for SSE and for 3DNow!. If it is possible, there should be one version of the executable for interactive comparison of the non-optimized and optimized algorithms and—if the CPU allows it—for comparison of the SSE and 3DNow! optimized versions between each other.

4.1.1 Implementation Notes

As was mentioned earlier, we have decided to implement the Bruckner's approach (see Section 2.4).
There are more 3D formats for storing volume data, e.g. DICOM [15]. We have chosen **f3d** [16] because of its simplicity and efficiency.
As it is usual in computer graphics, the major code will be written in ANSI C++. Thanks to this, our program should be platform (Windows/Linux) independent. However, to exploit the SIMD features, the core will be written in assembler. There are more alternatives, e.g TASM [17] or MASM [18] but we have decided for **NASM** [19]. It is free, platform independent, has simple syntax and what is its main advantage, it natively recognizes MMX, 3DNow! and SSE/SSE2 instructions.
To be platform independent also with GUI, the **wxWindows** library [22] will be used for user interface programming.

4.2 General Optimization

One of the basic optimization rules when working with memory is to align data on natural operand size address boundaries (see [14], page 2-28). According to this rule, addresses for MMX and 3DNow! should be aligned on 8 byte boundaries (MMX and 3DNow! registers are 64 bits wide) and for SSE (which has 128 bits wide registers) the addresses should be multiple of sixteen. The common solution is to align important addresses on 16 byte boundaries so both 3DNow! and SSE can load and store data without incurring significant performance penalties.
We also have to mention the well known classical optimization rule the *'Loop Unrolling'* (see [14]). The benefits are obvious: after unrolling a loop the number of branches (if-tests) significantly decreases, thus reducing the possibility of pipeline flush occurrences. Unrolling also enables to hide latencies, as mostly important instructions (such as `fmul` or all SIMD instructions) today are very well pipelined. Consider for example multiplying of two arrays:

```
for(int i=0;i<N;++i)
    c[i]=a[i]*b[i];
```

The loop body has to load (`fld`) a[i], multiply it with b[i] (`fmul`) and store the result (`fst`), which takes about $1 + 3 + 2 = 6$ cycles per one array element. Rewriting into assembler (for 3DNow!) may look as follows:

```
mov    ecx,N          ; number of iterations (assuming that
shr    N,1            ;   N is even - processing 2 per loop)
.loop
    movq mm0, [a]     ;  1 (2) load first operand
    movq mm1, [b]     ;  2 (2) load second operand
```

```
    pfmul mm0, mm1     ;  4 (4) multiply them
    movq [c], mm0      ;  8 (2) and store result

    dec  ecx           ; decrement counter
    jnz  .loop         ; repeat again if not zero
```

(Notice that result of multiplication is not available until cycle 8, because `pfmul` has latency of four.) This version takes about 10 cycles to process two array elements, giving 5 cycles for one array element. After unrolling four-times the code may look like this:

```
mov ecx,N              ; number of iterations (assuming that
shr ecx,3              ;   N is divisible by 8)
.loop
    movq mm0,[a]       ;  1 (2) load operands ...
    movq mm1,[a+8]     ;  2 (2)
    movq mm2,[a+16]    ;  3 (2)
    movq mm3,[a+24]    ;  4 (2)
    movq mm4,[b]       ;  5 (2)
    movq mm5,[b+8]     ;  6 (2)
    movq mm6,[b+16]    ;  7 (2)
    movq mm7,[b+24]    ;  8 (2)

    pfmul mm0,mm4      ;  9 (4) multiply them ...
    pfmul mm1,mm5      ; 10 (4)
    pfmul mm2,mm6      ; 11 (4)
    pfmul mm3,mm7      ; 12 (4)

    movq [c],mm0       ; 13 (2) and store results...
    movq [c+8],mm1     ; 14 (2)
    movq [c+16],mm2    ; 15 (2)
    movq [c+24],mm3    ; 16 (2)

    dec ecx
    jnz .loop
```

This version processes eight array elements in about 18 cycles, which means 2.25 per one element!

However, excessive unrolling, especially unrolling of large loops can lead to increased code size and decreased performance, if the unrolled loop no longer fits in the cache.

Another optimization rule is *prefetching*. It is closely related to the use of the cache (see Section 4.3). If we have some 'SIMD-natural' loop, i.e. loop which is hungry for data, we can add (usually at the end of the loop body before decreasing the counter and condition jump) additional instruction (some from the `prefetch` family) which can ensure, that

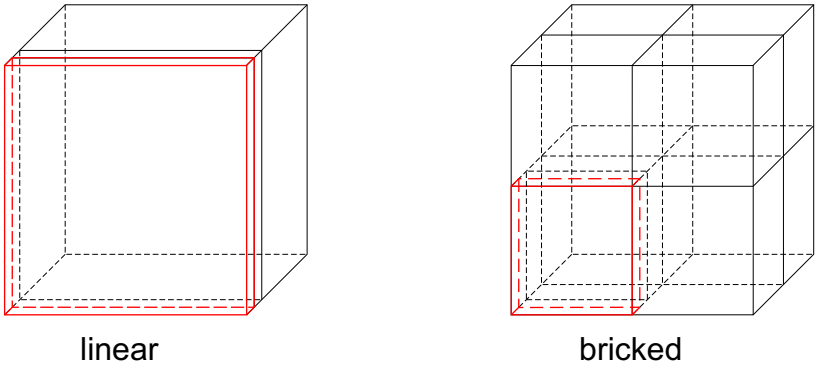

Figure 4.1: *Volume layouts. Linear volume stores slice after slice. Bricked volume stores block after block, each block is stored linear.*

after jumping again into the loop body the demanded data will be already in cache—this is so-called prefetching. This way we can avoid many 'cache-miss' events (Section 4.3).
We can start the whole optimization with the two basic functions `memcpy` and `memset`. It is important to work with memory as quickly as possible, and this functions are very well optimizable for SIMD (see results in Chapter 5).
The SIMD optimization is especially suitable for batch-like tasks. We have tried to find this 'batch-pattern' in all important parts of ray-casting. The results showed that this is the right way for speeding up the calculations times, even when SIMD optimization does not follow immediately.

4.3 Bricking

The increasing gap in speed between CPU and memory is often the bottleneck in current real-time application. This is particularly true when doing SIMD optimization because the SIMD instructions naturally tend to increase algorithm's data throughput. This bottleneck can be moderated by properly using the CPU caches. The cache is small (typically 16 - 2048 kB) but fast memory between the CPU and RAM that can cover the RAM speed limits. The reason why to consider cache optimization is apparent from the *locality concept*. "Temporal locality is the idea that if an item is referenced, it will tend to be referenced again soon. Spatial locality is the idea that if an item is referenced, nearby items will tend to be referenced soon." (see [10], p. 141)
Volume data are usually stored in memory as an array of 2D images, i.e. as a linear array. During ray casting we often search for neighbouring voxels (spatial locality concept). Because the volume data are huge and the caches are price compromise between speed and size the neighbours are often out of cache (so-called cache-miss happens) and CPU has to wait for the data. To avoid cache-misses and CPU stalls, volume data can be bricked and stored blockwise (see Fig. 4.1). Addressing in the bricked data can be effectively done with

help of a lookup table ([9]). Bricking and calculating the lookup table is done only once in the initialization phase.
The brick dimension can be an arbitrary positive number. However, it is suitable to choose it to be a power of two. The reasons emerges from the following optimization rules.
When addressing the volume, the optimization rule *"Shift, not multiply!"* can be successfully used. For example, if brick dimensions are power of two, then, in C++ notation, instead of doing

```
p = data[cx + Bx*(cy + cz*By) + v]
```

it is faster to do

```
p = data[cx + (cy<<logBx) + (cz<<logBxy) + v],
```

where logBx $= \log_2 B_x$, logBxy $= \log_2 B_x + \log_2 B_y$ and B_x and B_y are brick dimensions.
Another optimization rule when working with 'nice' integers (i.e. which are power of two) is that binary function *modulo* can be done more efficiently with masking, namely

```
int cx = sx % Bx; //relative x-coord in the block
```

can be replaced with

```
int cx = sx & (Bx-1); //relative x-coord in the block
```

if B_x is power of two.
The success of the last two optimization rules dwells in instructions latencies. The multiplication is obviously translated into `imul` instruction, which has latency[1] of 10 (cycles), whereas the shifting is done via `shl` instruction with latency of one single cycle. Implementing function *modulo* with help of masking (instruction `and`) takes 1 cycle, but doing `idiv` instruction to get the modulo takes 66–80 cycles! If we become aware how often we multiply or do modulo with 'nice' integers—especially when doing addressing arithmetics—we understand that performing of these operations is one big wasting of CPU cycles. Of course, if multiplication or modulo is performed by constants then a lot compilers translate this code effectively, but this is not the case if we multiply (or do modulo) by a variable, content of which the compiler cannot predict.
Rewriting the code using these optimizations has caused not only direct speedup but it has also opened the way for SIMD optimization. The reason is that the only way to optimize integer code for SIMD is using MMX, but MMX does not contain instructions for parallel multiplying of two DWORDs. However, shifting is done very well.

4.4 VR Pipeline

The standard ray-casting algorithm works as follows. Every pixel on the screen defines one ray. After initialization, the ray advances through the volume until leaving it. During

[1]Instruction latencies depend on the processor type and model encoding, we state latencies for Pentium IV processors with model encoding 3, see [14].

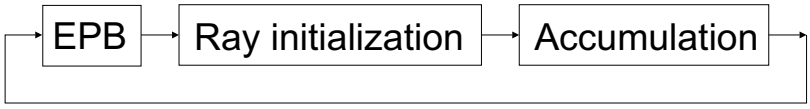

Figure 4.2: *Volume rendering pipeline.*

advancing, the ray's color and alpha values are accumulated. When all rays have been processed their color is written into framebuffer and the frame can be rendered. Volume rendering pipeline (see Fig. 4.2) therefore consists of three key parts: **Creating Entry Point Buffer(EPB)**, **Ray Initialization** and **Accumulation**.

4.4.1 Entry Point Buffer

Ray-casting is an image-order algorithm, thus we have to process one ray for every screen pixel. If we consider standard resolution 1024×768, then about 780 000 rays travel for every frame through the scene. This is pretty much and often unnecessary. The solution is to create an *Entry Point Buffer (EPB)* which holds for every ray its distance from the camera when the ray hits the volume for the first time. The EPB contains a special value for those rays which do not hit the volume and therefore will not be processed. Once the ray hits the volume and has to be processed, it is important to set its initial world position as closest as possible to the non-transparent part of the volume according to the current transfer function. This is especially true for medical datasets, because the object of interest is usually in the middle and surrounded with air (transparent empty space). After choosing the transfer function (in preprocessing step), every block is classified as transparent or non-transparent. Only non-transparent blocks are rasterized into EPB (like z-buffer). Because parallel projection is used, it is sufficient to project and rasterize only one block (three visible faces of the cube or box chosen according the viewing direction)—we obtain the template buffer—and the remaining part of EPB can be obtained by just copying the translated template buffer.

To determine EPB in higher resolution than the block granularity a *min-max octree* is used. For performance reasons, the depth of the octree is restricted to three.

SIMD optimization of this stage lies mainly in using optimized `memset` function (see Section 3.2.1) and vector arithmetic. When rasterizing a block, eight cube vertices of the block need to be transformed from world space into camera space. For this purpose, function `xformCube` was written. This function uses pipelined matrix-vector transformation (see Section 3.2.2) and prefetching the data to be in the cache before the CPU will demand them.

4.4.2 Ray Initialization

The next stage in the ray-casting pipeline is *Ray Initialization*. This stage is responsible for initializing all rays whose colors will contribute to the final rendered frame. Every ray is of `TRay` type and its declaration look as follows:

```
struct TRay
{
    float sx, sy, sz;    //resample position
    int index;           //index into framebuffer
    float alpha;         //accumulated alpha
    float color;         //accumulated color
}
```

In a naive implementation, one processes all the rays in one loop (one for every pixel). Initialization consists of setting `alpha` and `color` to zero (fully transparent black color) and `sx`, `sy` and `sz` to the corresponding entry sample position according to the EPB. To find the entry sample position we have to transform ray's origin into world space, which is simple matrix-vector transformation. After this the initialized ray is added into its block (See 2.2.5):

```
for all pixels[x,y]
{
    calculate z from EPB;
    if (ray from [x,y,z] will contribute to frame)
    {
        transform ray into world space;
        add ray into corresponding block;
    }
}
```

At first glance, nothing except the matrix-vector transformation can be optimized here. However, we can split the entire ray initialization phase in three step: gathering information about the rays that will be transformed into a temporary array, then transforming them all and finally to distribute them into blocks which they belong to. To keep the temporary array size not too high, we do the rays initialization in batches until all rays are initialized:

```
for all pixels[x,y]
{
    calculate z from EPB;
    if (ray from [x,y,z] will contribute to frame)
        add ray into the temporary array;

    if (array is full)
    {
```

```
        //process whole batch
        Transform all rays from array;
        Add all transformed rays into corresponding blocks;
    }
}
```

The measuring showed that this was the right way. Already a non-SIMD-optimized version gave better results (for the exact results see Chapter 5).
For the purpose of SIMD optimization, let `INIT_RAYS_BATCH` be the parameter holding the size of the temporary array into which are the rays added. It is important to choose this parameter properly. If `INIT_RAYS_BATCH` is too low we do not use the SIMD power of the CPU. On the other side, if `INIT_RAYS_BATCH` is chosen too high, then, although the transformation is done fast, we loose the cycles while waiting for the data, because a too big array does not fit into the cache, violates the locality concept and causes cache-misses and CPU stalls (for concrete results see Chapter 5).
The core of the optimized Ray Initialization assembler function lies (again) in the pipelined matrix-vector transformation, now for any number of vectors. Also the fast conversion between integer and float data types, which is offered by the SIMD extended instruction sets, was used. Finally, the last loop (the third step - distributing the initialized rays into their blocks) was unrolled to decrease the number of condition jumps which cause CPU stalls (due to pipeline flushes).

4.4.3 Accumulation

The third stage in the ray-casting pipeline is *Accumulation*. It is very simple. Blocks are grouped into blocklists—levels of block (see Section 2.2.5). The levels are traversed from front to back. For each block in the current level all rays are processed, i.e. ray is resampled and color and alpha values are accumulated until the ray leaves the block and enters a new one or leaves the scene. The correct block grouping into levels ensures that no ray can be added into a block that has been processed before. After processing the last ray of the last block of the last level the frame can be displayed.
There is nothing to optimize when processing rays in this way. It is just one quite short loop with color and alpha accumulation and couple of statements responsible for managing the rays when they leave the current block. If we look at the accumulation code (below) we can see the strong read-after-write dependency which keeps all efforts from optimizing this most important stage of ray-casting pipeline.

```
//get alpha and color value for current ray's resample position
GetVoxel(ray, tmpColor, tmpAlpha);
//accumulate color and alpha
tmpAlpha*=(1.0f-alpha);
color+=tmpAlpha*tmpColor;
alpha+=tmpAlpha;
```

However, being enlightened from optimizing the Ray Initialization phase, we have transformed the classic SISD approach into SIMD approach—instead of processing one ray at a time we work with two rays at the time. Now, the loop is a little bit more complicated because the two rays have not to spend the same time in the loop at all, thus there is a little overhead to keep the rays pipe (of length two) full. This approach has immediately led to SIMD optimization of the accumulation code.

Accumulation in this form is especially suitable for parallel computations using 3DNow!. As the SSE instructions can work with four single float values at once (not only two as the 3DNow!) this loop does not take advantage of full computational power of SSE. Therefore we finally rewrote the loop to process four rays at once. The SSE version of accumulation looks as follows:

```
; excerpt from SSE Accumulation code
...
; register xmm1 contains tmpColor0 ... tmpColor3
; register xmm2 contains alpha0 ... alpha3
; register xmm3 contains color0 ... color3
; register xmm7 contains [1.0f | 1.0f | 1.0f | 1.0f]

subps xmm7,xmm2        ; 1.0f-alpha
mulps xmm7,xmm0        ; tmpAlpha*=(1.0f-alpha)

mulps xmm1,xmm7        ; tmpColor*tmpAlpha
addps xmm3,xmm1        ; color+=tmpColor*tmpAlpha
addps xmm2,xmm7        ; alpha+=tmpAlpha

...
```

4.5 Conclusion

This chapter has briefly described optimization details of the final code evolution. The practical results (the timing and comparison) is the subject of the next chapter.

Chapter 5

Results

This chapter describes comparison results between non-optimized and optimized versions of our program. The measurings have been done on several computers, we are presenting three of them. Let us denote them as PC1, PC2 and PC3. More details about these computers are in Table 5.1.

All measurings have been done for brain dataset `brain.f3d` (see Fig. 5.1). The dataset with dimensions $256 \times 256 \times 109 \times 8$ bits is after loading converted into one with dimensions $256 \times 256 \times 128 \times 8$ bits. The images of size 1152×854 are rendered without shading with nearest neighbour interpolation. The results are mostly in milliseconds (absolute values), some of them are in cycles (relative values). As the rendering times depend on viewing direction (due to the number of rays, not the memory alignment), we have measured the longest rendering times, which are for our case for values $\beta = 0$ and α around zero, namely $\alpha = 0.3$. The $\alpha \in < 0, 2\pi)$ and $\beta \in (-\pi, \pi)$ parameters are angles in radians and determine the camera's rotation about y-axis (α-value) and its inclination (β-value).

At first we have measured the impact of the bricking technique on the final rendering time (Section 5.1). Next sections describe the measurings we have done for all three volume rendering stages and also the total rendering times for the optimized and non-optimized versions.

Table 5.1: *Details of PCs we have used for testing.*

Name	CPU	Clock Speed	L2 Cache	Instruction Sets
PC1	AMD	1460.4MHz	256 kBytes	MMX, 3DNow!
PC2	Intel	551.3MHz	512 kBytes	MMX, SSE
PC3	AMD	1659.1MHz	256 kBytes	MMX, 3DNow!, SSE

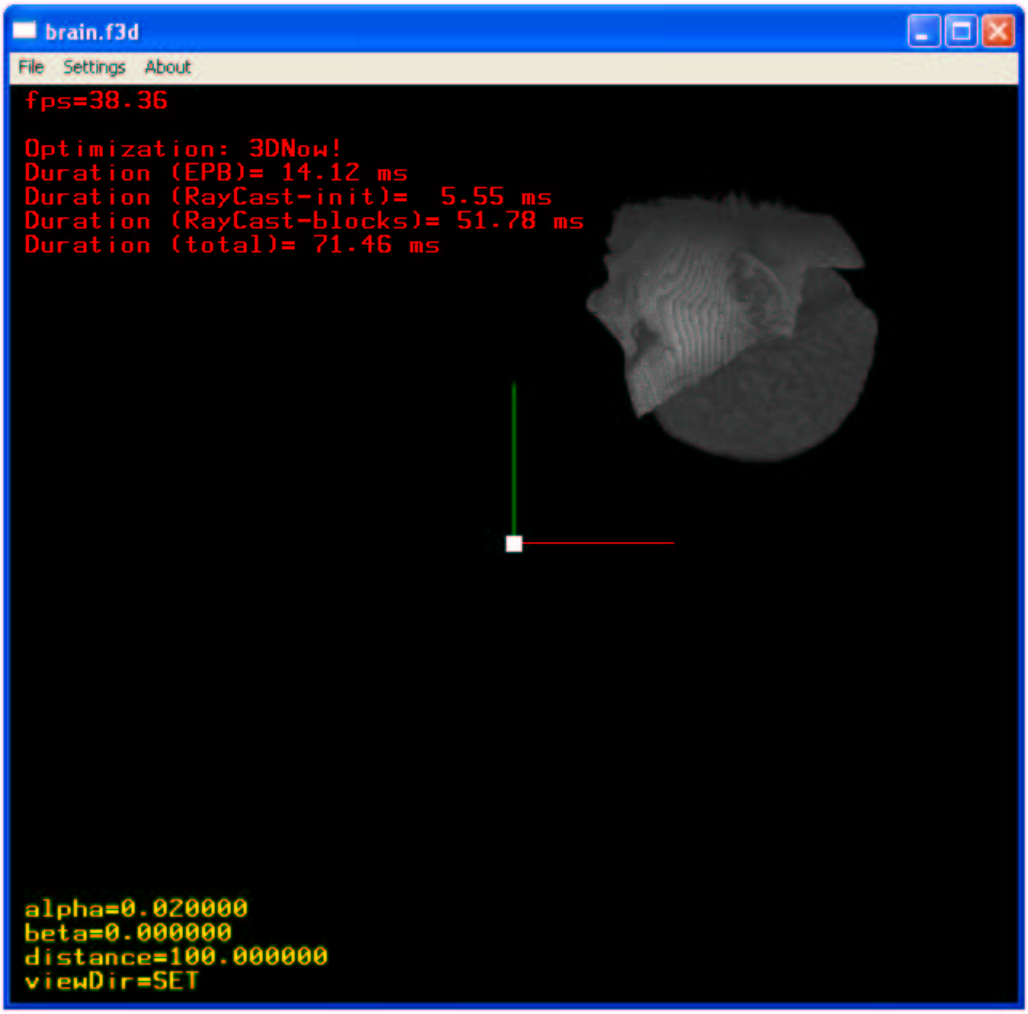

Figure 5.1: *A Screenshot from the testing program with the brain dataset.*

5.1 Bricking

The theoretical reasons why to use bricked rather than linear volume layout was described in Section 4.3. This section shows whether the practical results confirm the theory or not.

Table 5.2 shows the measurement obtained from PC 1 running a non-SIMD-optimized version. The B_x, B_y and B_z values are dimensions (in voxels) of one brick. EPB, RC-Init, RC-Accum and Total are given in milliseconds. From the table can be easily seen

Table 5.2: *Impact of Bricking on Calculation Times* [*ms*].

B_x	B_y	B_z	EPB	RC-Init	RC-Accum	Total
256	256	128	9.94	15.27	1426.91	1452.12
128	128	128	7.33	11.08	743.73	762.14
64	64	64	7.28	10.13	148.25	165.66
32	32	32	8.93	9.67	73.95	92.55
16	16	16	18.72	9.62	54.07	82.41
8	16	16	30.34	9.88	51.89	92.11

that bricking significantly influences all parts of volume rendering pipeline, especially Ray-Accumulation. Let's examine these interesting results.

The first row represent the case with no bricking, as the brick dimensions equal exactly the volume dimensions. Creating the EPB in this case is extraordinary simple—just parallel projection and subsequent rasterization of one cube (brick, i.e. block). This block contains all rays which are initialized in the second stage pipeline stage. Every ray is followed as it advances through the scene and its color is written into the framebuffer when it leaves the volume. It is the standard ray-casting.

The second row shows the results after decreasing B_x and B_y by half. Now we have four blocks. The rasterization is faster, because only one block (which area is four times smaller than in the previous case) has to be rasterized, the remaining three are only copied. The RC-Init stage contains less rays as more octree nodes are now classified as transparent. The most important impact of bricking can be seen on the RC-Accum stage. The calculating time decreased nearly two times. Although the whole block still does not fit into the cache ($128^3 > 256$ kB), we can suppose that decreasing block size is the right direction.

The next rows only confirm our assumptions. With the decreasing brick dimensions the number of blocks increases, but common calculating time decreases. After the brick dimensions fall bellow 16, the EPB time starts to grow, because there are too many small blocks which have to be copied, and the cache is not the bottleneck anymore. The RC-Init times are quite stabilized, the total number of discarded rays significantly changes only when not considering bricking. This can be easily explained by the number of transparent blocks, which is certainly higher when the volume is more bricked, and by considering, that the rendered volume contains only one object surrounded with air. This stage is only

minimally influenced by bricking.
The impact of bricking can be extremely well seen on the last stage (the column RC-Accum in the Table 5.2). The color accumulation permanently accesses the volume. As the volume is too big for cache, a lot of cache-misses occur. Processing rays in the correct order (see Section 2.2.5) significantly decreases number of cache-misses. So it is possible to improve calculating time from about ∼1400 ms to ∼50 ms which gives approximately speedup of 28! Of course, not only cache is responsible for this speedup. With finer octree granularity in EPB, the rays are initialized quite near before they enter the volume, so the number of accumulations, and thus memory accesses, is decreased.
More precisely, the L2 cache on PC 1 is 256 kB. Theoretically, a block of dimensions $64 \times 64 \times 64$ fits into cache, but additional data needed by other calculations cause cache-misses. Therefore, when processing any of the $32 \times 32 \times 32$ block from any direction, it always fits into cache and resides there, so, theoretically, no more cache-misses occur when accumulating rays' colors. That is the reason why the dramatically increasing speedup stops at $16 \times 16 \times 16$ blocks. Bellow this, the cache does not play such a significant role in optimizing the calculations time. Until now, the CPU had to wait for data (stalling), now the CPU power is important, as the slower memory does not hinder the CPU.
When we consider the last table row, we can see that $16 \times 16 \times 16$ blocks are the right compromise between the EPB and Ray-Accumulation. With smaller blocks, the EPB time becomes the bottleneck of volume rendering, not the Ray-Accumulation.

5.2 Entry Point Buffer

The first stage of volume rendering pipeline, which creates Entry Point Buffer, successfully uses SIMD optimized `memset` routine (see Section 3.2.1). Highly unrolled loop, together with aligned pointers and data prefetching runs about three times faster than the standard (`rep stosd`) implementation. This optimization runs on all[1] current modern processors as it needs only the MMX extended instruction set.
Before the block can be rasterized, we need to transform its vertices from world into camera space. This transformation involves eight matrix-vector multiplications. This can be effectively optimized for SIMD. The code for matrix-vector transformation is (with little modifications) identical with the codes introduced in Sections 3.2.2 and 3.2.3. The following Table 5.3 shows that matrix-vector transformation belongs to SIMD-like tasks, because the obtained measurings give high speedup—about three for both CPU types.
The most expensive part, however, is template buffer projection into EPB according the octree structure. As the template buffer may be of arbitrary width, it cannot be successfully SIMD-optimized. The above mentioned optimizations cause speedup of about 1.2 (see Table 5.4).

[1]Note that we are considering only Intel and AMD processors.

Table 5.3: *Transforming 8 cube vertices from camera-space into world-space.*

PC	Average cycles count			Speedup	
	None	3DNow	SSE	3DNow	SSE
PC1	3720	1227	-	3.03	-
PC2	4227	-	633	-	6.67
PC3	3811	1380	892	2.76	4.27

Table 5.4: *Optimization of stage EPB.*

PC	Duration [ms]			Speedup	
	None	3DNow	SSE	3DNow	SSE
PC1	19.13	14.4	-	1.33	-
PC2	48.9	40.2	-	-	1.22
PC3	26.39	23.0	22.7	1.15	1.16

5.3 Ray Initialization

As mentioned in Section 4.4.2, the second stage of volume rendering pipeline–Ray Initialization–had to be rewritten into 'SIMD-like form' after realizing, that quite a lot of rays[2] had to be initialized before they could travel through the scene, and this initialization consists primarily of doing matrix-vector transformations. To observe the impact of SIMD on the calculating time we introduced new `INIT_RAYS_BATCH` parameter. Table 5.5 summarizes our observations for SSE:

Table 5.5: *Impact of the INIT_RAYS_BATCH parameter.*

IRB	1	2	32	64	128	256	512	1024	2048
None [ms]	20.5	18.07	17.5	17.4	17.13	17.3	17.8	18.28	18.5
SSE [ms]	14.4	13.3	12.8	12.7	12.6	12.7	13.5	14.4	14.8

We can see, that the best results are obtained when processing about 128 rays in one batch. This technique gives speedup of about 1.36.
The overall speedup of this stage is summarized in Table 5.6.

[2]It depends on size of the window client area, e.g. for window 1024×768 about 780 000 rays have to be processed!

Table 5.6: *Optimization of the Ray Initialization stage.*

PC	Duration [ms]			Speedup	
	None	3DNow!	SSE	3Now!	SSE
PC1	9.5	6.0	-	1.6	-
PC2	22.0	-	16.0	-	1.4
PC3	10.03	7.06	7.05	1.4	1.4

5.4 Accumulation

The last and the most important stage in our volume rendering pipeline is Ray Accumulation. This stage is often the bottleneck when trying to reach real time rendering. According to the Section 4.4.3 we implemented three versions of the accumulation code. The first version iterates through all the rays, processing one ray per loop iteration. In this simple loop, the only place, where the SIMD can be used, is the color accumulation. Simply putting all color and alpha values, obtained by resampling the ray, into a temporary array and then doing accumulation on this array (like we did in Ray Initialization stage) is not the right way, because the accumulation is inherently sequential. Therefore, we have unrolled the loop and processed two (later four) rays per one loop iteration. It is very important to realize, that this is not the true loop unrolling as we have mentioned in Section 4.3. Each iteration of the new loop's body now has to perform two (or four–according to the version) times more tests, so we did not decrease the number of potential pipeline stalls. All what we have done is that we have grouped the same operations together. Instead of one multiplication or addition we do four. This can be, if we have the data in proper data structures, done in parallel. Actually, that was the reason why we did so—to prepare the code for SIMD optimization. Table 5.7 shows the measurings obtained after rewriting the accumulation for 3DNow! and SSE, both processing 4 rays per one loop iteration.

Table 5.7: *SIMD optimization of the Accumulation stage.*

PC	Duration [ms]			Speedup	
	None	3DNow!	SSE	3Now!	SSE
PC1	55.7	54.7	-	1.02	-
PC2	122.1	-	116.3	-	1.05
PC3	53.53	52.23	50.7	1.02	1.06

We can see that processing four rays in parallel (SIMD optimized version) did not speed up the calculations a lot. This can be explained similarly like in the previous section. Accumulation of four rays constitutes very small batch. As the SIMD instructions are especially successful when doing long batch tasks, the impact of SIMD optimizations is

very low (only 5%).
But very interesting results are in the Table 5.8. It compares processing one, two and four rays per loop iteration without SIMD-optimization. This table demonstrates that already the single C++ code reorganization for SIMD itself causes speedup about 24% even without successive assembler SIMD optimization.

Table 5.8: *Three versions of Accumulation stage (non-SIMD-optimized) - processing one, two and four rays per one loop iteration.*

Rays per loop	Duration [ms]	Speedup
1	67.4	-
2	59.3	1.14
4	54.2	1.24

5.5 Conclusion

After choosing the best parameters (`INIT_RAYS_BATCH` = 128, $B_x = B_y = B_z = 16$) we have measured the following values:

Table 5.9: *Total rendering times.*

Total	Duration [ms]			Speedup	
	None	3DNow!	SSE	3Now!	SSE
PC1	83.2	74.7	-	1.11	-
PC2	192.0	-	172.0	-	1.12
PC3	89.95	82.29	80.45	1.10	1.12

This table shows that using SIMD we are able to speed the overall rendering time by 10–12 %.
What can we conclude from this?
First, software ray-casting cannot beat the hardware assisted ray-casting using the newest graphics cards. They are running at around 40 fps [13], whereas our approach only at around 14 fps. Second, it is good to write the code with SIMD optimizing on mind, because (as we could have seen in Section 5.4) this optimization itself can lead to global optimizations and overall speedup even when the code is finally not rewritten into assembler.
The SIMD-optimization is especially suitable for batch-like tasks. We have tried to find this 'batch-pattern' in all stages of ray-casting. However, the analysis and implementation

showed, that ray-casting is not a typical batch-like task. Although we tried to reorganize the code in this way, there were found not many such parts. Optimized `memset` runs twice faster than standard memset routine. Optimized matrix-vector transformation runs about three times faster, but overall the Ray Initialization stage runs only 1.5 times faster. The bottleneck, however, remains the last stage— the Ray Accumulation. This stage cannot be transformed into typical SIMD batch task, so this part nearly does not exploit the SIMD power offered by the CPU.

The answer to our question, whether *to do or not to do the SIMD optimization in volume rendering*, cannot be definitive yes or no. If we look for interactive real time volume rendering then we recommend GPU volume rendering. Number of graphics cards supporting 3D textures is permanently growing, so in a few years it will be the common hardware. On the other side, the GPU suffers from lower quality and limited flexibility. Therefore, software volume rendering (whether SIMD optimized or not) will still be the best choice for rendering high quality images, especially since multi-CPU computers can become commonplace soon.

Although the SIMD optimization did not significantly improve the rendering time, its principles are worthful and should be taken into consideration when optimizing software volume rendering. The practice has shown, that the SIMD optimization of such easy functions like `memset` or transforming batch of vectors can bring a lot if used often. However, a function written in assembler is compiled directly into machine code, many non-trivial optimization rules must be therefore obeyed—e.g. calculating instructions' latencies to avoid partial stalls or pairing instructions according to the pipe the instructions will go in to effectively use both execution pipes—all this is bypassed by assembler programming and not automatically done by compiler. For longer parts of code the compiler obviously does better work.

5.6 Future work

SIMD (like many other things) is double edged sword. You can dramatically speed up some portions of your code, but also slow it down, when used not carefully. This work has shown, that some calculations are naturally suitable for SIMD, some of them have to be changed to be suitable for SIMD, but some are completely unsuitable for SIMD-optimization. We tried to beat GPU power using SIMD-optimized CPU code but the reality has shown that it is impossible. However if we completely forget the rivalry, we could use the software volume rendering for quality reasons. This would involve to do trilinear interpolation, Phong shading and the support for more than 8-bit dataset should be added, too. Perhaps afterwards SIMD will show its real power.

Bibliography

[1] LEVOY, M. *Display of Surfaces from Volume Data*. 1988. IEEE Computer Graphics and Applications, 8(3):29-37, May 1988

[2] PORTER, T. - DUFF, T. *Compositing digital images*. 1984. In Hank Christiansen, editor, Computer Graphics (SIGGRAPH '84 Proceedings), volume 18, pages 253-259, July 1984.

[3] SRAMEK, M. *Visualization of Volumetric Data by Ray Tracing*. 1998. Wien: Österreichische Computer Gesellschaft 1998. ISBN 3-85403-112-2.

[4] LACROUTE, P. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transforamation*. 1995. Technical report CSL-TR-95-678. Stanford University

[5] LEVOY, M. *Efficient Ray Tracing of Volume Data*. 1990. ACM Transaction on Graphics, 9(3):245-261, July 1990

[6] LACROUTE, P. - LEVOY, M. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*. 1994. Proc SIGGPRAPH '94, pp. 451-458, July 1994

[7] DONG, F. - KROKOS, M. - CLAPWORTHY, G. *Fast Volume Rendering and Data Classification Using Multiresolution Min-Max Octrees*. 2000. EUROGRAPHICS 2000, 19(3)

[8] YAGEL, R. - KAUFMAN, A. *Template-based Volume Viewing*. 1992. Proc. EUROGRAPHICS'92, p. C153-C157, June 1992

[9] BRUCKNER, S. *Efficient Volume Visualization of Large Medical Datasets*. 2004. Master's Thesis. Vienna University of Technology

[10] BISTRY, D. - DELONG, C. et al. *The Complete Guide to MMX Technology*. 1997. McGraw-Hill. ISBN 0-07-006192-0.

[11] CABRAL, B. - CAM, N. - FORAN, J. *Accelerated volume rendering and tomographic reconstruction using textuge mapping hardware*. 1994. In Proceedings of the Symposium on Volume Visualization 1994, p. 91-98, 1994.

[12] ENGEL, K. - KRAU, M. - ERTL, T. *High-quality pre-integrated volume rendering using hardware-accelerated pixel shading*. 2001. In Proceedings of the Workshop on Graphics Hardware 2001, p. 9-16, 2001.

[13] CERVENANSKY, M. *Využitie komerčných grafických akcelerátorov pre vizualizáciu a spracovanie objemových dát*. 2004. Master's Thesis. Faculty of Mathematics, Physics and Informatics, Comenius University.

[14] *IA-32 Intel Architecture Optimization*. 2004. Reference Manual. Order number 248966-011. *http://www.developer.intel.com*

[15] DICOM - *http://medical.nema.org/dicom/2003.html*

[16] f3d - *www.viskom.oeaw.ac.at/ milos/page/Download.html*

[17] TASM - *http://info.borland.com/borlandcpp/cppcomp/tasmfact.html*

[18] MASM - *www.masm32.com*

[19] NASM - *http://nasm.sourceforge.net/*

[20] AMD - *www.amd.com*

[21] INTEL - *www.intel.com*

[22] wxWindows - *http://www.wxwindows.org*

[23] *www.tommesani.com*