Evaluation of a Bricked Volume Layout for a Medical Workstation based on Java

Peter Kohlmann[†], Stefan Bruckner[†], Armin Kanitsar[‡], M. Eduard Gröller[†]

[†]Vienna University of Technology
Institute of Computer Graphics and Algorithms
Favoritenstrasse 9-11/E186
1040 Wien, Austria
{kohlmann | bruckner | groeller}@cg.tuwien.ac.at

[‡]AGFA
Diefenbachgasse 35
1150 Wien, Austria
armin.kanitsar@gwi-ag.com

ABSTRACT

Volumes acquired for medical examination purposes are constantly increasing in size. For this reason, the computer's memory is the limiting factor for visualizing the data. Bricking is a well-known concept used for rendering large data sets. The volume data is subdivided into smaller blocks to achieve better memory utilization. Until now, the vast majority of medical workstations use a linear volume layout. We implemented a bricked volume layout for such a workstation based on Java as required by our collaborative company partner to evaluate different common access patterns to the volume data. For rendering, we were mainly interested to see how the performance will differ from the traditional linear volume layout if we generate images of arbitrarily oriented slices via Multi-Planar Reformatting (MPR). Furthermore, we tested access patterns which are crucial for segmentation issues like a random access to data values and a simulated region growing. Our goal was to find out if it makes sense to change the volume layout of a medical workstation to benefit from bricking. We were also interested to identify the tasks where problems might occur if bricking is applied. Overall, our results show that it is feasible to use a bricked volume layout in the stringent context of a medical workstation implemented in Java.

Keywords: Medical Visualization, Bricked Volume Layout, MPR, Medical Workstation.

1 INTRODUCTION

Usually, medical volume data sets are available as stacks of two-dimensional images (slices). In a linear volume layout these values are stored in a single array. The rendering of enormously large data sets becomes problematic with this storing approach. For instance, the male data set of the National Library of Medicine's Visible Human Project [NLM] consists of 1871 axial anatomical images. Each is composed by 2048 x 1216 pixels with a color depth of 24 bit, which amounts to about 14 GB. As this is considerably more than the address space of a typical PC, the data set has to be stored on hard disk and needs to be transferred to main memory on demand. Because of limited bandwidth these transfers are quite costly and result in undesirable latency.

Bricking is a technique to subdivide the volume into smaller parts to overcome the mentioned problem. A single brick contains a fixed number of data values in x-, y- and z-dimension. Accessing a certain data value in a bricked volume is inevitably more costly than in a linear volume layout. After it is determined in which brick the value is located, its position inside this brick has to be calculated. Additional computational effort is necessary if interpolation and gradient calculation come into play. For instance, if the interpolation at a certain location needs data values from several bricks, an intelligent structure to ensure fast access to the according values is required. Solving this task becomes especially difficult within a Java-based implementation as required by our collaborating company partner. There are efficient addressing solutions implemented in C++ which require a lot of array accesses. An array access is quite cheap in C++, but it is significantly slower in Java because of costly boundary checks. However, the implementation of a PACS (Picture Archiving and Communications System) in Java has several benefits. Java offers tools for a convenient plug-in development. In addition, it provides platform independence.

For the evaluation whether a bricked volume layout is applicable for a medical workstation based on Java we investigate some common access patterns to the volume data. Important tasks of such a workstation are efficient handling, manipulation and display of images. As most radiologists still prefer to examine two-dimensional slices, our approach focuses on a visualization technique called Multi-Planar Reformatting (MPR). MPR provides an arbitrary reformation of

a given two-dimensional image stack. Basically, a two-dimensional plane is positioned and oriented inside the three-dimensional volume and the interpolated data values are displayed on this plane as shown in Figure 1. The left image shows a three-dimensional view of the data set and the slicing plane. In the right image the corresponding MPR slice is presented.
In various PACSs it is possible to display medical data by using three two-dimensional planes as shown in Figure 2. These planes are aligned with the three major axes to provide the axial, the sagittal and the coronal view simultaneously.

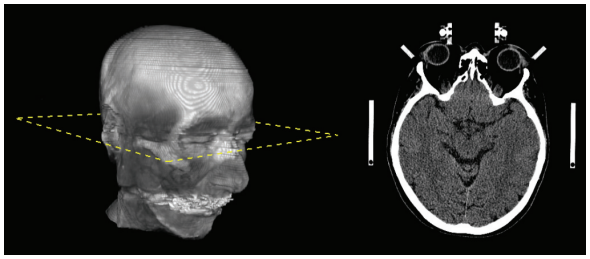

Figure 1: To generate an MPR-slice, a plane is defined which intersects the volume (left). Interpolated data values are shown on this plane (right).

With a linear volume layout the time for computing these different views varies according to the main data alignment. As all views are displayed at the same time, the slowest one is the performance bottleneck. For this reason, it is important that our approach provides not only a high, but also a rather constant frame rate for the different views.

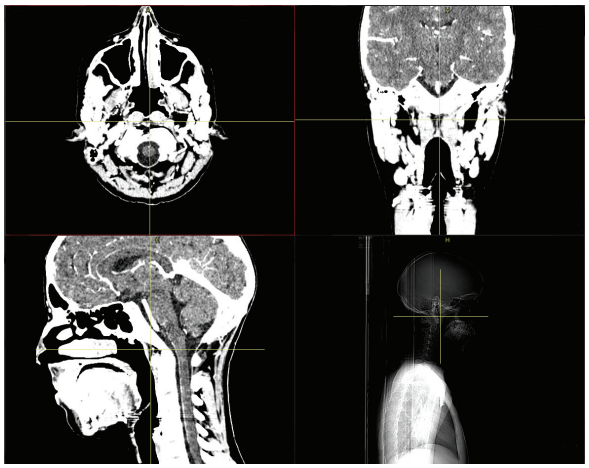

Figure 2: Axial, sagittal and coronal view of a CT head data set. They are displayed simultaneously by a PACS. In the lower right part of the display area an overview image (scout view) is provided.

Segmentation is a very important task in medical visualization. Only if the access to the values of the data set is possible in an adequate time it makes sense to change the volume layout of a medical workstation. Therefore, we compare the linear and the bricked volume layout in respect to important access patterns for segmentation.
The remainder of this paper is structured as follows. Section 2 provides an overview of the relevant previous work. In Section 3, our algorithms for the MPR calculation are presented in more detail. We show the different steps starting with the generation of the bricks up to the final MPR image. A discussion of our results concerning the performance of different access patterns is provided in Section 4. Finally, Section 5 concludes the paper and presents some ideas for future work.

2 RELATED WORK

Several approaches address bricking for ray casting. Law and Yagel [LY96a] presented a distributed ray tracing system. They identify coherency (data locality) as a very important factor which highly influences the performance. As their work employs an object data-flow approach, objects which are once fetched have to be fully processed before they are replaced by other objects. To ensure multi-frame thrashless ray casting, they divide the volume into equally sized cells and advance a ray front to generate the image. In addition, the screen is subdivided into a number of stripes of equal width, which are distributed to different available processors. Beside these stripes, each brick is randomly assigned to a certain processor. A linked-list data structure handles the information how the rays are advanced through the cells.
Our work is also inspired by the approach of Grimm et al. [GBKG04]. They focused on memory-efficient CPU-based volume rendering and presented several high-level optimizations for this purpose. As minimization of memory usage is crucial for their approach, several computations are performed on the fly. A bricked volume layout is used along with refined data addressing techniques to accelerate the on-the-fly computations. Costly computations to address the data values and performance-decreasing if-else statements are avoided by using elaborated shift-operations in addition to look-up tables. Efficient utilization of the CPU like thread-level parallelism enables a significant speedup compared to other techniques.
Approaches for interactive ray tracing based on bricking which are optimized for distributed systems have been presented by Parker et al. [PPL+99] and by DeMarle et al. [DPH+03]. Guthe et al. [GWGS02] accomplished an interactive walkthrough of large data sets on standard PC hardware. They apply wavelet filters to the subdivided (bricked) volume to get a compression of the volume data. This representation can be decompressed on-the-fly. Hardware texture mapping is used for the rendering.
Weiskopf et al. [WWE04] presented a solution to maintain constant frame rates in 3D texture-based volume rendering. For direct volume visualization the

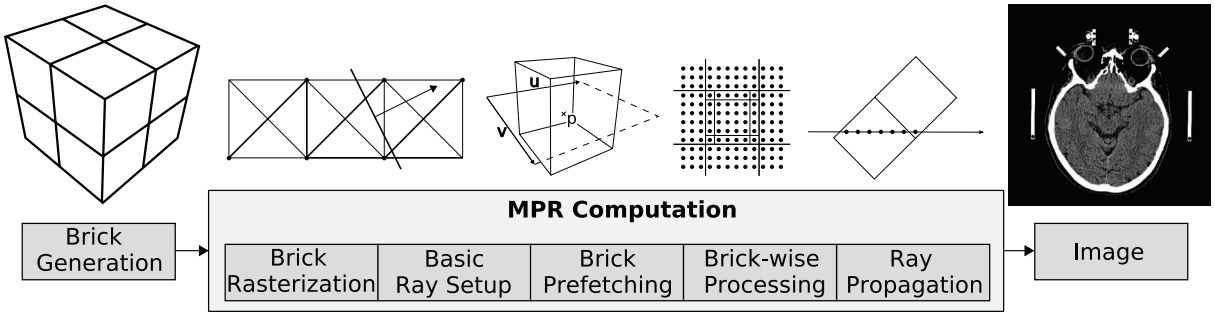

Figure 3: The MPR pipeline.

performance is highly view-dependent because of the texture memory layout of current graphics hardware. Using bricks with alternating orientations helps to avoid this varying performance.
A rendering pipeline for real-time rendering of isosurfaces was introduced by Hadwiger et al. [HSS+05]. With bricking and acceleration techniques like empty space skipping, they achieve interactive frame rates for large volumes which exceed the GPU texture memory.

3 BASIC ALGORITHMS

Most of the approaches mentioned in the previous section are focused on volume ray casting. As many radiologists prefer to examine two-dimensional slices we decided to evaluate the performance of an MPR implementation based on the different volume layouts. Our goal was to achieve an implementation with a high and rather constant frame rate. In this section, the basic algorithms for the generation of the bricks and the MPR implementation based on a bricked volume layout are presented.

3.1 Brick Generation

To benefit from an efficient data addressing, the brick size has to be a power of two. Grimm et al. [GBKG04] experimented with various sizes and came to the conclusion that 64 KB (32 x 32 x 32 * 16 bit) is an appropriate size in their hardware setup. Law and Yagel [LY96b] also showed that this size is a good choice. If the bricks are smaller, this is helpful for acceleration techniques such as empty-space skipping. But as a drawback additional computational effort is necessary to manage smaller bricks.
In our case a brick is quite a simple data structure with only few attributes. It has a unique ID to reference the brick, the min- and the max-value of the contained data and the information if it is padded. Padding has to be performed if a certain brick is not completely filled with data values. This occurs if the number of data values is not a multiple of the brick dimension (32) in one of the volume dimensions. As we want to store several bricks instead of the monolithic volume, an important question is how the bricks are generated.

First of all, we consider from where the data is extracted. The DICOM (Digital Imaging and Communications in Medicine) format is an open standard for medical images. It provides a container for image data and meta information like parameters of the scanner and patient information and it contains a single file for every slice. These slices are loaded successively but not necessarily in the correct order. If a file is read, its data array is extracted and the values are written to the corresponding brick arrays. A layer of bricks in xy-dimension is called a slab. For a volume in which each slice is recorded with a resolution of 512 x 512, a slab is built by 256 (16 x 16) bricks. The total number of slabs corresponds to the number of bricks in the z-direction. As soon as all the brick arrays for a single slab are filled with data values a notification is sent out. Then, the MPR renderer can start to generate part of the image. With this approach, it is not necessary to wait until the whole set of DICOM images is loaded to start the rendering process.

3.2 MPR Computation

After the generation of the bricks we focus on an important access pattern to medical volume data. In Figure 3 our MPR pipeline shows the steps which are necessary to produce the final image. The presented approach is based on a brick-wise resampling of the volume along rays. At first, a brick rasterization is performed to identify the bricks which are intersected by the MPR plane. In the next step, rays within the plane are cast to determine enter- and exit points where the rays hit the volume. Then, the list of intersected bricks is traversed and trilinear interpolation is performed to calculate the values at the sample positions along the rays. A ray is propagated to the next brick as soon as it is completely processed for the current brick.
To ensure high frame rates we avoid floating-point operations as much as possible. Therefore, several floating-point variables are converted into a fix-point representation via bit shifting. With this approach it is possible to perform the whole interpolation process and many intersection tests exclusively based on fix-point arithmetic. In the following sections, the steps of the MPR pipeline will be described in detail.

3.2.1 Brick Rasterization

MPR visualizes the information which is resampled on an arbitrarily oriented plane that intersects the volume. It is important to efficiently determine all the bricks which have to be processed to render the resulting image. An efficient method to detect plane and axis-aligned bounding box intersections, presented by Möller and Haines [MH99], is used for these calculations.
At this point we do not need to know where exactly the bricks are intersected. In a loop over all bricks, the relevant ones are extracted. The used algorithm exploits the fact that only a single diagonal of the box has to be tested for intersection. It is the one which is most closely aligned with the normal of the plane. In Figure 4 this is illustrated for the two-dimensional case. The three gray squares represent bricks and the black line is the two-dimensional version of the plane. The thickened gray lines are the diagonals of the squares which are most closely aligned with the plane's normal (black vector). It is sufficient to check if the black line intersects these selected diagonals to determine if the corresponding square is intersected.

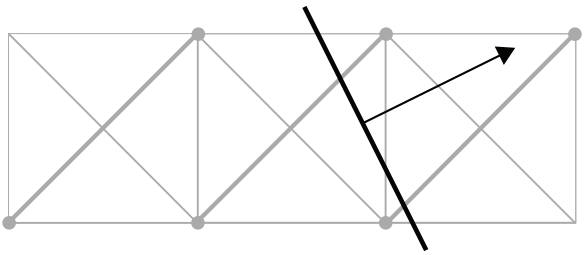

Figure 4: Brick rasterization in the two-dimensional case. Only the thickened diagonals of the squares (the ones which are most closely aligned with the normal of the plane) have to be checked for intersections with the plane.

In the three-dimensional case we have to identify the vertices of the *diagonal of interest* only for a single brick. The diagonals of the other bricks which need to be checked are calculated by adding the offset of the specific brick in x-, y-, and z-direction. With this algorithm the brick rasterization is performed very efficiently.

3.2.2 Basic Ray Setup

This section describes the basic setup of the rays and introduces some data structures. Two vectors and a point are used to define the MPR plane as shown in Figure 5 (top). Position p defines the center of the plane. The vectors u and v are orthogonal to each other and span the plane. In Figure 5 (bottom) the mapping of this plane to the image space is illustrated. A ray which is lying within the MPR plane is cast through the volume on each scan line. The pixels in image space are filled by an equidistant sampling along each ray. In contrast to ray casting, a ray is not utilized to gather the value for a single pixel but to *collect* the values for a complete scan line. To store some required information for resampling, the rays are initially cast through the volume. Each ray is an object with the following attributes:

- int enteringBrick
- int firstVolumeSample
- int lastVolumeSample
- int currentSamplePos

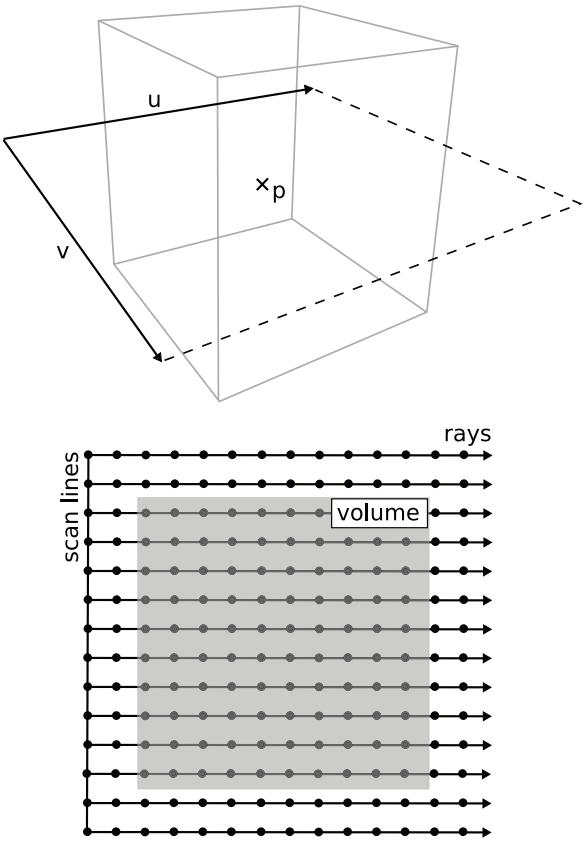

Figure 5: The point p and vectors u and v define the MPR plane in object space (top). This plane is mapped to the image space and rays are cast through the volume (bottom). The volume is resampled along these rays.

Figure 6 illustrates the values which are stored by these variables. We do a test if the ray intersects the volume. If this is the case, there is a volume-entry- and a volume-exit point. The ID of the first hit brick is assigned to *enteringBrick*. As resampling is performed along the ray, the first and the last of the ray's sample positions inside the volume are assigned to $firstVolumeSample$ and $lastVolumeSample$. All the sample positions of a ray which are outside the volume

are set to the background color. The current sample position *currentSamplePos* is initially set to the value of $firstVolumeSample$.

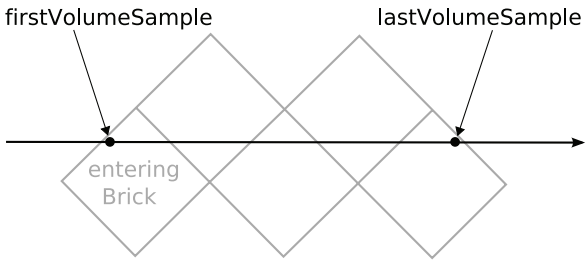

Figure 6: Two-dimensional illustration of a ray which is initially cast through the volume.

In addition, two arrays are used to store brick-relevant information:

- short[] brick_fromRay
- short[] brick_toRay

The size of these arrays corresponds to the total number of bricks. A loop over all rays is performed to determine the attributes of each ray. Each of these ray tests results in an update of the two brick arrays. The first ray which hits a certain brick is stored in *brick_fromRay* and the respective last one is written to *brick_toRay*. This structure keeps track of all the rays which intersect a certain brick. In Figure 7 this is shown for an example brick. It is intersected by the rays 231 to 235. The according entries in the *brick_fromRay* and *brick_toRay* arrays are set to 231 and 235.

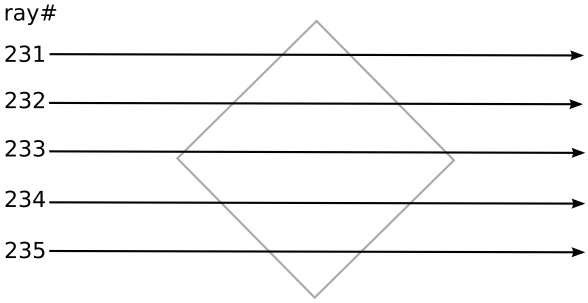

Figure 7: The brick is intersected by a number of rays. Two *brick_fromRay* and *brick_toRay* arrays keep track of this by storing the number of the first and of the last ray.

3.2.3 Brick Prefetching

In the brick rasterization step we identified all the bricks which are intersected by the defined plane. The bricks are organized in a cache implementation and they can be addressed with unique IDs. There is a function call to fetch and to release a single brick. It is necessary to minimize the number of function calls as much as possible to achieve optimal performance. Therefore, we fetch all the bricks which are needed to calculate an image before we start with the resampling. After one image is calculated, all the bricks are released again to ensure an efficient usage of the available memory.
The image-relevant bricks are those which were identified by the brick rasterization and all their neighboring bricks. The brick neighbors are needed for access patterns to the data during resampling or the computation of gradients as described by Grimm [Gri05]. In Figure 8 this is illustrated for the two-dimensional scenario.

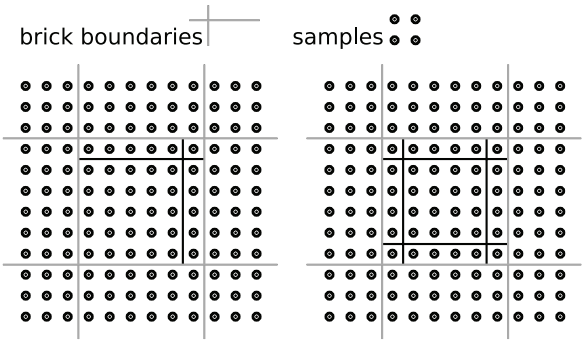

Figure 8: Subdivision of the sample positions inside a brick in 2D for the access patterns during resampling and gradient computation. For resampling the samples can be divided into 4 subsets (left). To calculate gradients 9 subsets can be built (right).

The left image shows that the sample positions of a brick can be divided into 4 subsets if resampling has to be performed. This subdivision is based on the fact that for a resampling operation either only samples from the same brick suffice or (in 2D) samples from one or three neighboring bricks are necessary. For the majority of the sample positions the needed neighboring samples are available inside the same brick. But for the sample positions on the top edge, the right edge and the top-right corner, samples from neighboring bricks are needed. In three dimensions an 8-neighborhood is used for resampling. Figure 8 (right) shows the subdivision of the sample positions for the gradient calculation. An 8-neighborhood is needed for two dimensions. This leads to 26 neighbors which have to be addressed in three dimensions.

3.2.4 Brick-Wise Processing

For performance reasons it is not sufficient to traverse one ray after another. This is very inefficient because it is likely that consecutive rays partially process the same data. As the cache size is limited, the same data is read from main memory several times and slows down the image computation. It is necessary to process the volume data brick wise to benefit from the bricked memory layout and to improve data locality. The bricks, which were identified during the brick rasterization step, need

to be ordered in a front-to-back manner according to the ray direction.
Afterwards, one brick after another is processed in the determined order. For each brick, references to its 26 neighbors point to the according prefetched bricks. Now, it is possible to look up the intersecting rays from the *brick_fromRay* and *brick_toRay* arrays for the currently active brick. A further loop is used to process this list of rays. Depending on the three components of the ray direction, a ray has to be tested with three sides of the brick to determine how many ray samples are inside the current brick. In a third loop the samples along the ray are traversed within the brick. A brick is entirely processed as soon as the contribution of all sample positions along its intersecting rays to the final image is computed.
Grimm et al. [GBKG04] presented a very efficient way to address the values within a brick. To facilitate trilinear interpolation they precompute the offsets for the eight neighboring samples and store them in a look-up table. With this approach they avoid to compute several Boolean conditions in costly if-else constructs. These look ups are on the one hand used to determine the brick in which a certain sample position is located and on the other hand to get the offset inside this brick. However, they have an implementation in C++ where the access of array elements is quite cheap. As the number of array accesses in their approach is rather large for the interpolation case, it is not applicable to our Java implementation. The high number of look ups would lead to poor performance because of the array implementation of Java. For each access a boundary check is performed, with the result that the performance drawback compared with an array access in C++ is significant.
Depending on the sample position within a brick, we can determine if the needed values for the trilinear interpolation are entirely inside the brick or if they are spread over neighboring bricks. To identify the involved bricks we use a method presented by Grimm et al. [GBKG04]. We assume to have the x-, y- and z-position where the sample is located inside a brick. Then it is possible to calculate the *case* of the location inside the brick with the equation

```
case = 9*((((x-1)&(b))|1)+1)>>5)
       + 3*((((y-1)&(b))|1)+1)>>5)
       + ((((z-1)&(b))|1)+1)>>5),
```

where b is two times the brick dimension (32) minus 1 and $>> 5$ corresponds to a division by the brick dimension. The *case* represents the subset of the brick where the specific sample position is located as shown in Figure 8 (right). Extended to three dimensions there are 27 brick subsets. The case calculation provides a value within the range $[0,26]$. This value defines the location of the sample position inside the active brick. With this information we know which neighboring bricks are needed for the resampling process. The case computation helps to avoid the evaluation of a number of long Boolean statements to determine the sample position inside the brick. A switch/case construct is used to select the adequate equation for the trilinear interpolation.

3.2.5 Ray Propagation

As soon as all the sample positions along a ray are processed for the active brick, this ray is propagated to the adjacent brick it enters as shown in Figure 9.

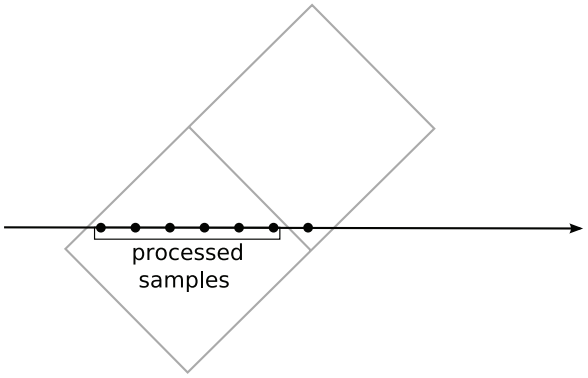

Figure 9: A ray is propagated to the adjacent brick it enters after it is processed for the active brick. To achieve this, the entries of the corresponding brick arrays *brick_fromRay* and *brick_toRay* are updated.

This has to be done because we have only registered the rays at the bricks which they are entering first at their way through the volume. We have already calculated the number of samples along the ray within the active brick, the current sample position and the last sample position of the ray which is inside the volume. With this information we can determine the position of the first sample along the ray which is outside the current brick. This position is used to calculate the ID of the next brick which is intersected by the current ray. Afterwards, the ray is propagated by updating the two arrays *brick_fromRay* and *brick_toRay* and the current sample position of the ray.

4 RESULTS

This section provides an overview of the results of the presented implementation. At first, we will evaluate the performance of the MPR computation for the bricked volume layout versus the linear volume layout. Secondly, we will examine how the bricking influences other important access patterns like the random access to data values or a simulated region growing approach. The PC for the performance tests is configured with an *AMD Athlon 64 Dual Core Processor 4400+*, 2 GB of main memory and an *NVIDIA GeForce 7800 GTX* graphics card with 256 MB of internal memory. On the

software side, the used Java version is the *Java Runtime Environment Version 5.0 Update 6*.

4.1 MPR Computation

To compare the performance of an MPR implementation based on a linear volume layout with our implementation we measured the time to calculate a single image. The specifications of the CT data set we used for these tests are:

- Resolution: 512 x 512 x 333
- Spacing: 0.40/0.40/0.90 mm

We measured the time for the computation of a single image for the cases where the slices are parallel to the xy-plane (coronal), the xz-plane (axial) and the yz-plane (sagittal). Additionally, the performance for the computation of an arbitrarily orientated slice is of interest. For the axial, sagittal and coronal test case the plane is moved through the volume and the averaged time per slice is calculated. In the case of the arbitrarily oriented slice, a plane, which is spanned by two randomly generated vectors that are orthogonal to each other, is defined within the volume. The directions of these vectors are changed in a loop and the averaged time is taken as the result. The size of the output images is 512 pixels in height and width. In Figure 10 the results of these tests are listed.

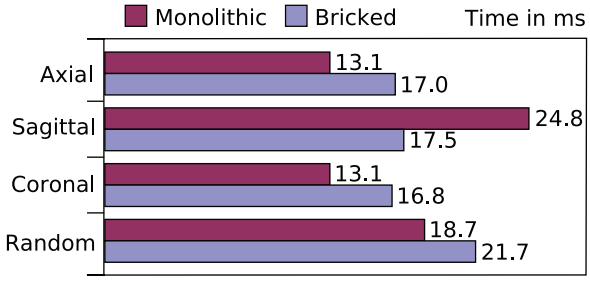

Figure 10: The averaged time in milliseconds which is needed to compute one MPR image for the axial, the sagittal, the coronal and the random case.

Whereas we have a performance loss in the axial and the coronal case of about 30 %, the sagittal case is accelerated by about 30 %. In the case of the randomly oriented plane the loss is about 16 %. This rather high performance gap between the axial and coronal versus the sagittal case using a linear volume layout is caused by different memory access patterns. The CT scanner that recorded the used data set generated primary images which were axially aligned. Thus, the cache hit ratio for the calculation of an axial slice is very good in contrast to the sagittal case. The utilization of a bricked volume layout offers a much better data locality.

4.2 Random Access

The worst case scenario to access the data values concerning data locality is a random access. We compared the time to access 512 x 512 values which are randomly distributed within the volume. The needed time for this access pattern is 21.4 ms in the monolithic versus 41.4 ms in the bricked case. For a bricked volume layout more address computations have to be performed to get a certain value. We assume that we have three random values x, y and z, the number of values per data dimension (*xValues*, *yValues*, *zValues*) and the array with all the data values (*data*). For a monolithic volume layout it is easy to access the value through

```
val = data[z*xValues*yValues
       + y*xValues + x];
```

Compared to this, the following effort is necessary to access one value in a bricked volume layout. We know the number of bricks in the three dimensions (*xBricks*, *yBricks*, *zBricks*). At first, the number of the brick (*brickNum*) which contains the sample position has to be calculated.

```
int brickNumX = x >> 5;
int brickNumY = y >> 5;
int brickNumZ = z >> 5;

int brickNum = brickNumX
       + brickNumY*xBricks
       + brickNumZ*xBricks*yBricks;
```

After this, it is necessary to calculate the position inside the active brick (*posInBrick*) to access the value.

```
int xPosBrick = x%32;
int yPosBrick = y%32;
int zPosBrick = z%32;

int posInBrick = xPosBrick
       + (yPosBrick << 5)
       + (zPosBrick << 10);

val = bricks[brickNum][posInBrick];
```

The increased effort for this hierarchical address computation compared to the simple calculation for the linear volume layout causes the measured performance difference of a factor two.

4.3 Spherical Access

The last access pattern we evaluated is a spherical access. We used a parameterized sphere to simulate region growing which is a popular segmentation algorithm to identify homogeneous areas inside the volume. Therefore, the center of a sphere is randomly placed inside the volume, with the constraint that the whole sphere fits into the volume. For the test volume with resolution 512 x 512 x 333, we measured the time to access 512 x 512 data values on the parameterized surface of the sphere. To simulate region growing, the radius of the sphere is varied between 5 and 150. With a

linear volume layout the access times are quite stable. They increase from 10.5 ms (radius 5) to 13.6 ms (radius 150). The reason for this is the worse cache hit ratio if the values are more widespread within the volume. With bricking we have a constant access time of 15.5 ms in the case that all the bricks are prefetched. But this strategy simulates a monolithic volume and counteracts the benefit of bricking. In another scenario, only the brick which holds the currently needed value is fetched. One optimization ensures that no brick is fetched if consecutive values are inside the same brick. The needed time for the access of all the values takes 32 ms for a sphere with the radius 5 and increases up to 260 ms with a radius of 150.

5 CONCLUSIONS AND FUTURE WORK

We have presented an implementation of a bricked volume layout and evaluated different access patterns to medical volume data. Our overall goal was to investigate the question if bricking is a good choice for a medical workstation based on Java. The previous work was almost exclusively based on ray casting. As many radiologists prefer the examination of two-dimensional slices, we focused on an MPR implementation. Compared to a linear volume layout we achieved a very good performance for this access pattern. Many PACSs divide the screen into different sections to display MPR images. Because of this splitting, the axial, the sagittal and the coronal view can be displayed simultaneously. As the computation of the different views can be easily parallelized if a machine with several CPUs is available, the view which needs the most time to be computed is the performance bottleneck. Provided that this parallelization is done, we can compare the frame rates of the implementation based on a linear volume layout with the ones which are based on the bricked volume layout. Therefore, it is enough to compare the frame rates of the according *slowest views* - the sagittal ones. In this case we have an improvement for the bricked volume layout from 40 to 57 fps or 42.5 %.

Beside this acceleration, it can be seen that the frame rates for the different views (axial: 76 fps, sagittal: 40 fps, coronal: 76 fps) are varying quite a lot using a linear volume layout. With the bricked volume layout we achieve almost constant frame rates for these views because of a better data locality. Another important point is, that the benefits of the bricked layout will be more pronounced if the data set is large enough so that it does not fit into the computer's main memory anymore.

The performance for the other access patterns is not yet fully satisfying. Random access to the data values takes about twice the time when bricking is used. For the spherical access the radius of the sphere is crucial for the performance. We are sure that optimizations by taking into account and prefetching only affected bricks improve the performance significantly. For instance, the sphere can be subdivided and the bricks which contain the surface of one part can be prefetched and fully processed before moving to the next part.

Overall, we can recommend the application of a bricked volume layout to medical workstations based on Java. Future work needs to be done for different segmentation algorithms like watershed or edge-based techniques. Furthermore tracking algorithms and the masking of certain areas of the volume have to be adapted to ensure good performance for the bricked volume layout.

ACKNOWLEDGMENTS

The work presented in this paper has been funded by AGFA in the scope of the DiagVis project. We would like to thank Rainer Wegenkittl and Lukas Mroz of AGFA Wien for their collaboration and for providing different CT data sets.

REFERENCES

[DPH+03] D. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed interactive ray tracing for large volume visualization. In *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 87–94, 2003.

[GBKG04] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. A refined data addressing and processing scheme to accelerate volume raycasting. *Computers and Graphics*, 28(5):719–729, 2004.

[Gri05] S. Grimm. *Real-Time Mono- and Multi-Volume Rendering of Large Medical Datasets on Standard PC Hardware*. PhD thesis, Vienna University of Technology, 2005.

[GWGS02] S. Guthe, M. Wand, J. Gonser, and W. Straßer. Interactive rendering of large volume data sets. In *Proceedings of IEEE Visualization*, pages 53–60, 2002.

[HSS+05] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proceedings of Eurographics 2005*, pages 303–312, 2005.

[LY96a] A. Law and R. Yagel. Multi-frame thrashless ray casting with advancing ray-front. In *Proceedings of Graphics Interfaces*, pages 70–77, 1996.

[LY96b] A. Law and R. Yagel. An optimal ray traversal scheme for visualizing colossal medical volumes. In *Proceedings of Visualization in Biomedical Computing*, pages 33–43, 1996.

[MH99] T. Möller and E. Haines. *Real-Time Rendering*. AK Peters, Ltd., Natick, MA, 1999.

[NLM] The National Library of Medicine. The Visible Human Project. Available online at http://www.nlm.nih.gov/research/visible/.

[PPL+99] S. Parker, M. Parker, Y. Livant, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, 1999.

[WWE04] D. Weiskopf, M. Weiler, and T. Ertl. Maintaining constant frame rates in 3D texture-based volume rendering. In *Proceedings of IEEE Computer Graphics International*, pages 604–607, 2004.