Advanced GPU Raycasting

Henning Scharsach*

VRVis Research Center
Vienna, Austria

Abstract

Modern GPUs offer a degree of programmability that opens up a wide field of applications far beyond processing millions of triangles at ever increasing speed. Raycasting is one of these applications that can make heavy use of the built-in features of todays graphics cards. With the possibilities offered by this technology, there is a lot of room for new techniques that do not simply convert existing algorithms to the GPU, but use the very strengths of this architecture to create more realistic images at interactive frame rates.

This paper presents an approach to hardware based raycasting in the fragment shader of a shader model 3 compatible graphics card that not only allows for both orthogonal and perspective projection, but enables the user to move the viewpoint into the dataset for fly-through applications like virtual endoscopy. This hardware-based approach can also be used to correctly intersect the rendered dataset with normal OpenGL geometry, allowing arbitrary 3D-meshes, pointers or grids to be rendered in the same scene.

The last section deals with the biggest problem of GPU-based raycasting - the limited amount of available video RAM - and how it can be circumvented by applying a cached blocking scheme that loads only blocks of interest into the memory.

Keywords: raycasting, GPU, shader model 3, virtual endoscopy, blocking

*henning@vrvis.at

1 Introduction

In the field of hardware-based volume rendering, there are two distinct approaches for rendering datasets at highly interactive framerates. The first approach, as originally presented by Cullip and Neumann [2] and further developed by Cabral et al. [1], is directly exploiting the GPUs texture mapping capabilities by creating some kind of (usually planar) sampling surface - either viewport [9] aligned with one 3D-texture or axis aligned [7] with a set of 2D-textures - and resampling the original data at this so-called proxy geometry. This technique is widely accepted now as a common way to render medium sized datasets in acceptable quality at interactive framerates and has been revisited, finetuned and extended many times, e.g. [9, 3, 8, 6].

Though this approach is very similar to the way computer games make use of the GPU, which ensures that it runs at the highest possible speed, it has a couple of serious drawbacks: First, everything that needs to be calculated for the final result, every texture fetch, gradient or lighting calculation, has to be done for every single fragment, no matter if it contributes to the final image or not. Second, advanced techniques like empty space skipping are very difficult to implement because of the unflexible nature of the algorithm. And finally, implementing perspective projection (or even fly-through modes) and dealing with the resulting sampling artefacts imposes some difficulties thus being infeasible for most cases.

The second approach would be to implement a raycaster in the fragment shader of the GPU, as proposed by Krüger

and Westermann [5]. Since this algorithm uses the graphics card in a very different way than most games do, there is often some additional effort required to find the most efficient solution for a certain task. Still, this approach is far more flexible and can be extended in a number of ways, which is what the main part of this paper is about, and this enables us to utilize the specific advantages a GPU has over a CPU in the best possible way, namely:

- A massively parallel architecture
- A separation into two distinct units (vertex and fragment shader) that can double performance if the workload can be split
- Incredibly fast memory and memory interface
- Dedicated instructions for graphical tasks
- Vector operations on 4 floats that are as fast as scalar operations
- Trilinear interpolation is automatically (and extremely fast) implemented in the 3D-texture

Many more advantages may arise through the specific nature of a GPU-based algorithm. As we will show in the next sections, there are a couple of advantages that our raycasting algorithm has over a similar CPU techniques, like the possibility of very efficient emtpy-space-skipping via the outer bounding geometry, the implicit support for perspective projection or the possibility to intersect our dataset correctly with normal OpenGL primitives by simply modifying the z-buffer accordingly.

On the other hand, there are still some disadvantages of GPU-based raycasting that one should not forget and that still limit the set of possible applications, the biggest one being the limitation of available video memory. Though we can improve on that by applying a sophisticated caching and blocking scheme like the one presented in section 8 (which only stores blocks of interest), the overall amount of important data we can display is still limited by the available memory.

In the next section, we will first introduce the basic idea of hardware raycasting and how the algorithm works. In section 3 we will extend this technique by adding a more sophisticated bounding geometry for efficient empty space skipping. Section 4 deals with Hitpoint Refinement, which significantly improves quality for iso-surface raycasting without a noticable speed penalty. Interleaved Sampling is presented as a solution to heavy sampling artefacts in Section 5, and in Section 6 we add the possibility to intersect the volume with arbitrary OpenGL geometry. Section 7 deals with the problems when trying to fly into the dataset for virtual endoscopy applications and how this can be solved. Section 8 shows one approach to circumvent the biggest drawback hardware-based approaches face: The limited amount of available video memory. Finally, sections 9 and 10 show some results we were able to achieve and give a short outlook into our future work.

2 Algorithm Overview

The basic idea of hardware raycasting, as proposed by Krüger and Westermann [5], is simple: The dataset is stored in a 3D-texture, in order to take advantage of the built-in trilinear filtering. Then, a bounding box for this dataset is created where the position inside the dataset (i.e. the texture coordinates) is encoded in the color channel, as shown in the left picture in figure 1.

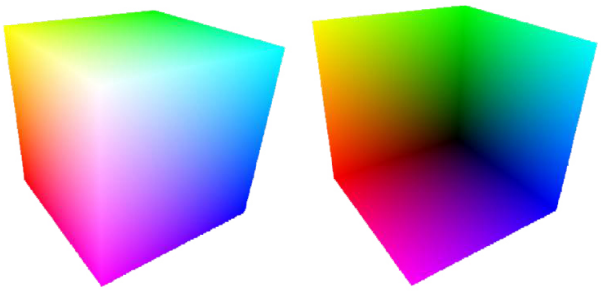

Figure 1: Front and back faces of our simple bounding geometry encoding the current position in the color channel. Subtracting these two images will yield the viewing vectors for the raycasting step.

Now the viewing vector at any given pixel can be easily computed by subtracting the color of the front faces of the color cube at this pixel (which is the entry point into the dataset) from the color of the back faces at this pixel (which is the exit point) as shown in figure 1. Normalizing and storing this vector in a 2D-texture of the exact size of the current viewport (together with its initial length in the alpha channel) yields a 'direction texture' for every screen pixel.

Casting through the volume is easy now: Render the front faces again (the entry points into the dataset) and step along the viewing vector for this pixel (stored at the same position in the direction texture) until the ray has left the bounding box again (i.e. the casted distance is greater than the alpha value of the texture, where the initial length was stored). Compositing the final color can be done in a separate texture, which is blended onto the screen at the very end.

The fragment shader of a modern GPU is perfectly suitable for accomplishing that, since the front faces can be drawn as normal OpenGL geometry and for every pixel of the bounding box that will be drawn, the fragment shader is automatically called with the current color (the starting position) and the current pixel position (for the direction texture lookup) as input parameters. The possibility to have loops and conditionals within a fragment shader (as of shader model 3) makes it possible to cast through the volume with a single function call.

To sum this all up, the basic hardware raycasting algorithm is a 4-step-process:

1. Draw front faces of the color cube into an intermediate texture.

2. Draw back faces, subtract the color value of the front faces, normalize the outcome and store this vector together with its initial length in a separate 'direction texture'.

3. Draw front faces again, taking the colors as an input parameter for the fragment program, and cast along the viewing vector (that is stored in the direction texture). Store the intermediate steps in a separate compositing texture. Terminate the ray if we leave the bounding box or as soon as the opacity has reached a certain threshold (early ray termination).

4. Blend the result back to screen. It would be possible to composite to the screen directly, but a separate blending step makes the approach more flexible, extensions like geometry intersection (see section 6) a lot easier, and doesn't impose a significant speed penalty.

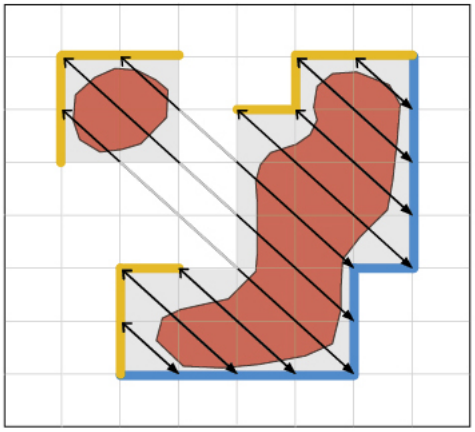

Figure 3: Front and back faces of our blocked bounding geometry, with the grey boxes being active blocks. Note that the ray always starts at the first front face and ends at the last back face, even if there are inactive blocks inbetween.

3 Empty Space Skipping

The basic algorithm presented in section 2 leaves a lot of room for improvement and possible extensions like empty space skipping, which can significantly speed up rendering time - in this section we propose a blocked scheme for bounding geometry creation which significantly speeds up rendering time.

As of now, only the pixel processing pipeline of our GPU is used, while the vertex pipeline is lying idle. So whether the bounding box consists of 12 (as in our basic algorithm) or 100,000 triangles doesn't make much difference - even more since graphics cards nowadays are specially designed to handle the massive (and ever increasing) amount of geometry modern games throw at them without suffering huge performance hits. So the idea of increasing the complexity of the bounding box - making it a data dependent bounding geometry - is obvious.

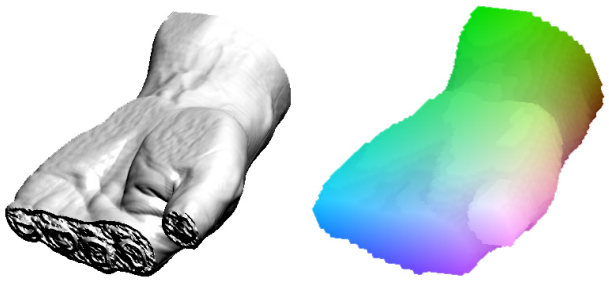

Figure 2: Example for a data-dependent bounding geometry of a hand dataset with the colors encoding the position in the dataset. In this view, only 68% of the bounding blocks are discarded es empty and are skipped without any performance loss, increasing the framerate to 56fps.

When modern graphics cards render a scene, they try to process vertices and pixels in parallel, in the best case leading to an equal distribution of the workload. Of course, if there's a small number of large triangles, the vertex processing engine will be partly idle, while only the pixel processing engine is busy - such a scene is said to be fill limited.

On the other hand, if there is a huge amount of visible triangles that consists of only one or two pixels, then the vertex engine will be the bottleneck, making the scene geometry limited. Both of this is of course undesirable, meaning that it is best to keep the average triangle size at a very constant rate that is, in the best case, close to the optimal size for the current generation of GPUs. With todays graphics cards, this optimal size is highly dependant on the shader complexity, but usually somewhere between four and eight.

Considering this, a blocking scheme with equally sized blocks (shown in Figure 2) seems to be a good idea, which decides for every block consisting of a number of voxels from the original dataset whether this block is of any interest or not. This can be done by culling each block either against the iso-value or the transfer function using summed area tables, depending on the current rendering mode. If this test reveals that the current block needs to be rendered, it is marked as active - otherwise, it will be discarded. It is important to note that have to test one border voxel outside the current block as well, because the filtering could cause an interpolated value inside the block to be greater than the threshold even if none of the voxel values in the block is.

With this blocking scheme enabled, the border between active and non-active blocks of our new bounding geometry is rendered now instead of the simple color cube. However, the separation into front faces and back faces is a little bit more complicated now, because the possibly non-convex bounding geometry doesn't guarantee for exactly *one* front and back face anymore. Thus, we need to retrieve the *first* front face to start our rays and the *last* back face to stop them - this can be done with a simple depth test.

As shown in Figure 3, this scheme does not necessarily skip all inactive blocks - another lookup in the fragment shader is necessary to determine whether there are empty blocks *between* the first front and last back face. However, this check is easy and doesn't slow the fragment program down noticably, and the implicit empty space skipping via the bounding geometry has no performance hit at all.

4 Hitpoint Refinement

In this section, we propose a simple way to get a better estimation of the real intersection point with the iso-surface after casting with constant sampling distance.

Starting from the first estimation of the intersection (i.e. the first sample point along the ray where the density is greater than our threshold), the algorithm goes one halfstep back (half the previous sampling distance) and checks the density value at the new position. The next step will again be half the previous stepsize (making this one fourth of the original sampling distance), and will depend on the density on this new position: If it is stil above the threshold, the next halfstep will also be taken backwards, otherwise forwards. This bisection can be repeated 5 or 6 times, until we have an intersection that is 64 times more precise than our original estimation (that should be sufficient for most applications, no matter how low the original sampling distance is).

In our implementation, we're always performing six bisection steps, each time multiplying the stepsize with either 0.5 or -0.5. This way, we don't need a real conditional statement (thus minimizing the impact on performance). Interestingly enough, performance slightly increases when turning on Hitpoint Refinement in our implementation, which is probably due to some pipelining issue.

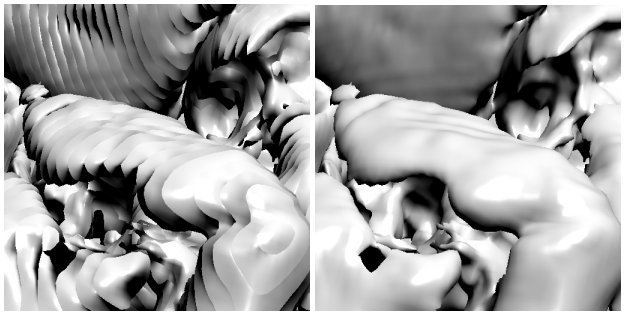

Figure 4: Highly undersampling this dataset leads to heavy sampling artefacts (left). Turning Hitpoint Refinement on removes those artefacs without lowering performance (right).

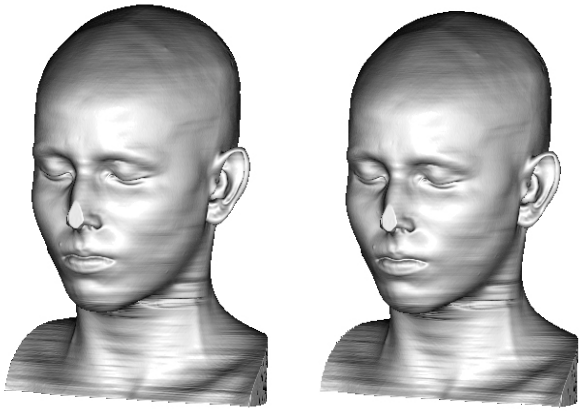

Figure 5: This head-to-head comparison of a 512x512x333 dataset shows that the results for sampling distance 1.0 (left, 24fps) and sampling distance 5.0 (right, 66fps) are almost identical when Hitpoint Refinement is turned on.

As shown in Figure 4, Hitpoint Refinement dramatically increases the image quality if iso-surface renderings - even more when the sampling distance is very low. Because of this, it is easily possible to increase the sampling distance to 400 or 500%, as long as no important features are completely missed by the ray. As shown in Figure 5, the sampling distance can even be five times the voxel distance for certain datasets without any visible difference. This makes it extremely useful for interaction renderings while the user is moving or rotating the dataset, because even *if* tiny details were missing because they are skipped by the ray, the user would hardly notice while moving around. As soon as the mouse is released, the sampling distance should be reduced again to ensure that all the details are rendered correctly.

5 Interleaved Sampling

Keller and Heidrich [4] proposed Interleaved Sampling as a solution to bridge the gap between regular and irregular sampling patterns. We extend this approach to GPU raycasting where calculating a small z-offset results in a large reduction of sampling artefacts.

Regular sampling patterns are easy and fast to compute, but prone to producing sampling artefacts, while irregular sampling patterns achieve much better results at the cost of higher computational demands. The solution is now to use an irregular sampling pattern that covers *multiple pixels*, so

that two adjacent pixels will never have the same pattern. Though this approach was basically meant to improve on the results of multisampling, the authors suggest using interleaved sampling for volume rendering as well. Since supersampling is not an option in computationally expensive algorithms like raycasting, the sampling positions are only interleaved in z-direction (the view direction of the camera).

This means choosing a small offset in z-direction that is different for adjacent pixels, but will repeat after a number of pixels.

The easiest way to do this would be to have some kind of modulo function of the screen coordinate. In the fragment shader, this can only be done by using the FRC command, which returns the fraction of a float value. Calculating an repeating offset this way only requires a couple of easy computations at the beginning of our fragment program, so there's hardly any performance hit. Still, there's a huge impact on the final image quality, which can be seen in Figure 6.

However, interleaved sampling does not always deliver the most visually appealing result. Consider the case of two very thin structures that map to completely different colors in the transfer function and are roughly viewport-aligned. Without interleaved sampling, only one of these colors may be visible, because the highest intensities of the other color may be missed by our ray samples. Turning interleaved sampling on, a strong dithering pattern consisting of these two colors will be visible, depending on which of the two structures was hit by the ray with a particular offset, as shown in Figure 7.

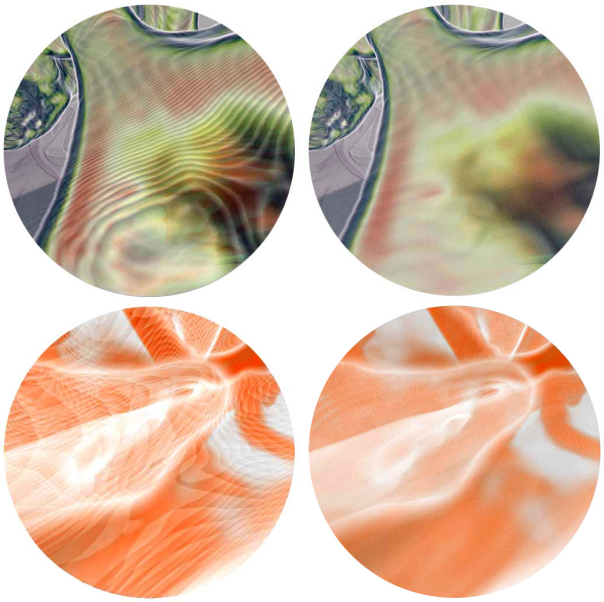

Figure 6: Two scenes with rather difficult transfer functions, rendered without (left) and with Interleaved Sampling (right). Notice how virtually all of the sampling artefacts disappear.

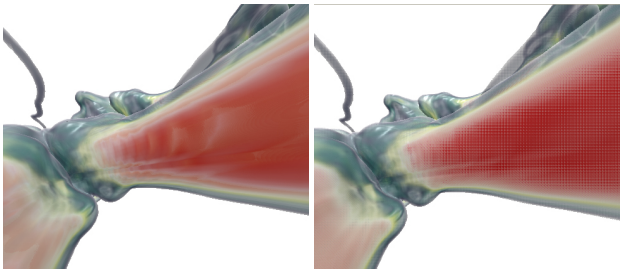

Figure 7: Turning Interleaved Sampling on does not always produce a more appealing result, but is a good indicator of undersampling in the other cases. In this example, the thin white tissue over the red bone is hardly visible in the left picture, except for a few sampling artefacts. Turning Interleaved Sampling on in the right picture suggests that the sampling rate should be increased.

Though the first result is definitely more appealing, the second picture is closer to the correct rendering (which would be achieved by sampling at an indefinitely small sampling distance or one that is at least a lot smaller than the smallest feature present in the dataset) because it shows both colors that are present at these sampling positions. Interleaved sampling can be taken then as an indication that given the current dataset and transfer function, the current sampling rate is not sufficient and we might be missing important features.

6 Geometry Intersection

Being able to intersect the rendered volume with normal OpenGL geometry allows for a number of interesting applications, like 3D-Pointers that correctly blend into the scene, a 3D-grid that gives additional information about the position in the dataset or arbitrary meshes that could cut away part of the dataset for easier navigation. In this section we propose an approach that makes sure that parts of the volume that are not visible will not be rendered at all.

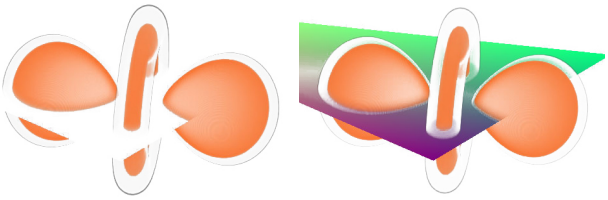

Figure 8: Modifying the ending geometry avoids rendering unnecessary parts of the volume, as shown in the left picture. Blending the clipped volume with the geometry then gives the correct result.

So before thinking about adding geometry, there should be a way to arbitrarily clip the volume. Fortunately, the bounding geometry introduced in section 3 offers a convenient way to do so. Modifying the starting points would result in clipping parts of the dataset from the viewer's side, which could be useful for 'opening up' the dataset if one wants to look inside without modifying the transfer function accordingly. Changing the ending points would result in clipping parts on the back side, which would be the same as putting a completely opaque object there (like our geometry for example).

Implementing this is straightforward: After the rendering of the front or back faces only the direction of the depth test needs to be changed and the clipping geometry must be rendered with the position inside the dataset encoded in the color channel. In the case of the back faces, this means that our initial algorithm retrieved the *last* backface, so the depth test was set to GL_GREATER, which ensures that only pixels with a z-value greater than the current value are drawn. Reversing this test to GL_LESS now makes sure that the bounding geometry is modified only where the clipping geometry is nearer to the viewpoint - all other parts will be discarded. Color-coding the clipping geometry with its position in the dataset ensures that whenever a value is modified, the correct ending positions for the ray will be written into the texture.

As shown in Figure 8, modifying the ending geometry results in occluded parts of the volume not being rendered, which gives an additional speedup of rendering time. All that is left to do is to draw the geometry *before* the volume is drawn and then blend the clipped volume onto the screen accordingly.

It is important to note that with this extension to our initial algorithm, the ending geometry could easily be *in front of* our starting geometry (because the clipping geometry could be in front of our front faces), yielding a negative direction vector. This makes it neccessary to check the direction vector before casting, which can be done in the fragment shader as well.

7 Fly-Through Applications

So far, we have the possibility of perspective projection that is an implicit feature of our basic algorithm. For some applications it might be interesting to move the viewpoint into the volume end explore the dataset in a fly-through mode like in Figure 9.

There shouldn't be a problem doing so as long as the camera doesn't touch the geometry, but as soon as the near clipping plane intersects the bounding geometry, holes will start to appear in our front faces, resulting in rays not being started where they should. What needs to be done is that whenever such an intersecion happens, all of these holes must be filled with the correct colors from the near clipping plane. In this section, we propose a novel aproach that fills all these holes with a number of simple tests in the

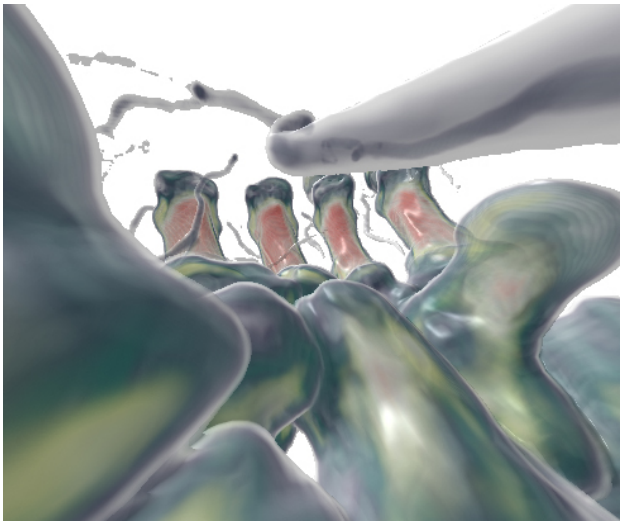

Figure 9: Exploring the dataset in fly-through mode.

depth buffer of the GPU.

A simple approach would be to draw the near clipping plane first (again with the colors encoding the absolute position in the dataset) and the front faces afterwards, ensuring that whenever there are no front faces to start from, the position of the near clipping plane will be taken. Unfortunately, this approach can only detect holes where no front faces are drawn at all (i.e. where the background color would shine through). If there is another object behind the current one, the front faces of this object would be visible and would be taken as starting positions for the ray (thus completely skipping the current object).

One way to avoid that is to first draw the backfaces to the depth buffer only, retrieving the z-value of the nearest

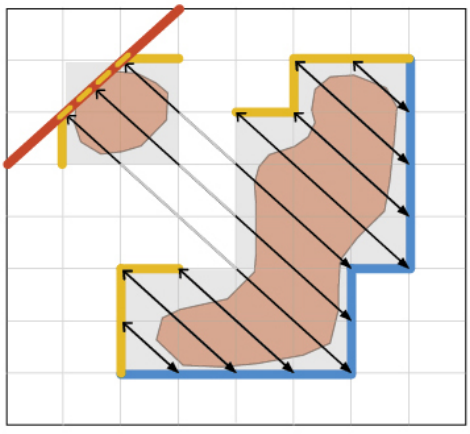

Figure 10: Example of the near clipping plane (red line) intersecting the bounding geometry. The dotted line is the hole in the bounding geometry that we need to fill with the near clipping plane.

backface, and render the front faces afterwards - this way, no front face behind the first object will be drawn, because its z-value would be greater than that of the nearest backface (see Figure 10). To sum this up again, drawing the front faces consists of three steps now:

1. Draw the color-coded near clipping plane with the depth buffer turned off.
2. Draw the back faces *only* to the depth buffer, ensuring that only the *first* front faces will be drawn.
3. Finally draw the front faces with depth buffer enabled, resulting in the correct starting positions for all rays.

8 Rendering of Large Datasets

As mentioned before, the biggest restriction of GPU based raycasting compared to CPU based approaches is the limitation of available memory. As of now, 256MB has been established as a standard for new graphics cards, which is just enough to store a 512x512x512 dataset in 16bit. Unfortunately, not all of the graphics card memory can be reserved for our dataset - the geometry information and the textures needed in the process of rendering take up some space as well. As shown in Figure 11, we're able to render a 580 MB dataset with only about 190 MB of space available for storing the volume data.

Considering the way our bounding geometry was created in section 3, inactive blocks don't contain important

Figure 11: Rendering this 512x512x1112 dataset is possible with our cached blocking scheme, as long as the active blocks fit into the memory.

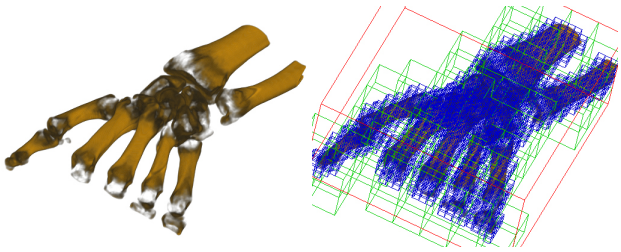

Figure 12: Rendering of a dataset with our 2-way blocking structure - cache blocks are marked green and bounding blocks blue.

information and shouldn't be stored. Having the dataset in a 3D-texture is convenient for our algorithm, so the 3D-texture should be preserved, but this texture does not have to be in the same order as the original dataset, nor does it have to have the same size. Storing only the active blocks in a new 3D cache texture would be one possible solution - unfortunately, in order to preserve correct trilinear filtering, blocks have to be stored with 1 extra border voxel. Otherwise a sample could be interpolated between voxels from different blocks, which of course would result in a wrong value. This means that for every 4x4x4 block of data from the original dataset, a 6x6x6 block has to be reserved in the cache texture. Obviously, with a block size of four more space would be lost than saved.

This suggests that the sweet spot for our bounding geometry (which is usually around 4) and the one for our cache texture are of a different magnitude. That said, a 2-way-blocking scheme looks like the best idea, with larger blocks (e.g. 32x32x32) for caching and smaller structures (4x4x4) for the bounding geometry. For the sake of efficiency (and simplicity), the small block size should be a factor of the large block size.

Of course the structure of the cache texture needs to be stored as well - this can be done with another low-resolution 3D-texture, that stores the position in the cache texture for every original cache block. All it takes then is another intermediary texture lookup in our fragment program to find the position of the current block.

Figure 12 illustrates what this 2-way blocking scheme looks like on a regular dataset - in this particular example, only 30.5% of the cache blocks are active, leading to a memory consumption of only 36.6% of the original texture size. Even better, only 7.2% of the bounding blocks are active, meaning that 92.8% of the dataset can be implicitly skipped via the bounding geometry.

9 Results

Compared to the original algorithm, significant speedups can be achieved by applying the techniques presented in the previous sections. Table 1 compares the basic algorithm presented in section 2 to the optimized algorithm

dataset size	BA[fps]	BG[fps]	BG&HR[fps]
256x256x128	14.6	41.7	350.0
512x512x333	4.4	24.3	66.4
512x512x1112*	1.5	5.1	15.1

Table 1: Iso-surface rendering of different datasets, comparing the basic algorithm to the presented optimizations. BA = the basic algorithm presented in section 2, BG = with bounding geometry, HR = hitpoint refinement and increasing sampling distance to 5.0. *With activated blocking.

dataset size	DVR [fps]	shaded DVR [fps]
256x256x128	40.5	26.7
512x512x333	23.9	11.3
512x512x1112*	4.4	1.5

Table 2: Comparing DVR and Shaded DVR of different datasets, bounding geometry is enabled. *With activated blocking.

with empty space skipping enabled. The third column shows the framerates that can be achieved by increasing the sampling distance to 5.0, which in the case of our datasets can be done without any visible differences when hitpoint refinement is enabled. In any case, the detail should be sufficient for moving and rotating the dataset, which can then be done at highly interactive framerates even for the large datasets. In general, the framerate is mostly dependant on screen resolution (i.e. the number of generated fragments) and sampling distance.

What can be seen in Table 2 is that unshaded DVR is almost as fast as iso-surface rendering, which is mainly due to the efficient empty space skipping preventing most of the dataset to be even looked at. Shaded DVR slows down rendering considerably, mostly because of the many samples needed for gradient reconstruction at each sampling point.

10 Conclusions and Future Work

This paper demonstrates that hardware based raycasting is much more than the conversion of well-known algorithms from the CPU to the GPU. Graphics processors have their own strengths and weaknesses, and exploiting these strengths while avoiding the weaknesses leads to completely different techniques than in CPU based approaches. Fortunately, the ongoing evolution of graphics cards will allow for even more efficient algorithms in the near future, and with the speed of GPUs growing at a much faster pace than that of CPUs, we're looking into a bright future for GPU-based approaches.

We have shown that it is already possible to implement a full-fledged raycasting environment on a GPU for all kinds of possible applications, minimizing most of the restrictions and achieving highly interactive framerates far above similar CPU approaches. This enables us to introduce new rendering modes like interactive fly-through DVR, which was next to impossible until now.

Finally, having the possibility to easily intersect the dataset with OpenGL-geometry efficiently is an important advantage as well, allowing for a number of interesting future applications like interactive 3D-pointers, augmentation tools or VR-objects in the same scene.

Other future work will include making the blocking scheme even more flexible, allowing for rendering of data that will *not* fit into the video memory as a whole, and full support of segmented datasets.

Acknowledgments

The VRVis research center is funded in part by the Austrian Kplus project. The medical data sets are courtesy of Tiani MedGraph.

References

[1] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 91–98, 1994.

[2] T. Cullip and U. Neumann. Accelerating volume reconstruction with 3D texture mapping hardware. Technical Report TR93-027, Department of Computer Science, University of North Carolina, Chapel Hill, 1993.

[3] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of Graphics Hardware 2001*, pages 9–16, 2001.

[4] A. Keller and W. Heidrich. Interleaved sampling. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 269–276, 2001.

[5] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization 2003*, pages 287–292, 2003.

[6] M. Meißner, U. Hoffmann, and W. Straßer. Enabling classification and shading for 3D texture mapping based volume rendering. In *Proceedings of IEEE Visualization '99*, pages 207–214, 1999.

[7] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of Graphics Hardware 2000*, pages 109–118, 2000.

[8] A. Van Gelder and K. Kim. Direct volume rendering with shading via three-dimensional textures. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 23–ff., 1996.

[9] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH '98*, pages 169–178, 1998.