Eurographics Symposium on Parallel Graphics and Visualization (2007)
Jean M. Favre, Luis Paulo dos Santos, and Dirk Reiners (Editors)

Scalable Sort-First Parallel Direct Volume Rendering with Dynamic Load Balancing

Brendan Moloney[1], Daniel Weiskopf[1], Torsten Möller[1], Magnus Strengert[2]

[1]GrUVi Lab, Simon Fraser University, Vancouver, Canada
[2]Visualization and Interactive Systems Group, University of Stuttgart, Stuttgart, Germany

Abstract

We describe a sort-first algorithm for parallel direct volume rendering on GPUs, with the intent of high scalability in regards to both performance and data set size. We explore three novel techniques for estimating the computation time for rendering each pixel, so that we can guarantee a good load balancing regardless of the level of frame-to-frame coherence. A bricking technique is used to subdivide the object space, thus allowing each rendering node to load only the bricks of data that are needed to render their respective portions of the image space. This enables us to render data sets larger than an individual GPU's texture memory. We cull bricks that do not contribute to the final image in order to reduce the data that is loaded and provide a coarse method of empty space leaping. We introduce a novel method of eliminating the overhead of generating vertices for the proxy geometry of each brick, by creating a single template of vertices that are used to render all bricks of the same size. Finally, detailed performance measurements document the various aspects of our algorithm.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Distributed/network graphics I.3.1 [Computer Graphics]: Parallel processing

1. Introduction

Three-dimensional texture slicing [CN93] is an easy way to interactively perform a high quality direct volume rendering (DVR) of small to medium sized data sets on almost any consumer level PC with a GPU. However, larger data sets often require more memory and performance than what is available on a single GPU. Using a number of GPUs to increase the rendering speed and memory is an attractive solution due to its price-to-performance ratio. We give an overview of previous work on parallel GPU based DVR in Section 2.

The texture memory attached to GPUs adds another level to the memory hierarchy (in addition to main memory, disk storage, etc.), which makes it difficult to consistently distribute both the data set and the rendering workload evenly. Algorithms for parallel DVR can be classified by their partitioning strategies. In Section 3 we compare the rendering pipelines that are required for three different partitioning strategies.

In Section 4 we detail our sort-first algorithm, which can guarantee a good distribution of the rendering workload regardless of the level of frame-to-frame coherence. Similar to previous work, we brick the data set so that we can cull based on the transfer function [MSE06, KSH03, CMCL06] and cache only the bricks that are needed by each GPU to render its portion of the image space [BHPB03]. With our current caching scheme we can handle data sets up to the size of the system RAM available to each node, but this restriction could be removed by caching bricks over the network or on local disk storage. For a detailed look at texture bandwidth in this scenario, we refer our readers to [BHPB03].

Our main contribution is a novel dynamic load balancing strategy that provides a guaranteed level of load balancing regardless of frame-to-frame coherence. We present three variations on how to estimate the rendering cost of a pixel. We analyze all three methods with a series of detailed experiments in Section 6. We also introduce a template based method of rendering a bricked data set using a slice based rendering engine, without incurring the penalty of increased vertex computations for the proxy geometry.

2. Previous Work

A variety of methods have been proposed for distributing a rendering workload among a number of machines. Molnar et al. [MCEF94] classify these into groups based on where in the rendering pipeline primitives are sorted in regards to viewing conditions. In sort-first methods the screen space is divided into regions and object-space primitives are sorted into these regions and distributed before rendering. In sort-last methods the object-space primitives are distributed, processed, and rasterized independently. Then overlapping pixels are sorted in depth order and composited together. For the special case of DVR, the object space primitives are textures. These textures generally require little to no processing, but they do have significant storage requirements.

The sort-last method probably is the most common for parallel GPU accelerated DVR. One of the primary research topics for sort-last algorithms is how to efficiently transfer and composite the intermediate images. Binary swap [MPHK94] and direct send [Hsu93] compositing schemes are easy to implement and do a fair job of distributing the compositing workload among the render nodes. SLIC [SML*03] improves direct send compositing primarily through better load balancing and scheduling. Hardware solutions to the compositing problem are also available [SEP*01, LMS*01, NKS*04] but they are expensive alternatives.

Sort-first methods for parallel GPU accelerated DVR generally either replicate the data set across all render nodes [ACCE04] or transfer data over the network [BHPB03]. Algorithms that replicate the full data set can only scale performance, but not the maximum data set size. Algorithms that transfer data over the network, or cache data on local storage, can allow for data sets close to the size of the total combined GPU memory.

Load balancing is another important research subject for parallel volume rendering, as the overall performance is limited by the slowest component. A common technique uses the relative performance of each rendering node in the previous frame. This has been done with sort-last algorithms [MSE06, MMD06] that subdivide the volume into bricks and reassign bricks to nodes that had a higher frame rate (less workload) in previous frames. Despite being sort last, these methods require data to be sent over the network and/or cached on the local hard disk. Sort-first algorithms have also used this method of load balancing [ACCE04], redistributing the image space instead of the object space. Any such method relies on frame-to-frame coherence and thus cannot guarantee any tight bounds on the level of load balancing.

3. Parallel DVR Pipeline

In Figure 1 we illustrate a generic parallel DVR pipeline, as well as the paths that three representative algorithms take. The (black) arrows along the center show the path taken by a typical sort-last algorithm, where the data is distributed among the render nodes once and then the intermediate images are composited together. The (red) arrows along the left show the path taken by a typical sort-first algorithm, where the full data set is loaded onto each GPU. The (blue) arrows along the right show the path taken by our algorithm, where we assume that each node has enough system RAM to hold the full data set although the GPU may not.

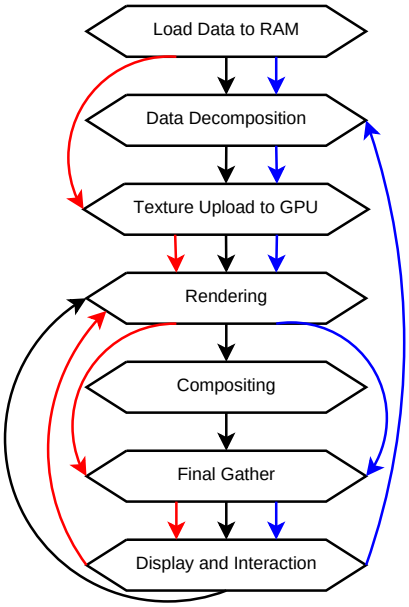

Figure 1: *A generic overview of the parallel DVR pipeline, with the paths of three algorithms traced through it.*

One obvious advantage to sort-first techniques is that we can completely skip the compositing stage. The number of pixels that need to be composited grows not only with image size but also with the number of rendering nodes. While the compositing computations can be well distributed, the number of intermediate images that must be sent over the network also grows with the number of nodes. This is a major issue on networks that have high-latency or are susceptible to packet collisions.

However, sort-first algorithms are inherently bad at distributing the data set. If we want to render a data set that is larger than the amount of memory available on the GPU, then we must loop all the way back to the data distribution stage of the pipeline and potentially load some data into textures before we can render the next frame. The motivation for taking this path is threefold: the overhead of loading data into textures should decrease (rather than increase) as we add more rendering nodes, data can be pre-cached in parallel to the rendering process (rendering and compositing is strictly sequential), and data redistribution is required for fully dynamic load balancing.

4. Algorithm

We divide our data set into a uniform grid of evenly sized bricks. We do this once, when the data is loaded, based on a user defined parameter for the size of the bricks. Each rendering node loads only the bricks intersected by its view frustum, as illustrated by a 2D example in Figure 2. By subdividing the data set in this manner, we can reduce the amount of data that must be replicated among the render nodes. We have a single level caching scheme where the full data set is held in main memory, and bricks that are needed are loaded from main memory into textures on the GPU. For load balancing, we estimate the computational cost of each pixel before rendering each frame. We then compute a kd-tree over the image space, such that each inner node of the kd-tree splits the image space into two portions, each with the same total rendering cost. Each rendering node is then responsible for rendering a portion of the image space corresponding to a leaf node in the kd-tree.

Our algorithm consists of three main components: the bricking technique used to divide up the object space, the computation of a per-pixel rendering cost, and the dynamic distribution of the image space based on this rendering cost. We explain each component, and how it interacts with the other components, in the following three subsections.

4.1. Object Space Bricking

By dividing the object space into bricks we can use a sort-first algorithm to render data sets that are larger than the memory available on any single GPU. We use an LRU (least recently used) caching scheme to determine which textures to reuse when we run out of texture memory. We currently assume that each rendering node has access to enough system RAM to hold the entire data set, because we only have a single layer of caching between local system memory and GPU memory. This caching mechanism imposes an additional overhead as data might need to be transferred to the GPU during the rendering. However, as mentioned in the last section, the amount of data to be transferred should decrease as the number of nodes increases and we could potentially pre-cache bricks in parallel to the rendering process.

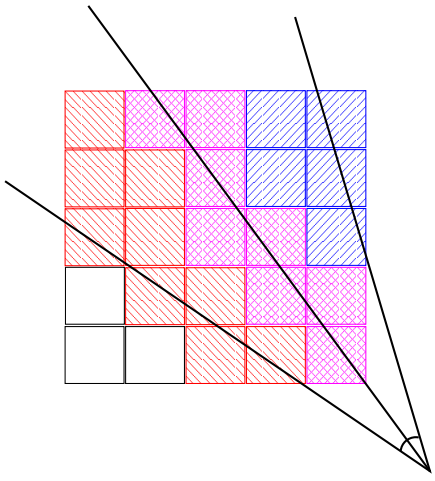

Figure 2: *The bricks with a (red) backward hatch pattern are loaded by the node rendering the left frustum, the bricks with a (blue) forward hatch pattern are loaded by the node rendering the right frustum, and the bricks with a (purple) cross hatch pattern are loaded by both.*

When choosing a brick size, we must balance the benefits of having a finer granularity in object space and the increased overheads from having a larger number of bricks. A finer granularity reduces data replication between rendering nodes along shared planes of the nodes' view frustums. This replication is illustrated by a 2D example in Figure 2. However, there is a per-brick memory overhead since adjacent bricks must share one data value at every point on their border so that the trilinear interpolation is consistent across bricks. When using a pre-integrated transfer function [EKE01], two data values must be replicated so that you can access the values for the back sides of the slabs at the boundary. When bricks are culled based on the transfer function, having a finer granularity can allow us to perform a more accurate culling, thus reducing the rendering workload and the amount of data that must be loaded. However, due to cache coherence and other overheads, rendering one large brick is always faster than rendering the same amount of data as a number of smaller bricks.A hierarchy of brick sizes has often been used to help balance these factors but in turn has it's own associated overheads (in particular memory usage).

The most significant per-brick performance overhead (when using a slice based rendering engine) is the increased number of vertices that must be generated on the CPU, and sent to the GPU, for the proxy geometry of each brick. To tackle this issue we devise a novel technique that computes a single vertex 'template' for each frame, which can be used to render every brick of the same size. This reduces the amount of computation on the CPU as well as the amount of data that must be transferred to, and stored on, the GPU. Without this technique, the rendering speed is severely CPU limited when rendering thousands of bricks.

4.2. Calculating Per-Pixel Rendering Cost

One of the advantages of sort-first techniques over sort-last is that one can generally do a better job of distributing the rendering cost evenly. Since we are using a slice based rendering engine, good load balancing is equivalent to having each GPU render the same number of fragments. Thus we need an estimate of how many fragments contribute to each pixel. This is directly proportional to the total length of the ray through a given pixel, that lies within some brick. We present three methods of computing the rendering cost for

the full image space, each with varying levels of accuracy and computational cost. The methods differ in the manner they compute the rendering cost for a single brick, but all three methods use additive blending to sum up the contributions of individual bricks.

The first method is completely accurate. This technique draws the back faces of the brick and for each fragment it computes the nearest intersection between the ray from the fragment to the eye and the front faces of the brick. This is about twice as slow as the other two methods on our target architecture (NVIDIA 6800 Ultra). However, on the latest generation NVIDIA 8800GTX this method is as fast as the two approximating methods and thus it would definitely be the method of choice.

The second method splats a sphere for each brick, with a diameter equal to the longest diagonal of the brick. This method will obviously assign a rendering cost outside of the brick's image space footprint, and even within the brick's footprint the assigned rendering cost will be inaccurate. This method is quite straightforward to implement, is much faster than the accurate method, and actually gives good load balancing results under certain viewing conditions (see Section 6).

The third and final method we implemented draws only the back faces of each brick, with an estimated rendering cost assigned to each vertex and then linearly interpolated for each fragment. We estimate the rendering cost for vertices, both inside the image space footprint and on the silhouette, based on the angle between the view direction and the normals of the brick faces. This method will obviously not assign any rendering cost outside of a bricks footprint, unlike the sphere splatting. Although the rendering cost will not be completely accurate within the footprint (due to us not dealing with the front faces accurately), it gives us better load balancing performance than splatting for some viewing conditions, and has a very similar computation time.

We also experiment with parallelizing the computation of the rendering cost. Trying to do this in a manner that balances the computational load on each rendering node leads to a chicken and egg scenario: we are trying to load balance the computation of our load balancing. However, assuming frame-to-frame coherence, we can reuse the image space decomposition from the previous frame. This approach should balance the computation of the rendering cost fairly well, because the time it takes to compute the rendering cost is proportional to the number of bricks that must be processed, which is in turn loosely proportional to the number of fragments that must be rendered for that same portion of image space. However, if we do not have good frame-to-frame coherence, the best we can do is tile the image space bounding box of the volume evenly.

4.3. Distributing the Image Space

Once we have computed the rendering cost for each pixel, we want to use this information to update the image space decomposition in a manner that distributes the workload evenly. We recursively construct a kd-tree, from top to bottom, over the image space to divide the rendering cost evenly. To do this efficiently we need to be able to quickly compute the total rendering cost for an area of the image space. Therefore, we compute a Summed Area Table (SAT) of the rendering cost, which allows us to query an area of the image space for its total rendering cost in constant time. For each inner node of the kd-tree, we perform a binary search along the direction perpendicular to the splitting direction in order to find the location of the balanced split.

For a parallel computation of the rendering cost, each render node computes the SAT only for the portion of the image space for which it computed the rendering cost. Then each render node broadcasts its SAT to all the other nodes. The SATs are attached to the corresponding leaf nodes in the kd-tree (of the previous frame) for that same portion of image space. Inner nodes of the kd-tree just hold the sum of the total rendering costs of its two children. We never fully combine these results into a SAT for the full image space, but rather we just evaluate the full SAT value for the positions we check in the binary search. The full combined SAT value for a specific position in image space is computed by a tree traversal, as described in Section 5.3.

5. Implementation

Our system is written in C and C++ with GLSL for GPU programming. We have one viewer node which interacts with the user, dispatches render requests to one render node, and receives render results from all render nodes, over TCP/IP. The viewer is responsible for tiling the intermediate images together for final display. The render nodes use MPI for communication, and OpenGL for rendering. The core rendering is based on 3D texture slicing with pre-integration [EKE01]. We describe the implementation for each of the three main components of the algorithm in the following subsections.

5.1. Object Space Bricking

We subdivide the data set into bricks in a preprocessing step. At the same time we compute a bit mask for each brick, corresponding to the scalar values that occur within that brick. This allows us to quickly and accurately cull bricks in the same manner as used in [CMCL06].

We found that our rendering times were heavily CPU limited when rendering several hundreds or thousands of bricks. This was because hundreds of slice vertices were being computed for every brick for each frame. Since the slices intersect all of the bricks at the same angle, the only information that is potentially different for each brick is an offset in the

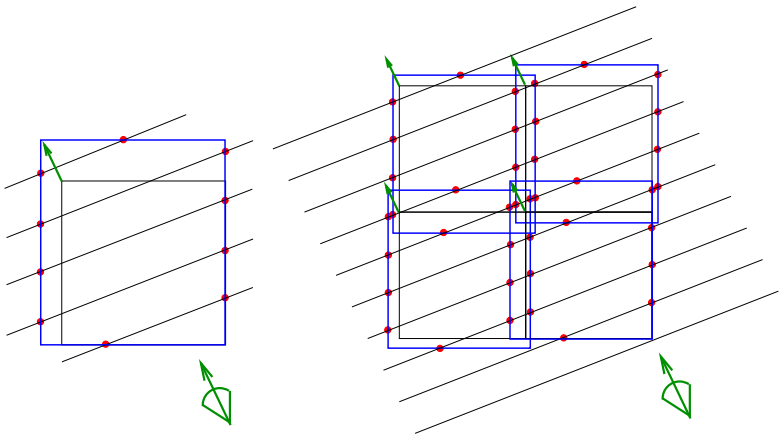

Figure 3: *The illustration on the left shows how we generate the slice vertex template. The small thin (black) box is a brick and the large thick (blue) box is the template bounding box, which is the size of the brick after being extended by one slice distance along the view direction (the green arrow). The (red) dots mark the actual vertices of the template. The figure to the right shows how these templates can be made to line up along brick boundaries, by translating them by some offset along the negative view direction.*

view direction that determines where the first slice intersects the brick. Since the number of slices for each brick differs by at most one, we can compute a slice template by expanding the brick along the view direction by one slice distance, and then use the vertices at the intersection points of this expanded brick and the slice planes. Figure 3 illustrates this concept.

We can use this slice template to render any brick by simply computing the offset along the view direction for the first intersection, and then translating the templated slice vertices along the direction opposite of the view direction. Since the vertices themselves never change, they can be loaded onto the GPU as a vertex buffer object once at the beginning of each frame. The pre-integration texture coordinates are computed in a vertex shader program on the GPU. Since the templated slices are larger than the actual bricks, we apply user defined OpenGL clip planes to kill any fragments that lie outside of the actual brick. In order to minimize the number of fragments that we need to kill, we compute one template for each brick size (at most eight, due to the data set size not being evenly divisible by the brick size).

5.2. Calculating Per-Pixel Rendering Cost

For all three methods of computing the rendering cost, we use a Frame Buffer Object (FBO) with one 16-bit floating point render target and additive blending to sum up the contributions of each brick.

For an initial attempt at an accurate method we tried rendering the front faces' depth into a buffer and then rendering back faces and taking the difference in the depths. This was too slow for significant numbers of bricks because it requires two passes for each brick. Instead, we compute the distance between the front and back faces in a single pass by rendering the back faces and intersecting the ray from each fragment to the eye with the front faces. We do the intersections in object space, so dot products with the normals of the planes is just selecting a single component.

For the second method, we generate a single texture containing a spherical footprint. We choose a texture size of 32^2 to maximize cache coherence. For each brick we render a quad, textured with this spherical footprint, at the brick's center with a width and height equal to the longest diagonal of the brick. In order to accommodate multiple brick sizes, the fragment shader scales the values from the texture by the size of the brick. We do not perform any kind of perspective correction to the texture.

For the third method, we identify the front and back faces of each brick as well as silhouette and non-silhouette (interior) vertices. At the same time we find the maximum dot product between the viewing direction and the normals for the faces of the bricks, as well as the corresponding face that yields the maximum dot product. We use the value of the maximum dot product to estimate the rendering cost that we attach to the silhouette and interior vertices. An interior vertex is assigned a smaller value when the dot product is large (looking at the brick head on) and a larger value when the dot product is small (looking along the diagonal). Therefore we use the inverse of the dot product, scaled to the range of zero to one, and scaled again by the size of the brick.

The rendering cost at the silhouette vertices is actually always zero, but since we are not taking the front faces into account this can cause the rendering cost to drop off too quickly when it is interpolated between vertices. If a silhouette vertex is just barely outside of the front face with the maximum dot product in image space (maximum dot product is larger), we assign a larger value to that vertex since the rendering cost is actually quite high near this vertex (even if it is actually zero at the vertex). We achieve this behaviour by using the compliment of the weight assigned to the interior vertex (before being scaled by the brick size) as the weight assigned to the silhouette vertices that do not lie on the face with the maximum dot product. Silhouette vertices that do lie on the face with the maximum dot product are assigned a weight of zero.

5.3. Distributing the Image Space

We use an axis aligned bounding box in the image space to reduce the number of pixels that are read back from the FBO and processed in the SAT computation. If we are not parallelizing the computation of the rendering cost, then each node computes a SAT for the rendering cost over the full image space. Then we compute a kd-tree over the image space by doing a binary search for the new split point along the appropriate row or column of the SAT. The end result is that each leaf node has the same total rendering cost. Each rendering node is then assigned a portion of the image space corresponding to a leaf in the kd-tree.

If we are parallelizing the computation of the rendering cost, then after each node computes the rendering cost for its portion of the image space, it also computes a SAT for that same portion of image space. Then each node traverses the kd-tree from the last frame and performs an MPI Bcast for each leaf node to either send or receive the SAT for that portion of the image space (depending on whether it is the portion of image space that was computed locally). We compute the new kd-tree in a manner similar to what we described above for the non-parallelized case. The only difference is that when we do the binary search, we need to query more than one SAT for each search position. For each search point we traverse the old kd-tree from top to bottom looking for inner nodes that are completely within, or leaf nodes that are intersected by the area we are querying. Nodes that lie completely within the area just report their total cost; leaf nodes that are intersected by the area bounds report the SAT value at the intersection point. Combining these results gives us the total SAT cost for the search position.

Broadcasting the full SATs is actually unneccesary, as you could just broadcast the total values and then let each node search for the split points in their region of image space. We did not test this method, but it should allow for better scalability.

6. Experiments and Results

For testing we used a cluster of 8 render nodes, each with AMD Opteron 248 CPUs and an NVIDIA GeForce 6800 Ultra GPU (256MB AGP). The interconnect was provided by an Infiniband network. For each test we ran an animation with 1024 frames and averaged the results.

The first experiment tests the performance impact of our templated slice technique. Since the overhead we are targeting corresponds to the number of bricks, not the image size, we use a 128^2 viewport and 5 different brick sizes on the same 256^3 volume. Table 1 shows that the templated slice technique out performs the standard slicing technique by as much as a factor of 7.

The second experiment compares the computation times for the three methods of computing the per-pixel rendering cost.

Table 1: *The performance of the templated slicing technique compared to the standard slicing technique.*

	Rendering Time (ms)		
# Bricks	Standard	Templated	Speed Up
1	3.78	3.77	1.00
9	9.05	5.23	1.73
25	24.21	5.84	4.15
729	46.41	9.21	5.04
6859	367.70	53.33	6.89

We did this on a single node with 4 different brick sizes and 5 different image resolutions. The graph in Figure 4 shows that the back face method for computing the rendering cost is slightly faster than the splatting method except for when we are rendering 5,832 bricks with a low resolution viewport. This is as expected since the back face method has a higher vertex cost (as many as 20 vertices per brick versus 4 for splatting), while the splatting method has a higher fragment cost due to the fact that it draws outside of the brick's footprint in image space. The graph also shows that all three methods share a linear behaviour in the number of pixels, and differ by an offset along the y-axis that corresponds to the number of bricks being processed. These offsets are shown in more detail in Table 2. Clearly the offset is much worse for the accurate method, with the back face and splatting methods being rather similar. We noticed that the splatting and back face methods' performance increased by almost a factor of two when using a PCI-E GeForce 6800 Ultra, which is due to the much faster framebuffer readback.

The next experiment looked at how well the performance of our rendering cost computation methods scaled in regards to the number of nodes. In Figure 5 we show the results for the accurate and back face methods on a data set with 5,832 bricks. For the backface method we show results for two res-

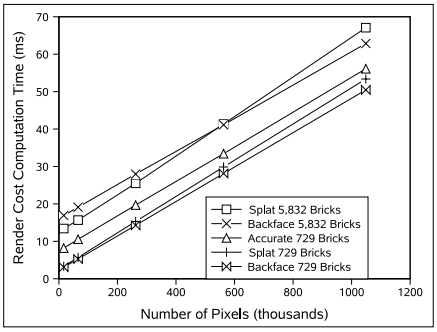

Figure 4: *The performance of all three methods for computing the rendering cost for various image sizes and 729 bricks. Two of the methods are also plotted using 5,832 bricks.*

Table 2: *Computation times for all three methods of computing the rendering cost with four different brick sizes. The viewport resolution is* 128^2.

	Rendering Cost Computation Time (ms)		
# Bricks	Accurate	Splatting	Back Face
9	1.33	1.06	0.99
25	2.37	1.44	1.29
729	8.21	3.30	3.10
5832	46.79	13.42	16.92

Table 3: *The deviation of render times among nodes, for all three methods of computing the rendering cost. We use two different types of animations, and both full resolution and quarter resolution for the rendering cost function.*

	Average Render Time Deviation		
Animation	Accurate	Splatting	Back Face
Global (Full)	0.089	0.175	0.142
Global (Quarter)	0.108	0.166	0.143
Local (Full)	0.064	0.093	0.147
Local (Quarter)	0.072	0.093	0.147

olutions. We do not show the splatting results because they are very similar to the back face results. The accurate method benefits the most from the parallelization, with the back face method showing almost no improvement moving from 4 to 8 nodes. Avoiding the SAT broadcast, as previously mentioned, should give better scaling results.

The quality of a load balancing algorithm is best characterized by the relative deviation in the rendering time among different nodes; there are too many variables to give a single FPS or speed up factor. Therefore we took the average of the differences between the maximum and minimum render time for each frame. Then we normalized this by the average render time to obtain a measure of how much the rendering times deviated on different nodes relative to the average render time. For all three methods we used 8 nodes to render a 512^3 volume with a 1024^2 viewport. We used two animations, a 'global' one that rotates the volume while keeping it fully in the view frustum, and a 'local' one that zooms in on the volume and then pans and rotates around. We ran each experiment twice, once with the rendering cost computed at the same resolution as the image and again with the resolution set to one quarter of the image resolution (half in each dimension). Our results are given in Table 3. As expected, the accurate method did the best job of load balancing, and the back face method outperformed the splatting method for the global animation. The splatting method's results varied greatly for the different animations, which is due to its highly approximative nature. The back face method provided consistent results for both animations and resolutions. The lower resolution did not hurt the load balancing much for any method.

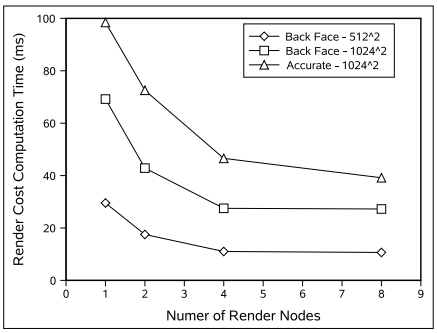

Figure 5: *A plot of the scaling behaviour of the accurate and back face methods of computing the rendering cost. We plot two resolutions for the backface method.*

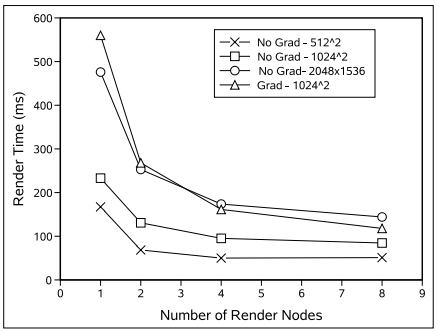

Figure 6: *A plot of the scaling behaviour of the overall performance of our algorithm, using a real world data set and transfer function. We show three image resolutions, and the effect of loading gradients into the textures.*

A visual comparison of the three methods for computing the rendering cost is shown in Figure 7. The corresponding rendered image is also shown, with and without the brick outlines displayed. The scaling of the overall performance of our algorithm for this same data set (512x512x400) and transfer function is shown in Figure 6. We plot the performance for three image resolutions without gradients being loaded into the textures, and for the middle resolution we also plot a run with gradients being loaded. The difference in performance between the run with gradients versus the run without, shows the effect of data loading on performance (we are quadrupling the texture sizes). The difference between the runs with and without gradients diminishes as we add more render nodes, confirming our prediction that the data loading overhead grows smaller as we add rendering nodes. Since the data set with gradients is too large for a single GPU, OpenGL itself is swapping the textures in and out of memory for the single node run. The global animation

was used for this test, so we picked the back face method with a quarter resolution for our load balancing. For our target architecture, we recommend the back face method due to its fast computation and small variation in quality of load balancing. Using a quarter resolution for the rendering cost lowers the overhead of the load balancing without significantly impacting the quality. For this choice, the load balancing computation consumes between 6.5 and 13.4 percent of the total rendering time. Therefore we achieve good load balancing, regardless of the level of frame-to-frame coherence, for an acceptable cost.

7. Conclusions and Future Work

We have presented a sort-first algorithm for parallel DVR with a load balancing strategy that can guarantee a good level of load balancing under all conditions. Our algorithm scales not only performance, but also the maximum size of a data set that can be rendered. We have shown how to eliminate one of the biggest overheads incurred when using bricking techniques and slice based rendering engines, using our novel templated slice technique.

We have presented three different methods of computing the rendering cost function that drives our load balancing algorithm, each with different performance and load balancing characteristics. We have analyzed these differences through a detailed series of experiments, exposing each method's strengths and weaknesses.

As mentioned previously, adding a second layer of caching (where bricks that are needed and not currently available in system RAM would be fetched over the network or from disk) would allow us to render larger data sets in distributed memory environments. Using a sort-first partitioning allows for certain techniques, such as early ray termination, that are not feasible with sort-last partitioning. Exploring ways to incorporate such techniques into our algorithm could be another avenue of future work. Our algorithm is also extremely well fit to time-varying data since it allows for fast culling, good load balancing regardless of temporal coherence, and data caching is no longer a penalty since it is required. A focus on time-varying data would certainly be interesting.

8. Acknowledgements

We would like to thank Brown & Herbranson Imaging, Stanford Radiology, and The Rosicrucian museum for the mummy data set. This work was funded in part by the Natural Sciences and Engineering Research Council of Canada.

References

[ACCE04] ABRAHAM F. R., CELES W., CERQUEIRA R., ELIAS J. L.: A load-balancing strategy for sort-first distributed rendering. In *Proc. SIBGRAPI* (2004), pp. 292–299.

[BHPB03] BETHEL E. W., HUMPHREYS G., PAUL B., BREDERSON J. D.: Sort-first, distributed memory parallel visualization and rendering. In *Proc. IEEE Symp. Parallel Large-Data Vis. Graphics (PVG)* (2003), p. 7.

[CMCL06] CASTANIE L., MION C., CAVIN X., LEVY B.: Distributed shared memory for roaming large volumes. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1299–1306.

[CN93] CULLIP T. J., NEUMANN U.: *Accelerating Volume Reconstruction With 3D Texture Hardware*. Tech. rep., Chapel Hill, NC, USA, 1993.

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. ACM SIGGRAPH/EG Workshop Graphics Hardware (HWWS)* (2001), pp. 9–16.

[Hsu93] HSU W. M.: Segmented ray casting for data parallel volume rendering. In *Proc. Symp. Parallel Rendering* (1993), pp. 7–14.

[KSH03] KAHLER R., SIMON M., HEGE H.-C.: Interactive volume rendering of large sparse data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics 9*, 3 (2003), 341–351.

[LMS*01] LOMBEYDA S., MOLL L., SHAND M., BREEN D., HEIRICH A.: Scalable interactive volume rendering using off-the-shelf components. In *Proc. IEEE Symp. Parallel Large-Data Vis. Graphics (PVG)* (2001), pp. 115–121.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Comput. Graphics Appl. 14*, 4 (1994), 23–32.

[MMD06] MARCHESIN S., MONGENET C., DISCHLER J.: Dynamic load balancing for parallel volume rendering. In *Proc. EG Symp. Parallel Graphics Vis. (PGV)* (2006), pp. 43–50.

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graphics Appl. 14*, 4 (1994), 59–68.

[MSE06] MÜLLER C., STRENGERT M., ERTL T.: Optimized volume raycasting for graphics-hardware-based cluster systems. In *Proc. EG Symp. Parallel Graphics Vis. (PGV)* (2006), pp. 59–66.

[NKS*04] NONAKA J., KUKIMOTO N., SAKAMOTO N., HAZAMA H., WATASHIBA Y., LIU X., OGATA M., KANAZAWA M., KOYAMADA K.: Hybrid hardware-accelerated image composition for sort-last parallel rendering on graphics clusters with commodity image compositor. In *Proc. IEEE Symp. Vol. Vis. Graphics (VolVis)* (2004), pp. 17–24.

[SEP*01] STOLL G., ELDRIDGE M., PATTERSON D., WEBB A., BERMAN S., LEVY R., CAYWOOD C., TAVEIRA M., HUNT S., HANRAHAN P.: Lightning-2: A high-performance display subsystem for PC clusters. In *Proc. ACM SIGGRAPH* (2001), pp. 141–148.

[SML*03] STOMPEL A., MA K.-L., LUM E. B., AHRENS J., PATCHETT J.: SLIC: scheduled linear image compositing for parallel volume rendering. In *Proc. IEEE Symp. Parallel Large-Data Vis. Graphics (PVG)* (2003), pp. 33–40.

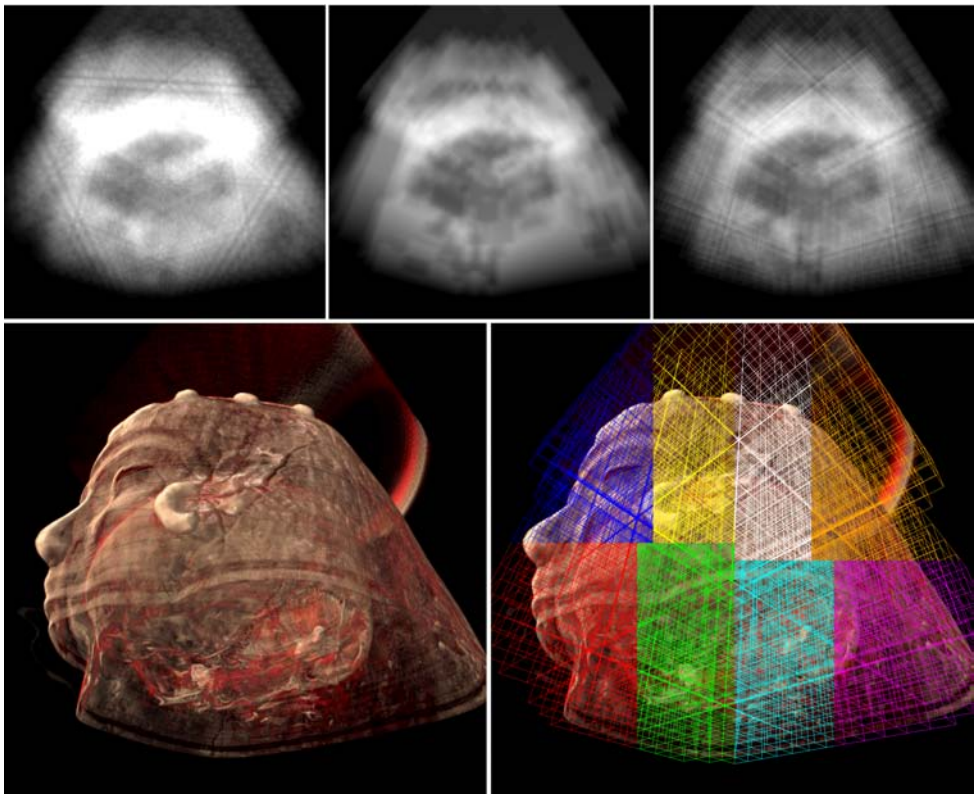

Figure 7: *Images generated with the Mummy data set and the same viewing conditions. Top row from left to right: greyscale visualization of the rendering cost estimated by the splatting method, accurate method, and back face method. Brightness is normalized for print quality. Bottom row: The final volume rendered with and without the brick boundaries shown. In the right image, each node uses a unique color for the brick boundaries in its portion of the image space, illustrating the kd-tree decomposition.*