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

Direct Send Compositing for Parallel Sort-Last Rendering

Stefan Eilemann and Renato Pajarola

Visualization and MultiMedia Lab, Department of Informatics, University of Zürich

Abstract

In contrast to sort-first, sort-last *parallel rendering has the distinct advantage that the task division for parallel geometry processing and rasterization is simple, and can easily be incorporated into most visualization systems. However, the efficient final depth-compositing for polygonal data, or alpha-blending for volume data of partial rendering results is the key to achieve scalability in sort-last parallel rendering. In this paper, we demonstrate the efficiency as well as flexibility of the* direct send *sort-last compositing algorithm, and compare it to existing approaches, both in a theoretical analysis and in an experimental setting.*

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Graphics Systems]: Distributed Graphics; I.3.m [Miscellaneous]: Parallel Rendering; I.3.7 [Three-Dimensional Graphics and Realism]: Virtual Reality

1. Introduction

1.1. Motivation

Among the basic parallel rendering approaches outlined in Figure 2 [MCEF94], *sort-last* has received increasing interest from visualization application developers, see. for example [CKS02, SML*03, CMF05, CM06]. On one hand this is due to the high scalability and good load-balancing it can offer, but at least as important is its simplicity of task decomposition which makes it a prime candidate to extend visualization software to high-performance parallel rendering.

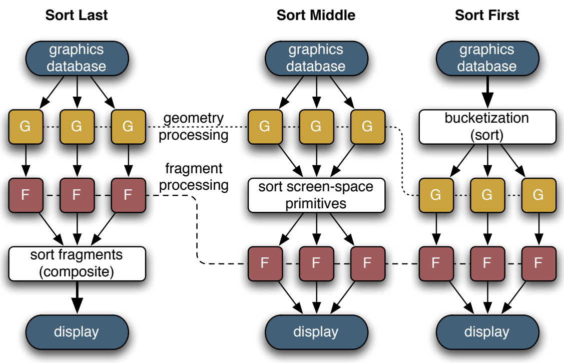


Figure 2: *Fundamental sort-last, sort-middle and sort-first rendering data processing stages.*

The main *sort-first* parallel rendering's bottleneck is its overhead for processing split primitives on multiple tiles. Moreover, besides this additional processing cost, the sort-first bucketization is more complex to integrate into existing visualization applications and non-trivial to load-balance, hence limiting its scalability and general applicability. *Sort-middle* solutions can only be implemented if intercepting projected geometry before rasterization is supported by the graphics system. Furthermore, it requires that rasterization units are re-configurable on-the-fly to different viewport regions for efficient load-balancing.

The compositing stage is the key element of the *sort-last* parallel rendering pipeline. To benefit from a simple and effective load-balanced task decomposition of geometry processing and rasterization, the rendered intermediate images have to be composited together efficiently for final display. Fundamentally, the amount of pixel data to composite is a linear function of the number of parallel rendering units as each generates a full-size partially complete frame. In general the composition task is a per-pixel z-buffer depth-visibility test, for polygonal data, or back-to-front α-blending, for volume data. Several algorithms to efficiently parallelize the image composition task have been proposed, with binary swap [MPHK94] being the most commonly used. In theory it achieves a constant compositing time by distributing the work equally across all rendering units. However, it requires a power-of-two number of parallel processing units and exhibits a significant synchronization overhead.

In this paper, we present an improved and more flexible sort-last compositing algorithm called *direct send* (Figure 1). We analyze its theoretical cost model with respect to com-

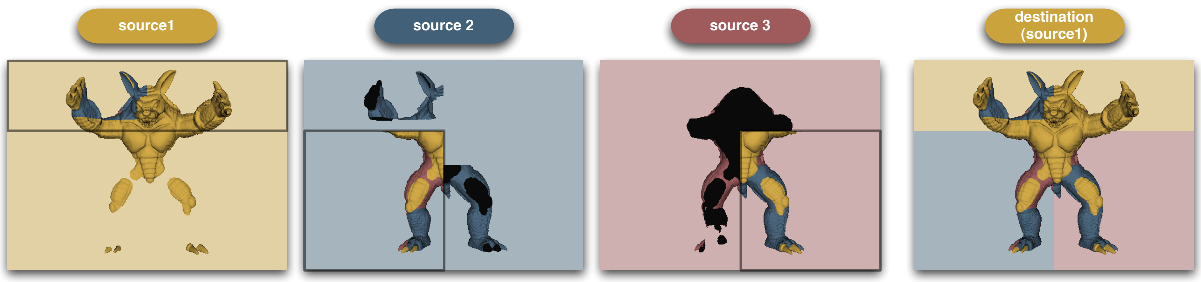


Figure 1: *Sort-last rendering with direct send compositing using three channels. The first three images show the rendering buffers which contain the rendering of the partial database and the composed tile of this channel. The last image shows the final, visible destination view with the fully composited image. The channels use different clear colors to identify the tile layout.*

positing time, and we provide experimental evidence that it achieves the same or better performance as binary swap while supporting any number of parallel rendering units.

2. Related Work

Most approaches to sort-last parallel rendering have developed special-purpose hardware solutions for the image compositing process. This trend was mainly due to the limited pixel read-back rates and narrow bandwidth of the CPU-GPU interface of contemporary graphics hardware. Several hardware architectures have been designed in the past for sort-last parallel rendering, such as Sepia [MSH99], Sepia 2 [LMS*01], Lightning 2 [SEP*01], Metabuffer [ZBB01], MPC Compositor [MOM*01] and PixelFlow [MEP92, EMP*97], of which only a few have reached the commercial product stage (i.e. Sepia 2 and MPC Compositor). However, the inherent inflexibility and setup overhead have limited their distribution and application support.

Software-based algorithms have two fundamental bottlenecks in the composition: the CPU-GPU transfer and the transmission from the source to the destination node. Recent advances in the speed of CPU-GPU interfaces, such as PCI Express, and system interconnects, such as InfiniBand, made software-based sort-last composition feasible for interactive applications.

The most simple sort-last compositing solution is the *serial* approach which combines and merges all intermediate images on the destination rendering unit responsible for the final display. Several parallelization schemes for software composition have been proposed. Most notably, these include direct send [SML*03], binary tree [SGS91], binary swap [MPHK94] and parallel pipeline [LRN95], among which binary swap is the most commonly used algorithm.

Several improvements to the various parallel sort-last compositing algorithms have been proposed. One approach is to reduce the pixel data to cover only the screen-space bounding rectangle, as in [YYC01]. Furthermore, to accelerate the image transport several compression-based methods have been proposed [AP98,TIH03,SKN04]. While these optimizations can dramatically improve the framerate, sort-last rendering on high-resolution displays at interactive framerates is still hard to achieve.

In the context of the proposed *direct send* compositing algorithm we exploit fragment level optimizations using a stencil-based z-buffer visibility compositing, or back-to-front ordered α-blending method for polygonal and volume data respectively.

3. Theoretical Analysis

Throughout this paper we use the parallel and distributed rendering terminology introduced in the Equalizer framework [Equ06]. Thus a *channel* is a single OpenGL view within an OpenGL drawable, assumed to be executed on a GPU and thread separate from other channels. A *compound* is the structure used to describe the task decomposition and partial-result recomposition in parallel rendering as described in [BRE05].

3.1. Algorithm Overview

In sort-last parallel rendering, the main task decomposition for geometry processing and rasterization is a simple *database partitioning*. This partitioning is often trivial, in contrast to sort-first parallel rendering, and not further discussed here [MCEF94]. On the other hand, the task of final image compositing, which is a simple image mosaic assembly in sort-first, is more demanding in sort-last parallel rendering. Generally, n rendering channels will generate n full-size partial images, containing color and potentially depth. These n images have to be merged considering per-fragment z-visibility, or α-blending in volume rendering.

Direct send compositing divides this final image gathering task into n screen-space tiles to avoid exchanging full-size images between the n compositing channels. Each tile is associated to and composited by one channel, and the composited tiles are eventually assembled together to form the final image. Note that while we assume the drawing – geometry transform, lighting, and rasterization – and compositing channels to coincide, this is not a general restriction as compositing can be assigned to a subset of the drawing channels or to a different set of channels altogether. An important observation though, is that the layout and shape of the tiles can be arbitrary and optimized for the best frame-buffer read-back and write access path provided by the graphics hardware.

Figure 3 illustrates the z-compositing stages for a 3-channel parallel polygonal rendering compound. After drawing the partial databases – the scene divided into n parts – the n image tiles are exchanged between the channels and composited one on each channel. Finally the $n-1$ missing tiles are assembled on the channel responsible for final display.

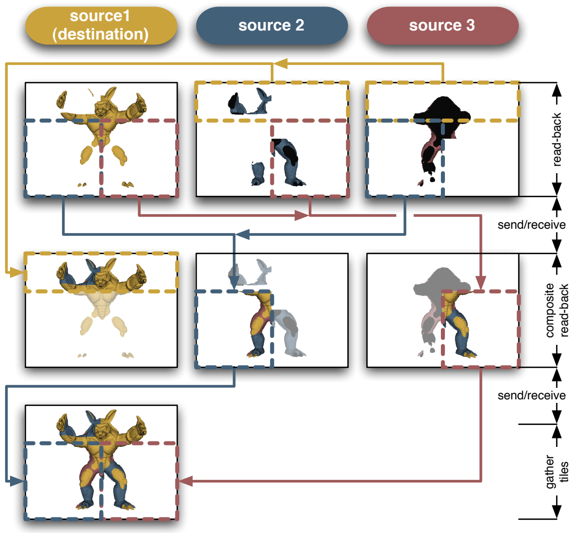


Figure 3: *Direct send compositing with three channels.*

Figure 4 shows the corresponding fragment operations of the three channels from Figure 3. Each channel has to read back $n-1$ image tiles from its own frame-buffer and send them to the appropriate compositors, and in turn receives $n-1$ image parts for its 'own' tile z-composition.

In the following theoretical analysis we will focus on the total time cost for the composition stage. As mentioned earlier, we assume that the drawing channels are also the compositing channels. The other operations to form the image, such as the draw update, are not relevant in the context of this theoretical discussion. We compare our algorithm against the simple serial composition and binary swap. We assume a similar rendering cost per channel, full-frame pixel coverage and equal tile distribution.

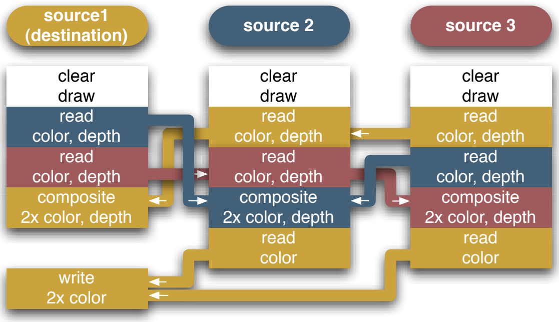


Figure 4: *Three channel per-frame update operations.*

3.2. Serial Composition

For serial composition, all channel images are directly composited on the destination channel. As expected, the complexity of this algorithm is linear $O(n)$. Let us assume that t_r is the time to perform a full frame read-back (color and depth), t_l is the latency introduced by transmitting a full frame from one channel to another and t_d is the time to do a full frame composition. Considering that the read-back is done in parallel on all channels, the total serial compositing time t_{serial} using n channels is then:

$$t_{serial} = t_r + (n-1) \cdot \max(t_l, t_d) \qquad (1)$$

The delay $(n-1) \cdot \max(t_l, t_d)$ is caused by the fact that $n-1$ full frame images have to be sent to and composited on a single destination channel one-by-one. Without pipelining the transmission and composition tasks, the delay would effectively increase to $(n-1) \cdot (t_l + t_d)$. The overall image data exchanged is simply $n-1$ times a full frame.

3.3. Direct Send

Direct send executes six operations to form the final display image: (i) read-back of $n-1$ tiles, (ii) send the tiles to their compositing channels, (iii) composition of $n-1$ copies of the tile it 'owns', (iv) read-back of $n-1$ composited tiles, (v) send tiles to final display destination channel, (vi) assemble the $n-1$ received tiles into the final image on the destination channel. Steps i to iii are performed on each channel, Steps iv and v on $n-1$ (source) channels and Step vi is executed only on the final display (destination) channel. The direct send compositing time t_{ds} then consists of:

$$t_{ds} = t_{composite} + t_{read\ tile} + t_{gather} \qquad (2)$$

The individual times are given below. The composition time $t_{composite}$ (3) consists of reading back and sending $n-1$ tiles, and pipelined composition of $n-1$ tiles from the other channels, each of size $\frac{1}{n}$. The read-back and compositing is performed in parallel on all channels without overhead. Moreover, also the exchange of tiles between channels does not incur any other overhead, besides $\frac{n-1}{n} \cdot t_l$, since each channel sends and receives exactly $n-1$ partial frames of size $\frac{1}{n}$. The time $t_{read\ tile}$ (4) consists of reading back a single tile of size $\frac{1}{n}$ on each source channel from the frame buffer, with t_{rc} being the time to read a (color only) full frame. The gathering step with t_{gather} (5) receives $n-1$ tiles of size $\frac{1}{n}$ on the destination channel from all source channels and draws them (pipelined) side-by-side into the final destination frame buffer, with t_{dc} being the time to draw and t_{lc} the latency of receiving a full frame.

$$t_{composite} = \frac{n-1}{n} \cdot (t_r + \max(t_l, t_d)) \tag{3}$$

$$t_{read\ tile} = \frac{t_{rc}}{n} \tag{4}$$

$$t_{gather} = \frac{n-1}{n} \cdot \max(t_{lc}, t_{dc}) \tag{5}$$

In contrast to serial compositing, image transmission is performed concurrently as tiles of size $\frac{1}{n}$ are exchanged between all channels. Thus there is only $\frac{t_l}{n}$ latency to be introduced per tile. Furthermore, the amortized image data transmitted between all channels is only two full frames[†], in contrary to the $n-1$ full frames for serial compositing. One full frame of image data is exchanged in both, the compositing and the gathering stage.

Note that direct send is capable of using a different number of channels for compositing and drawing. The compositing time changes when using n compositing channels on m draw channels ($n \leq m$). Each compositing channel has to read back $n-1$ tiles of size $\frac{1}{n}$, and to composite $m-1$ tiles of the same size. Thus we can exchange the term (3) from above in Equation 2 by

$$t_{composite} = \frac{n-1}{n} \cdot t_r + \frac{m-1}{n} \cdot \max(t_l, t_d). \tag{6}$$

The readback time of the draw-only channels (n tiles of size $\frac{1}{n}$, i.e., a full frame) is completely hidden by the compositing time of the other channels.

3.4. Binary Swap

Binary swap [MPHK94] consists of the same steps for the tile read-back, compositing and final gathering as direct send. However, the composition step differs in that a series of $\log_2 n$ composition steps are executed, as illustrated in Figure 5 for $n = 4$. After each step, half of the assembled image region is swapped with a partner channel for composition, until all tiles have been fully assembled. Since only pairs of nodes exchange image tiles for composition in each step, no pipelining of transmission and compositing is possible. The term (3) is thus replaced for binary swap by:

$$t_{composite} = \sum_{i=1}^{\lfloor \log_2 n \rfloor} \frac{1}{2^i} \cdot (t_r + t_l + t_d). \tag{7}$$

[†] actually $2 \cdot \frac{n-1}{n}$ full frames

In each step i a fractional image frame region of size $\frac{1}{2^i}$ is exchanged between partner channels which amortizes to a full image frame during the compositing stage eventually ($\sum_{i=1}^{\log_2 n} \frac{1}{2^i} \leq 1$). Another full frame is to be received by the destination channel during the gathering stage for final assembly and display.

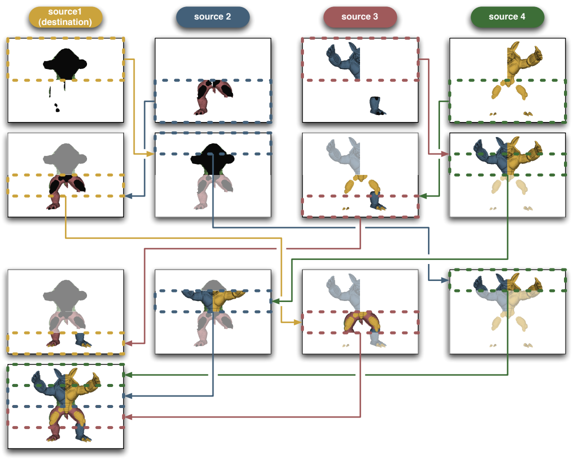


Figure 5: *Binary swap compositing with four channels.*

3.5. Comparison

In theory, the composition time for binary swap and direct send only differ in the main compositing time $t_{composite}$. The series $\sum_{i=1}^{\lfloor \log_2 n \rfloor} \frac{1}{2^i}$ in (7) approaches 1.0. If a power-of-two number n of channels for drawing and compositing is used, the total composition time t_{binary} then converges to:

$$t_{binary} \approx t_r + t_l + t_d + \max(t_{lc}, t_{dc}). \tag{8}$$

On the other hand, for direct send and for any number n of channels, due to pipelining of the transmission and compositing tasks, the total composition time t_{send} converges to:

$$t_{send} \approx t_r + \max(t_l, t_d) + \max(t_{lc}, t_{dc}) \tag{9}$$

Binary swap is based on $\log_2 n$ tiles per channel, whereas direct send uses $n-1$ tiles. Note that the higher number of tiles used by direct send could have a negative impact on performance only if the compositing or frame buffer readback operations have a significant static setup cost.

Transmission of image data is equivalent for both algorithms as two times a full frame of image data is exchanged between the contributing channels. An advantage of direct send is that image regions exchanged between channels can be limited to a constant $1/n$ of a full frame, hence limiting peak point-to-point image transfer bandwidth. In contrast, binary swap starts out with $1/2$ sized tiles and reaches $1/n$

regions only in its last of the $\log_2 n$ steps. Note that we assumed here that the total communication bandwidth, i.e. in the network switch, is sufficiently high to service the concurrent point-to-point image transmissions during the compositing stage of both direct send and binary swap. If this assumption does not hold, the communication patterns and the peak point-to-point image transfer bandwidth may have to be considered in more detail.

However, the main difference between binary swap and direct send lies in the synchronization and flexibility. While direct send only needs two synchronization points, independently of the number n of channels, binary swap depends on $\log_2 n + 1$ synchronization points. As observed in [CMF05] this can have a significant negative impact on the parallelism, in particular as n grows.

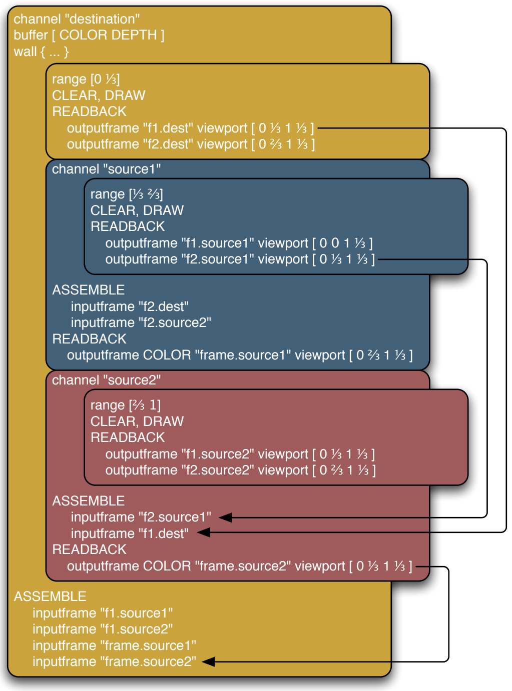
Moreover, the direct send algorithm adapts to any number of channels with the same performance characteristic, and easily supports any combination of different numbers of drawing and compositing channels. Additionally, the low constant synchronization overhead scales well and is a significant advantage in cluster-parallel environments where any latency due to network transfer synchronization may cause dramatic slow downs.

4. Implementation

We have implemented the proposed direct send compositing algorithm along with binary swap and serial compositing in the *Equalizer* parallel rendering framework [Equ06]. Equalizer provides a generic toolkit for parallel, scalable multipipe rendering and is designed to work transparently on a single workstation, shared memory multipipe graphics system or a rendering cluster.

Equalizer applications are configured by a central resource server. A server configuration consists of two parts: the resource description and the usage description of the resources. The resource description is a hierarchical structure. On the top level, nodes define the machines of the rendering cluster. Each node has pipes which describe the graphics cards and are an execution thread in Equalizer. Each pipe has windows, which encapsulate the OpenGL context and drawable. Each window has channels, which are two-dimensional viewports within the window. In the typical deployment case, one channel and one window per pipe is used for optimal performance. Channels are used by compounds, which describe the resource usage. Compounds in Equalizer [EqC06] form a tree, where the top-level compound typically defines the final display. Each compound has tasks and a channel, which is used to execute the tasks. Possible tasks are: clear, draw, assemble and readback.

In this context, sort-last parallel rendering is defined by a multi-level compound. The top-level compound describes the final display setup as a single channel and has n children defining multiple full-frame viewports for parallel sort-last image compositing. Each of the children, with the exception of the destination channel, has a child which executes the rendering, as well as the first step of reading back the tiles for composition. The intermediate compound then assembles the tiles from the other channels and reads back his complete color tile. The destination channel does not need the intermediate compound, as the top-level compound executes the task of assembling the destination channel's tile and, in addition, the gathering of the complete color tiles from all sources. Figure 6 illustrates the compound tree for a three channel direct send configuration shown in Figure 3, using all contributing channels for drawing.



```
channel "destination"
buffer [ COLOR DEPTH ]
wall { ... }
    range [0 ⅓]
    CLEAR, DRAW
    READBACK
       outputframe "f1.dest" viewport [ 0 ⅓ 1 ⅓ ]
       outputframe "f2.dest" viewport [ 0 ⅔ 1 ⅓ ]
    channel "source1"
        range [⅓ ⅔]
        CLEAR, DRAW
        READBACK
           outputframe "f1.source1" viewport [ 0 0 1 ⅓ ]
           outputframe "f2.source1" viewport [ 0 ⅓ 1 ⅓ ]
    ASSEMBLE
       inputframe "f2.dest"
       inputframe "f2.source2"
    READBACK
       outputframe COLOR "frame.source1" viewport [ 0 ⅔ 1 ⅓ ]
    channel "source2"
        range [⅔ 1]
        CLEAR, DRAW
        READBACK
           outputframe "f1.source2" viewport [ 0 ⅓ 1 ⅓ ]
           outputframe "f2.source2" viewport [ 0 ⅔ 1 ⅓ ]
    ASSEMBLE
       inputframe "f2.source1"
       inputframe "f1.dest"
    READBACK
       outputframe COLOR "frame.source2" viewport [ 0 ⅓ 1 ⅓ ]
ASSEMBLE
   inputframe "f1.source1"
   inputframe "f1.source2"
   inputframe "frame.source1"
   inputframe "frame.source2"
```

Figure 6: *A compound tree for a three-channel direct send decomposition. Compounds with the same color use the same channel and a separate execution thread. The image data flow for one tile is illustrated.*

The Equalizer server traverses the compound tree once per frame. During this traversal tasks are generated from the compound description. These tasks are sent to the rendering clients, which execute them in the order they arrive, in a separate thread for each pipe.

For network transmission Equalizer compresses the image data using a fast, modified RLE algorithm, which exploits certain characteristics of the image data. Our compression algorithm achieves approximately a compression of 50% at a rate of 650MB/s on a 2.2Ghz AMD Opteron processor. Furthermore, the compression rate generally improves with the number of nodes in sort-last rendering since the images become more sparse as the number of nodes increases. The receiving node decompresses the data immediately upon reception, which happens in a separate thread from the pipe thread(s). Therefore the decompression can be overlapped with rendering. All sort-last compositing algorithms compared in this paper benefit equally from this image transmission optimization.

The implementation of direct send in Equalizer is an outcome of the flexible compound configuration, not a hard-coded feature. Using the same implementation, we were able to configure the serial assembly compositing, as well as binary swap compositing using a slightly more complicated compound tree. Furthermore, the implementation allows for a variety of parallel compositing algorithms, for example by using a subset or different set of channels for composition.

5. Results

We conducted our experiments on a six node rendering cluster with the following characteristics: dual 2.2GHz AMD Opteron CPUs, 4GB of RAM, Geforce 7800 GTX PCIe graphics and a high-resolution 2560x1600 pixel LCD panel per node; 1GB network and switch. For most tests we used a destination channel with a resolution of 1280x800, since this is closer to a typical window size for scalable parallel rendering. Pixel read, write and network transmission performances for the full-screen content of such a window are:

GL Format, Type	read	write	transmit
RGBA, UNSIGNED_BYTE	12.14*ms*	5.64*ms*	42.05*ms*
DEPTH_COMPONENT, FLOAT	9.86*ms*	33.96*ms*	36.41*ms*

Our test application renders polygonal data (Figures 1 and 9), organized spatially in an octree for efficient view frustum culling and sort-last range selection. The data is rendered using display lists, and each vertex consist of 24 bytes (position+normal). We use a fixed camera path of 100 frames to obtain the total (accumulated) rendering time as the result.

5.1. Scalability Benchmarks

The first three charts show one data set rendered on *n* pipes using four different configurations. (1) 'DB binary swap' uses binary swap compositing, and consequently only datapoints at a power-of-two number of nodes are available. (2) 'DB direct send' is our new algorithm using any number of channels for the composition. (3) 'DB serial' uses serial composition where all data is assembled on the destination channel. (4) 'DB baseline' performs no composition and provides an upper limit for the achievable performance by measuring the parallelism of the draw operation. Additionally, the linear speedup line is marked for comparison.

We have also conducted a sample using eight pipes on our six node cluster. The two nodes running two threads did render 10% of the database per thread, all others did render 15%. Note that the results for 8 nodes are only a qualitative indicator, and can not directly be compared to the other results.

5.1.1. Model Size

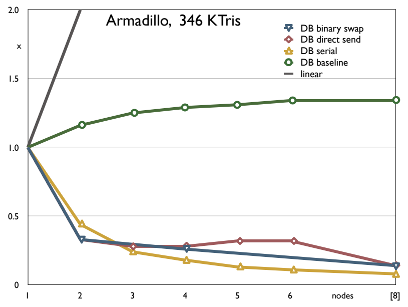
Figure 7a) shows the results of our first test run which uses a small model with less than one million triangles. The draw time of this model is negligible compared to the composition time. This benchmark illustrates the composition overhead of the various algorithms. The singlepipe rendering performance for this model is 42 frames per second. The baseline shows that almost no scalability is possible with such a small model: the rendering with six nodes is only 1.34 times faster than singlepipe rendering. Any compositing algorithm actually decreases the rendering performance. The serial composition shows a linear increase in rendering time, as expected from the theoretical discussion. When using direct send composition, the rendering performance increases again slightly when using three nodes or more due to the parallelism in the composition. Binary swap has the same behaviour, though with slightly less performance.

Figure 7b) shows the results of a medium-sized model. The draw time becomes significant, especially since it is not possible to fit all the data onto a single GPU, as shown by the super-linear speedup of the baseline. The singlepipe rendering performance for this data set is 1.3 frames per second. The serial configuration scales up to only four nodes with a performance of 4.3 FPS (3.2x speedup), afterwards the composition term increases the total rendering time. Direct send shows good scalability up to six nodes with a performance of 7.7 FPS (5.9x speedup), but at five and six node counts we can observe some composition and synchronization overhead. Binary swap is slightly slower due to the higher synchronization overhead, with a 4.6x speedup compared to a 4.9x speedup for direct send at four nodes.

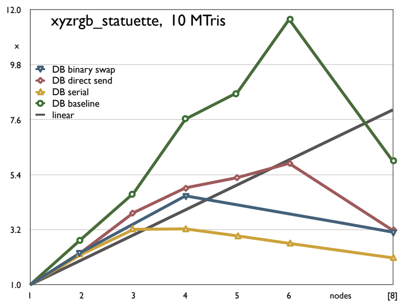
The last scalability benchmark in Figure 7c) uses a large model. Again, the model shows the expected superlinear speedup for the baseline. The singlepipe rendering time for this data set is 0.39 frames per second. The composition time is less relevant for this model, as the rendering time increases. We can again exhibit the increasing bottleneck of the serial composition with a higher number of nodes. The constant composition time of direct send allows a superlinear speedup of the total rendering time. Binary swap shows the same behaviour, with almost exactly the same performance. This model achieves a 6.8x speedup when using six pipes with direct send compositing.

5.1.2. Viewport Size

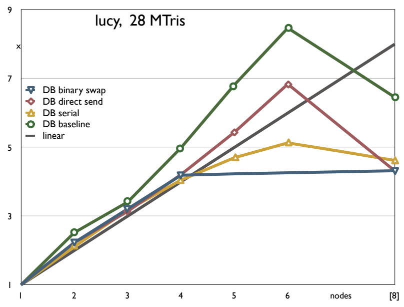
In this benchmark we measured the influence of the viewport size, and therefore the amount of transferred pixel data, on the total rendering time. We used the same model as in Figure 7b) to measure the total rendering time with direct send



a)



b)



c)

Figure 7: *Parallel speedup of serial, binary swap and direct send compositing for a) small, b) medium and c) large data.*

compositing. The graph shows the expected asymptotic behaviour towards the constant composition cost of direct send, regardless of the viewport size. As outlined in the theoretical discussion, the composition cost is directly dependent on the viewport size, which we increased quadratically here.

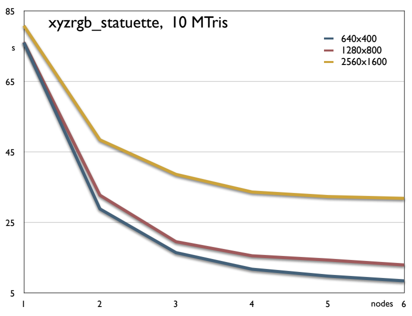


Figure 8: *Influence of the viewport size on the total rendering time.*

5.2. Interpretation

Overall, the scalability tests show that the direct send algorithm behaves as expected. Direct send shows better performance and scalability than serial compositing, and marginally better performance than binary swap. Depending on the model size and screen-space distribution, there is a tradeoff point where the composition time becomes dominant and performance declines.

We have shown that with direct send compositing we can scale the rendering time up to several parallel rendering nodes. Especially with the medium model size we can see that with a higher number of nodes the synchronization overhead may become a factor limiting the parallelism. The flexibility of direct send, however, allows to use the optimal number of compositing channels independently from the number of parallel rendering channels. Furthermore, it allows to optimally use rendering clusters with a non-power-of-two number of nodes.

6. Conclusions

In this paper, we have presented the direct send sort-last compositing algorithm and evaluated its implementation in a generic parallel rendering framework. We have shown in the theoretical discussion that our algorithm provides the same, if not better performance as binary swap, and far superior performance than serial compositing. In the results section we have supported the theoretical results by experiments on a parallel rendering cluster.

The implementation of a flexible compositing engine in Equalizer is a first step towards a generic, scalable rendering engine. We will in the future focus on a number of further optimizations to provide better compositing performance, and therefore parallel rendering scalability.

Currently the network transfer is a major bottleneck. By optimizing the network performance, either directly by tuning the socket code, or indirectly by decreasing the amount of pixels transferred, we plan to decrease the impact of this bottleneck. The network transfer speed can be increased by using zero-copy transfers, asynchronous transmission or by upgrading to a high-performance interconnect bypassing the TCP/IP stack. The amount of pixel data transferred can be improved by improved fast compression algorithms and by limiting the data to the actual screen-space viewport updated by the draw operation.

We are currently updating our benchmark application to provide better raw rendering performance. In particular, we are looking into using vertex buffer objects (VBO) to improve rendering speed, and into using a three-dimensional kd-tree instead of the current octree. The latter optimization will provide better spatial distribution of the data base during sort-last rendering, and therefore sparser images, and a more equal data distribution across nodes.

Acknowledgements

We would like to thank and acknowledge the Stanford 3D Scanning Repository for providing the 3D geometric test data sets.

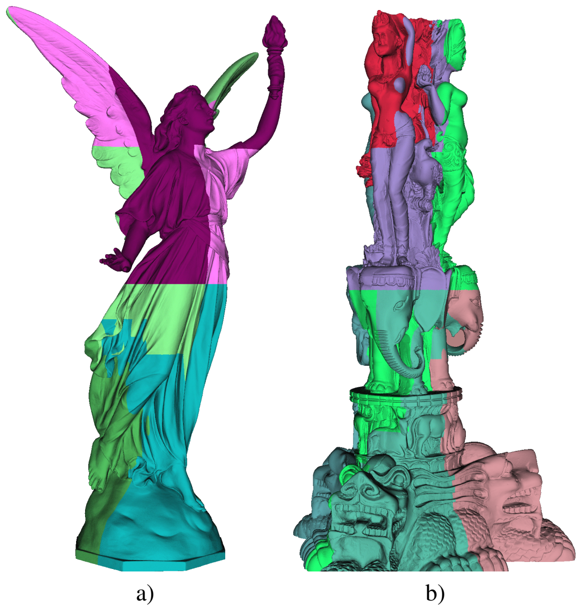


Figure 9: *Destination view of the a) large and b) medium model of a six node sort-last configuration, using a different draw color for each node.*

References

[AP98] AHRENS J., PAINTER J.: Efficient sort-last rendering using compression-based image compositing. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization* (1998), pp. 145–151.

[BRE05] BHANIRAMKA P., ROBERT P. C. D., EILEMANN S.: OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization* (2005), pp. 119–126.

[CKS02] CORREA W. T., KLOSOWSKI J. T., SILVA C. T.: Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *Proceedings Eurographics Workshop on Parallel Graphics and Visualization* (2002), pp. 89–96.

[CM06] CAVIN X., MION C.: Pipelined sort-last rendering: Scalability, performance and beyond. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization* (2006).

[CMF05] CAVIN X., MION C., FILBOIS A.: COTS cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *Proceedings IEEE Visualization* (2005), pp. 111–118.

[EMP*97] EYLES J., MOLNAR S., POULTON J., GREER T., LASTRA A., ENGLAND N., WESTOVER L.: PixelFlow: The realization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics Hardware* (1997), pp. 57–68.

[EqC06] Equalizer compound specification. http://www.equalizergraphics.com/documents/design/compounds.html, 2006.

[Equ06] Equalizer. http://www.equalizergraphics.com/, 2006.

[LMS*01] LOMBEYDA S., MOLL L., SHAND M., BREEN D., HEIRICH A.: Scalable interactive volume rendering using off-the-shelf components. In *Proceedings of the IEEE Symposium on Pparallel and large-data Visualization and Graphics* (2001), pp. 115–121.

[LRN95] LEE T.-Y., RAGHAVENDRA C. S., NICHOLAS J. N.: Image composition methods for sort-last polygon rendering on 2-d mesh architectures. In *Proceedings of the IEEE Symposium on Parallel Rendering* (1995), pp. 55–62.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Algorithms* (July 1994), 23–32.

[MEP92] MOLNAR S., EYLES J., POULTON J.: PixelFlow: High-speed rendering using image composition. In *Proceedings ACM SIGGRAPH* (1992), pp. 231–240.

[MOM*01] MURAKI S., OGATA M., MA K.-L., KOSHIZUKA K., KAJIHARA K., LIU X., NAGANO Y., SHIMOKAWA K.: Next-generation visual supercomputing using PC clusters with volume graphics hardware devices. In *Proceedings ACM/IEEE Conference on Supercomputing* (2001), pp. 51–51.

[MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel Volume Rendering Using Binary-Swap Image Composition. *IEEE Computer Graphics and Algorithms* (July 1994).

[MSH99] MOLL L., SHAND M., HEIRICH A.: Sepia: Scalable 3D compositing using PCI pamette. In *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines* (1999), p. 146.

[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 *Proceedings ACM SIGGRAPH* (2001), pp. 141–148.

[SGS91] SHAW C. D., GREEN M., SCHAEFFER J.: A VLSI architecture for image composition. In *Advances in Computer Graphics Hardware III (Eurographics'88 Workshop)* (London, UK, 1991), Springer-Verlag, pp. 183–200.

[SKN04] SANO K., KOBAYASHI Y., NAKAMURA T.: Differential coding scheme for efficient parallel image composition on a pc cluster system. *Parallel Computing 30*, 2 (2004), 285–299.

[SML*03] STOMPEL A., MA K.-L., LUM E. B., AHRENS J., PATCHETT J.: SLIC: Scheduled linear image compositing for parallel volume rendering. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 33–40.

[TIH03] TAKEUCHI A., INO F., HAGIHARA K.: An improved binary-swap compositing for sort-last parallel rendering on distributed memory multiprocessors. *Parallel Computing 29*, 11-12 (2003), 1745–1762.

[YYC01] YANG D.-L., YU J.-C., CHUNG Y.-C.: Efficient compositing methods for the sort-last-sparse parallel volume rendering system on distributed memory multicomputers. *Journal of Supercomputing 18*, 2 (2001), 201–220.

[ZBB01] ZHANG X., BAJAJ C., BLANKE W.: Scalable isosurface visualization of massive datasets on cots clusters. In *Proceedings IEEE Symposium on Parallel and Large Data Visualization and Graphics* (2001), pp. 51–58.

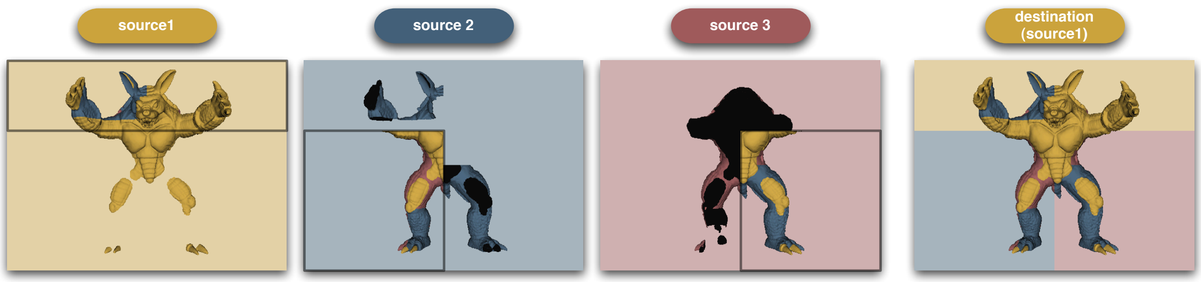


Figure 1: *Sort-last rendering with direct send compositing using three channels.*

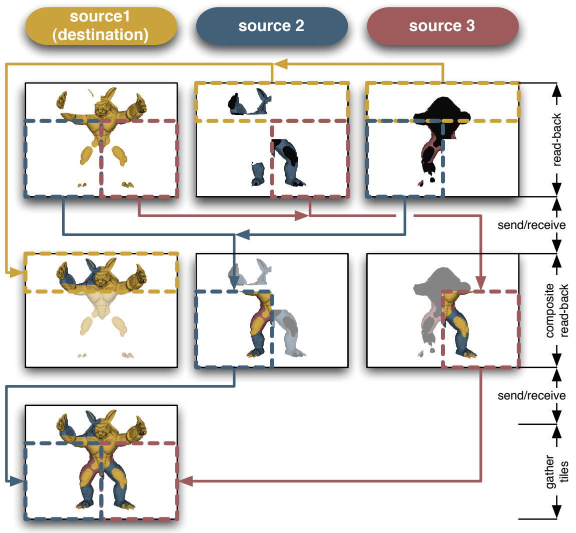


Figure 3: *Direct send compositing data flow.*

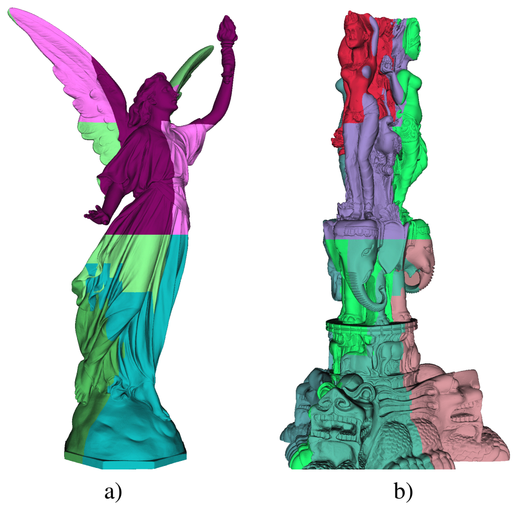


Figure 9: *Destination view of the a) large and b) medium model of a six node sort-last configuration, using a different draw color for each node.*