Equalizer: A Scalable Parallel Rendering Framework

Stefan Eilemann, Maxim Makhinya, and Renato Pajarola, *Member*, *IEEE Computer Society*

Abstract—Continuing improvements in CPU and GPU performances as well as increasing multicore processor and cluster-based parallelism demand for flexible and scalable parallel rendering solutions that can exploit multipipe hardware accelerated graphics. In fact, to achieve interactive visualization, scalable rendering systems are essential to cope with the rapid growth of data sets. However, parallel rendering systems are nontrivial to develop and often only application specific implementations have been proposed. The task of developing a scalable parallel rendering framework is even more difficult if it should be generic to support various types of data and visualization applications and at the same time work efficiently on a cluster with distributed graphics cards. In this paper, we introduce a novel system called *Equalizer*, a toolkit for scalable parallel rendering based on OpenGL, which provides an application programming interface (API) to develop scalable graphics applications for a wide range of systems ranging from large distributed visualization clusters and multiprocessor multipipe graphics systems to single-processor single-pipe desktop machines. We describe the system architecture and the basic API, discuss its advantages over previous approaches, and present sample configurations and usage scenarios as well as scalability results.

Index Terms—Parallel rendering, scalable visualization, cluster graphics, immersive environments, display walls.

1 Introduction

The continuing improvements in hardware integration lead to ever faster CPUs and GPUs, as well as higher resolution sensor and display devices. Moreover, increased hardware parallelism is applied in form of multicore CPU workstations, massive parallel supercomputers, or cluster systems. Hand in hand goes the rapid growth in complexity of data sets from numerical simulations, high-resolution 3D scanning systems, or biomedical imaging, which causes interactive exploration and visualization of such large data sets to become a serious challenge. It is thus crucial for a visualization solution to take advantage of hardware-accelerated scalable parallel rendering. In this systems paper, we describe a new scalable parallel rendering framework called *Equalizer* that is aimed primarily at cluster-parallel rendering but works as well in a shared-memory system. Cluster systems are the main focus because workstation graphics hardware is developing faster than high-end graphics (super-) computers can absorb new developments, and also because clusters offer a better cost-performance balance.

Previous parallel rendering approaches typically failed in one of the following system requirements:

1. generic application support, instead of special domain solution,
2. scalable abstraction of the graphics layer,
3. exploit existing code infrastructure, such as proprietary scene graphs, molecular data structures, level-of-detail (LOD), and geometry databases.

To date, generic and scalable parallel rendering frameworks that can be adopted to a wide range of scientific visualization domains are not yet readily available. Furthermore, flexible configurability to arbitrary cluster and display wall configurations has also not been addressed in the past but is of immense practical importance to scientists depending high-performance interactive visualization as a scientific tool. In this paper, we present Equalizer, which is a novel flexible framework for parallel rendering that supports scalable performance, configuration flexibility, is *minimally invasive* with respect to adapting existing visualization applications, and is applicable to virtually any scientific visualization application domain.

The main contributions that Equalizer introduces in a single parallel rendering system and which are presented in this paper are given as follows:

1. novel concept of compound trees for flexible configuration of graphics system resources,
2. easy specification of parallel task decomposition and image compositing choice through compound tree layouts,
3. automatic decomposition and distributed execution of rendering tasks according to compound tree,
4. support for parallel surface as well as transparent (volume) rendering through z-visibility as well as α-blending compositing,
5. fully decentralized architecture providing network swap barrier (synchronization) and distributed objects functionality,
6. support for low-latency distributed frame synchronization and image compositing,
7. minimally invasive programming model.

- S. Eilemann is with the Eyescale Software, and Visualization and MultiMedia Laboratory (VMML), University of Zurich, Faubourg de Hopital 12, 2000 Neuchatel, Switzerland. E-mail: eilemann@gmail.com.
- M. Makhinya and R. Pajarola are with the Visualization and MultiMedia Laboratory (VMML), Department of Informatics, University of Zurich, Binzmühlestrasse 14, 8050 Zurich, Switzerland. E-mail: makhinya@ifi.uzh.ch, pajarola@acm.org.

Manuscript received 5 Feb. 2008; revised 11 July 2008; accepted 22 Aug. 2008; published online 4 Sept. 2008.
Recommended for acceptance by K.-L. Ma.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2008-02-0021. Digital Object Identifier no. 10.1109/TVCG.2008.104.

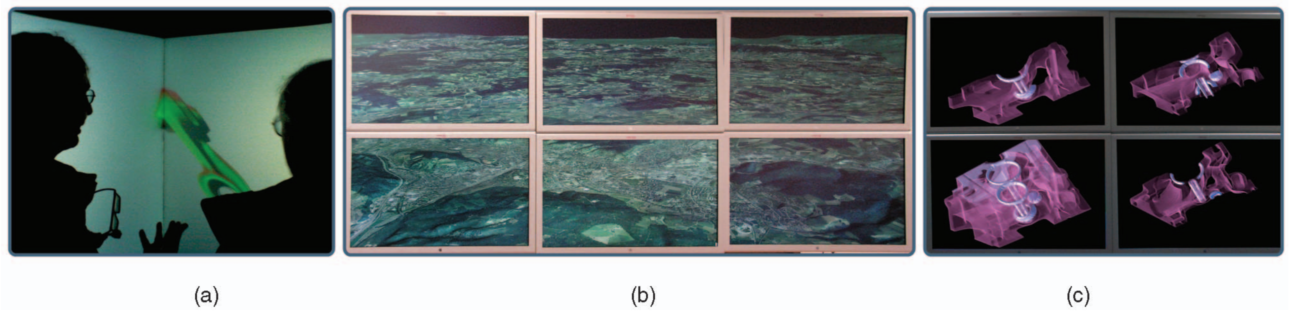


Fig. 1. Various Equalizer use cases. (a) Immersive CAVE. (b) Display wall. (c) Scalable volume rendering.

Equalizer is open source, available under the LGPL license from http://www.equalizergraphics.com/, which allows it to be used both for open source and commercial applications. It is source-code portable and has been tested on Linux, Microsoft Windows, and Mac OS X in 32 and 64-bit modes using both little endian and big endian processors.

2 Related Work

The early fundamental concepts of parallel rendering have been laid down in [39] and [13]. A number of domain-specific parallel rendering algorithms and special-purpose hardware solutions have been proposed in the past; however, only few generic parallel rendering frameworks have been developed.

2.1 Domain-Specific Solutions

Cluster-based parallel rendering has been commercialized for offline rendering (i.e., distributed ray-tracing) for computer-generated animated movies or special effects, since the ray-tracing technique is inherently amenable to parallelization for offline processing. Other special-purpose solutions exist for parallel rendering in specific application domains such as volume rendering [34], [56], [23], [50], [18], [44] or geovisualization [55], [2], [33], [29]. However, such specific solutions are typically not applicable as a generic parallel rendering paradigm and do not translate to arbitrary scientific visualization and distributed graphics problems.

Recently in [45], parallel rendering of hierarchical LOD data has been addressed and a solution specific to sort-first tile-based parallel rendering has been presented. While the presented approach is not a generic parallel rendering system, basic concepts presented in [45] such as load management and adaptive LOD data traversal can be carried over to other sort-first parallel rendering solutions.

2.2 Special-Purpose Architectures

Traditionally, high-performance real-time rendering systems have relied on an integrated proprietary system architecture, such as the SGI graphics supercomputers. These special-purpose solutions have become a niche product as their graphics performance does not keep up with off-the-shelf workstation graphics hardware and scalability of clusters. However, cluster systems need more sophisticated parallel graphics rendering libraries, such as the one proposed in this paper.

Due to its conceptual simplicity, a number of special-purpose image compositing hardware solutions for sort-last parallel rendering have been developed. The proposed hardware architectures include Sepia [38], [32], Sepia 2 [35], [36], Lightning 2 [52], Metabuffer [9], [59], MPC Compositor [43], and PixelFlow [40], [17], 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. Moreover, with the recent advances in the speed of CPU-GPU interfaces, such as PCI Express and other modern interconnects, combinations of software and GPU-based solutions offer more flexibility at comparable performance.

2.3 Generic Approaches

A number of algorithms and systems for parallel rendering have been developed in the past. On the one hand, some general concepts applicable to cluster parallel rendering have been presented in [41], [42] (sort-first architecture), [49], [48] (load balancing), [47] (data replication), or [11], [10] (scalability). On the other hand, specific algorithms have been developed for cluster-based rendering and compositing such as [3], [12] and [57], [53]. However, these approaches do not constitute APIs and libraries that can readily be integrated into existing visualization applications, although the issue of the design of a parallel graphics interface has been addressed in [28]. Only few generic APIs and (cluster-) parallel rendering systems exist, which include VR Juggler [8] (and its derivatives), Chromium [27] (an evolution of [26], [24], and [25]), and OpenGL Multipipe SDK (MPK) [30], [6], [1].

VR Juggler [8], [31] is a graphics framework for virtual reality applications, which shields the application developer from the underlying hardware architecture, devices, and operating system. Its main aim is to make virtual reality configurations easy to set up and use without the need to know details about the devices and hardware configuration but not specifically to provide scalable parallel rendering. Extensions of VR Juggler, such as for example ClusterJuggler [7] and NetJuggler [4], are typically based on the replication of application and data on each cluster node and basically take care of synchronization issues but fail to provide a flexible and powerful configuration mechanism that efficiently supports scalable rendering as also noted in [51]. The presented system is different from VR Juggler in that it fully supports scalable parallel rendering such as sort-first and sort-last task decomposition and image compositing, and it provides more flexible node configurations, which, for example, allow specifying arbitrary task decomposition and image compositing combinations as simple compound layouts. Furthermore, it is fully distributed, which includes support for network swap barriers (synchronization), distributed objects, as well as image compression and transmission. In contrast to VR Juggler, Equalizer supports multiple rendering threads per process, which is important for multi-GPU systems.

While Chromium [27] provides a powerful and transparent abstraction of the OpenGL API, which allows a flexible configuration of display resources, its main limitation with respect to scalable rendering is that it is focused on streaming OpenGL commands through a network of nodes, often initiated from a single source. This has also been observed in [51]. The problem comes in when the OpenGL stream is large in size, due to not only containing OpenGL calls but also the rendered data such as geometry and image data. Only if the geometry and textures are mostly static and can be kept in GPU memory on the graphics card, no significant bottleneck can be expected as then the OpenGL stream is composed of a relatively small number of rendering instructions. However, as it is typical in real-world visualization applications, display and object settings are interactively manipulated, data and parameters may change dynamically, and large data sets do not fit statically in GPU memory but are often dynamically loaded from out-of-core and/or multiresolution data structures. This can lead to frequent updates not only of commands and parameters that have to be distributed but also of the rendered data itself (geometry and texture), thus causing the OpenGL stream to expand dramatically. Furthermore, this stream of function calls and data must be packaged and broadcast in real-time over the network to multiple nodes for each rendered frame. This makes CPU performance and network bandwidth a more likely limiting factor. While preserving a minimally invasive API, the novel proposed system is better aimed at scalability as the actual data access is decentralized in the distributed rendering clients.

The performance experiments in [27] indicate that Chromium is working quite well when the rendering problem is fill-rate limited. This is due to the fact that the OpenGL commands and a noncritical amount of rendering data can be distributed to multiple nodes without significant problems and since the critical fill-rate work is then performed locally on the graphics hardware.

Chromium also provides some facilities for parallel application development, namely a sort-last, binary-swap compositing SPU, and an OpenGL extension providing synchronization primitives, such as a barrier and semaphore. It leaves other problems, such as configuration, task decomposition, as well as process and thread management unaddressed, thus making the development of parallel OpenGL applications harder than with Equalizer. Parallel Chromium applications tend to be written for one specific parallel rendering use case, such as for example the sort-first distributed memory volume renderer [5] or the sort-last parallel volume renderer Raptor [22]. We are not aware of a generic Chromium-based application using many-to-one sort-first or stereo decompositions. This is another difference to Equalizer, which provides a much more flexible task decomposition configuration. Applications written once for

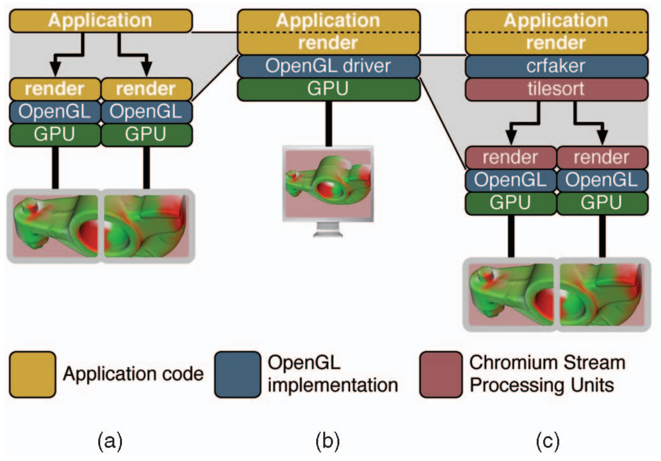


Fig. 2. A traditional OpenGL application (b) and its equivalents when using Equalizer (a) or Chromium (c).

Equalizer can easily be run in any different task decomposition mode and for any physical display configuration without any changes to the application itself. While Equalizer provides an abstraction of all entities of the rendering pipeline (see Sections 4 and 5), Chromium's infrastructure is primarily the compositing stage.

MPK [6] implements an effective parallel rendering API for a shared memory multi-CPU/GPU system. It is similar to IRIS Performer [46] in that it handles multipipe rendering by a lean abstraction layer via a conceptual callback mechanism, and that it runs different application tasks in parallel. However, MPK is not designed nor meant for rendering nodes separated by a network. MPK focuses on providing a parallel rendering framework for a single application, parts of which are run in parallel on multiple rendering channels, such as the culling, rendering, and final image compositing processes. Compared to MPK, Equalizer supports a fully distributed parallel rendering paradigm and features a more flexible task decomposition approach.

3 Basic Concepts

Besides the API, one of the major differences of Equalizer to Chromium is that it is fully distributed and runs the application code in parallel. For example, one can set up a multiscreen display wall with Chromium, streaming the OpenGL calls to a number of render nodes assigned to screen tiles of the display wall, as illustrated in Fig. 2c. One instance of the application is running. In contrast, Equalizer runs parts of the application in parallel on multiple rendering channels as illustrated in Fig. 2a.

Equalizer takes care of distributed execution, synchronization, and final image compositing, while the application programmer identifies and encapsulates critical parts of the application, such as culling and rendering. This approach is considered to be *minimally invasive* since the existing and proprietary rendering code can basically be retained. All rendering is executed directly to an OpenGL context, and at no point are OpenGL commands sent over the network.

This minimally invasive approach allows the application to retain its OpenGL rendering code but structures the implementation to allow for optimal performance. The network bandwidth is freed from unnecessary transmission of excessive graphics commands and data since only the basic rendering parameters are exchanged between nodes. Only for the unavoidable final image compositing step in scalable rendering, frame buffer data between the nodes must be exchanged. The application can implement efficient dynamic database updates based on distributed objects or message passing as these distributed system primitives are provided by Equalizer.

A major strength of Equalizer is its flexible and scalable configuration of the parallel rendering tasks, which takes the notion of a compound tree introduced in MPK [6] to a distributed cluster environment as discussed in Section 4.5. Hence, different parallel rendering task decomposition and image compositing configurations can easily be specified, see also Fig. 11. For example, efficient direct-send sort-last image compositing has been demonstrated in [15].

The Equalizer framework does not impose any constraints on how the application handles and accesses the data to be visualized. As such, Equalizer does not provide a solution to the parallel data access and distribution problem, which has to be addressed by the application itself, for example via mechanisms to limit data replication (e.g., [47]), or out-of-core access to large data sets and multiresolution representations (e.g., [12]). As demonstrated in [12], out-of-core data structures are well suited to provide efficient parallel access to the 3D data from all rendering nodes, and a wealth of out-of-core approaches have been provided for volume, polygonal, or point data sets (e.g., [54], [21], [58], [20], or [19]). Equalizer does not interfere with or inhibit any solution to this problem, as it is an orthogonal issue.

Equalizer does address some fundamental problems to help application developers distribute their data effectively in the context of parallel rendering. The Equalizer networking layer supports message passing and the creation of distributed objects. By subclassing a distributed object class, static and versioned objects can be created. Objects are addressed on the cluster using a unique identifier, which allows the remote mapping of the object. Versioned objects are typically used for frame-specific data, where a new version for each new frame is created. This version information is passed correctly by Equalizer to the application rendering code. This mechanism allows simple distribution and multibuffering of data.

Our *eqPly* and *eVolve* sample applications use static distributed objects for submitting the initialization parameters, e.g., the model filename, and a versioned distributed object for the camera position and other frame-specific data.

4 System Architecture

Equalizer is a parallel rendering framework using a similar task description concept as MPK [6]. In the following, we will focus on the basic system aspects of Equalizer, starting with the interface and application structure followed by the client-server model employed. One of the main Equalizer contributions, the compound tree concept, which describes the hardware resource setup and parallel task decomposition, is then introduced in detail.

4.1 Interface

Equalizer provides a framework to facilitate the development of distributed as well as nondistributed parallel rendering applications. The programming interface is based on a set of C++ classes, modeled closely to the resource hierarchy of a graphics rendering system. The application subclasses these objects and overrides C++ task methods, similar to C callbacks. These task methods will be called in parallel by the framework, depending on the current configuration. A wrapper interface could be written to provide C bindings. This parallel rendering interface is significantly different from Chromium [27] and more similar to VR Juggler [8] or MPK [6]. The class framework and in particular its use is described in more detail in Section 5.

An Equalizer application does not have to select a particular rendering configuration itself; it is configured by a system-wide configuration server. The application is written only against a client library, communicating with the server, which does not have to be touched by the developer. The parallel rendering configuration is initialized by the server based on guidelines from the application or a user-supplied configuration file. The server also launches and controls the distributed rendering clients provided by the application. Thus, the application itself can run unmodified on any configuration, which has been initialized by the server, and if none is given, the application will run as a stand-alone process on the node it has been started.

While on a higher level Equalizer uses a client-server approach, it is built on a peer-to-peer network layer. This network layer provides a message-based communication interface, as needed between any two nodes in the cluster, e.g., to transmit image data for result recomposition during scalable rendering. Currently, Equalizer provides an implementation for TCP/IP sockets and InfiniBand. The usage of MPI as a low-level communication library was not feasible in the context of Equalizer. Dynamic process management is only available in MPI 2, which still is not widespread enough. Furthermore, the communication patterns for which MPI was designed are significantly different from Equalizer's use case. However, this does not prohibit coupling MPI-based programs with Equalizer.

4.2 Application

The application in Equalizer solely drives the rendering, that is, it carries out the main rendering loop only but does not actually execute any rendering. Although depending on the configuration, the application process may also host one or more render client threads, as described below. When a configuration has no additional nodes besides the application node, all application code is executed in the same process, and no network data distribution has to be implemented.

During initialization of the server, the application provides a rendering client. The rendering client is often, especially for simple applications, the same executable as the application. However, in more sophisticated implementations, the rendering client may be a thin renderer, which only contains the application-specific rendering code. The server deploys this rendering client on all nodes specified in the configuration. The main rendering loop is quite simple: The application requests a new frame to be rendered, synchronizes on the completion of a frame, and processes

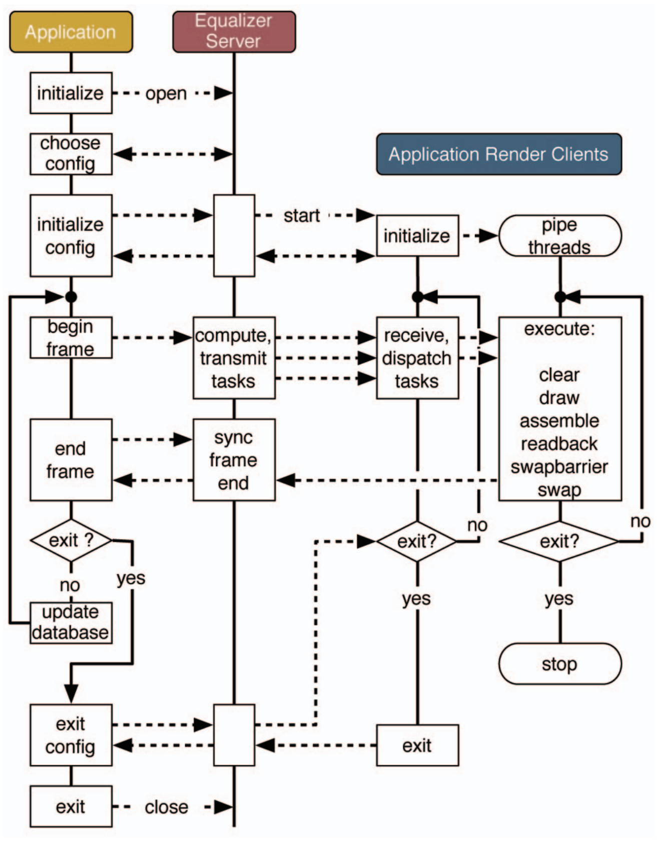


Fig. 3. Simplified execution flow of an Equalizer application, omitting event handling and application-node rendering threads.

events received from the render clients. Fig. 3 shows a simplified execution model of an Equalizer application. The rendering client and server are further described in the following sections.

4.3 Rendering Client

Each Equalizer application provides a rendering client, which can be the same executable as the application code itself. In contrast to the application, however, the rendering client does not need to have a main loop and is completely controlled by the Equalizer framework. If a configuration also uses the application node for rendering, then the rendering happens in different threads within the application process. A render client consists of the following threads: the node main thread, one network receive thread, and one thread for each graphics card (GPU) to execute rendering tasks.

The client library implements the main loop, which receives network events and processes them. Most importantly, the network data contains the rendering task parameters computed by the server. Based on this data, the client library sets up the rendering context and calls the application-provided task methods. Setting up the rendering context consists of using the correct rendering thread, making the drawable and graphics context current, as well as providing the task methods with the 2D viewport, frustum, view matrix, and the data range for sort-last rendering. The task methods clear the frame buffer as necessary, execute the OpenGL rendering commands as well as readback, and assemble partial frame results for scalable rendering. All tasks have default implementations so that only the application specific methods have to be implemented, which typically includes the `frameDraw()` method. For example, the default callbacks for frame recomposition during scalable rendering implement tile-based assembly for sort-first and stereo decompositions, and z-buffer or compositing for sort-last rendering of polygonal data. A detailed description of the API and all methods can be found in the programming guide [14].

Event handling is implemented by listening asynchronously for events from all windows. Events are transformed from window-system specific events into generic window events and dispatched to the correct window. The window either processes the event locally or converts it into a config-event to be sent to the application node. The application node processes the config-events as part of its main rendering loop. A more detailed description of event handling can also be found in [14].

In addition to executing the application code in the right context, the client library implements image compression and transmission, network swap barrier support, and distributed object support.

TABLE 1
Correspondence between Physical and Logical System Entities and Equalizer Resources

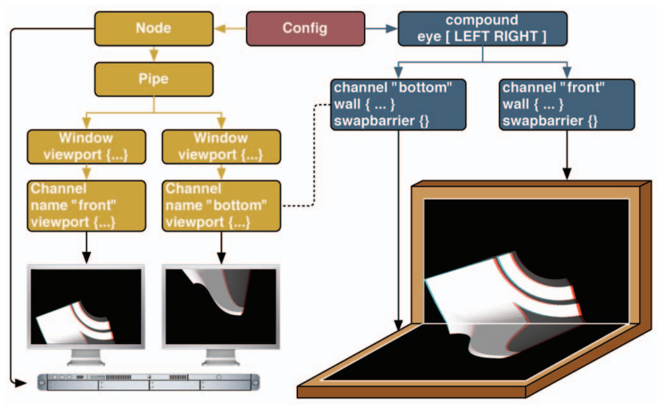
Physical entity:	CPU	GPU	Drawable	Viewport
Equalizer resource:	*node*	*pipe*	*window*	*channel*

4.4 Equalizer Server

The Equalizer server receives requests from the application on the visualization system. It serves these requests using the application's specific configuration, launching rendering clients on the nodes, determining the rendering tasks for a frame, and synchronizing the completion of frames.

The server maintains the configuration for the application. Maintaining the configuration on the server facilitates an extension to cross-application load balancing, resource reservation, and further system-wide resource management. Each configuration consists of two parts. The first part is a hierarchical resource description derived from the physical and logical environment of the application. The second part consists of the compound tree, which declares how the resources are used for rendering. The compounds are the heart of the scalable rendering engine and are described in detail in the following section.

The resources description given in Table 1 includes the intuitive entities that make up a typical graphics system, of which several are used in parallel in a rendering cluster. At the top of the hierarchy are *nodes*, which represent a process, possibly one per CPU-core on each computer within a cluster. A node contains one or more *pipes*, which are threads representing the GPUs in a machine. In turn, a pipe can have multiple *windows*, which correspond to OpenGL onscreen or offscreen drawables. By default, all windows of a pipe share display lists and other OpenGL objects. Eventually, a window has one or more *channels*, which encapsulate a particular OpenGL viewport in a window. Note that all tasks for a pipe and its children are executed in a separate thread.



```
node {
  pipe {
    window {
      viewport [ 0 0 1280 1024 ]
      channel { name ”front” }
    }
    window {
      viewport [ 0 0 1280 1024 ]
      channel { name ”bottom” }
    }
}}
```

Fig. 4. A sample Equalizer resource configuration for a TAN Holobench with the associated render resources.

A simple example of resource description and configuration is given in Fig. 4, which shows a one-node, single-pipe, two-window, two-channel resource configuration driving a TAN Holobench with two projection surfaces. The corresponding resources configuration file that is read by the server is also given below. The leaf-node channels declared in the resource section on the left are used by the compounds to describe the rendering processes. Another resource configuration is illustrated in Fig. 12. The corresponding compounds configuration file is further detailed in the following section.

4.5 Compound Trees

To describe the parallel rendering task decomposition, Equalizer uses a *compound tree* structure similar to MPK [6]. However, the compound definition has been improved in a few key points to provide a more flexible and powerful configuration.

First, it does not rely on a hard-coded mode, which determines the task decomposition and image compositing stages. Instead, it describes the rendering and compositing tasks via the compound tree's structure.

Second, the rendering is asynchronous and not frame-synchronized as in MPK, where all rendering threads are synchronized at the end of each frame. Asynchronous rendering avoids idle times for rendering threads that finish early. Equalizer introduces a config-latency l_{config}, which defines how many frames the slowest rendering thread is allowed to fall behind. Hence, at the end of frame i, the completion of frame $i - l_{\text{config}}$ will be synchronized. Note that setting $l_{\text{config}} = 0$ enforces a frame synchronicity if desired. Other synchronization points in Equalizer only include the completion of image transfers for compositing

```
compound {
  channel ”draw”
  buffer [ COLOR DEPTH ]
  range [0 ½]
  viewport [ 0 0 ½ 1 ]
  outputframe {name ”left_half” }
}
```

Fig. 5. Compound executing rendering of a part of the data set into a given region of the viewport.

and optional *swap barriers* explicitly defined in the compound tree.

Compounds are a data structure to describe the parallel execution of rendering tasks in a form of a tree. Each compound corresponds to some *tasks* (clear, draw, assemble, readback) and references a *channel* from the resource description, which executes the tasks in the given order. A compound may provide *output frames* from the readback task to others and can request *input frames* from others for its own assembly task, and output frames are linked to input frames by name.

Compound trees are a logical description of the rendering pipeline and only reference the actual physical resources through their channels. This allows mapping a compound tree to different physical configurations by simply replacing the channel IDs. For example, one can test the functionality of a sort-last configuration by using channels of different windows on one local workstation.

A simple leaf compound description for rendering a part of the data set, given by the data *range*, into a particular region of the *viewport* is given in Fig. 5. The data range is a logical mapping of the data set onto the unit interval and is left to the application to interpret appropriately. Hence, the range $[0\ \frac{1}{2}]$ indicates that the first half of the data set should be rendered, for example, the first $\frac{n}{2}$ triangles of a polygonal mesh with n faces. The viewport is indicated by the parameters $[x\ y\ width\ height]$ as fraction of the parent's viewport, and in the example, the data are thus rendered into the left half of the viewport. The resulting frame buffer data—including per-pixel color and depth—of the rendering executed on this channel is read back and made available to other compounds by the name left_half.

A nonleaf compound performing some image assembly and compositing task is indicated in Fig. 6. Frame buffer data are read from two other compounds, which supposedly execute rendering for part_a and part_b of the data set in parallel. The compound itself executes, for example, z-depth visibility compositing of the two input images on its channel and returns the resulting color frame buffer.

```
compound {
  channel ”display”
  inputframe { name ”part_a” }
  inputframe { name ”part_b” }
  outputframe { buffer [ COLOR ] }
}
```

Fig. 6. Compound performing image compositing.

```
compound {
  eye [ LEFT RIGHT ]
  compound {
    channel ”front”
    wall {
      bottom_left [ 0.0 0.5 0.0 ]
      bottom_right [ 1.0 0.5 0.0 ]
      top_left [ 0.0 0.5 0.5 ]
  }}
  compound {
    channel ”bottom”
    wall {
      bottom_left [ 0.0 0.0 0.0 ]
      bottom_right [ 1.0 0.0 0.0 ]
      top_left [ 0.0 0.5 0.0 ]
  }}
}
```

Fig. 7. Wall compound.

An example showing how to set up a specific physical display configuration is given in Fig. 7, which corresponds to the TAN Holobench configuration shown in Fig. 4 above. Using the *wall* parameter, the physical configuration of a display can be specified, here given in meters where the coordinate system's x, y-plane is the horizontal bottom screen, z extending vertically up, and the origin is the front-leftmost corner of the two-screen display. Together with an observer position and orientation, the wall parameters fully define the view frustum for each output screen.

Leaf compounds execute all tasks by default, but the focus is often on the draw task with a default assemble and standard readback task used to pass the resulting image data on to other compounds for further compositing. Hence, while leaf compounds execute the rendering in parallel, nonleaf compounds often correspond, but are not restricted, to the (parallel) image compositing and assembly part. The readback or assemble tasks are only active if output or input frames have been specified, respectively. Otherwise, the rendered image frame is left in place for further processing in a parent compound sharing the same channel.

Note that nonleaf nodes in the compound tree structure traverse their children first before performing their default assemble and readback tasks. Furthermore, compounds only define the logical task decomposition structure, while its execution is actually performed on the referenced channels. Therefore, since compounds can share channels, as often done between a parent and one of its child compounds, rendered image data can sometimes be left in place, avoiding readback and transfer to another node.

All attributes as well as the channel are inherited from the parent compound if not specified otherwise. The *viewport*, *data range*, and *eye* attributes are used to describe the decomposition of the parent's 2D viewport, database range, and eye passes, respectively. To synchronize the buffer swap among a group of channels, *swap barriers* can be used, which is typically used for multiscreen setups such as CAVEs or display walls.

In the following, we describe several use-case examples of the compound tree structure introduced above that demonstrate how different task decomposition modes can be specified. More complex configurations can be achieved by combining these strategies. Note that the physical resources description, the first part of the configuration (see also previous section), is omitted in these examples.

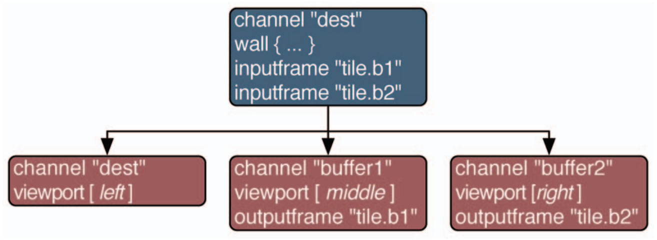


Fig. 8. Compound tree for a three-to-one sort-first decomposition.

4.5.1 Sort-First Configuration

A sort-first compound configuration is shown in Figs. 8 and 11a. The root compound defines the viewport size of the channel and the frustum from the wall description. While the first child compound inherits the channel, the other compounds are executed on different channels. However, each defines a partial viewport, affecting its local view frustum corresponding to the sort-first screen subdivision. All leaf compounds execute the basic clear and draw tasks and, except for the first child, have to readback the result into the specified output frames. The root compound executes the assemble task (sort-first tiled image compositing) once the output frames are available.

4.5.2 Sort-Last Configuration

Figs. 9 and 11b show a sort-last configuration with parallel image compositing. The leaf compounds execute the rendering and read back two tiles each to be z-composited by the other channels. The intermediate compounds execute the z-compositing in parallel using frame buffer data from the other channels via the indicated output-input frame mapping. Once a channel has completed this assemble task (sort-last z-buffer image compositing) on its tile, the color frame buffer content is handed over to the root compound, which puts together the tiles to form the final image. Note that a compound does not need to read back a tile that is processed in a parent on the same channel since it is already in place (e.g., the compounds executed on the "dest"

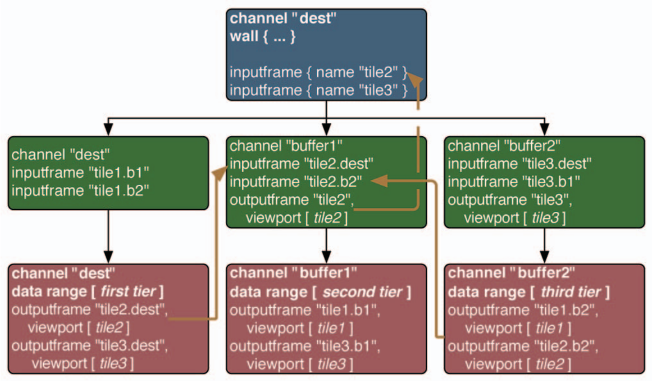


Fig. 9. Compound tree for a three-to-one direct-send sort-last configuration.

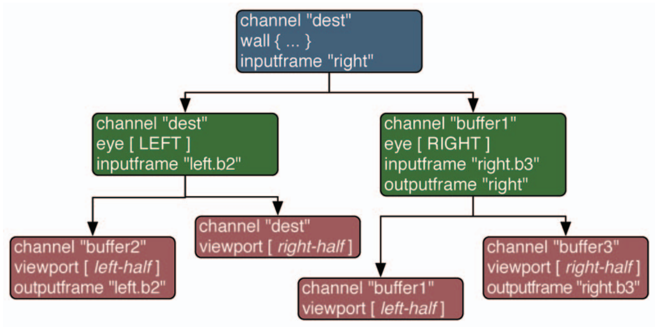


Fig. 10. Compound tree for a four-to-one stereo/sort-first configuration.

channel in Fig. 9). The arrows illustrate the data flow for the tile being z-composited by the channel named "buffer1," according to a direct-send sort-last image compositing [15].

4.5.3 Stereo Sort-First Configuration

Figs. 10 and 11c show a mixture of decomposition algorithms in a multilevel compound tree. Stereo rendering is mixed with sort-first decomposition. The first level is a stereo decomposition for the left and right eyes, which is in turn parallelized for each eye on two channels using a sort-first decomposition. The channels used for rendering are again also used for compositing, which again allows some image transfer optimizations. Fig. 11c uses anaglyphic stereo for better readability, but the compound works the same for quad-buffered stereo.

4.5.4 Multiscreen Configurations

Multiscreen display systems can easily be configured with Equalizer by assigning one destination channel to each screen and additionally specifying the rendering decomposition to generate the different screen images. For example, it is straightforward to set up any sized display wall configuration that uses its own nodes that drive the tiled screens or projectors, or for that matter any additional nodes not directly driving a display, for parallel rendering and compositing. Nodes can freely be combined to share the task of rendering and in a different way to perform the image compositing task. Thus, the use of physical resources can be tailored to the particular system and use.

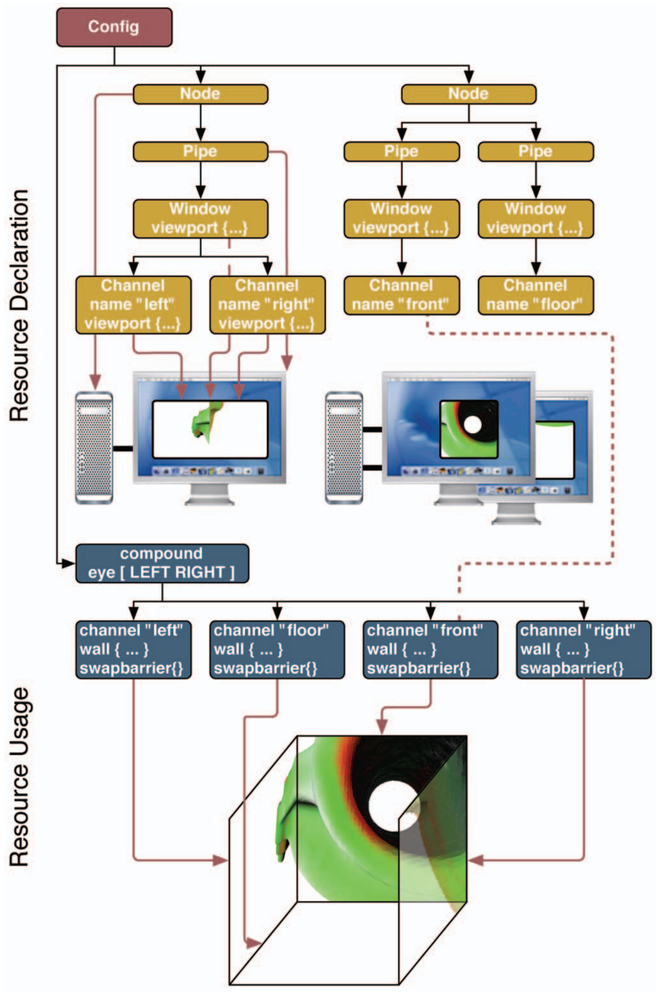


Fig. 12. A sample Equalizer CAVE configuration with the associated real-world counterparts.

Fig. 12 shows a two-node, three-pipe, three-window, four-channel configuration driving a four-sided CAVE. In this example, we again show the mapping to physical resources where two channels are mapped to one single pipe and one node contains two pipes. The channels declared in the resource section are used by the compounds for rendering. The leaf compounds, which execute the rendering, use a *swap barrier* to synchronize their output.

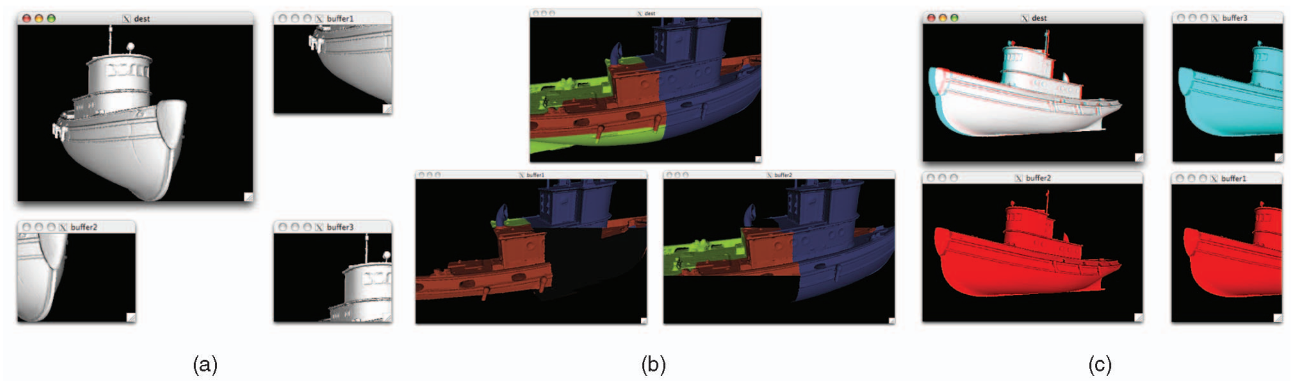


Fig. 11. (a) Sort-first scalable rendering—compound tree in Fig. 8. (b) Sort-last scalable rendering—compound tree in Fig. 9. (c) Stereo separation and sort-first decomposition—compound tree in Fig. 10.

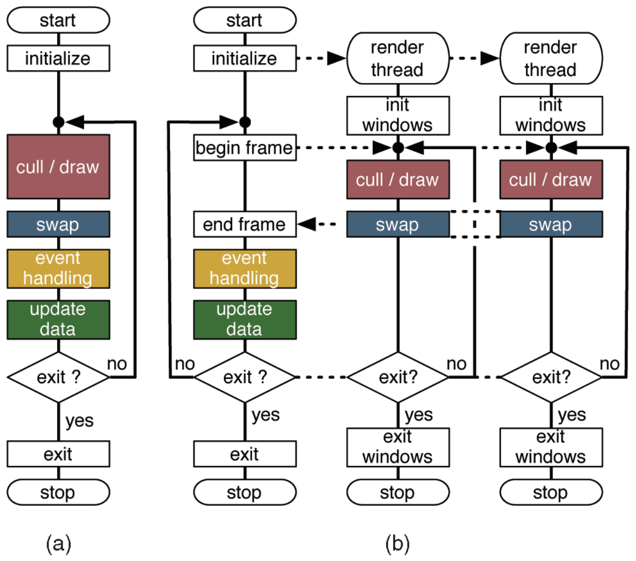


Fig. 13. (a) A typical execution flow for a single-pipe and (b) a parallel rendering application.

The root compound specifies that the left and right eyes are used for stereo rendering.

Equalizer's compound description is extremely flexible and powerful and can be used to define parallel image compositing algorithms, such as direct-send or binary-swap, as well as multilevel decompositions using different decomposition modes to balance the bottlenecks of the individual algorithms. A detailed specification can be found in [14]. Numerous sample configurations are included with the Equalizer distribution.

5 Application Development

A typical, e.g., OpenGL-based, interactive visualization application's main loop conceptually looks something like Fig. 13a. Equalizer extends this model by separating the rendering operations from the application's main loop to be executed in parallel, as shown in Fig. 13b. An Equalizer-based application subclasses from the provided C++ classes, which represent typical rendering entities, such as a node, pipe (GPU), window, and channel (view). The base Equalizer classes implement the typical use case, so that the programmer can focus on implementing the application-dependent code (more details are given in the programming guide [14]).

The hierarchical *node-pipe-window-channel* resource description (see also Fig. 14) results in more flexible applications than the single "application" class used by VR Juggler. For example, one node process in Equalizer might have two pipes, thus using two rendering threads. In VR Juggler, two processes need to be instantiated on such a dual-GPU configuration. Furthermore, it allows the programmer to store the data with the logical entity, for example, context-specific data in the window class and thread-specific data in the pipe class.

The most important change for a rendering application to take advantage of Equalizer is to provide an implementation of the `Channel::frameDraw()` method, the principal rendering routine executed in parallel by Equalizer. Equalizer provides a rendering context to this routine, which consists of the drawable and its OpenGL context, view frustum parameters, viewport, stereo buffer, and a data range for sort-last rendering. Based on these parameters, the application should implement efficient view frustum culling and rendering of the indicated part of the database. Therefore, the `cull()` and `draw()` functions indicated in Fig. 13 are called from the `frameDraw()` method.

Rendering parameters, such as the camera data, are implemented as a distributed object. The application subclasses from the base `eqNet::Object` class and provides the pointer and size of data to the base class for network distribution. During initialization, the object is registered within the rendering session. At the beginning of each frame, a new version of the object is committed and the new version is passed to the rendering callbacks by Equalizer, which synchronize their instance of the object to the given version.

Fig. 14 shows an UML diagram of the principal Equalizer classes and how they are subclassed in the polygonal rendering example (eqPly). Most of the methods overwritten by eqPly just add minor functionality and call the superclass method to do most of the work. The exception is the aforementioned method `Channel::frameDraw`, which contains the rendering code.

The implementation of multiview rendering, sort-first, and stereo task decompositions is straightforward: Based on the resources configuration, Equalizer computes the view frusta, draw buffer, and rendering tasks for the application rendering clients. `Channel::frameDraw()` is executed in parallel by the framework and should implement efficient view frustum culling for performance. The resulting image tiles are gathered and assembled automatically by Equalizer, based on the compound tree configuration. For sort-first decomposition, each contributing compound child specifies a fractional viewport of the destination compound's channel, e.g., $[0, 0, \frac{1}{3}, 1]$, $[\frac{1}{3}, 0, \frac{1}{3}, 1]$, and $[\frac{2}{3}, 0, \frac{1}{3}, 1]$ for a 2D compound three-way split in the x dimension.[1] For a stereo compound, one compound child only renders the left eye, whereas the other child renders the right eye.

For sort-last rendering, the application only has to support the ability to render a subset of the application-specific database, given by a 1D *range* interval. A range of [0, 1] indicates the entire database, while a range of $[a, b]$ with $0 \leq a < b \leq 1$ indicates a proportional subset of the database. Therefore, a simple sort-last parallel task distribution for three nodes is achieved by specifying the three data ranges $[0, \frac{1}{3}]$, $[\frac{1}{3}, \frac{2}{3}]$, and $[\frac{2}{3}, 1]$ in the compound tree of the resource configuration, each indicating one third of the full range. The mapping of the range [0, 1] to the actual data is left to the application.

Advanced applications can provide implementations for any stage of the rendering, e.g., volume rendering applications (such as our eVolve demo) can override `Channel::frameAssemble()` in order to implement a back-to-front sorted α-blended assembly of the provided frame image data.

Sort-first and sort-last rendering can be load-balanced by updating the viewport split or data range subdivision, respectively. These values are currently fixed in the

1. Viewport decomposition syntax is $[x, y, width, height]$.

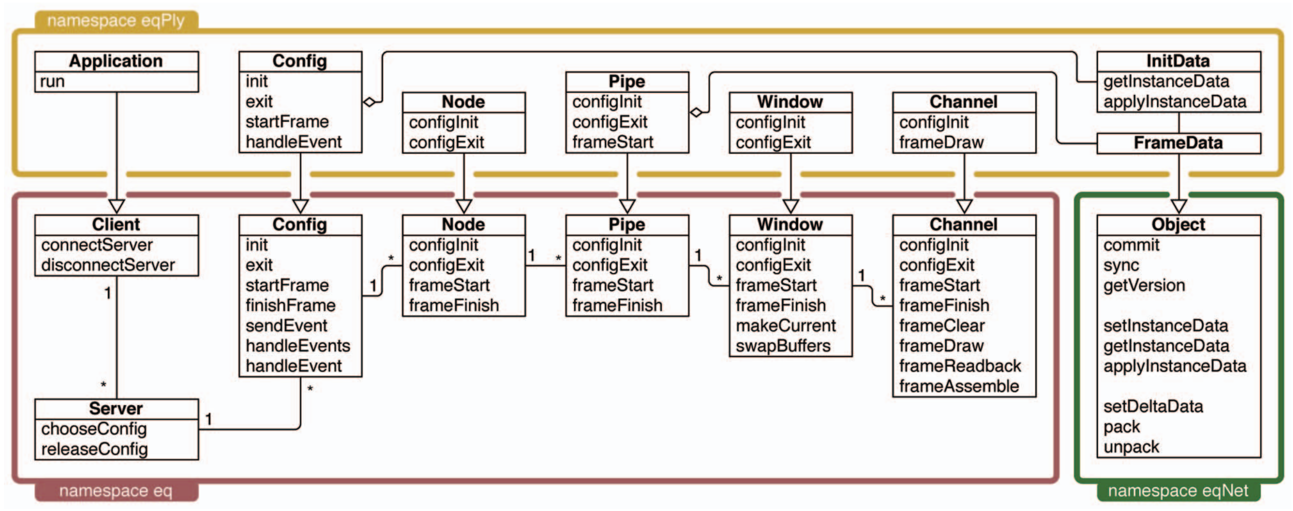


Fig. 14. UML diagram of the base Equalizer and extended eqPly classes.

compound tree but can be updated by the application based on its internal rendering statistics if desired. Equalizer is scheduled to provide simple automatic load balancing strategies based on its own internal statistics in the near future.

6 Experimental Results

We conducted our experiments on two different clusters, which exhibit different GPU performance and network bandwidth. The first, *Hactar*, is a six-node rendering cluster with the following technical specifications: dual 2.2-GHz AMD Opteron CPUs, 4 Gbytes of RAM, Geforce 7800 GTX PCIe graphics, and a high-resolution 2,560 $\times$ 1,600 pixel LCD panel per node; Myrinet network and switch. The second configuration, *Horus*, consisted of 16 nodes with the following technical details: dual 2.4-GHz AMD Opteron CPUs, 4 Gbytes of RAM (one node has two dual-Core 2-GHz AMD Opterons and 32-Gbyte RAM), Quadro FX4500 PCIe graphics; 1-Gbps ethernet network and switch.

For most tests, we used a full-size destination channel with a resolution of 1,280 $\times$ 800 on Hactar and 1,280 $\times$ 1,024 on Horus, since these are typical window sizes for scalable parallel rendering. Pixel read, write, and network transmission performances are given in Table 2. The slower network image transmission on Horus is due to missing SDP support, thus showing the influence of network bandwidth.

Our prototype test applications included two 3D viewers: *eqPly* for rendering simple polygonal data, organized spatially in an octree for better view frustum culling and sort-last data range selection, and *eVolve* for 3D texture-based direct volume rendering. The polygonal data are rendered using display lists, and each vertex consists of 24 bytes (position + normal). The volume renderer keeps the volume data in GPU texture memory using 4 bytes per voxel (packed scalar + gradient). Table 3 lists our experimental test models.

Due to the limitations of the scope of this paper, our experimental results provide the basic evidence of the flexibility and scalability potential of Equalizer but do not cover an extensive range of test data sets, compound configurations, or cluster sizes. This requires an additional dedicated and comprehensive performance study. The used test applications eqPly and eVolve are also not yet fully optimized with respect to large-scale data management, culling, or GPU usage.

6.1 Decomposition Modes

The power of Equalizer lies in its flexibility to configure different scalable task decomposition and image compositing strategies efficiently using the introduced compound tree structure. Various exemplary use cases have already been shown, demonstrating the power of the compound structure in Section 4.5 and also Fig. 1, including tiled screen rendering (e.g., for display walls or CAVEs), partitioned rendering of the geometry database (mostly for scalability), or an eye-separated sort-first parallelized stereo rendering. The quintessential benefit of Equalizer's process model and compound tree structure lies in an easy-to-configure and very scalable parallel rendering system. Therefore, we demonstrate various use cases of the flexible task decomposition possibilities in Equalizer that demonstrate the potential of the presented system.

TABLE 2
Pixel Transfer Timings for a Full-Size Image

Cluster	*GL Format, Type*	*read*	*write*	*transmit*
Hactar	BGRA, UNSIGNED_BYTE	5.2*ms*	4.1*ms*	9.0*ms*
	DEPTH_COMPONENT, FLOAT	5.8*ms*	37*ms*	8.9*ms*
Horus	BGRA, UNSIGNED_BYTE	5.5*ms*	2.8*ms*	22.7*ms*
	DEPTH_COMPONENT, FLOAT	5.7*ms*	48*ms*	22.7*ms*

TABLE 3
Size in Number of Polygons or Voxels of Our Test Models

Model	*Polygons*
David head	$4 \cdot 10^6$
David 2mm	$8 \cdot 10^6$
Thai statue	$10 \cdot 10^6$
Lucy	$28 \cdot 10^6$
David 1mm	$56 \cdot 10^6$

Model	*Size*
Skull	512^3
Skull	256^3
MRI Head	256^3
Engine	256^3
VisMale	$256^2 \times 128$

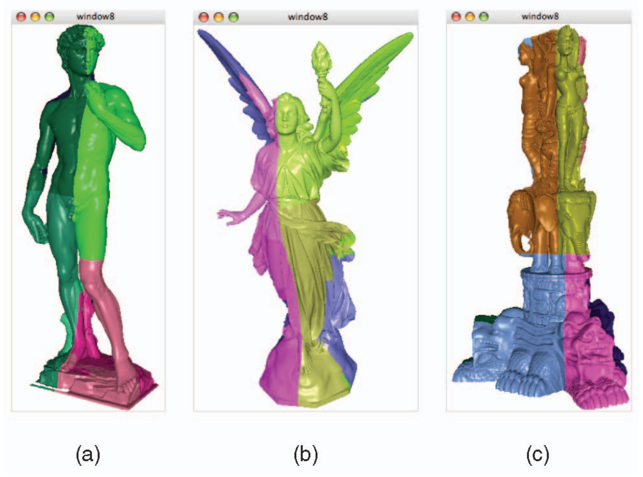


Fig. 15. Destination views of large polygonal models using an eight node sort-last configuration, with color-coded node contributions for illustration purposes. (a) David1. (b) Lucy. (c) Thai statue.

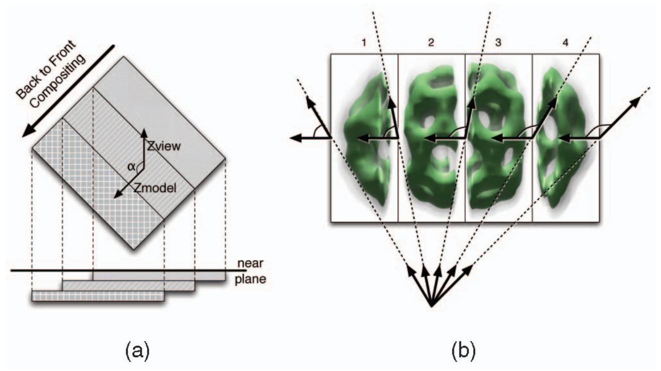


Fig. 16. (a) Basic back-to-front compositing order of parallel volume slabs. (b) Volume divided into a number of slabs. Perspective compositing order is 4-3-1-2 or 1-4-3-2.

6.1.1 Sort-Last

Scalable parallel rendering is demonstrated in Fig. 15, which shows screenshots of eqPly using an eight-to-one node sort-last rendering setup. The compound tree configuration is similar to the example given in Fig. 9 but with eight instead of three rendering and compositing channels. Corresponding sort-last scalability results obtained on Hactar are shown in Fig. 22a.

The eVolve demo application uses a hardware-accelerated 3D texture-based volume rendering algorithm [37], where the 3D texture is intersected by some proxy geometry, a series of view-aligned clipped quadrilaterals. The scalar and gradient values are interpolated from the 3D texture, and the slices are α-blended back to front. To improve visual quality, classification of the scalars is done by preintegration [16].

For sort-last rendering, the volume data range is divided uniformly into slabs along one dimension, as illustrated in Fig. 16. Each node renders one slab into a partial image, and final image assembly is performed by perspective-correct back-to-front α compositing of the partial frame data based on the relative positions of the slabs with respect to the viewer, see also Fig. 16. Such sort-last volume rendering has

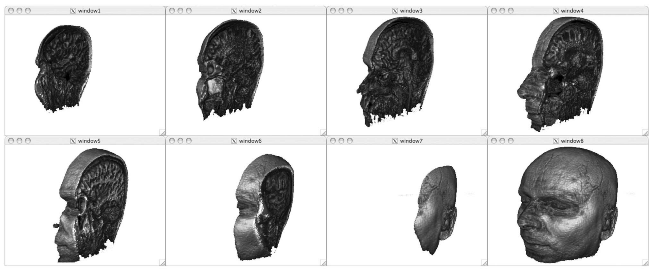


Fig. 17. Sort-last parallel rendering of a large volume data set divided uniformly into slabs. Lower right window shows final destination channel with back-to-front α-blended slab images.

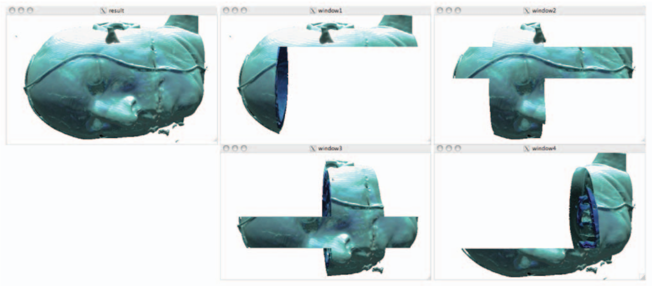


Fig. 18. Demonstration of direct-send image compositing in combination with α-blended volume rendering. Each node renders one volume slab as well as composites one horizontal image stripe for final assembly, which is displayed in the upper left window.

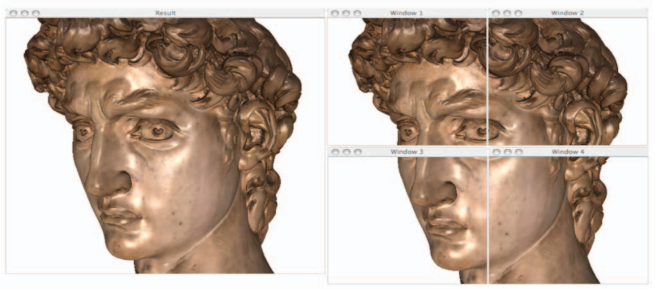


Fig. 19. Tiled sort-first parallel rendering using four channels and showing the final assembled image on the left.

the advantage of scaling both texture and main memory usage as well as pixel fill rate.

Fig. 17 demonstrates scalable sort-last rendering with eVolve using an eight-to-one node compound setup. In this example, final α compositing of the rendered volume slabs is performed on the destination display channel. In contrast, Fig. 18 demonstrates the combination of the (in-place) direct-send compositing principle [15] with back-to-front α blending, required by the above outlined direct volume rendering. This example provides further evidence how basic parallel rendering features of Equalizer can orthogonally be exploited for specific visualization tasks.

6.1.2 Sort-First

Sort-first parallel rendering can directly be applied for tiled multiscreen display systems, and it offers the benefit of simple final image assembly, which does not require a costly z-depth or α-opacity compositing stage. Fig. 19 shows a simple four-split tiled sort-first rendering of a polygonal model that can be used to drive multiple displays of a tiled

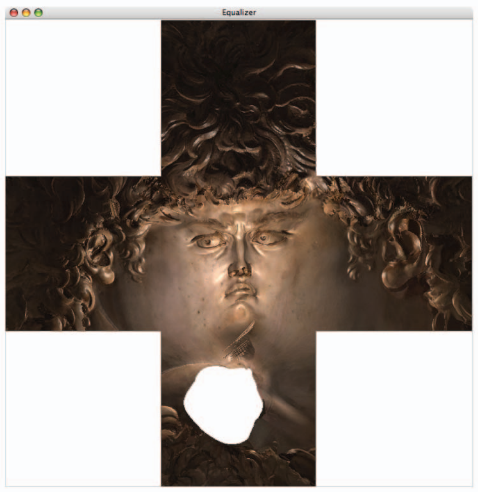


Fig. 20. Environment cube map frame buffer image of a five-sided CAVE display configuration. Five sort-first rendering channels generate the different views in a single window.

wall or render subregions of one single screen as shown in this example.

A five-sided CAVE configuration is demonstrated in Fig. 20. A sort-first compound tree distributes the rendering tasks to five different channels, each rendering and driving the display of one of the views of the five-sided CAVE. Final image assembly in a form of a cube environment map is performed to illustrate the result.

For volume rendering, typically no special programming is needed when targeting sort-first or stereo decompositions, since volume rendering is mostly fill-rate limited and thus scales nicely in this mode. In Fig. 21, we demonstrate yet another combination of task decomposition modes, where a red-blue stereo image is generated by: first, stereo separation of the rendering task for the left and right eye views, and second, sort-first decomposition of the screen (see also compound structure in Fig. 10).

6.2 Performance

Performance experiments were performed on the Hactar and Horus parallel rendering clusters mentioned above, which exhibit different graphics and network bandwidth characteristics. Specific sort-last direct-send image compositing scalability results can also be found in [15]. In the performance charts, sort-first decomposition is also denoted

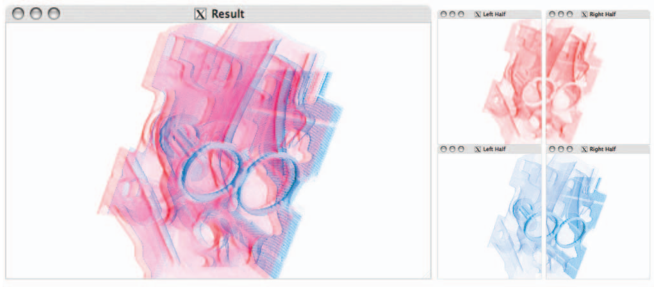


Fig. 21. Four-to-one stereo/sort-first parallel volume rendering.

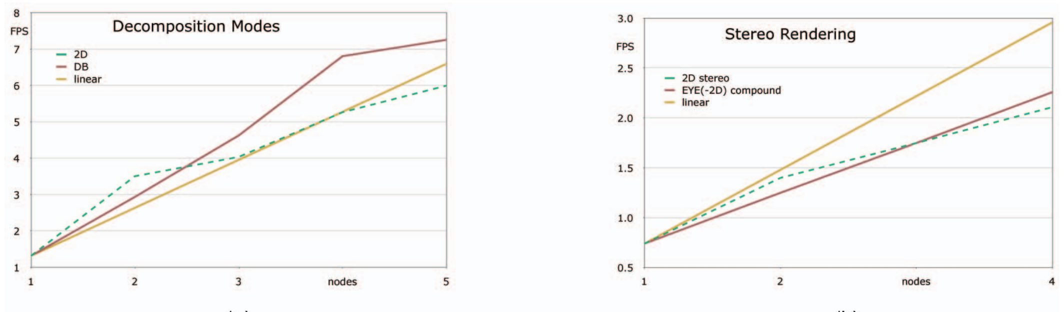
by the shortcut *2D* and sort-last parallel rendering is indicated by *DB*.

6.2.1 Hactar

In the first benchmarks on Hactar, we measured the performance of different task decomposition modes. The Thai statue was used in these experiments and a fixed camera path of 100 frames was used to obtain the average frames per second as the result.

In Fig. 22a, we tested n-to-one sort-first as well as sort-last decompositions. The sort-first compounds use a trivial tile assembly on the destination channel, while the sort-last compounds use direct-send compositing. For sort-first parallel rendering, the speedup heavily depends on the decomposition of the view frustum and, hence, the tiling of the window. For this study, the data set is roughly placed in the middle of the screen such that a simple tiling results in a fair, though not perfect, load distribution. The graph 2D in Fig. 22a shows a nice close-to linear speedup for sort-first rendering, and as expected, the overhead from clipped primitives is not dominating for small numbers of tiles. Equalizer also shows excellent scalability with respect to sort-last rendering, graph DB in Fig. 22a. Image compositing overhead is not manifested at this level of parallelism, partly also due to the efficient direct-send compositing algorithm (see also [15]).

The second set of benchmarks in Fig. 22b uses different approaches to scale the performance during stereo rendering. The first graph 2D stereo uses a sort-first decomposition, where the image is split in half and then assigned to two nodes for each of the two eye passes, which are assembled on the destination channel in the parent node into the correct stereo buffers. The second graph EYE-2D does first a stereo decomposition, separating into left and right eye rendering



(a) (b)

Fig. 22. (a) Sort-first and sort-last many-to-one rendering performance. (b) Different stereo rendering decompositions. (*Hactar*).

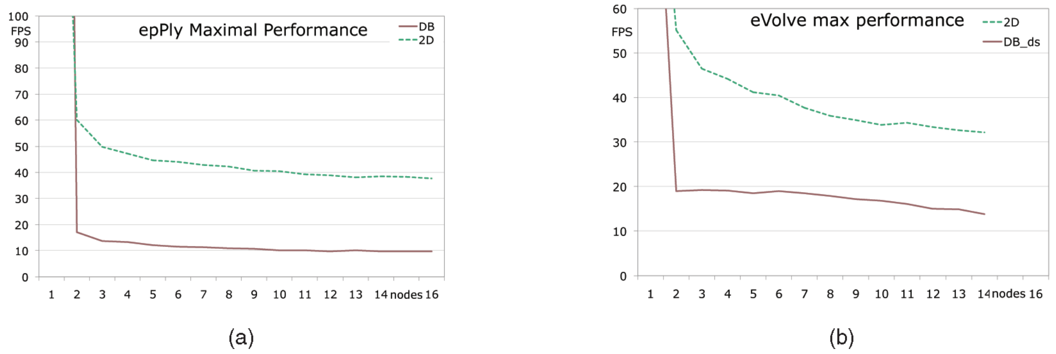


Fig. 23. Maximal frame rate performance considering only the distributed image assembly, compositing, and final display, using trivial geometry for sort-first and sort-last rendering. (*Horus*). (a) Polygonal rendering. (b) Volume rendering.

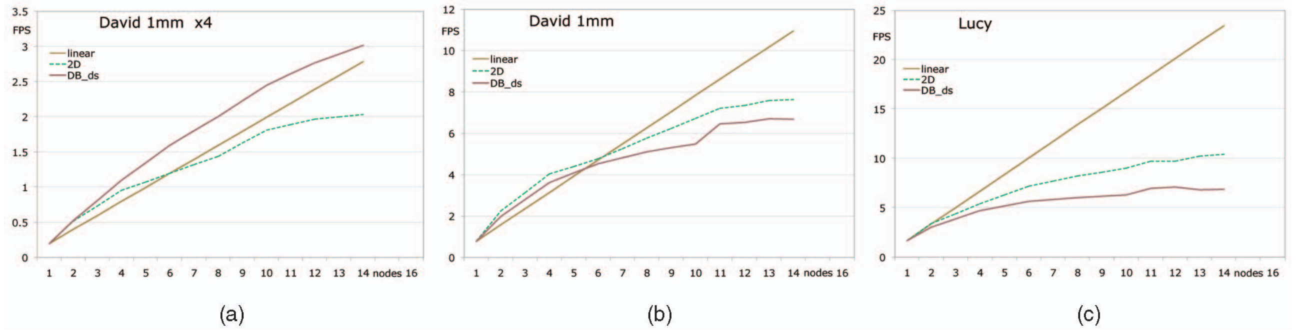


Fig. 24. Frame rate performance of sort-first and sort-last parallel rendering of large polygonal models. (*Horus*). (a) 225M triangles. (b) 56M triangles. (c) 28M triangles.

tasks, and then a sort-first decomposition into screen tiles. The graphs in Fig. 22b show a good linear speedup but also indicate that the more complicated stereo image assembly and compositing incurs a small overhead factor.

6.2.2 Horus

To evaluate the basic scalability of parallel rendering and separating out the networking and compositing costs, we performed some baseline experiments, as reported in Fig. 23. In this test, we rendered some screen full of trivial geometry to measure the overall system bottleneck with respect to pixel readback, transmission, z/α compositing, and pixel-draw for final display. It is clear that on a single node this overhead is negligible as the frame buffer data never leaves the GPU memory. Only for distributed parallel rendering using multiple nodes the overhead actually limits the achievable frame rate.

For up to 16 nodes on Horus, we can observe that polygonal rendering with eqPly is bounded by around 10 FPS for sort-last and 35 for sort-first rendering (Fig. 23a). Despite different frame data and compositing—back-to-front α-blending instead of z-depth visibility culling—a similar trend can also be observed for our volume renderer eVolve in Fig. 23b. The difference between sort-first (2D) and sort-last (DB) can be attributed to the significantly different image assembly stages. For 2D, overall the assembly only needs to draw one full-resolution image into the destination channel (although one in parts). On the other hand, the final DB image assembly consists of combining many full-resolution images using z-depth visibility culling (polygonal rendering) or α blending (volume rendering).

In fact, these maximal distributed rendering frame rates depend on a number of parameters including: pixel readback rate, network transmission, pixel draw rate (compositing), as well as binary frame buffer formats. Most of these parameters are not yet fully optimized in Equalizer. In particular, the pixel draw rate is severely limiting the current frame buffer assembly and image display stage. This is partly due to a slow (driver) implementation of the OpenGL `glDrawPixels` functionality, which may be improved by implementing z/α compositing using asynchronous texture uploads and fragment shaders or CPU-based compositing. Furthermore, the binary frame buffer number format and packing of color, alpha, and depth channels can also have a significant impact as implicit format transformations could be caused in the drivers and these may be executed in software (on the CPU instead of the GPU). From our experiments, a number of signs indicate that the latter two issues are currently the major limiting factors. Furthermore, network transmission can be improved in the future by more sophisticated frame buffer data compression and region-of-interest selection methods.

The scalability tests reported in Fig. 24 show excellent speedup factors for large polygonal data sets. Combining four large models (4 × David 1 mm) to a 225M triangle mesh, eqPly demonstrates full linear speedup for (direct-send) sort-last (DB_ds) and near-linear speedup for sort-first (2D) rendering, as shown in Fig. 24a. Using only a single 56M triangle David 1-mm model, we can observe in Fig. 24b that the parallel rendering speedup is dampened as soon as the individual nodes reach internal frame rates that approach the maximal distributed rendering bounds. For the 56M David 1

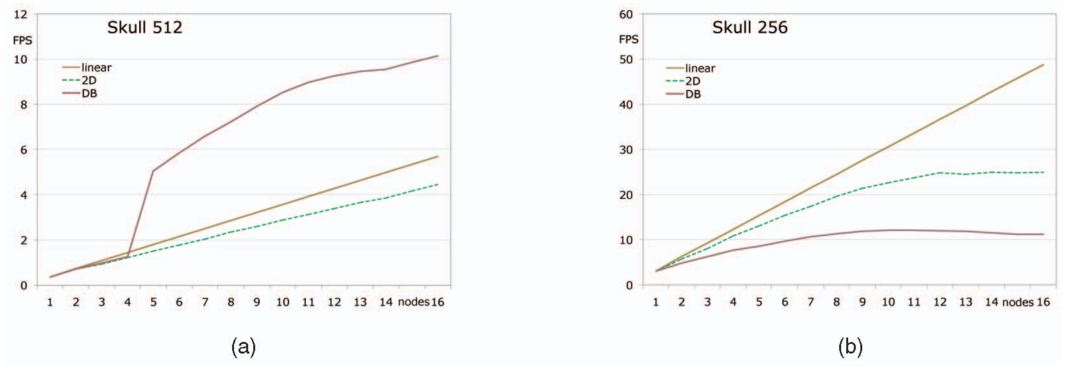


Fig. 25. Frame rate performance of sort-first and sort-last parallel volume rendering. (*Horus*). (a) 512^3 voxels. (b) 256^3 voxels.

mm, this is the case at around 8 to 10 nodes, and for smaller models such as the 28M triangles Lucy, this limit is hit earlier, already at around four to six nodes as shown in Fig. 24c.

In Fig. 25, we report our experimental performance speedup results for 3D texture-based volume rendering. The achieved numbers demonstrate very good scalability, up to the maximal distributed rendering performance. In fact, for the large 512^3 voxel volume, we can observe a drastic performance jump at five nodes, which is most likely due to the fact that the reduced volume slabs fit more optimally into the GPU 3D texture memory. The smaller 256^3 volume test shows a similar behavior as the smaller polygonal models with the performance approaching the maximal bounds after a certain number of added parallel rendering nodes.

One observation from the above tests is that the sort-first (2D) polygonal rendering performance does not reach the maximal performance limit, compare Figs. 24b and 24c with Fig. 23a, while the sort-last (DB) generally does. From our current tests and investigations, we conclude that this is mostly due to the view frustum culling costs, which add an additional overhead that is not included in Fig. 23a. Our current hierarchical polygonal mesh management and view frustum culling has some potential for optimization in that respect. Optimized hierarchical and multiresolution data structures and culling methods may reduce this extra overhead largely. Thus, the advantage of simpler 2D image compositing, as mentioned above along with Fig. 23, can be compensated by view frustum culling if it is not fully optimized.

On the other hand, the simpler view frustum culling in 3D texture-based volume rendering—bounding the proxy geometry to the view frustum—allows it to better approach the maximal performance. This is indicated in Fig. 25b, where 2D and DB reach a performance much closer to the maximal reported in Fig. 23b.

6.2.3 Comparison to Chromium

Despite Equalizer and Chromium having slightly different main targets, flexible configuration and scalability on one side and transparent abstraction of the OpenGL API on the other side, we provide a limited experimental evaluation here. For this test, we used a simple display wall configuration as shown in Fig. 26, with a static model, rotating about its vertical axis, placed such that it nicely covers the different screens. A standard tile-sort Chromium configuration has been compared to a simple Equalizer display-wall compound setup. The polygonal model is rendered using eqPly and uses display lists for the static geometry. Using display lists allows Chromium to send geometry and texture data once to the rendering nodes (retained mode rendering) and display them repeatedly using *glCallLists()*, which is inexpensive in terms of network overhead [5].

According to [27], [5], [51], as well as our own understanding, a tile-sort display-wall setup with static geometry rendered in retained mode should be reasonably favorable for Chromium because the display lists have to be transmitted only once over the network, and only simple display calls will be processed and distributed by Chromium for each rendered frame. Fig. 27 shows the experimental results of the display-wall comparison between Chromium and Equalizer. One can clearly observe that while Chromium initially increases performance when adding nodes, it quickly stagnates and even decreases when more nodes are added. In contrast, Equalizer continually improves performance with more added nodes and only exhibits a smooth drop-off in speedup, due to the expected synchronization and network overhead as the

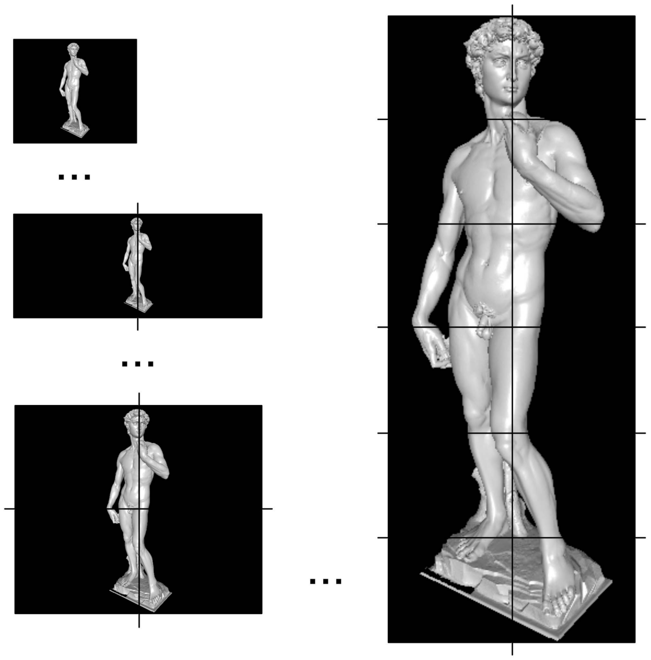


Fig. 26. Display wall configurations to compare Equalizer and Chromium using 1, 2, 4, 6, . . . , and 12 screens and rendering nodes.

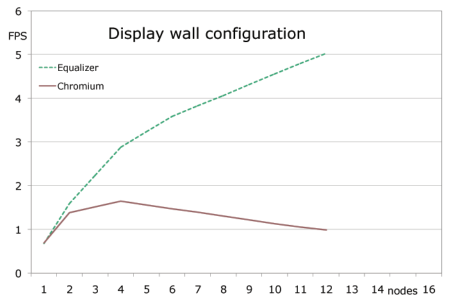


Fig. 27. Frame rate performance comparison between Chromium and Equalizer for tiled display wall configurations of up to 12 screens and nodes. (*Horus*).

rendered data gets negligible in size per node. This performance difference may also be due to the fact that Equalizer can benefit from distributed parallel view frustum culling.

6.3 Latency and Viewport Size

In these benchmarks, we measure the influence of the viewport size and latency on the performance, tested with polygonal rendering using eqPly on Hactar. All tests were conducted using a sort-last direct-send configuration with five nodes. Fig. 28a varies the config-latency l_{config} from 0 to 6. One can observe that increasing the latency from a strict frame synchronization with $l_{\mathrm{config}} = 0$ immediately increases the performance by about 15 percent. This is achieved through reduced synchronization bottlenecks and better task pipelining as rendering channels can overlap their draw tasks between frames. We also notice, as expected, that further increasing the latency does not further improve rendering performance, due to other synchronization constraints such as image transfers. We can conclude that a small latency of only one or two frames is sufficient to avoid most drawbacks of a strictly frame synchronized parallel rendering execution.

In Fig. 28b, experiments with different viewport sizes for the destination window are shown, and hence, the amount of transferred and z-composited pixel data varies accordingly. The graph exhibits the expected asymptotic behavior toward the constant time composition cost of direct send, as analyzed in [15], regardless of the viewport size. Since the composition cost is directly dependent on the viewport size, the performance approaches and is limited by the constant time compositing as soon as the draw cost is reduced sufficiently by parallel load distribution. This is the normal expected behavior. However, we would like to point out here that the flexible compound structure allows for complex combinations of parallel rendering and parallel compositing where the number of contributing channels can vary and thus allows for optimized resource usage.

7 Discussion and Conclusion

In this paper, we have presented a state-of-the art distributed parallel rendering framework, which has been designed to be minimally invasive in order to facilitate the porting and development of real-world visualization applications. Equalizer has also been designed to be as generic as possible to support development of parallel rendering applications for different data types.

The major strengths of Equalizer are its flexible compound tree structures, fully distributed rendering support, as well as efficient compositing algorithms. Compound trees allow for easy specification of complex parallel task decomposition strategies, which are automatically implemented and executed by the Equalizer system. The parallel task decomposition and efficient compositing achieves great scalability for large data sets as demonstrated by the 225M polygonal mesh and 512^3 volume data sets. The fully distributed design supports effective network synchronization as well as shared objects and remote method invocation, which facilitate the development of decentralized applications.

Parallel rendering of transparent data is not only supported for sort-first configurations with application-side back-to-front traversal but also for sort-last configurations given the data partitioning enables a back-to-front spatial ordering. This is demonstrated in our eVolve volume rendering application, which exploits the efficient α-compositing compound provided in Equalizer.

Scalable sort-first rendering depends on a balanced distribution of the rendering cost across the different screen tiles. To achieve this, dynamic tile decomposition must be supported as well as some basic rendering cost heuristics for effective load balancing. These extensions pose interesting but also tractable challenges and are lined up for integration into Equalizer. In fact, efficient load balancing is an important aspect for parallel applications, and with its flexible task decomposition abilities, Equalizer offers the basic structural support that applications can readily use.

Equalizer efficiently supports but does not solve all problems of parallel rendering for the programmer. As

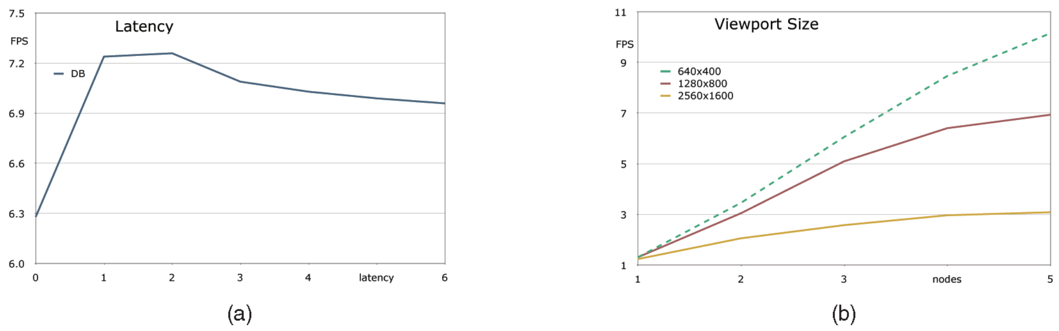


Fig. 28. Influence of (a) latency and (b) viewport size on rendering performance, using five nodes. (*Hactar*).

mentioned before, load balancing is a focus area, as is an improved image compression and transport (readback-transfer-draw) pipeline. While these two problems are going to be addressed directly in Equalizer, the data distribution and replication problem may be more of an application-dependent challenge, which will be supported by facilitating distributed objects.

The current Equalizer system already goes beyond just the necessary basic scalable rendering functionality. Nevertheless, we plan to extend the functionality to include also time-multiplex support, sophisticated automatic load balancing for sort-first and sort-last task decompositions, as well as an API to compress and mask the channels' screen frames for optimized image transport.

Aside from the core parallel rendering API, in the long term, we plan to improve the resource management capabilities of the server by enabling it to handle multiple applications, resource reservation, and cross-application load balancing. Furthermore, the creation of a transparent OpenGL layer with Equalizer as the back end could allow running existing applications alongside with parallel applications. Eventually, we will integrate remote visualization capabilities, for example by supporting the VNC protocol.

Acknowledgments

This work was supported in part by the Swiss National Science Foundation under Grant 200021-116329/1. The authors would like to thank the following institutions and projects for providing the 3D geometry and volume test data sets: the Digital Michelangelo Project, Stanford 3D Scanning Repository, Cyberware, volvis.org, and the Visual Human Project.

References

[1] SGI, "OpenGL Multipipe SDK," http://www.sgi.com/products/software/multipipe/sdk/, Technical Publication 007-4516-002, 2002.

[2] G. Agranov and C. Gotsman, "Algorithms for Rendering Realistic Terrain Image Sequences and Their Parallel Implementation," *The Visual Computer,* vol. 11, no. 9, pp. 455-464, 1995.

[3] J. Ahrens and J. Painter, "Efficient Sort-Last Rendering Using Compression-Based Image Compositing," *Proc. Eurographics Workshop Parallel Graphics and Visualization (EGWPGV),* 1998.

[4] J. Allard, V. Gouranton, L. Lecointre, E. Melin, and B. Raffin, "Netjuggler: Running VR Juggler with Multiple Displays on a Commodity Component Cluster," *Proc. IEEE Virtual Reality Conf. (VR '02),* pp. 275-276, 2002.

[5] W.E. Bethel, G. Humphreys, B. Paul, and J.D. Brederson, "Sort-First, Distributed Memory Parallel Visualization and Rendering," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG '03),* pp. 41-50, 2003.

[6] P. Bhaniramka, P.C.D. Robert, and S. Eilemann, "OpenGL Multipipe SDK: A Toolkit for Scalable Parallel Rendering," *Proc. IEEE Visualization (VIS '05),* pp. 119-126, 2005.

[7] A. Bierbaum and C. Cruz-Neira, "ClusterJuggler: A Modular Architecture for Immersive Clustering," *Proc. Workshop Commodity Clusters for Virtual Reality, IEEE Virtual Reality Conf.,* 2003.

[8] A. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira, "VR Juggler: A Virtual Platform for Virtual Reality Application Development," *Proc. IEEE Virtual Reality Conf. (VR '01),* pp. 89-96, 2001.

[9] W. Blanke, C. Bajaj, D. Fussel, and X. Zhang, "The Metabuffer: A Scalable Multi-Resolution 3-D Graphics System Using Commodity Rendering Engines," Technical Report TR2000-16, Univ. of Texas at Austin, 2000.

[10] X. Cavin and C. Mion, "Pipelined Sort-Last Rendering: Scalability, Performance and Beyond," *Proc. Eurographics Symp. Parallel Graphics and Visualization (EGPGV),* 2006.

[11] X. Cavin, C. Mion, and A. Filbois, "COTS Cluster-Based Sort-Last Rendering: Performance Evaluation and Pipelined Implementation," *Proc. IEEE Visualization (VIS '05),* pp. 111-118, 2005.

[12] W.T. Correa, J.T. Klosowski, and C.T. Silva, "Out-of-Core Sort-First Parallel Rendering for Cluster-Based Tiled Displays," *Proc. Eurographics Workshop Parallel Graphics and Visualization (EGWPGV '02),* pp. 89-96, 2002.

[13] T.W. Crockett, "An Introduction to Parallel Rendering," *Parallel Computing,* vol. 23, pp. 819-843, 1997.

[14] S. Eilemann, "Equalizer Programming Guide," Technical Report IFI-2007.11, Dept. of Informatics, Univ. of Zurich, 2007.

[15] S. Eilemann and R. Pajarola, "Direct Send Compositing for Parallel Sort-Last Rendering," *Proc. Eurographics Symp. Parallel Graphics and Visualization (EGPGV),* 2007.

[16] K. Engel, M. Kraus, and T. Ertl, "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware (GH '01),* pp. 9-16, 2001.

[17] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover, "PixelFlow: The Realization," *Proc. ACM SIGGRAPH/Eurographics Workshop Graphics Hardware (GH '97),* pp. 57-68, 1997.

[18] A. Garcia and H.-W. Shen, "An Interleaved Parallel Volume Renderer with PC-Clusters," *Proc. Eurographics Workshop Parallel Graphics and Visualization (EGWPGV '02),* pp. 51-60, 2002.

[19] E. Gobbetti and F. Marton, "Layered Point Clouds: A Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-Sampled Models," *Computers and Graphics,* vol. 28, no. 1, pp. 815-826, Feb. 2004.

[20] M. Guthe, P. Borodin, Ä. Balazs, and R. Klein, "Real-Time Appearance Preserving Out-of-Core Rendering with Shadows," *Proc. Eurographics Workshop Rendering Techniques,* pp. 69-80, 2004.

[21] S. Guthe, M. Wand, J. Gonser, and W. Strasser, "Interactive Rendering of Large Volume Data Sets," *Proc. IEEE Visualization (VIS '02),* pp. 53-60, 2002.

[22] M. Houston, *Raptor,* http://graphics.stanford.edu/projects/raptor/, 2005.

[23] J. Huang, N. Shareef, R. Crawfis, P. Sadayappan, and K. Mueller, "A Parallel Splatting Algorithm with Occlusion Culling," *Proc. Eurographics Workshop Parallel Graphics and Visualization (EGWPGV),* 2000.

[24] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan, "Distributed Rendering for Scalable Displays," *Proc. IEEE Supercomputing,* Oct. 2000.

[25] G. Humphreys, M. Eldridge, I. Buck, G. Stoll, M. Everett, and P. Hanrahan, "WireGL: A Scalable Graphics System for Clusters," *Proc. ACM SIGGRAPH '01,* pp. 129-140, 2001.

[26] G. Humphreys and P. Hanrahan, "A Distributed Graphics System for Large Tiled Displays," *Proc. IEEE Visualization (VIS '99),* pp. 215-224, Oct. 1999.

[27] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P.D. Kirchner, and J.T. Klosowski, "Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters," *ACM Trans. Graphics,* vol. 21, no. 3, pp. 693-702, 2002.

[28] H. Igehy, G. Stoll, and P. Hanrahan, "The Design of a Parallel Graphics Interface," *Proc. ACM SIGGRAPH '98,* pp. 141-150, July 1998.

[29] A. Johnson, J. Leigh, P. Morin, and P. Van Keken, "GeoWall: Stereoscopic Visualization for Geoscience Research and Education," *IEEE Computer Graphics and Applications,* vol. 26, no. 6, pp. 10-14, Nov./Dec. 2006.

[30] K. Jones, C. Danzer, J. Byrnes, K. Jacobson, P. Bouchaud, D. Courvoisier, S. Eilemann, and P. Robert, "SGI OpenGL Multipipe SDK User's Guide," Technical Report 007-4239-004, Silicon Graphics, 2004.

[31] C. Just, A. Bierbaum, A. Baker, and C. Cruz-Neira, "VR Juggler: A Framework for Virtual Reality Development," *Proc. Immersive Projection Technology Workshop (IPT),* 1998.

[32] P.G. Lever, *SEPIA—Applicability to MVC,* white paper, Manchester Visualization Centre (MVC), Univ. of Manchester, 2004.

[33] P. Li, W.H. Duquette, and D.W. Curkendall, "RIVA: A Versatile Parallel Rendering System for Interactive Scientific Visualization," *IEEE Trans. Visualization and Computer Graphics,* vol. 2, no. 3, pp. 186-201, Sept. 1996.

[34] P.P. Li, S. Whitman, R. Mendoza, and J. Tsiao, "ParVox: A Parallel Splatting Volume Rendering System for Distributed Visualization," *Proc. IEEE Parallel Rendering Symp. (PRS '97),* pp. 7-14, 1997.
[35] S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich, "Scalable Interactive Volume Rendering Using Off-the-Shelf Components," Technical Report CACR-2001-189, California Inst. of Technology, 2001.
[36] S. Lombeyda, L. Moll, M. Shand, D. Breen, and A. Heirich, "Scalable Interactive Volume Rendering Using Off-the-Shelf Components," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG '01),* pp. 115-121, 2001.
[37] M. Meissner, U. Hoffmann, and W. Strasser, "Enabling Classification and Shading for 3D Texture Mapping Based Volume Rendering Using OpenGL and Extensions," *Proc. IEEE Visualization (VIS '99),* pp. 207-214, 1999.
[38] L. Moll, A. Heirich, and M. Shand, "Sepia: Scalable 3D Compositing Using PCI Pamette," *Proc. Seventh IEEE Symp. Field-Programmable Custom Computing Machines (FCCM '99),* pp. 146-155, 1999.
[39] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications,* vol. 14, no. 4, pp. 23-32, 1994.
[40] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition," *Proc. ACM SIGGRAPH '92,* pp. 231-240, 1992.
[41] C. Mueller, "The Sort-Frst Rendering Architecture for High-Performance Graphics," *Proc. Symp. Interactive 3D Graphics (I3D), ACM SIGGRAPH '95,* pp. 75-84, 1995.
[42] C. Mueller, "Hierarchical Graphics Databases in Sort-First," *Proc. IEEE Parallel Rendering Symp. (PRS '97),* p. 49, 1997.
[43] S. Muraki, M. Ogata, K.-L. Ma, K. Koshizuka, K. Kajihara, X. Liu, Y. Nagano, and K. Shimokawa, "Next-Generation Visual Supercomputing Using PC Clusters with Volume Graphics Hardware Devices," *Proc. ACM/IEEE Conf. Supercomputing (SC '01),* p. 51, 2001.
[44] W. Nie, J. Sun, J. Jin, X. Li, J. Yang, and J. Zhang, "A Dynamic Parallel Volume Rendering Computation Mode Based on Cluster," *Proc. Int'l Conf. Computational Science and Its Applications (ICCSA '05),* vol. 3482, pp. 416-425, 2005.
[45] K. Niski and J.D. Cohen, "Tile-Based Level of Detail for the Parallel Age," *IEEE Trans. Visualization and Computer Graphics,* vol. 13, no. 6, pp. 1352-1359, Nov./Dec. 2007.
[46] J. Rohlf and J. Helman, "IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics," *Proc. ACM SIGGRAPH '94,* pp. 381-394, 1994.
[47] R. Samanta, T. Funkhouser, and K. Li, "Parallel Rendering with K-Way Replication," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG '01),* 2001.
[48] R. Samanta, T. Funkhouser, K. Li, and J.P. Singh, "Hybrid Sort-First and Sort-Last Parallel Rendering with a Cluster of PCs," *Proc. Eurographics Workshop Graphics Hardware,* pp. 97-108, 2000.
[49] R. Samanta, J. Zheng, T. Funkhouser, K. Li, and J.P. Singh, "Load Balancing for Multi-Projector Rendering Systems," *Proc. Eurographics Workshop Graphics Hardware,* pp. 107-116, 1999.
[50] J.P. Schulze and U. Lang, "The Parallelization of the Perspective Shear-Warp Volume Rendering Algorithm," *Proc. Eurographics Workshop Parallel Graphics and Visualization (EGWPGV '02),* pp. 61-70, 2002.
[51] O.G. Staadt, J. Walker, C. Nuber, and B. Hamann, "A Survey and Performance Analysis of Software Platforms for Interactive Cluster-Based Multi-Screen Rendering," *Proc. Eurographics Workshop Virtual Environments,* pp. 261-270, 2003.
[52] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan, "Lightning-2: A High-Performance Display Subsystem for PC Clusters," *Proc. ACM SIGGRAPH '01,* pp. 141-148, 2001.
[53] A. Stompel, K.-L. Ma, E.B. Lum, J. Ahrens, and J. Patchett, "SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG '03),* pp. 33-40, 2003.
[54] X. Tong, W. Wang, W. Tsang, and Z. Tang, "Efficiently Rendering Large Volume Data Using Texture Mapping Hardware," *Proc. Joint Eurographics-IEEE TCVG Symp. Visualization (VisSym),* 1999.
[55] G. Vezina and P.K. Robertson, "Terrain Perspectives on a Massively Parallel SIMD Computer," *Proc. Computer Graphics Int'l (CGI '91),* pp. 163-188, 1991.
[56] C.M. Wittenbrink, "Survey of Parallel Volume Rendering Algorithms," *Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA '98),* pp. 1329-1336, 1998.
[57] D.-L. Yang, J.-C. Yu, and Y.-C. Chung, "Efficient Compositing Methods for the Sort-Last-Sparse Parallel Volume Rendering System on Distributed Memory Multicomputers," *J. Supercomputing,* vol. 18, no. 2, pp. 201-220, Feb. 2001.
[58] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha, "Quick-VDR: Out-of-Core View-Dependent Rendering of Gigantic Models," *IEEE Trans. Visualization and Computer Graphics,* vol. 11, no. 4, pp. 369-382, July/Aug. 2005.
[59] X. Zhang, C. Bajaj, and W. Blanke, "Scalable Isosurface Visualization of Massive Datasets on COTS Clusters," *Proc. IEEE Symp. Parallel and Large-Data Visualization and Graphics (PVG '01),* pp. 51-58, 2001.

Stefan Eilemann received the Engineering Diploma in computer science in 1998. He is a senior software engineer and CEO at Eyescale Software, with a specialization in high-performance 3D graphics, C++, parallelization of applications, and distributed systems. He developed Equalizer as a researcher in the Visualization and Multimedia Laboratory, University of Zurich. Previously, he was the technical lead of OpenGL Multipipe SDK in the Advanced Graphics Division, SGI.

Maxim Makhinya received the BSc and MSc degrees in computer science from the Moscow State University in 2004 and 2005, respectively. He is currently a research assistant and doctoral student in the Visualization and MultiMedia Laboratory (VMML), Department of Informatics, University of Zurich. His research interests include real-time graphics, parallel rendering, and high-performance visualization.

Renato Pajarola received the Dipl Inf-Ing ETH and DrSc Techn degrees in computer science from the Swiss Federal Institute of Technology (ETH), Zurich, in 1994 and 1998, respectively. Following his dissertation, he was a postdoctoral researcher and lecturer in the Graphics, Visualization and Usability (GVU) Center, Georgia Institute of Technology. In 1999, he joined the University of California, Irvine, as an assistant professor and founded the Computer Graphics Laboratory. Since 2005, he has been leading the Visualization and MultiMedia Laboratory (VMML), University of Zurich as an associate professor in the Department of Informatics. His research interests include real-time 3D graphics, multiresolution modeling, point-based graphics, interactive scientific visualization, remote and parallel rendering, compression, and interactive 3D multimedia. He has published a wide range of more than 50 peer-reviewed research articles in top journals and conferences. He frequently serves on program committees, such as the IEEE Visualization Conference (2004-2006), Pacific Graphics (2002-2003, 2007-2008), and Visualization Symposium (EuroVis; 2001, 2006-2008), and is cochairing the 2007 IEEE/Eurographics PBG Symposium papers program. He was papers cochair of the IEEE/Eurographics Symposium on Point-Based Computer Graphics in 2007 and 2008. He also received a Eurographics Second Best Paper Award (as coauthor) in 2005. He is a member of the ACM, ACM SIGGRAPH, and the IEEE Computer Society.