Equalizer: A Mature Parallel Rendering Framework

Stefan Eilemann* Renato Pajarola†

Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

Abstract— We present the features, algorithms and system integration necessary to implement a parallel rendering framework usable in a wide range of real-world research and industry applications, based on the basic architecture and implementation of the Equalizer parallel rendering framework presented in [5].

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 multi-core CPU workstations, massive parallel super computers, or cluster systems. Hand in hand goes the rapid growth in complexity of data sets from numerical simulations, high-resolution 3D scanning systems or bio-medical 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:

a) generic application support, instead of special domain solution
b) scalable abstraction of the graphics layer
c) exploit existing code infrastructure, such as proprietary scene graphs, molecular data structures, level-of-detail 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:

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

Equalizer is open source, available under the LGPL license from http://www.equalizergraphcis.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 mode using both little endian and big endian processors.

2 Related Work

In [5], we presented the Equalizer parallel rendering framework. This paper also summarized the work in parallel rendering up to 2009. In the following we will present the related work published since then.

[6] cross-segment load balancing

[3]: Framework to develop apps for multitile, thus targeted specifically for tiled-wall visualization systems: grid configuration, scalability (?), multiple work sessions, multiuser events. Master-slave approach like Eq. with grid nodes running instances of the application. Is maybe more like a GLUT and window replacement, or enhancement to deal with the OpenGL contexts and events

[2] tiled display wall virtual environment

[**?**] DisplayCluster

[10] ClusterGL?

Omegalib

3 Usability

In this section we present features motivated by real-world application use cases, i.e., new functionalities rather then performance improvements. We motivate the use case, explain the architecture and integration into our parallel rendering framework, and, where applicable, show the steps needed to use this functionality in applications.

3.1 Physical and Logical Visualization Setup

Real-world visualization setups are often complex, and having an abstract representation of the display system can simplify the configuration process. Real-world applications often have the need to be aware of spatial relationship of the display setup, for example to render 2D overlays or to configure multiple views on a tiled display wall.

We addressed this need through a new configuration section interspersed between the node/pipe/window/channel hardware resources and the compound trees configurating the resource usage for parallel rendering.

A typical installation consists of one projection canvas, which is one aggregated projection surface, e.g., a tiled display wall or a CAVE. Desktop windows are considered a canvas. Each canvas is made of one or more segments, which are the individual outputs connected to

*email: eilemann@gmail.com
†email: pajarola@acm.org

Manuscript received 5 February 2008; accepted ? April 2008; posted online ? June 2008.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

(a) (b) (c)

Fig. 1. Example Equalizer application use cases: (a) 192 Megapixel CAVE at KAUST running RTNeuron, (b) Immersive HMD with external tracked and untracked views running RTT DeltaGen for virtual car usability studies (c) Cave2 running a molecular visualization build using Omegalib.

a display or projector. Segments can be planar or non-planar to each other, and can overlap or have gaps between each other. A segment is referencing a channel, which defines the output area of this segment, e.g., on a DVI connector connected to a projector.

A canvas can define a frustum, which will create default, planar sub-frusta for all of its segments. A segment can also define a frustum, which overrides the canvas frustum, e.g., for non-planar setups such as CAVEs or curved screens. These frusta describe a physically-correct display setup for a Virtual Reality installation. A canvas may have a software or hardware swap barrier, which will synchronize the rendering of all contributing GPUs.

On each canvas, the application can display one or more views. A view is a view on a model, in the sense used by the MVC pattern. The view class is used by Equalizer applications to define view-specific data for rendering, e.g., a scene, viewing mode or camera. The application process manages this data, and the render clients receive it for rendering.

A layout groups one or more views which logically belong together. A layout is applied to a canvas. The layout assignment can be changed at run-time by the application. The intersection between views and segments defines which output channels are available, and which frustum they should use for rendering. These output channels are then used as destination channels in a compound. They are automatically created during configuration.

A view may have a frustum description. The view's frustum overrides frusta specified at the canvas or segment level. This is used for non-physically correct rendering, e.g., to compare two models side-by-side on a tiled display wall. If the view does not specify a frustum, the corresponding destination channels will use the physically correct sub-frustum resulting from the view/segment intersection.

An observer looks at one or more views. It is described by the observer position in the world and its eye separation. Each observer will have its own stereo mode, focus distance and frame loop (framerate). This allows to have untracked views and multiple tracked views, e.g., two HMDs, in the same application.

3.2 Automatic Configuration

Automatic configuration implements the discovery of local and remote resources as well as the creation of typical configurations using the discovered resources at application launch time.

The discovery is implemented in a separate library, hwsd (Hard-Ware Service Discovery), which uses a plugin-based approach to discover GPUs for GLX, AGL or WGL windowing systems, as well as network interfaces on Linux, Mac OS X and Windows. Furthermore, it detects the presence of VirtualGL to allow optimal configuration of remote visualization clusters. The resources can be discovered on the local workstation, and through the help of a simple daemon using the zeroconf protocol, on a set of remote nodes within a visualization cluster. A session identifier may be used to support multiple users on a single cluster.

The Equalizer server uses the hwsd library to discover local and remote resources when an hwsd session name instead of a .eqc configuration file is provided. A set of standard decomposition modes is configured, which can be selected through activating the corresponding layout.

This versatile mechanism allows non-experts to configure asnd profit from multi-GPU workstations and visualization clusters, as well as to provide system administrators with the tools to implement easy to use integration with cluster schedulers. This feature is transparent to Equalizer application developers.

3.3 Qt Windowing

Challenges in threading model, architecture

3.4 Tide Integration

Tiled interactive display environment, parallel pixel streaming, events

3.5 Sequel

application, renderer, view data

4 The Collage Network Library

4.1 Distributed, Versioned Objects

types (instance, delta, static), versioning, multicast, compression, serializable with dirty bits, mapping, blocking commits

4.2 Reliable Stream Protocol

UDP-based reliability protocol

4.3 Infiniband RDMA

reverse-engr impl

5 Virtual Reality

Virtual Reality is an important field for parallel rendering. It does however require special attention to support it as a first-class citizen in a generic parallel rendering framework. Equalizer has been used in many virtual reality installations, such as the Cave2 ([7]), the high-resolution C6 CAVE at the KAUST visualization laboratory and head-mounted displays (Figure 1). In the following we lay out the features needed support these installations. All features presented were motivated by application use cases and have been validated with their respective users.

5.1 Head Tracking

Head tracking is the minimal feature needed to support immersive installations. Equalizer does support multiple, independent tracked views through the observer abstraction introduced in Section 3.1. Built-in VRPN support enables the direct, application-transparent configuration of a VRPN tracker device. Alternatively, an application can provide a 4×4 tracking matrix defining the transformation from the canvas coordinate system to the observer. Both CAVE-like tracking with fixed projection surfaces and HMD tracking modes are implemented.

5.2 Dynamic Focus Distance

To our knowledge, all parallel rendering systems have the focal plane coincide with the physical display surface. For better viewing comfort, we introduce a new dynamic focus mode, where the application defines the distance of the focal plane from the observer, based on the current view direction af the user. Initial experiments show that this is particularly effective for objects placed within the immersive space, that is, in front of all display segments.

5.3 Asymmetric Eye Position

Traditional head tracking computes the left and right eye positions by using a configurable interocular distance and the tracking matrix. However, human heads are not symmetric, and by measuring individual users a more precise frustum can be computed. Equalizer supports this through the optional configuration (file or programmatically) of individual 3D positions for the left and right eye.

5.4 Application-specific Scaling

Gullivers world

5.5 Runtime Stereo Switch

5.6 Swap Synchronization and GPU affinity

6 Performance

6.1 New Decomposition Modes

The initial version of Equalizer implemented sort-first (2D), sort-last (DB) and stereo (EYE) decomposition. In the following we present new decomposition modes and motivate their use case.

6.1.1 Time-Multiplex

Time-multiplexing, or DPlex, was already implemented in [1]. While it increases the framerate linearly, it does not decrease the latency between user input and the corresponding output. Consequently, this decomposition mode is mostly useful for non-interactive movie generation. It is transparent to Equalizer applications, but does require the configuration latency to be equal or greater than the number of source channels. Furthermore, to work in a multi-threaded, multi-GPU configuration, the application needs to support running the rendering threads asynchronously, as outlined in Section 6.3.4. The output frame rate of the destination channel is smoothened using a frame rate equalizer (Section 6.2.4).

6.1.2 Tiles and Chunks

Tile and chunk decompositions are a variant of sort-first and sort-last rendering, respectively. They decompose the scene into a predefined set of fixed-size image tiles or database ranges. These tasks are queued and processed by all source channels by polling a server-central queue. Prefetching ensures that the task communication overlaps with rendering. As shown in [11], these modes provide better performance due to being inherently load-balanced, as long as there is an insignificant overhead for the render task setup. This mode is transparent to Equalizer applications.

6.1.3 Pixel

Pixel compounds decompose the destination channel by interleaving rows or columns in image space. They are a variant of sort-first decomposition which works well for fill-limited applications which are not geometry bound. Source channels cannot reduce geometry load through view frustum culling, since each source channel has almost the same frustum (only shifted by some pixels), but applied to a reduced 2D viewport. However, the fragment load on all source channels is very similar due to the interleaved distribution of pixels. This functionality is transparent to Equalizer applications, and the default compositing implementation uses the OpenGL stencil buffer to blit pixels onto the destination channel.

6.1.4 Subpixel

Subpixel compounds are similar to pixel compounds, but they decompose the work for a single pixel, for example when using multisampling or depth of field. Composition typically uses accumulation and averaging of all computed fragments for a pixel. This feature is not fully transparent to the application, since it needs to adapt (jitter or tilt) the frustum based on the iteration executed. Furthermore, subpixel compounds interact with idle image refinements, e.g., they can accelerate idle anti-aliasing of a scene when the camera and scene are not changed.

6.2 Equalizers

Equalizer are an addition to compound trees. They modify parameters of their respective subtree at runtime to optimize one aspect of the decomposition. Due to their nature, they are transparent to application developers, but might have application-accessible parameters to tune their behaviour.

6.2.1 Sort-First and Sort-Last Load Equalizer

Sort-first and sort-last load balancing is the most obvious optimization for these parallel rendering modes. Our load equalizer is fully transparent for application developers, that is, it uses a reactive approach based on past rendering times. This assumes a reasonable frame-to-frame coherency. Our implementation stores a 2D or 1D grid of the load, mapping the load of each channel. The load is stored in normalized 2D/1D coordinates using $\frac{time}{area}$ as the load, the contributing source channels are organized in a binary tree, and then the algorithm balances the two branches of each level by equalizing the integral over the area on each side.

We have implemented various tunable parameters allowing application developers to optimize the load balancing based on the characteristics of their rendering algorithm:

Damping reduces frame-to-frame oscillations. It is a normalized scalar defining how much of the computed delta from the previous position is applied. The equal load distribution within the region of interest assumed by the load equalizer is in reality not equal, causing the load balancing to overshoot.

Resistance eliminates small deltas in the load balancing step. This might help the application to cache some computations since the frustum does not change each frame.

Boundaries define the modulo factor in pixels onto which a load split may fall. Some rendering algorithms produce artefacts related to the OpenGL raster position, e.g., screen door transparency, which can be eliminated by aligning the boundary to the pixel repetition. Furthermore, some rendering algorithms are sensitive to cache alignments, which can again be exploited by chosing the corresponding boundary.

6.2.2 Cross-Segment Load Balancing

Cross-segment load balancing addresses the optimal resource allocation of n rendering resources to m output channels (with $n \geq m$). The view equalizer works in conjunction with load equalizer balancing the individual output channels. It monitors the usage of shared source channels (across outputs) and activates them to balance the rendering time of all outputs. In [6], we provide a detailed description and evaluation of our algorithm.

6.2.3 Dynamic Frame Resolution

The DFR equalizer provides a functionality similar to dynamic video resizing [9], that is, it maintains a constant framerate by adapting the rendering resolution of a fill-limited application. In Equalizer, this works by rendering into a source channel (typically on a FBO) separate to the destination channel, and then scaling the rendering during the transfer (typically through an on-GPU texture) to the destination channel. The DFR equalizer monitors the rendering performance and accordingly adapts the resolution of the source channel and zoom factor for the source to destination transfer. If the performance and source channel resolutions allows, this will not only subsample, but also supersample the destination channel to reduce aliasing artefacts.

6.2.4 Frame Rate Equalizer

The framerate equalizer smoothens the output frame rate of a destination channel by instructing the corresponding window to delay its buffer swap to a minimum time between swaps. This is regularly used for time-multiplexed decompostions, where source channels tend to drift and finish their rendering not evenly distributed over time. This equalizer is however fully independent of DPlex compounds, and may be used to smoothen irregular application rendering algorithms.

6.2.5 Monitoring

Control workstation in VR setups

6.3 Optimizations

6.3.1 Region of Interest

The region of interest is the screen-space 2D bounding box enclosing the geometry rendered by a single resource. We have extended the core parallel rendering framework to use an application-provided ROI to optimize the load equalizer as well as image compositing performance. The load equalizer uses the ROI to refine its load grid to the regions containing data. The compositing code uses the ROI to minimize image readback and network transmission. In [4], we provide the details of the algorithm, and show that this improves rendering performance of up to 25%.

6.3.2 Asynchronous Compositing

Asynchronous compositing pipelines rendering with compositing operations, by executing the image readback, network transfer and image assembly from threads running in parallel to the rendering threads. In [4], we provide the details of the implementation and experimental data showing an improvement of the rendering performance of over 25% for large node counts.

6.3.3 Download and Compression Plugins

GPU-CPU transfer plugins with optional compression (eg YUV) linked to CPU compression for network transfer

6.3.4 Thread Synchronization Modes

Per-node sync, draw sync, async

7 Applications

7.1 Livre

7.2 RTT Deltagen

7.3 RTNeuron

[8]

7.4 Raster

7.5 Omegalib

8 Experimental Results

8.1 Object Distribution

Distribute large ply model to 1..16 nodes using TCP, IB and RSP

8.2 Decomposition Modes

Two graphs: Time to render a 4k, 256-step MSAA image of largest ply model and Livre using 1..16 nodes with all modes (2D, 2D LB, DB, DPlex (over AA steps), tiles, chunks, pixel, subpixel) (+DFR for Livre)

9 Discussion and Conclusion

Acknowledgements

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

References

[1] P. Bhaniramka, P. C. D. Robert, and S. Eilemann. OpenGL Multipipe SDK: A toolkit for scalable parallel rendering. In *Proceedings IEEE Visualization*, pages 119–126, 2005.

[2] Y. Cho, M. Kim, and K. S. Park. LOTUS: Composing a multi-user interactive tiled display virtual environment. *The Visual Computer*, 28(1):99–109, 2012.

[3] K.-U. Doerr and F. Kuester. CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Transactions on Visualization and Computer Graphics*, 17(2):320–332, March 2011.

[4] S. Eilemann, A. Bilgili, M. Abdellah, J. Hernando, M. Makhinya, R. Pajarola, and F. Schürmann. Parallel Rendering on Hybrid Multi-GPU Clusters. In *Proceedings of the 12th Eurographics Symposium on Parallel Graphics and Visualization*, pages 109–117, 2012.

[5] S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):436–452, May/June 2009.

[6] F. Erol, S. Eilemann, and R. Pajarola. Cross-segment load balancing in parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, pages 41–50, 2011.

[7] A. Febretti, A. Nishimoto, T. Thigpen, J. Talandis, L. Long, J. Pirtle, T. Peterka, A. Verlo, M. Brown, D. Plepys, et al. Cave2: a hybrid reality environment for immersive simulation and information analysis. In *IS&T/SPIE Electronic Imaging*, pages 864903–864903. International Society for Optics and Photonics, 2013.

[8] J. B. Hernando, J. Biddiscombe, B. Bohara, S. Eilemann, and F. Schürmann. Practical Parallel Rendering of Detailed Neuron Simulations. In *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV, pages 49–56, Aire-la-Ville, Switzerland, Switzerland, 2013. Eurographics Association.

[9] J. S. Montrym, D. R. Baum, D. L. Dignam, and C. J. Migdal. InfiniteReality: A real-time graphics system. In *Proceedings ACM SIGGRAPH*, pages 293–302, 1997.

[10] B. Neal, P. Hunkin, and A. McGregor. Distributed OpenGL rendering in network bandwidth constrained environments. In T. Kuhlen, R. Pajarola, and K. Zhou, editors, *Proceedings Eurographics Conference on Parallel Graphics and Visualization*, pages 21–29. Eurographics Association, 2011.

[11] D. Steiner, E. G. Paredes, S. Eilemann, and R. Pajarola. Dynamic work packages in parallel rendering. In *Proceedings Eurographics Symposium on Parallel Graphics and Visualization*, number (number to appear), June 2016.