Equalizer: A Mature Parallel Rendering Framework

Stefan Eilemann, David Steiner and Renato Pajarola

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 [6].

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 [6], 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. An extensive Programming and User Guide provides in-depth documentation on using and programming Equalizer [4].

[7] 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

- *All authors are with the Visualization and MultiMedia Lab, Department of Informatics, University of Zürich.*
- *S. Eilemann is also with the Blue Brain Project, Ecole Polytechnique Federale de Lausanne.*

(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.

[?] DisplayCluster
[12] 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 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. The software barrier executes a glFinish to ensure the GPU is ready to swap, a Collage barrier (Section 4.5) to synchronize all segments, the swap buffers call followed by a glFlush to ensure timely execution of the swap command. The hardware swap barrier is implemented using the WGL_NV_swap_group or GLX_NV_swap_group extensions.

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 Runtime Reconfiguration

Switching a layout, as described above, or switching the stereo rendering mode, may involve a different set of resources after the change, including the launch and exit of render client processes. Equalizer solves this through a reconfiguration step at the beginning of each rendering frame. Each resource (channel, window, pipe, node) has an activation count, which is updated when the layout or any other relevant rendering parameter is changed. When a resource is found whose activation count does not match its current start/stopped state, the resource is created or destroyed and configInit or configExit are called accordingly. In the current implementation, a normal configuration initialization or exit, as described in [6], uses the same code path with all used resources transitioning to a running or stopped state, accordingly. Since starting new resources typically requires object mapping and associated data distribution, it is a costly operation. To improve performance and robustness, Equalizer will flush the rendering pipeline whenever a runtime reconfiguration is detected, before executing the change. This implementation allows the late join or early exit, transparent to application developers.

3.3 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 (HardWare 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.

3.4 Qt Windowing

Qt is a popular window system with many application developers. Unfortunately, it imposes a different threading model for window creation and event handling compared to Equalizer. In Equalizer, each GPU rendering thread is independently responsible for creating its windows, receiving the events and eventually dispatching them to the application process main thread. This design is motivated by the natural threading model of X11 and WGL, and allows simple sequential semantics between OpenGL rendering and event handling. In contrast, Qt requires all windows and QOpenGLContext to be created from the Qt main thread. An existing Qt window or context may be moved to a different thread, and events are signalled from the main thread.

Challenges in threading model, architecture

3.5 Tide Integration

Tiled interactive display environment, parallel pixel streaming, events

3.6 Sequel

Sequal is a simplification layer for Equalizer. It is based on the realization that while fully expressive, the verbatim abstraction layer of nodes, pipes, windows and channels in Equalizer requires significant learning to fully understand and exploit. In reality, a higher abstraction of Application and Renderer is sufficient for most use cases. In Sequel, the application class drives the configuration, and one renderer instance is created for each (pipe) render thread. They also provide the natural place to store and distribute data. Last, but not least, view data provides a convenient way to manage multiple views by storing the camera, model or any other view-specific information.

4 The Collage Network Library

An important part of writing a parallel rendering application is the communication layer between the individual processes. Equalizer relies on the Collage network library for its internal operation. Collage furthermore provides powerful abstractions for writing Equalizer applications, which are introduced in this section.

4.1 Architecture

Collage provides networking functionality of different abstraction layers, gradually providing higher level functionality for the programmer. The main primitives in Collage are:

Connection A stream-oriented point-to-point communication line. The connections transmit raw data reliably between two endpoints for unicast connections, and between a set of endpoints for multicast connections. For unicast, process-local pipes, TCP and Infiniband RDMA are implemented. For multicast, a reliable, UDP-based protocol is discussed in Section 4.3.

DataI/OStream Abstracts the input and output of C++ data types from or to a set of connections by implementing output stream operators. Uses buffering to aggregate data for network transmission. Performs endian swapping during input if the endianness differs between the originating and local node.

Node and LocalNode The abstraction of a process in the cluster. Nodes communicate with each other using connections. A LocalNode listens on various connections and processes requests for a given process. Received data is wrapped in ICommands and dispatched to command handler methods. A Node is a proxy for a remote LocalNode. The Equalizer Client object is a LocalNode.

Object Provides object-oriented, versioned data distribution of C++ objects between nodes. Objects are registered or mapped on a LocalNode.

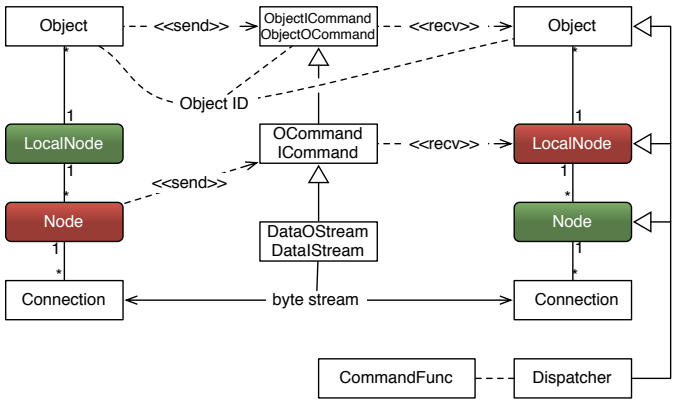

Fig. 2. Communication between two Collage Objects

Figure 2 illustrates this architecture.

4.2 Infiniband RDMA

needed? reverse-engr impl

4.3 Reliable Stream Protocol

RSP is an implementation of a reliable multicast protocol over an unreliable UDP transport. Compared to other protocols, such as pragmatic multicast (PGM), it provides full reliability, that

is, a slow receiver will eventually throttle the sender as in a TCP unicast connection. This property is important to guarantee delivery, e.g., for updates of distributed objects. RSP is optimized for LAN performance. In the following we will outline the RSP protocol and implementation and motivate the design decisions. Any defaults given below are for Linux or OS X, the Windows UDP stack requires different default which can be found in the implementation.

Our RSP implementation uses a separate protocol thread for each RSP connection, which handles all reads and writes on the multicast group. This thread implements the protocol handling and communicates with the application threads through thread-safe queues, as shown in Figure 3. These queues contain datagrams, which are prefixed by a type-specific header. Each connection has a configurable number of buffers, (1024 by default) of a configurable MTU (1470 bytes default) which are either free for use or in-use for reading or writing.

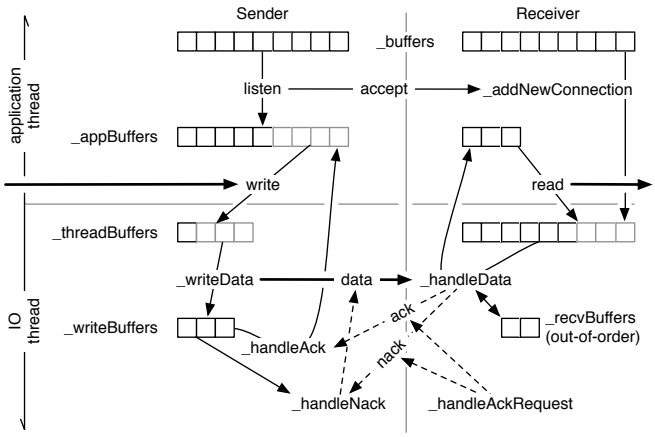

Fig. 3. Reliable Stream Protocol

A client opens a listening multicast RSP connection. Similar to TCP, this listener will create one connected RSP connection for each remote member in the multicast group, that is, on each node n RSP connection instances exist, 1 for the local listener and $n-1$ for all other nodes. Each member has a group-unique 16 bit identifier, which is randomly selected, announced to the group, and if no collisions are notified by any other member, confirmed for this connection. This mechanism allows addressing of connections with a low memory overhead. The datagrams used in this exchange contain no payload, and their head consists of two bytes type (HELLO, HELLO_REPLAY, HELLO_DENY, HELLO_CONFIRM), two bytes protocol version and two bytes connection identifier.

The _appBuffers queue, which is blocking, contains read UDP datagrams on incoming connections or empty buffer for writing on outgoing connections. The _threadBuffers queue is lock-free and used by the protocol thread to store empty buffers on incoming connections and pre-packaged UDP datagrams on outgoing connections. The application thread pushes buffers into this queue, consumed by the protocol thread. A datagram is up to 64 kilobytes (UDP limit, configurable) and contains a header (two bytes type, two bytes size, two bytes source identifier, two bytes sequence number) and its payload. The sequence number is used for positive (ack) and negative (nack) acknowledgments and reordering of packets.

The _writeBuffers std::deque is only used by the protocol thread. It contains already sent, but not yet acknowledged datagrams. Its datagrams are resent upon reception of a nack or recycled into _appBuffers when acknowledged by all receivers. Similarly, the _recvBuffers std::deque contains received out-of-order packets already received by an incoming connection. Upon each reception it is checked, and in-order datagrams are placed into _appBuffers for consumption by the application thread.

Handling a smooth packet flow is critical for performance. RSP actively flow control to advance the stream. Each incoming connection actively acknowledges every n (17 by default) packets fully received. The incoming connections shift this acknowledgment by their connection identifier to avoid bursts of acks. Any missed datagram is actively nack'ed as soon as it is detected. Write connections continuously retransmit packets from nack datagrams, and advance their _writeBuffers upon reception of acks from all readers. Furthermore, the writer will explicitly request an ack or nack when it runs out of empty buffers or finishes its write queue.

Congestion control is necessary to optimize bandwidth usage. While TCP uses the well-known AIMD (additive increase, multiplicative decrease) algorithm, we have chosen a more aggressive congestion control algorithm of additive increase and decrease. This has proven experimentally to be more optimal: UDP is often rate-limited by switches, that is, packets are discarded regularly and not exceptionally. Only slowly backing of the current send rate helps to stay close to this limit. Furthermore, our RSP traffic is limited to the local subnet, making cooparation between multiple data stream less of an issue. Send rate limiting uses a bucket algorithm, where over time the bucket fills with send credits, from which sends are substracted. If there are no available credits, the sender sleeps until sufficient credits are available.

4.4 Distributed, Versioned Objects

Adapting an existing application for parallel rendering requires the synchronization of application data across the processes in the parallel rendering setup. Existing parallel rendering frameworks address this often poorly, at best they rely on MPI to distribute data. Real-world, interactive visualization applications are typically written in C++ and have complex data models and class hierarchies to represent their application state. As outlined in [6], the parallel rendering code in an Equalizer application only needs access to the data needed for rendering, as all application logic is centralized in the application main thread. We have encountered two main approaches to address this distribution: Using a shared filesystem for static data and using data distribution for static and dynamic data.

Distributed objects in Collage provide powerful, object-oriented data distribution for C++ objects. They facilitate the implementation of data distribution in a cluster environment. Distributed objects are created by subclassing from co::Serializable or co::Object. The application programmer implements serialization and deserialization of the distributed data. Distributed objects can be static (immutable) or dynamic. Objects have a universally unique identifier (UUID) to address them cluster-wide. A master-slave model is used to establish mapping and data synchronization across processes. Typically, the application mail loop registers a master instance and communicates the UUID to the render clients, which map their instance to the given identifier. The following object types are available:

Static The object is not versioned nor buffered. The instance data is serialized whenever a new slave instance is mapped. No additional data is stored.

Instance The object is versioned and buffered. The instance and delta data are identical, that is, only instance data is serialized. Previous instance data is saved to be able to map old versions.

Delta The object is versioned and buffered. The delta data is typically smaller than the instance data. The delta data is transmitted to slave instances for synchronization. Previous instance and delta data is saved to be able to map and sync old versions.

Unbuffered The object is versioned and unbuffered. No data is stored, and no previous versions can be mapped. No old versions can be mapped and no additional data is stored.

All distributed objects have to implement getInstanceData and applyInstanceData to serialize and deserialize the object's distributed data. These methods provide an output or input stream as a parameter, which abstracts the data transmission and can be used like a std::stream. The data streams implement efficient buffering and compression, and automatically select the best connection for data transport. Custom data type serializers can be implemented by providing the appropriate serialization functions. No pointers should be directly transmitted through the data streams. For pointers, the corresponding object is typically a distributed object as well, and its identifier and potentially version is transmitted in place of the memory address.

Dynamic objects are versioned, and have to override getChangeType to indicate how they want to have changes to be handled. Upon commit the delta data from the previous version is sent, if available using multicast, to all mapped slave instances. The data is queued on the remote node, and is applied when the application calls sync to synchronize the object to a new version. The sync method might block if a version has not yet been committed or is still in transmission. All versioned objects the following characteristics:

- The master instance of the object generates new versions for all slaves. These versions are continuous. It is possible to commit on slave instances, but special care has to be taken to handle possible conflicts.
- Slave instance versions can only be advanced, that is, sync(version) with a version smaller than the current version will fail.
- Newly mapped slave instances are mapped to the oldest available version by default, or to the version specified when calling mapObject.

Besides the instance data serialization methods used to map an object, versioned objects may implement pack and unpack to serialize or deserialize the changes since the last version.

The Collage Serializable implements one convenient usage pattern for object data distribution. The co::Serializable data distribution is based on the concept of dirty bits, allowing inheritance with data distribution. Dirty bits form a 64-bit mask which marks the parts of the object to be distributed during the next commit. For serialization, the application developer implements serialize or deserialize, which are called with the bit mask specifying which data has to be transmitted or received. During a commit or sync, the current dirty bits are given, whereas during object mapping all dirty bits are passed to the serialization methods.

Blocking commits allow to limit the number of outstanding, queued versions on the slave nodes. A token-based protocol will block the commit on the master instance if to many unsynchronized versions exist.

4.5 Barriers, Queues and Object Maps

Collage implements a few generic distributed objects which are used by Equalizer and applicatons. A barrier is a distributed barrier primitive used for software swap barriers in Equalizer (Section **??**). Its implementation follows a simple master-slave approach, which has shown to by sufficient for this use case. Queues are single producer, multiple consumer FIFOs. To hide network latencies, the consumers use prefetching of queue items into a local queue. They are used for tile and chunk compounds (Section 6.1.2).

The object map facilitates distribution and synchronization of a collection of distributed objects. Master versions can be registered on a central node, e.g., the application node in Equalizer. Consumers, e.g., Equalizer render clients, can selectively map the objects they are interested in. Committing the object map will commit all registered objects and sync their new version to the slaves. Syncing the map on the slaves will synchronize all mapped instances to the new version recorded in the object map. This effective design allows data distribution with minimal application logic. It is used by Sequel (Section 3.6) and other Collage applications.

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 ([8]), 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 Model Unit

This feature allows applications to specify a scaling factor between the model and the real world, to allow exploration of macroscopic or microscopic worlds in virtual reality. The unit is per view, allowing different scale factors within the same configuration. It scales both the specified projection surface as well as the eye position (and therefore separation) to achieve the necessary effect.

5.5 Runtime Stereo Switch

Applications can switch each view between mono and stereo rendering at runtime, and can run both monoscopic and stereoscopic views concurrently (Figure 1 (b)). While this is trivial for active stereo rendering, it does potentially involve the start and stop of resources and processes for passive stereo or stereo-dependent task decompositions, as described in Section 3.2.

6 PERFORMANCE

6.1 New Decomposition Modes

The initial version of Equalizer implemented sort-first (2D), sort-last (DB) and stereo (EYE) decomposition [6]. 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 [13], 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 [7], 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 [11], 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

The monitor allows to reuse the rendering on another channel, typically for monitoring a larger setup on a control workstation. Output frames on the display channels are connected to input frames on a single monitoring channel. The monitor equalizer changes the scaling factor and offset between the output and input, so that the monitor channel has the same, but typically downscaled view, as the originating segments.

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 [10] and [5], we provide the details of the algorithm, and show that using ROI can quadruple the rendering performance, in particular for the costly compositing step in sort-last rendering.

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 [5], 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

Compression for the compositing step in parallel rendering is critical for performance. This not only applies to the well-researched network transfer step, but also for the transfer between GPU and CPU. Equalizer supports a variety of compression algorithms, from very fast RLE encoding, JPEG compression to YUV subsampling on the GPU. These algorithms are implemented as runtime-loaded plugins, allowing easy extension and customization to application-specific compression. In [10], we show this to be a critical step for interactive performance at scale.

6.3.4 Thread Synchronization Modes

Different applications have different degrees on how decoupled and thread-safe the rendering code is from the application logic. For full decoupling all mutable data has to have a copy in each render thread, which is not feasible in large data scenarios. To easily customize the synchronization of all threads on a single process, Equalizer implements three threading modes: Full synchronization, draw synchronization and asynchronous. Note that the execution between nodes is always asynchronous, for up to latency frames.

In full synchronization, all threads always execute the same frame, that is, the render threads are unlocked after Node::frameStart, and the node is blocked for all render threads to finish the frame before executing Node::frameFinish. This allows the render threads to read shared data from all their operations, but provides the slowest performance.

In draw synchronization, the node thread and all render threads are synchronized for all frameDraw operations, that is, Node::frameFinish is executed after the last channel is done drawing. This allows the render threads to read shared data during their draw operation, but not during compositing. Since compositing is typically independent of the rendered data, this is the default mode. This mode allows to overlap compositing with data synchronization on multi-GPU machines.

In asynchronous execution, all threads run asynchronously. Render threads may work on different frames at any given time. This mode is the fastest, and requires the application to have one instance of each mutable object in each render thread. It is required for using time-multiplex compounds in multithreaded rendering.

7 Applications

7.1 Livre

7.2 RTT Deltagen

7.3 RTNeuron

[9]

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

Acknowledgments

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.

PLACE PHOTO HERE

Michael Shell Biography text here.

John Doe Biography text here.

Jane Doe Biography text here.

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. Equalizer Programming and User Guide. Technical report, Eyescale Software GmbH, 2013.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] M. Makhinya, S. Eilemann, and R. Pajarola. Fast Compositing for Cluster-Parallel Rendering. In *Proceedings of the 10th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV, pages 111–120, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[11] 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.

[12] 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.

[13] 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.