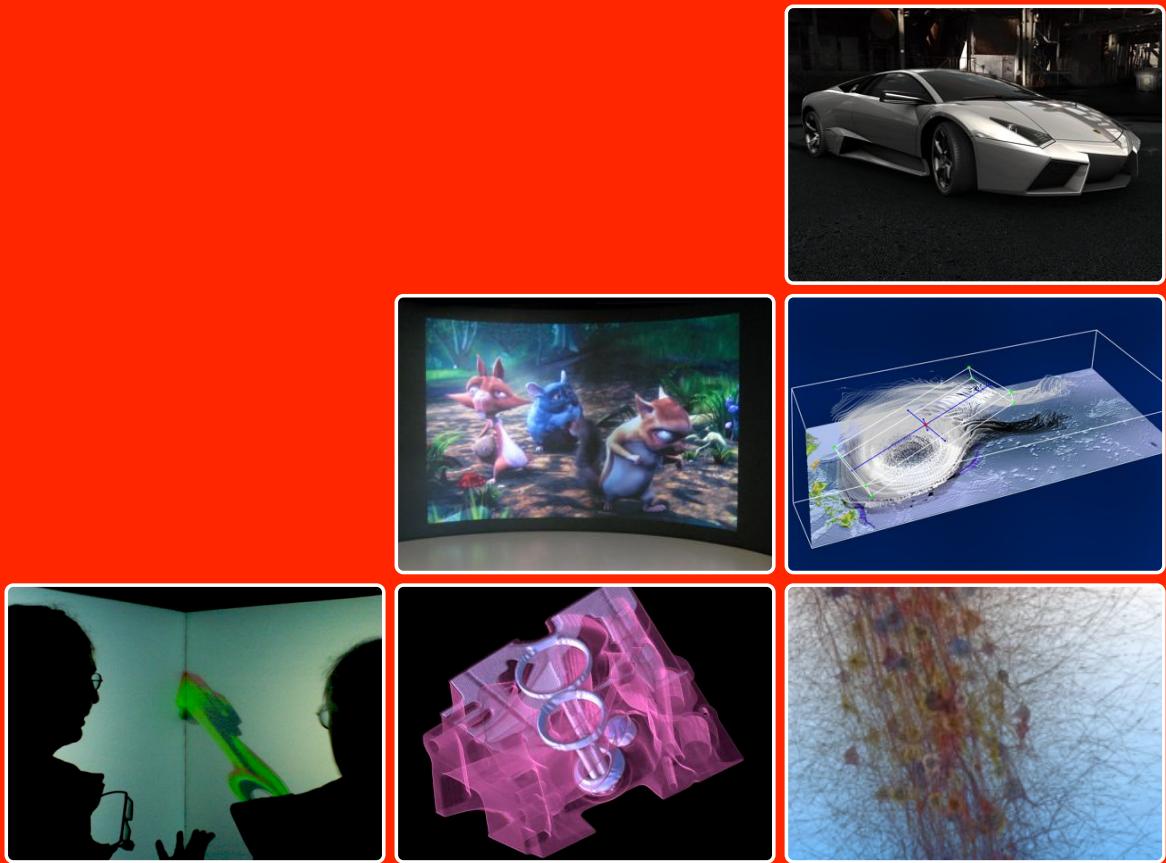


Equalizer

Programming and User Guide

Eyescale Software GmbH



The official reference for developing and deploying
parallel, scalable OpenGL™ applications using the
Equalizer parallel rendering framework

Version 1.12 for Equalizer 1.4

September 7, 2012

Equalizer Programming and User Guide

September 7, 2012

Contributors

Written by Stefan Eilemann.

Contributions by Maxim Makhinya, Jonas Bösch, Christian Marten, Sarah Am sellem, Patrick Bouchaud, Philippe Robert, Robert Hauck and Lucas Peetz Dulley.

Copyright

©2007-2012 Eyescale Software GmbH. All rights reserved. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Eyescale Software GmbH.

Trademarks and Attributions

OpenGL is a registered Trademark, OpenGL Multipipe is a Trademark of Silicon Graphics, Inc. Linux is a registered Trademark of Linus Torvalds. Mac OS is a Trademark of Apple Inc. CAVELib is a registered Trademark of the University of Illinois. The CAVE is a registered Trademark of the Board of Trustees of the University of Illinois at Chicago. Qt is a registered Trademark of Trolltech. TripleHead2Go is a Trademark of Matrox Graphics. PowerWall is a Trademark of Mechdyne Corporation. CUDA is a Trademark of NVIDIA Corporation. All other trademarks and copyrights herein are the property of their respective owners.

Feedback

If you have comments about the content, accuracy or comprehensibility of this Programming and User Guide, please contact eile@eyescale.ch.

Front and Back Page

The images on the front page show the following applications build using Equalizer: RTT DeltaGen¹ [top right], Bino, a stereo-capable, multi-display video player² [middle center], a flow visualization application for climate research³ [middle right], the eqPly polygonal renderer in a three-sided CAVE [bottom left], the eVolve volume renderer⁴ [bottom center], and the RTNeuron⁵ application to visualize cortical circuit simulations [bottom right].

The images on the back page show the following scalable rendering modes: Sub-pixel (FSAA) [top left], DPlex [middle left], Pixel [middle center], 2D [bottom left], DB [bottom center] and stereo [bottom right].

¹Image copyright Realtime Technology AG, 2008

²<http://bino.nongnu.org/>

³Image courtesy of Computer Graphics and Multimedia Systems, University of Siegen

⁴Data set courtesy of General Electric, USA

⁵Image courtesy of Cajal Blue Brain / Blue Brain Project

Contents

| | |
|--|-----------|
| I. User Guide | 1 |
| 1. Introduction | 1 |
| 1.1. Parallel Rendering | 1 |
| 1.2. Installing Equalizer and Running <code>eqPly</code> | 2 |
| 1.3. Equalizer Processes | 2 |
| 2. Scalable Rendering | 3 |
| 2.1. 2D or Sort-First Compounds | 3 |
| 2.2. DB or Sort-Last Compounds | 4 |
| 2.3. Stereo Compounds | 5 |
| 2.4. DPlex Compounds | 5 |
| 2.5. Tile Compounds | 6 |
| 2.6. Pixel Compounds | 6 |
| 2.7. Subpixel Compounds | 7 |
| 2.8. Automatic Runtime Adjustments | 8 |
| 3. Configuring Equalizer Clusters | 8 |
| 3.1. Auto-Configuration | 8 |
| 3.2. Preparation | 9 |
| 3.3. Summary | 10 |
| 3.4. Node | 10 |
| 3.5. Pipe | 11 |
| 3.6. Window | 12 |
| 3.7. Channel | 12 |
| 3.8. Canvases | 12 |
| 3.9. Layouts | 14 |
| 3.10. Observers | 16 |
| 3.11. Compounds | 16 |
| 4. Setting up a Visualization Cluster | 21 |
| 4.1. Name Resolution | 21 |
| 4.2. Server | 21 |
| 4.3. Starting Render Clients | 21 |
| 4.4. Debugging | 22 |
| II. Programming Guide | 23 |
| 5. Hello, World! | 23 |
| 6. Programming Interface | 24 |
| 6.1. Namespaces | 24 |
| 6.2. Task Methods | 26 |
| 6.3. Execution Model and Thread Safety | 26 |
| 7. Sequel: The <code>seqPly</code> Polygonal Renderer | 27 |
| 7.1. main Function | 28 |
| 7.2. Application | 28 |
| 7.3. Renderer | 29 |

List of Figures

| | |
|---|-----------|
| 8. Equalizer: The eqPly Polygonal Renderer | 30 |
| 8.1. main Function | 30 |
| 8.2. Application | 33 |
| 8.3. Distributed Objects | 36 |
| 8.4. Config | 41 |
| 8.5. Node | 46 |
| 8.6. Pipe | 47 |
| 8.7. Window | 49 |
| 8.8. Channel | 51 |
| 9. Advanced Features | 56 |
| 9.1. Event Handling | 56 |
| 9.2. Error Handling | 60 |
| 9.3. Thread Synchronization | 61 |
| 9.4. OpenGL Extension Handling | 65 |
| 9.5. Advanced Window Initialization | 66 |
| 9.6. Stereo and Immersive Rendering | 69 |
| 9.7. Layout API | 72 |
| 9.8. Region of Interest | 75 |
| 9.9. Image Compositing for Scalable Rendering | 77 |
| 9.10. Subpixel Processing | 82 |
| 9.11. Statistics | 85 |
| 9.12. GPU Computing with CUDA | 88 |
| 10. The Collage Network Library | 88 |
| 10.1. Connections | 88 |
| 10.2. Packet Handling | 89 |
| 10.3. Nodes | 89 |
| 10.4. Objects | 91 |
| 10.5. Barrier | 95 |
| A. Command Line Options | 96 |
| B. File Format | 96 |
| B.1. File Format Version | 96 |
| B.2. Global Section | 96 |
| B.3. Server Section | 103 |

List of Figures

| | |
|--|----|
| 1. Parallel Rendering | 1 |
| 2. Equalizer Processes | 2 |
| 3. 2D Compound | 3 |
| 4. Database Compound | 4 |
| 5. Stereo Compound | 5 |
| 6. A DPlex Compound | 5 |
| 7. Tile Compound | 6 |
| 8. Pixel Compound | 6 |
| 9. Pixel Compound Kernel | 7 |
| 10. Example Pixel Kernels for a four-to-one Pixel Compound | 7 |
| 11. Subpixel Compound | 8 |
| 12. GPU discovery for auto-configuration | 9 |
| 13. An Example Configuration | 11 |
| 14. Wall and Projection Parameters | 12 |

List of Figures

| | | |
|-----|--|----|
| 15. | A Canvas using four Segments | 13 |
| 16. | Layout with four Views | 15 |
| 17. | Display Wall using a six-Segment Canvas with a two-View Layout . . | 15 |
| 18. | 2D Load-Balancing | 18 |
| 19. | Cross-Segment Load-Balancing for two Segments using eight GPU's . . | 19 |
| 20. | Cross-Segment Load-Balancing for a CAVE | 19 |
| 21. | Dynamic Frame Resolution | 20 |
| 22. | Monitoring a Projection Wall | 20 |
| 23. | Hello, World! | 23 |
| 24. | Namespaces | 25 |
| 25. | Equalizer client UML map | 26 |
| 26. | Simplified Execution Model | 27 |
| 27. | UML Diagram eqPly and relevant Equalizer Classes | 31 |
| 28. | Synchronous and Asynchronous Execution | 35 |
| 29. | co::Serializable and co::Object | 38 |
| 30. | Scene Data in eqPly | 39 |
| 31. | Scene Graph Distribution | 40 |
| 32. | Config Initialization Sequence | 41 |
| 33. | SystemWindow UML Class Hierarchy | 49 |
| 34. | Destination View of a DB Compound using Demonstrative Coloring | 54 |
| 35. | Main Render Loop | 55 |
| 36. | Event Processing | 57 |
| 37. | UML Class Diagram for Event Handling | 58 |
| 38. | Threads within one Node Process | 62 |
| 39. | Async, draw_sync and local_sync Thread Synchronization Models . . | 63 |
| 40. | Per-Node Frame Synchronization | 64 |
| 41. | Synchronization of Frame Tasks | 65 |
| 42. | Monoscopic (a) and Stereoscopic (b) Frusta | 70 |
| 43. | Tracked (a) and HMD (b) Immersive Frusta | 70 |
| 44. | Fixed(a) and dynamic focus distance relative to origin(b) and ob- server(c) | 71 |
| 45. | Fixed(a), relative to origin(b) and observer(c) focus distance examples | 72 |
| 46. | UML Hierarchy of eqPly::View | 73 |
| 47. | Using two different decompositions during stereo and mono rendering | 74 |
| 48. | Event Flow during a View Update | 75 |
| 49. | ROI for a two-way (a) and four-way DB (b) compound | 76 |
| 50. | ROI for 2D load balancing | 76 |
| 51. | Direct Send Compositing | 78 |
| 52. | Hierarchy of Assembly Classes | 79 |
| 53. | Functional Diagram of the Compositor | 80 |
| 54. | Volume Rendering using 3D Textures | 81 |
| 55. | Result of Figure 56(b) | 82 |
| 56. | Back-to-Front Compositing for Orthogonal and Perspective Frusta . | 82 |
| 57. | Statistics for a two node 2D compound | 86 |
| 58. | Detail of the Statistics from Figure 57. | 86 |
| 59. | UML class diagram of the major Collage classes | 89 |
| 60. | Communication between two Nodes | 90 |
| 61. | Object Distribution using Subclassing, Proxies or Multiple Inheritance | 92 |
| 62. | Slave Commit Communication Sequence | 94 |
| 63. | Communication between two Objects | 95 |

List of Figures

Revision History

| Rev | Date | Changes |
|------|--------------|-----------------------------------|
| 1.0 | Oct 28, 2007 | Initial Version for Equalizer 0.4 |
| 1.2 | Apr 15, 2008 | Revision for Equalizer 0.5 |
| 1.4 | Nov 25, 2008 | Revision for Equalizer 0.6 |
| 1.6 | Aug 07, 2009 | Revision for Equalizer 0.9 |
| 1.8 | Mar 21, 2011 | Revision for Equalizer 1.0 |
| 1.10 | Feb 17, 2012 | Revision for Equalizer 1.2 |
| 1.12 | Jul 20, 2012 | Revision for Equalizer 1.4 |

Part I.

User Guide

1. Introduction

Equalizer is the standard middleware for the development and deployment of parallel OpenGL applications. It enables applications to benefit from multiple graphics cards, processors and computers to improve the rendering performance, visual quality and display size. An Equalizer-based application runs unmodified on any visualization system, from a simple workstation to large scale graphics clusters, multi-GPU workstations and Virtual Reality installations.

This User and Programming Guide introduces parallel rendering concepts, the configuration of Equalizer-based applications and programming using the Equalizer parallel rendering framework.

Equalizer is the most advanced middleware for scalable 3D visualization, providing the broadest set of parallel rendering features available in an open source library to any visualization application. Many commercial and open source applications in a variety of different markets rely on Equalizer for flexibility and scalability.

Equalizer provides the domain-specific parallel rendering expertise and abstracts configuration, threading, synchronization, windowing and event handling. It is a ‘GLUT on steroids’, providing parallel and distributed execution, scalable rendering features, an advanced network library and fully customizable event handling.

If you have any question regarding Equalizer programming, this guide, or other specific problems you encountered, please direct them to the eq-dev mailing list⁶.

1.1. Parallel Rendering

Figure 1 illustrates the basic principle of any parallel rendering application. The typical OpenGL application, for example using GLUT, has an event loop which redraws the scene, updates application data based on received events, and eventually renders a new frame.

A parallel rendering application uses the same basic execution model, extending it by separating the rendering code from the main event loop. The rendering code is then executed in parallel on different resources, depending on the configuration chosen at runtime.

This model is naturally followed by Equalizer, thus making application development

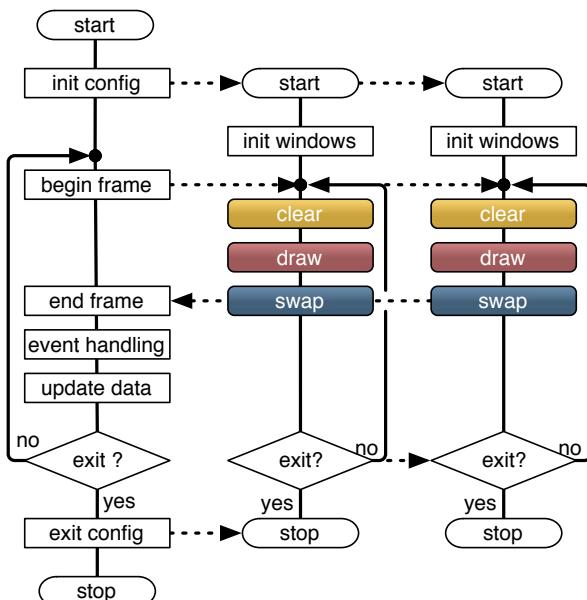


Figure 1: Parallel Rendering
is possible

⁶<http://www.equalizergraphics.com/lists.html>

1. Introduction

1.2. Installing Equalizer and Running eqPly

Equalizer can be installed by downloading a binary installer or source code⁷. After installing Equalizer, please take a look at the Quickstart Guide⁸ to get familiar with the capabilities of Equalizer and the eqPly example.

Equalizer uses CMake to generate platform-specific build files. Compiling Equalizer from source is as simple as running `make` on Linux or Mac OS X, running `make xcode` and building the generated XCode project on Mac OS X, or creating a Equalizer Visual Studio solution using CMake on Windows.

1.3. Equalizer Processes

The Equalizer architecture is based on a client-server model. The client library exposes all functionality discussed in this document to the programmer, and provides communication between the different Equalizer processes.

Collage is a cross-platform C++ library for building heterogenous, distributed applications. Collage provides an abstraction of different network connections, peer-to-peer messaging, discovery and synchronization as well as high-performance, object-oriented, versioned data distribution. Collage is designed for low-overhead multi-threaded execution which allows applications to easily exploit multi-core architectures. Equalizer uses Collage as the cluster backend, e.g., by setting up direct communication between two nodes when needed for image compositing or software swap barriers.

Figure 2 depicts the relationship between the server, application, render client and administrative processes, which are explained below.

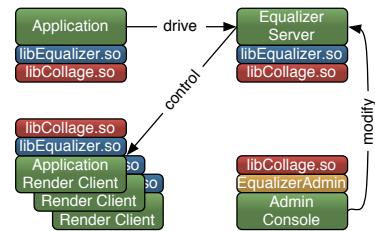


Figure 2: Equalizer Processes

1.3.1. Server

The Equalizer server is responsible for managing one visualization session on a shared memory system or graphics cluster. Based on its configuration and controlling input from the application, it computes the active resources, updates the configuration and generates tasks for all processes. Furthermore it controls and launches the application's rendering client processes. The Equalizer server is the entity in charge of the configuration, and all other processes receive their configuration from the server.

1.3.2. Application

The application connects to an Equalizer server and receives a configuration. Furthermore, the application also provides its render client, which will be controlled by the server. The application and render client may use the same executable. The application has a main loop, which reacts on events, updates its data and controls the rendering.

1.3.3. Render Clients

The render client implements the rendering part of an application. Its execution is passive, it has no main loop and is completely driven by Equalizer server. It executes

⁷<http://www.equalizergraphics.com/downloads.html>

⁸<http://www.equalizergraphics.com/documents/EqualizerGuide.html>

2. Scalable Rendering

the rendering tasks received from the server by the calling the appropriate task methods (see Section 6.2) in the correct thread and context. The application either implements the task methods with application-specific code or uses the default methods provided by Equalizer.

The application can also be a rendering client, in which case it can also contribute to the rendering. If it does not implement any render client code, it is reduced to be the application’s ‘master’ process without any OpenGL windows and 3D rendering.

The rendering client can be the same executable as the application, as it is the case with all provided examples. When it is started as a render client, the Equalizer initialization routine does not return and takes over the control by calling the render client task methods. Complex applications usually implement a separate, lightweight rendering client.

1.3.4. Administration Programs

Equalizer 1.0 introduced the admin library, which can be used to modify a running Equalizer server. The admin library is still in development, but already allows limited modifications such as adding new windows and changing layouts. The admin library may be used to create standalone administration tools or from within the application code. In any case, it has an independent view of the server’s configuration. Documentation for admin library is not yet part of this Programming and User Guide.

2. Scalable Rendering

Scalable rendering is a subset of parallel rendering, where more multiple resources are used to update a view.

Real-time visualization is an inherently parallel problem. Different applications have different rendering algorithms, which require different scalable rendering modes to address the bottlenecks correctly. Equalizer supports all important algorithms as listed below, and will continue to add new ones over time to meet application requirements.

This section gives an introduction to scalable rendering, providing some background for end users and application developers. The scalability modes offered by Equalizer are discussed, along with their advantages and disadvantages.

Choosing the right mode for the application profile is critical for performance. Equalizer uses the concept of compounds to describe the task decomposition and result recomposition. It allows the combination of the different compound modes in any possible way, which allows to address different bottlenecks in a flexible way.

2.1. 2D or Sort-First Compounds

2D decomposes the rendering in screen-space, that is, each contributing rendering unit processes a tile of the final view. The recomposition simply assembles the tiles side-by-side on the destination view. This mode is also known as sort-first or SFR.

The advantage of this mode is a low, constant IO overhead

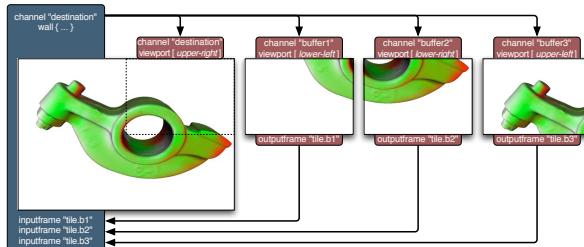


Figure 3: 2D Compound

2. Scalable Rendering

for the pixel transfers, since only color information has to be transmitted. The upper limit is the amount of pixel data for the destination view.

Its disadvantage is that it relies on view frustum culling to reduce the amount of data submitted for rendering. Depending on the application data structure, the overlap of some primitives between individual tiles limits the scalability of this mode, typically to around eight graphics cards. Each node has to potentially hold the full database for rendering.

2D decompositions can be used by all types of applications, but should be combined with DB compounds to reduce the data per node, if possible. In most cases, a `load_equalizer` should be used to automatically adjust the tiling each frame, based on the current rendering load.

2D compounds in Equalizer are configured using the `viewport` parameter, using the values [`x y width height`] in normalized coordinates. The viewport defines the area of the parent (destination) channel to be rendered for each child. Each child compound uses an output frame, which is connected to an input frame on the destination channel. The destination channel can also be used as a source channel, in which case it renders in place and no output frame is needed.

2.2. DB or Sort-Last Compounds

DB, as shown in Figure 4⁹, decomposes the rendered database so that all rendering units process a part of the scene in parallel. This mode is also known as sort-last, and is very similar to the data decomposition approach used by HPC applications.

Volume rendering applications use an ordered alpha-based blending to composite the result image. The depth buffer information is used to composite the individual images correctly for polygonal data.

This mode provides very good scalability, since each rendering unit processes only a part of the database. This allows to lower the requirements on all parts of the rendering pipeline: main memory usage, IO bandwidth, GPU memory usage, vertex processing and fill rate.

Unfortunately, the database recomposition has linear increasing IO requirements for the pixel transfer. Parallel compositing algorithms, such as direct-send, address this problem by keeping the per-node IO constant (see Figure 51).

The application has to partition the database so that the rendering units render only part of the database. Some OpenGL features do not work correctly (anti-aliasing) or need special attention (transparency, shadows).

The best use of database compounds is to divide the data to a manageable size, and then to use other decomposition modes to achieve further scalability. Volume rendering is one of the applications which can profit from database compounds.

DB compounds in Equalizer are configured using the `range` parameter, using the values [`begin end`] in normalized coordinates. The range defines the start and end

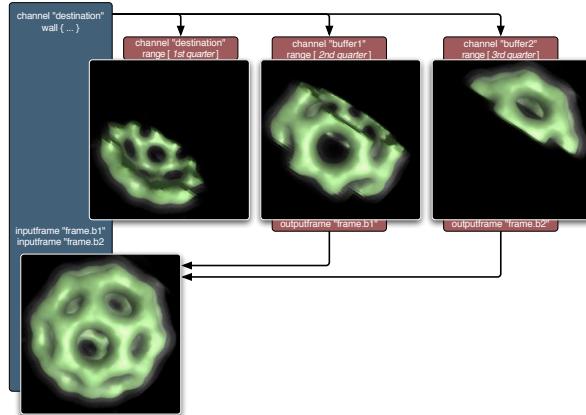


Figure 4: Database Compound

⁹3D model courtesy of AVS, USA.

2. Scalable Rendering

point of the application's database to be rendered. The value has to be interpreted by the application's rendering code accordingly. Each child compound uses an output frame, which is connected to an input frame on the destination channel. For more than two contributing channels, it is recommended to configure streaming or parallel direct send compositing, as described in Section 9.9.2.

2.3. Stereo Compounds

Stereo compounds, as shown in Figure 5¹⁰, assign each eye pass to individual rendering units. The resulting images are copied to the appropriate stereo buffer. This mode supports a variety of stereo modes, including active (quad-buffered) stereo, anaglyphic stereo and auto-stereo displays with multiple eye passes.

Due to the frame consistency between the eye views, this mode scales very well. The IO requirements for pixel transfer are small and constant. The number of rendering resources used by stereo compounds is limited by the number of eye passes, typically two.

Stereo compounds are used by all applications when rendering in stereo, and is often combined with other modes.

Eye compounds in Equalizer are configured using the `eye` parameter, limiting the child to render the [LEFT] or [RIGHT] eye. Each child compound uses an output frame, which is connected to an input frame on the destination channel. The destination channel can also be used to render an eye pass, in which case it renders in the correct stereo buffer and no output frame is needed.

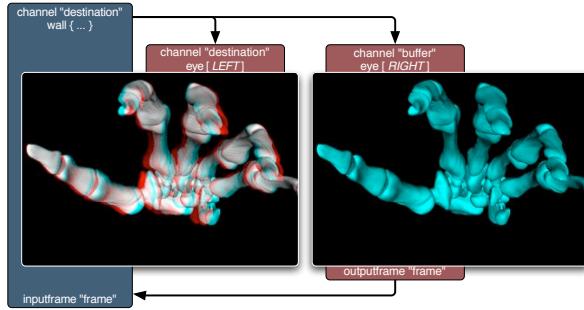


Figure 5: Stereo Compound

2.4. DPlex Compounds

DPlex compounds assign full, alternating frames to individual rendering units. The resulting images are copied to the destination channel, and Equalizer load-balancing is used to ensure a steady framerate on the destination window. This mode is also known as time-multiplex or AFR.

Due to the frame consistency between consecutive frames, this mode scales very well. The IO requirements for pixel transfer are small and constant.

DPlex requires a latency of at least n frames. This increased latency might be disturbing to the user, but it

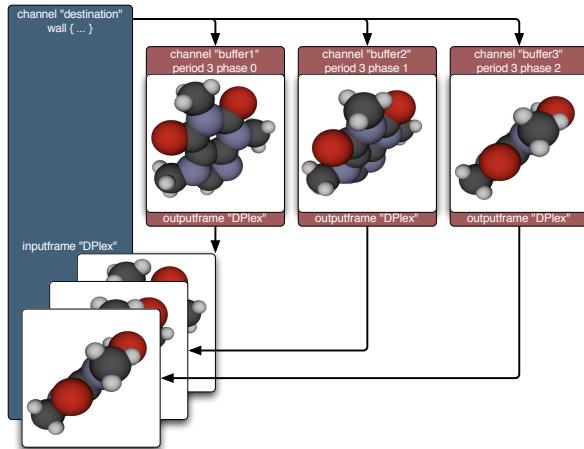


Figure 6: A DPlex Compound

¹⁰3D model courtesy of Stereolithography Archive at Clemson University.

2. Scalable Rendering

is often compensated by the higher frame rate. The frame rate typically increases linearly with the number of source channels, and therefore linearly with the latency.

DPlex compounds in Equalizer are configured using the `period` and `phase` parameter, limiting each child to render a subset of the frames. Each child compound uses an output frame, which is connected to an input frame on the destination channel. The destination channel uses a `framerate_equalizer` to smoothen the framerate.

2.5. Tile Compounds

Tile compounds are similar to 2D compounds. They decompose the rendering in screen-space, where each rendering unit pulls and processes regular tiles of the final view. Tile compounds are ideal for purely fill-limited applications such as volume rendering and raytracing.

Tile compounds have a low, constant IO overhead for the image transfers and can provide good scalability when used with fill-limited applications. Tile decomposition works transparently for all Equalizer-based applications.

Contrary to all other compounds, the work distribution is not decided before-hand using a push model, but each resource pulls work from a central work queue, until all tiles have been processed. Therefore the rendering is naturally load-balanced since all sources pull data on an as-needed basis. The queue implementation uses prefetching to hide the communication overhead of the central, distributed tile queue.

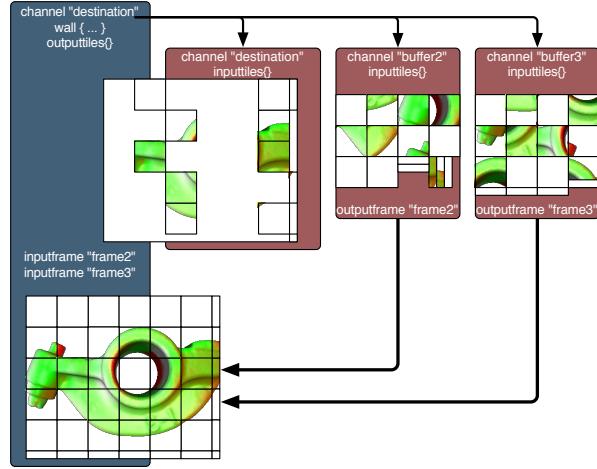


Figure 7: Tile Compound

2.6. Pixel Compounds

Pixel compounds are similar to 2D compounds. While 2D compounds decompose the screen space into contiguous regions, pixel compounds assign one pixel of a regular kernel to each resource. The frusta of the source rendering units are modified so that each unit renders an evenly distributed subset of pixels, as shown in Figure 8¹¹

As 2D compounds, pixel compounds have low, constant IO requirements for the

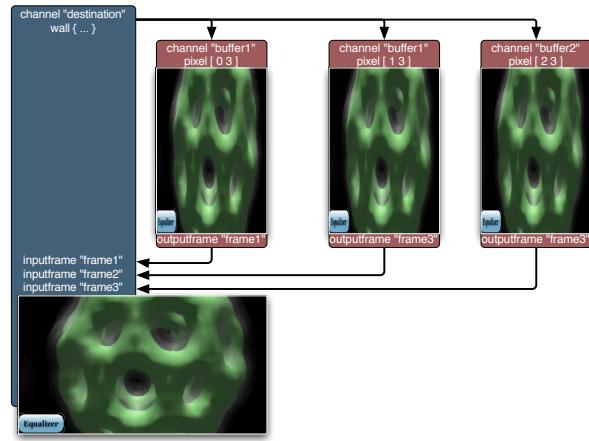


Figure 8: Pixel Compound

¹¹3D model courtesy of AVS, USA.

2. Scalable Rendering

pixel transfers during recomposition. They are naturally load-balanced for fill-limited operations, but do not scale geometry processing at all.

OpenGL functionality influenced by the raster position will not work correctly with pixel compounds, or needs at least special attention. Among them are: lines, points, sprites, `glDrawPixels`, `glBitmap`, `glPolygonStipple`. The application can query the current pixel parameters at runtime to adjust the rendering accordingly.

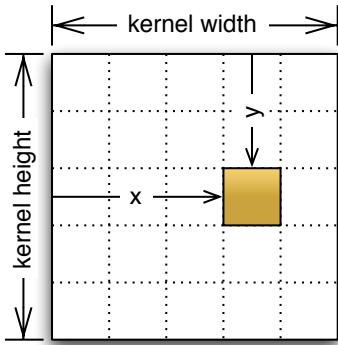


Figure 9: Pixel Compound Kernel

channel within the kernel. They have to be smaller than the size of the kernel. Figure 9 illustrates these parameters, and Figure 10 shows some example kernels for a four-to-one pixel decomposition.

The destination channel can also be used as a source channel. Contrary to the other compound modes, it also has to use an output and corresponding input frame. During rendering, the frustum is ‘squeezed’ to configure the pixel decomposition. The destination channel can therefore not be rendered in place, like with the other compound modes.

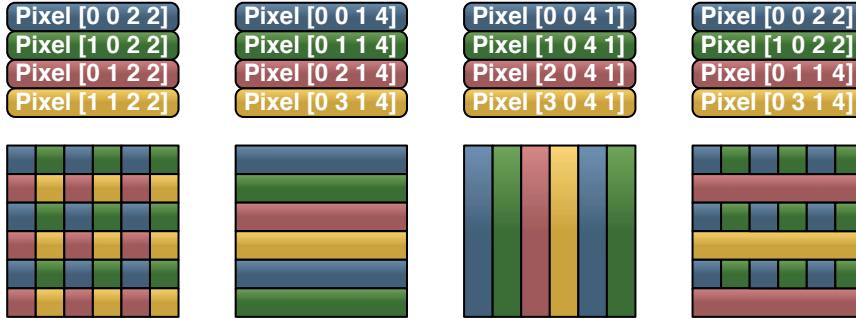


Figure 10: Example Pixel Kernels for a four-to-one Pixel Compound

2.7. Subpixel Compounds

Subpixel decomposes the rendering of multiple samples for one pixel, e.g., for anti-aliasing or depth-of-field rendering. The default implementation provides transparent, scalable software idle-antialiasing when configuring a subpixel compound.

Applications can use subpixel compounds to accelerate depth-of-field effects and software anti-aliasing, with potentially a different number of idle or non-idle samples per pixel.

3. Configuring Equalizer Clusters

As for the DB compound, the subpixel recomposition has linear increasing IO requirements for the pixel transfer, with the difference that only color information has to be transmitted.

Subpixel compounds are configured using the `subpixel` parameter, using the values `[index size]`. The `index` parameter corresponds to the current resource index and the `size` is the total number of resources used.

The `index` has to be smaller than the `size`. Figure 11 illustrates a three-way subpixel decomposition.

The destination channel can also be used as a source channel. As for the pixel compound, it has to use an output and corresponding input frame.

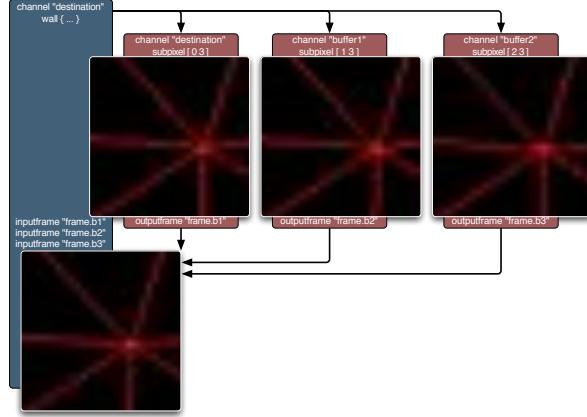


Figure 11: Subpixel Compound

2.8. Automatic Runtime Adjustments

Some scalable rendering parameters are updated at runtime to provide optimal performance or influence other scalable rendering features. These adjustments are often referred to as load-balancing, but are called **equalizers**, since their functionality is not limited to load-balancing in Equalizer.

The server provides a number of equalizers, which automatically update certain parameters of compounds based on runtime information. They balance the load of 2D and DB compounds, optimally distribute render resources for multi-display projection systems, adjust the resolution to provide a constant framerate or zoom images to allow monitoring of another view.

Equalizers are described in more detail in Section 3.11.8.

3. Configuring Equalizer Clusters

3.1. Auto-Configuration

3.1.1. GPU Service Discovery

Equalizer uses the GPU-SD library for discovering local and remote GPUs. Based on this information, a typical configuration is dynamically compiled during the initialization of the application. The auto-configuration uses the same format as the static configuration files. It can be used to create template configurations by running a test application or the `eqServer` binary and using the generated configuration, saved as `<session>.auto.eqc`.

Please note that both GPU-SD and the GPU-SD ZeroConf module are optional and requires GPU-SD and a ZeroConf implementation when Equalizer is compiled.

GPU-SD contains three components: a core library, local and remote discovery modules as well as a ZeroConf daemon. The daemon uses the appropriate module to query local GPUs and announces them using ZeroConf on the local network.

3. Configuring Equalizer Clusters

Equalizer uses the library with the dns_sd discovery module to gather the information announced by all daemons on the local network, as well as the local discovery modules for standalone auto-configuration. The default configuration is using all locally discovered GPUs.

Figure 12 shows how GPU-SD is used to discover four remote GPUs on two different machines to use them to render a single view on a laptop.

3.1.2. Usage

On each node contributing to the configuration, install and start the GPU-SD daemon. If multiple, disjoint configurations are used on the same network, provide the session name as a parameter when starting the daemon. Verify that all GPUs are visible on the application host using the `gpu_sd_list` tool. When starting the application, use the command-line parameter `-eq-config`:

- The default value is `local` which uses all local GPUs queried using the `cgl`, `glx` or `wgl` GPU-SD module.
- `-eq-config sessionname` uses the `dns_sd` ZeroConf module of GPU-SD to query all GPUs in the subnet for the given session. The default session of the GPU-SD daemon is `default`. The hostname provided by ZeroConf is used to connect the nodes.
- `-eq-config filename.eqc` loads a static configuration from the given ascii file. The following sections describe how to write configuration files.

The auto-configuration creates one display window on the local machine, and one off-screen channel for each GPU. The display window has one full-window channel used as an output channel for a single segment. It combines all GPUs into a scalability config with different layouts for each of the following scalability modes:

2D A dynamically load-balanced sort-first configuration using all GPUs as sources.

Simple A no-scalability configuration using only the display GPU for rendering.

Static DB A static sort-last configuration distributing the range evenly across all GPUs.

Dynamic DB A dynamically load-balanced sort-last configuration using all GPUs as sources.

The ZeroConf hostname is used to configure the node, that is, the launch command has to be able to resolve this hostname for starting the render client processes. Given a properly set up ZeroConf installation, the default launch command `ssh` should be able to start the render client correctly.

3.2. Preparation

Before writing a configuration, it is useful to assemble the following information:

- A list of all **computers** in your rendering cluster, including the **IP addresses** of all network interfaces to be used.

3. Configuring Equalizer Clusters

- The number of **graphics cards** in each computer.
- The **physical dimensions** of the display system, if applicable. These are typically the bottom-left, bottom-right and top-left corner points of each display surface in meters.
- The relative coordinates of all the **segments** belonging to each display surface, and the graphics card **output** used for each segment. For homogenous setups, it is often enough to know the number of rows and columns on each surface, as well as the overlap or underlap percentage, if applicable.
- The number of desired **application windows**. Application windows are typically destination windows for scalable rendering or 'control' windows paired with a view on a display system.
- **Characteristics** of the application, e.g., supported scalability modes and features.

3.3. Summary

Equalizer applications are configured at runtime by the Equalizer server. The server loads its configuration from a text file, which is a one-to-one representation of the configuration data structures at runtime.

For an extensive documentation of the file format please refer to Appendix B. This section gives an introduction on how to write configuration files.

A configuration consists of the declaration of the rendering resources, the description of the physical layout of the projection system, logical layouts on the projection canvases and an optional decomposition description using the aforementioned resources.

The rendering resources are represented in a hierarchical tree structure which corresponds to the physical and logical resources found in a 3D rendering environment: nodes (computers), pipes (graphics cards), windows, channels.

Physical layouts of display systems are configured using canvases with segments, which represent 2D rendering areas composed of multiple displays or projectors. Logical layouts are applied to canvases and define views on a canvas.

Scalable resource usage is configured using a compound tree, which is a hierarchical representation of the rendering decomposition and recomposition across the resources.

Figure 13 shows an example configuration for a four-side CAVE, running on two machines (nodes) using three graphics cards (pipes) with one window each to render to the four output channels connected to the projectors for each of the walls.

For testing and development purposes it is possible to use multiple instances for one resource, e.g. to run multiple render client nodes on one computer. For optimal performance during deployment, one node and pipe should be used for each computer and graphics card, respectively.

3.4. Node

For each machine in your cluster, create one **node**. Create one **appNode** for your application process. List all nodes, even if you are not planning to use them at first. Equalizer will only instantiate and access used nodes, that is, nodes which are referenced by an active compound.

In each node, list all **connections** through which this node is reachable. Typically a node uses only one connection, but it is possible to configure multiple connections if the machine and cluster is set up to use multiple, independent network interfaces.

3. Configuring Equalizer Clusters

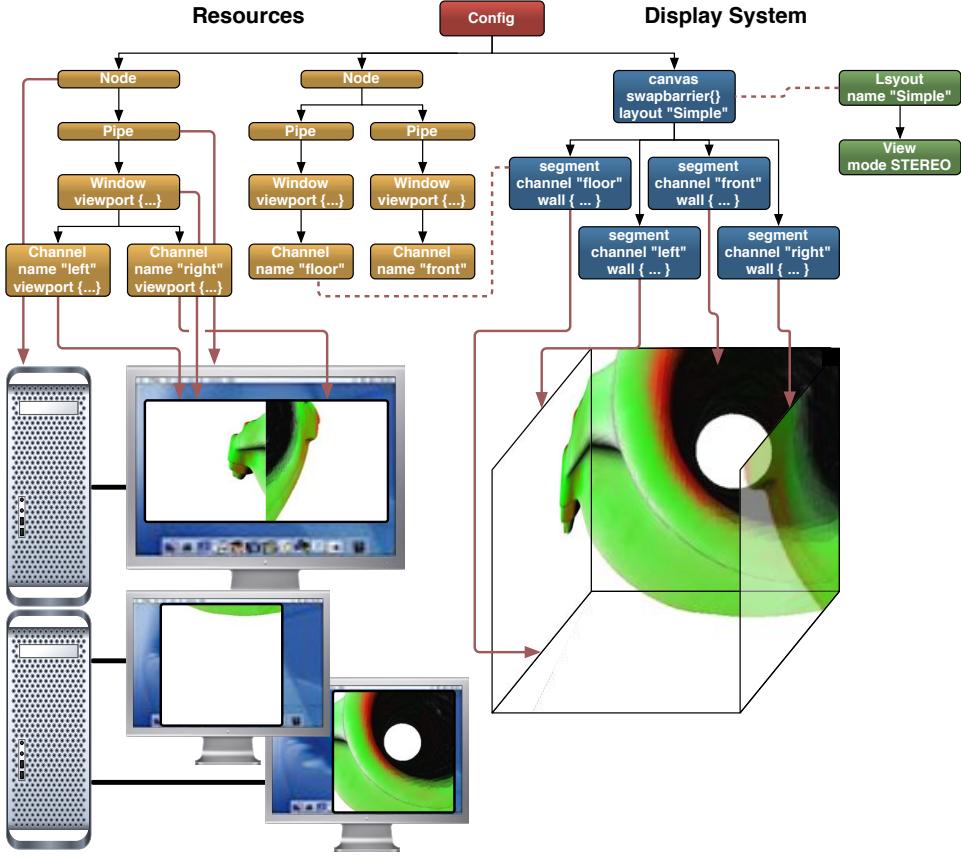


Figure 13: An Example Configuration

Make sure the configured hostname is reachable from all nodes. An IP address may be used as the hostname.

For cluster configurations with multiple nodes, configure at least one connection for the server. All render clients connect back to the server, for which this connection is needed.

The `eq::Node` class is the representation of a single computer in a cluster. One operating system process of the render client will be used for each node. Each configuration might also use an application node, in which case the application process is also used for rendering. All node-specific task methods are executed from the main application thread.

3.5. Pipe

For each node, create one pipe for each graphics card in the machine. Set the `device` number to the correct index. On operating systems using X11, e.g., Linux, also set the port number if your X-Server is running on a nonstandard port.

The `eq::Pipe` class is the abstraction of a graphics card (GPU), and uses one independent operating system thread for rendering. Non-threaded pipes are supported for integrating with thread-unsafe libraries, but have various performance caveats. They should only be used if using a different, synchronized rendering thread is not an option.

3. Configuring Equalizer Clusters

All pipe, window and channel task methods are executed from the pipe thread, or in the case of non-threaded pipes from the main application thread¹².

3.6. Window

Configure one **window** for each desired application window on the appNode. Configure one full-screen window for each display segment. Configure one off-screen window, typically a pbuffer, for each graphics card used as a source for scalable rendering. Provide a useful name to each on-screen window if you want to easily identify it at runtime.

Sometimes display segments cover only a part of the graphics card output. In this case it is advised to configure a non-fullscreen window without window decorations, using the correct window viewport.

The `eq::Window` class encapsulates a drawable and an OpenGL context. The drawable can be an on-screen window or an off-screen PBuffer or FrameBuffer Object (FBO).

3.7. Channel

Configure one **channel** for each desired rendering area in each window. Typically one full-screen channel per window is used. Name the channel using a unique, easily identifiable name, e.g., 'source-1', 'control-2' or 'segment-2_3'.

Multiple channels in application windows may be used to view the model from different viewports.

Sometimes, a single window is split across multiple projectors, e.g., by using an external splitter such as the Matrox TripleHead2Go. In this case configure one channel for each segment, using the channel's **viewport** to configure its position relative to the window.

The `eq::Channel` class is the abstraction of an OpenGL viewport within its parent window. It is the entity executing the actual rendering. The channel's viewport is overwritten when it is rendering for another channel during scalable rendering.

3.8. Canvases

If you are writing a configuration for workstation usage you can skip the following sections and restart with Section 3.11.

Configure one **canvas** for each display surface. For planar surfaces, e.g., a display wall, configure a frustum. For non-planar surfaces, the frustum will be configured on each display segment.

The frustum can be specified as a wall or projection description. Take care to choose your reference system for describing the frustum to be the same system as used by the head-tracking matrix calculated by the application.

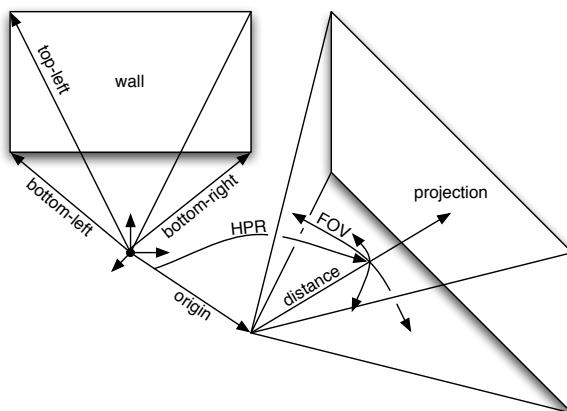


Figure 14: Wall and Projection Parameters

¹²see <http://www.equalizergraphics.com/documents/design/nonthreaded.html>

3. Configuring Equalizer Clusters

A wall is completely defined by the bottom-left, bottom-right and top-left coordinates relative to the origin.

A projection is defined by the position and head-pitch-roll orientation of the projector, as well as the horizontal and vertical field-of-view and distance of the projection wall.

Figure 14 illustrates the wall and projection frustum parameters.

A canvas represents one physical projection surface, e.g., a PowerWall, a curved screen or an immersive installation. One configuration might drive multiple canvases, for example an immersive installation and an operator station.

A canvas consists of one or more segments. A planar canvas typically has a frustum description (see Section 3.11.3), which is inherited by the segments. Non-planar frusta are configured using the segment frusta. These frusta typically describe a physically correct display setup for Virtual Reality installations.

A canvas has one or more layouts. One of the layouts is the active layout, that is, this set of views is currently used for rendering. It is possible to specify OFF as a layout, which deactivates the canvas. It is possible to use the same layout on different canvases.

A canvas may have a swap barrier, which is used as the default swap barrier by all its segments to synchronize the video output.

Canvases provide a convenient way to configure projection surfaces. A canvas uses layouts, which describe logical views. Typically, each desktop window uses one canvas, one segment, one layout and one view.

3.8.1. Segments

Configure one segment for each display or projector of each canvas. Configure the viewport of the segment to match the area covered by the segment on the physical canvas. Set the output channel to the resource driving the described projector.

For non-planar displays, configure the frustum as described in Section 3.8. For passive stereo installations, configure one segment per eye pass, where the segment for the left and right eye have the same viewport. Set the eyes displayed by the segment, i.e., left or right and potentially cyclop.

To synchronize the video output, configure either a canvas swap barrier or a swap barrier on each segment to be synchronized.

When using software swap synchronization, swap-lock all segments using a swap barrier. All windows with a swap barrier of the same name synchronize their swapbuffers. Software swap synchronization uses a distributed barrier, and works on all hardware.

When using hardware swap synchronization, use swap barriers for all segment to be synchronized, setting NV_group and NV_barrier appropriately. The swap barrier name is ignored in this case. All windows of the same group on a single node synchronize their swap buffer. All groups of the same barrier synchronize their swap buffer across nodes. Please note that the driver typically limits the number of groups and barriers to one, and that multiple windows per swap group are not supported by all drivers. Hardware swap barriers require support from the OpenGL driver, and has been tested on nVidia Quadro GPUs with the G-Sync option. Please refer to your OpenGL driver documentation for details.

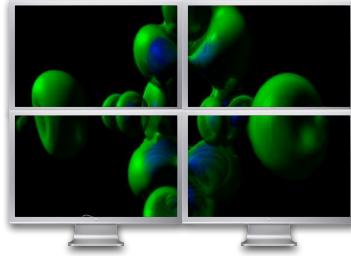


Figure 15: A Canvas using four Segments

3. Configuring Equalizer Clusters

A segment represents one output channel of the canvas, e.g., a projector or an LCD. A segment has an output channel, which references the channel to which the display device is connected.

A segment covers a part of its parent canvas, which is configured using the segment viewport. The viewport is in normalized coordinates relative to the canvas. Segments might overlap (edge-blended projectors) or have gaps between each other (display walls, Figure 15¹³). The viewport is used to configure the segment's default frustum from the canvas frustum description, and to place layout views correctly.

3.8.2. Swap and Frame Synchronization

Canvases and segments may have a software or hardware swap barrier to synchronize the buffer swap of multiple channels. The swap barriers are inherited from the canvas to the segment ultimately to all compounds using a destination channel of the segment.

A software swap barrier is configured by giving it a name. Windows with a swap barrier of the same name synchronize with each other before executing the swap buffer task. Before entering the barrier, `Window::finish` is called to ensure that all OpenGL commands have been executed. Swap barrier names are only valid within the compound tree, that is, a compound from one compound tree cannot be synchronized with a compound from another compound tree.

A hardware swap barrier uses a hardware component to synchronize the buffer swap of multiple windows. It guarantees that the swap happens at the same vertical retrace of all corresponding video outputs. It is configured by setting the `NV_group` and `NV_barrier` parameters. These parameters follow the `NV_swap_group` extension, which synchronizes all windows bound to the same group on a single machine, and all groups bound to the same barrier across systems.

Display synchronization uses different algorithms. Framelock synchronizes each vertical retrace of multiple graphic outputs using a hardware component. This is configured in the driver, independently of the application. Genlock synchronizes each horizontal and vertical retrace of multiple graphic outputs, but is not commonly used anymore for 3D graphics. Swap lock synchronizes the front and back buffer swap of multiple windows, either using a software solution based on network communication or a hardware solution based on a physical cable. It is independent of, but often used in conjunction with framelock.

Framelock is used to synchronize the vertical retrace in multi-display active stereo installations, e.g., for edge-blended projection systems or immersive installations. It is combined with a software or hardware swap barrier. Software barriers in this case cannot guarantee that the buffer swap of all outputs always happens with the same retrace. The time window for an output to miss the retrace for the buffer swap is however very small, and the output will simply swap on the next retrace, typically in 16 milliseconds.

Display walls made out of LCD's, monoscopic or passive stereo projection systems often only use a software swap barrier and no framelock. The display bezels make it very hard to notice the missing synchronization.

3.9. Layouts

Configure one layout for each configuration of logical views. Name the layout using a unique name. Often only one layout with a one view is used for all canvases.

Enable the layout on each desired canvas by adding it to the canvas. Since canvases reference layouts by name or index, layouts have to be configured before their respective canvases in the configuration file.

¹³Dataset courtesy of VolVis distribution of SUNY Stony Brook, NY, USA.

3. Configuring Equalizer Clusters

A layout is the grouping of logical views. It is used by one or more canvases. For all given layout/canvas combinations, Equalizer creates destination channels when the configuration file is loaded. These destination channels can be referenced by compounds to configure scalable rendering.

Layouts can be switched at runtime by the application. Switching a layout will activate different destination channels for rendering.

3.9.1. Views

Configure one view for each logical view in each layout. Set the `viewport` to position the view. Set the mode to stereo if appropriate.

A view is a logical view of the application data, in the sense used by the Model-View-Controller pattern. It can be a scene, viewing mode, viewing position, or any other representation of the application's data.

A view has a fractional viewport relative to its layout. A layout is often fully covered by its views, but this is not a requirement.

Each view can have a frustum description. The view's frustum overrides frusta specified at the canvas or segment level. This is typically used for non-physically correct rendering, e.g., to compare two models side-by-side on a canvas. If the view does not specify a frustum, it will use the sub-frustum resulting from the covered area on the canvas.

A view might have an observer, in which case its frustum is tracked by this observer.

Figure 16 shows an example layout using four views on a single segment. Figure 3.9.1 shows a real-world setup of a single canvas with six segments using underlap, with a two-view layout activated. This configuration generates eight destination channels.

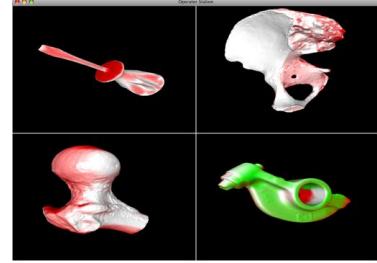


Figure 16: Layout with four Views



Figure 17: Display Wall using a six-Segment Canvas with a two-View Layout

3.10. Observers

Unless you have multiple tracked persons, or want to disable tracking on certain views, you can skip this section.

Configure one **observer** for each tracked person in the configuration. Most configurations have at most one observer. Assign the observer to all views which belong to this observer. Since the observer is referenced by its name or index, it has to be specified before the layout in the configuration file.

Views with no observer are not tracked. The config file loader will create one default observer and assign it to all views if the configuration has no observer.

An observer represents an actor looking at multiple views. It has a head matrix, defining its position and orientation within the world, an eye separation and focus distance parameters. Typically, a configuration has one observer. Configurations with multiple observers are used if multiple, head-tracked users are in the same configuration session, e.g., a non-tracked control host with two tracked head-mounted displays.

3.11. Compounds

Compound trees are used to describe how multiple rendering resources are combined to produce the desired output, especially how multiple GPUs are aggregated to increase the performance.

It is advised to study and understand the basic configuration files shipped with the Equalizer configuration, before attempting to write compound configurations. The auto-configuration code and command line program `configTool`, shipped with the Equalizer distribution, creates some standard configurations automatically. These are typically used as templates for custom configuration files.

For configurations using canvases and layouts without scalability, the configuration file loader will create the appropriate compounds. It is typically not necessary to write compounds for this use case.

The following subsection outlines the basic approach to writing compounds. The remaining subsections provide an in-depth explanation of the compound structure to give the necessary background for compound configuration.

3.11.1. Writing Compounds

The following steps are typically taken when writing compound configurations:

Root compound Define an empty top-level compound when synchronizing multiple destination views. Multiple destination views are used for multi-display systems, e.g., a PowerWall or CAVE. All windows used for one display surface should be swap-locked (see below) to provide a seamless image. A single destination view is typically used for providing scalability to a single workstation window.

Destination compound(s) Define one compound for each destination channel, either as a child of the empty group, or as a top-level compound. Set the destination channel by using the canvas, segment, layout and view name or index. The compound frustum will be calculated automatically based on the segment or view frustum. Note that one segment may create multiple view/segment channels, one for each view intersection of each layout used on the canvas. Only the compounds belonging to the active layout of a canvas are active at runtime.

3. Configuring Equalizer Clusters

Scalability If desired, define scalability for each of your destination compounds.

Add one compound using a source channel for each contributor to the rendering. The destination channel may also be used as a source.

Decomposition On each child compound, limit the rendering task of that child by setting the `viewport`, `range`, `period` and `phase`, `pixel`, `subpixel`, `eye` or `zoom` as desired.

Runtime Adjustments A `load_equalizer` may be used on the destination compounds to set the `viewport` or `range` of all children each frame, based on the current load. A `view_equalizer` may be used on the root compound to assign resources to all destination compounds, which have to use `load_equalizers`. A `framerate_equalizer` should be used to smoothen the framerate of DPlex compounds. A `DFR_equalizer` may be used to set the zoom of a compound to achieve a constant framerate. One compound may have multiple equalizers, e.g., a `load_equalizer` and a `DFR_equalizer` for a 2D compound with a constant framerate.

Recomposition For each source compound, define an `output_frame` to read back the result. Use this output frame as an `input_frame` on the destination compound. The frames are connected with each other by their name, which has to be unique within the root compound tree. For parallel compositing, describe your algorithm by defining multiple input and output frames across all source compounds.

3.11.2. Compound Channels

Each compound has a channel, which is used by the compound to execute the rendering tasks. One channel might be used by multiple compounds. Compounds are only active if their corresponding destination channel is active, that is, if the parent layout of the view which created the destination channel is active on at least one canvas.

Unused channels, windows, pipes and nodes are not instantiated during initialization. Switching an active layout may cause rendering resources to be stopped and started. The rendering tasks for the channels are computed by the server and send to the appropriate render client nodes at the beginning of each frame.

3.11.3. Frustum

Compounds have a frustum description to define the physical layout of the display environment. The frustum specification is described in Section 3.8. The frustum description is inherited by the children, therefore the frustum is defined on the topmost compound, typically by the corresponding segment.

3.11.4. Compound Classification

The channels of the leaf compounds in the compound tree are designated as source channels. The topmost channel in the tree is the destination channel. One compound tree might have multiple destination channels, e.g., for a swap-synchronized immersive installation.

All channels in a compound tree work for the destination channel. The destination channel defines the 2D pixel viewport rendered by all leaf compounds. The destination channel and pixel viewport cannot be overridden by child compounds.

3. Configuring Equalizer Clusters

3.11.5. Tasks

Compounds execute a number of tasks: clear, draw, assemble and readback. By default, a leaf compound executes all tasks and a non-leaf compound assemble and readback. A non-leaf compound will never execute the draw task.

A compound can be configured to execute a specific set of tasks, for example to configure the multiple steps used by binary-swap compositing.

3.11.6. Decomposition - Attributes

Compounds have attributes which configure the decomposition of the destination channel's rendering, which is defined by the viewport, frustum and database. A **viewport** decomposes the destination channel and frustum in screen space. A **range** tells the application to render a part of its database, and an eye rendering pass can selectively render different stereo passes. A **pixel** parameter adjusts the frustum so that the source channel renders an even subset of the parent's pixels. A **subpixel** parameter tells the source channels to render different samples for one pixel to perform anti-aliasing or depth-of-field rendering. Setting one or multiple attributes causes the parent's view to be decomposed accordingly. Attributes are cumulative, that is, intermediate compound attributes affect and therefore decompose the rendering of all their children.

3.11.7. Recomposition - Frames

Compounds use output and input frames to configure the recomposition of the resulting pixel data from the source channels. An output frame connects to an input frame of the same name. The selected frame buffer data is transported from the output channel to the input channel. The assembly routine of the input channel will block on the availability of the output frame. This composition process is extensively described in Section 9.9. Frame names are only valid within the compound tree, that is, an output frame from one compound tree cannot be used as an input frame of another compound tree.

3.11.8. Adjustments - Equalizers

Equalizers are used to update compound parameters based on runtime data. They are attached to a compound (sub-)tree, on which they operate. The Equalizer distribution contains numerous example configuration files using equalizers.

Load Equalizer While pixel, subpixel and stereo compounds are naturally load-balanced, 2D and DB compounds often need load-balancing for optimal rendering performance.

Using a load equalizer is transparent to the application, and can be used with any application for 2D, and with most applications for DB load-balancing. Some applications do not support dynamic updates of the database range, and therefore cannot be used with DB load-balancing.

Using a 2D or DB load-balancer will adjust the 2D split or database range automatically each frame. The 2D load-balancer exists in three flavors: 2D using tiles, VERTICAL using columns and HORIZONTAL using rows.

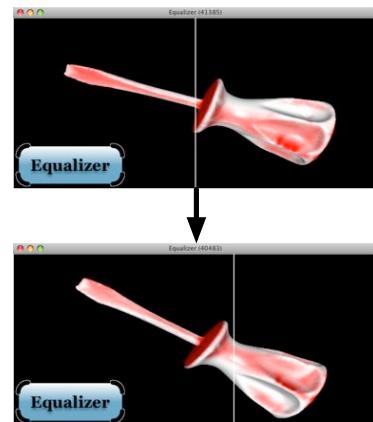


Figure 18: 2D Load-Balancing

3. Configuring Equalizer Clusters

2D load-balancing increases the framerate over a static decomposition in virtually all cases. It works best if the application data is relatively uniformly distributed in screen-space. A damping parameter can be used to fine-tune the algorithm.

DB load-balancing is beneficial for applications which cannot precisely predict the load for their scene data, e.g., when the data is nonuniform. Volume rendering is a counterexample, where the data is uniform and a static DB decomposition typically results in a better performance.

View Equalizer Depending on the model position and data structure, each segment of a multi-display system has a different rendering load. The segment with the biggest load determines the overall performance when using a static assignment of resources to segments. The view equalizer analyzes the load of all segments, and adjusts the resource usage each frame. It equalizes the load on all segments of a view.

Figure 19¹⁴ illustrates this process. On the left side, a static assignment of resources to display segments is used. The right-hand segment has a higher load than the left-hand segment, causing sub-optimal performance. The configuration on the right uses a view equalizer, which assigns two GPU's to the left segment and four GPU's to the right segment, which leads to optimal performance for this model and camera position.

The view equalizer can also use resources from another display resource, if this resource has little rendering load by itself. It is therefore possible to improve the rendering performance of a multi-display system without any additional resources. This is particularly useful for installations with a higher number of displays where the rendering load is typically in a few segments only, e.g., for a CAVE.

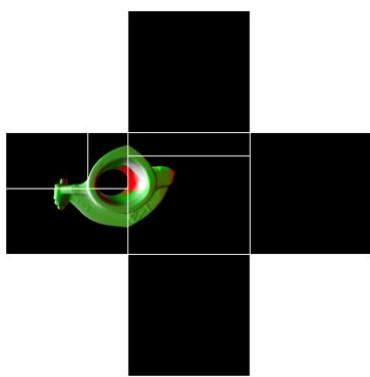


Figure 20: Cross-Segment Load Balancing for a CAVE

ing of the framerate at the destination channel, otherwise the framerate will become

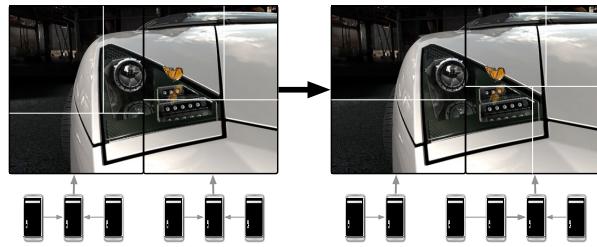


Figure 19: Cross-Segment Load-Balancing for two Segments using eight GPU's

Figure 20 shows cross-usage for a five-sided CAVE driven by five GPU's. The front and left segments show the model and have a significant rendering load. The view equalizer assigns the GPU's from the top, bottom and right wall for rendering the left and front wall in this configuration.

Cross-segment load-balancing is configured hierarchically. On the top compound level, a view equalizer assigns resources to each of its children, so that the optimal number of resources is used for each segment. On the next level, a load equalizer on each child computes the resource distribution within the segment, taking the resource usage given by the view equalizer into account.

Framerate Equalizer Certain configurations, in particular DPlex compounds, require a smoothing of the framerate at the destination channel, otherwise the framerate will become

¹⁴Image Copyright Realtime Technology AG, 2008

3. Configuring Equalizer Clusters

periodically faster and slower. Using a framerate equalizer will smoothen the swap buffer rate on the destination window for optimal user experience.

DFR Equalizer Dynamic Frame Resolution (DFR) trades rendering performance for visual quality. The rendering for a channel is done at a different resolution than the native channel resolution to keep the framerate constant. The DFR equalizer adjusts the zoom of a channel, based on the target and current framerate. It is typically used for fill-rate bound applications, such as volume rendering and ray-tracing.

Figure 21¹⁵ shows DFR for volume rendering. To achieve 10 frames per second, the model is rendered at a lower resolution, and upscaled to the native resolution for display. The rendering quality is slightly degraded, while the rendering performance remains interactive. When the application is idle, it renders a full-resolution view.

The dynamic frame resolution is not limited to subsampling the rendering resolution, it will also supersample the image if the source buffer is big enough. Upscaled rendering, which will down-sample the result for display, provides dynamic anti-aliasing at a constant framerate.

Monitor Equalizer The monitor equalizer allows the observation of another view, potentially made of multiple segments, in a different channel at a different resolution. This is typically used to reuse the rendering of a large-scale display on an operator station.

A monitor equalizer adjusts the frame zoom of the output frames used to observe the rendering, depending on the destination channel size. The output frames are down-scaled on the GPU before readback, which results in optimal performance.

Figure 22 shows a usage of the monitor equalizer. A two-segment display wall is driven by a separate control station. The rendering happens only on the display wall, and the control window receives the correctly down-scaled version of the rendering.

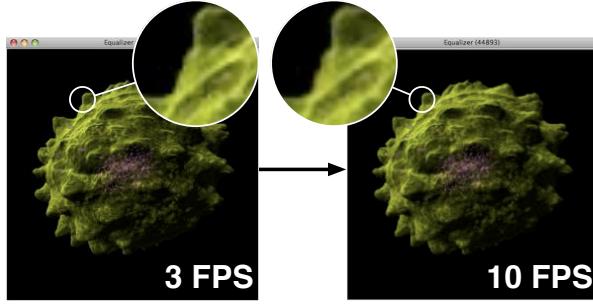


Figure 21: Dynamic Frame Resolution

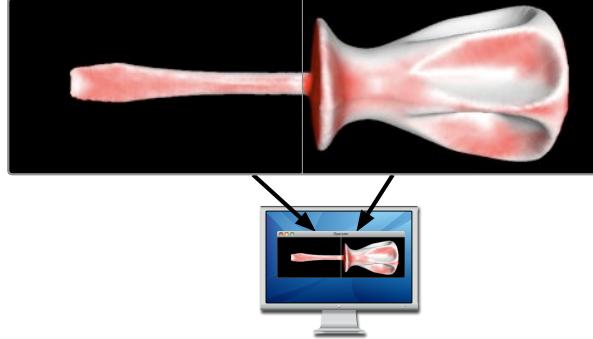


Figure 22: Monitoring a Projection Wall

¹⁵Data set courtesy of Olaf Ronneberger, Computer Science Institute, University of Freiburg, Germany

4. Setting up a Visualization Cluster

4. Setting up a Visualization Cluster

This section covers the setup of a visualization cluster to run Equalizer applications. It does not cover basic cluster management and driver installation. A prerequisite to the following steps is a preinstalled cluster, including network and graphics card drivers.

4.1. Name Resolution

Pay attention that your name resolution works properly, that is, the output of the hostname command should resolve to the same, non-local IP address on all machines. If this is not the case, you will have to provide the public IP address for all processes, in the following way:

Server Specify the server IP as the hostname field in the connection description in the server section.

Application Specify the application IP as the hostname field in the connection description in the appNode section. If the application should not contribute to the rendering, set up an appNode without a pipe section.

Render Clients Specify the client IP's as the hostname field in the connection description of each node section.

4.2. Server

The server may be started as a separate process or within the application process. If it is started separately, simply provide the desired configuration file as a parameter. It will listen on all network addresses, unless other connection parameters are specified in the configuration file. If the server is started within the application process, using the `-eq-config` parameter, you will have to specify a connection description for the server in the configuration file. Application-local servers do not create a listening socket by default for security reasons.

4.3. Starting Render Clients

Cluster configurations use multiple Equalizer nodes, where each node represents a separate process, typically on a different machine. These node processes have to be started for a visualization session, and need to communicate with each other.

Equalizer supports prelaunched render clients and automatic render client launching, if configured properly. Prelaunched render clients are started manually or by an external script on a predefined address. Auto-launched render client are started and stopped by the Equalizer server on demand.

The two mechanism can coexist. The server will first try to connect a prelaunched render client on the given connection descriptions for each node. If this fails, he will try to auto-launch the render client. If the render client does not connect back to the server within a certain timeout (default one minute), a failure to initialize the configuration is reported back to the application.

4.3.1. Prelaunched Render Clients

Prelaunched render clients are useful if setting up auto-launching is too time-consuming, e.g., on Windows which does not have a standard remote login procedure. The following steps are to be taken to use prelaunched render clients:

4. Setting up a Visualization Cluster

- Set the **connection** hostname and port parameters of each node in the configuration file.
- Start the **render clients** using the parameters `-eq-client` and `-eq-listen`, e.g., `./build/Linux/bin/eqPly -eq-client -eq-listen 192.168.0.2:1234`. Pay attention to use the same connection parameters as in the configuration file.
- Start the **application**. If the server is running on the same machine and user as the application, the application will connect to it automatically. Otherwise use the `-eq-server` parameter to specify the server address.

The render clients will automatically exit when the config is closed. The eqPly example application implements the option `-r` to keep the render client processes resident.

4.3.2. Auto-launched Render Clients

To automatically launch the render clients, the server needs to know the name of the render client and the command to launch them without user interaction.

The name of the render client is automatically set to the name of the application executable. This may be changed programmatically by the application. Normally it suffices to install the application in the same directory on all machines, ideally using a shared file system.

The default launch command is set to ssh, which is the most common solution for remote logins. To allow the server to launch the render clients without user interaction, password-less ssh needs to be set up. Please refer to the ssh documentation (cf. `ssh-keygen` and `~/.ssh/authorised_keys`) and verify the functionality by logging in to each machine from the host running the server.

4.4. Debugging

If your configuration does not work, simplify it as much as possible first. Normally this means that there is one server, application and render client. Failure to launch a cluster configuration often is caused by one of the following reasons:

- A firewall is blocking network connections.
- The render client can't access the GPU's on the remote host. Set up your X-Server or application rights correctly. Log into the remote machine using the launch command and try to execute a simple application, e.g., `glxinfo -display :0.1`.
- The server does not find the prelaunched render client. Verify that the client is listening on the correct IP and port, and that this IP and port are reachable from the server host.
- The server cannot launch a render client. Check the server log for the launch command used, and try to execute a simple application from the server host using this launch command. It should run without user interaction. Check that the render client is installed in the correct path. Pay attention to the launch command quotes used to separate arguments on Windows. Check that the same software versions, including Equalizer, are installed on all machines.
- A client can't connect back to the application. Check the client log, this is typically caused by a misconfigured host name resolution.

Part II.

Programming Guide

To modify an application for Equalizer, the programmer structures the source code so that the OpenGL rendering can be executed in parallel, potentially using multiple processes for cluster-based execution.

5. Hello, World!

The `eqHello` example is a minimal application to illustrate the basic principle of any Equalizer application: The application developer has to implement a rendering method similar to the `glutDisplayFunc` in GLUT applications.

It can be run as a stand-alone application from the command line or using an explicit configuration file for a visualization cluster. In the stand-alone case, any Equalizer application, including `eqHello`, will automatically configure itself to use all graphics cards found on the local system for scalability.

The `eqHello` example uses Sequel, a thin layer on top of the canonical Equalizer programming interface. Section 7 introduces Sequel in detail, and Section 8 introduces the full scope of the Equalizer API.

The main `eqHello` function instantiates an application object, initializes it, starts the main loop and finally de-initializes the application:

```
int main( const int argc , char** argv )
{
    eqHello :: ApplicationPtr app = new eqHello :: Application;

    if( app->init( argc , argv , 0 ) && app->run( 0 ) && app->exit( ) )
        return EXIT_SUCCESS;

    return EXIT_FAILURE;
}
```

The application object represents one process in the cluster. The primary application instance has the rendering loop and controls all execution. All other instances used for render client processes are passive and driven by Equalizer. The application is responsible for creating the renderers, of which one per GPU is used:

```
class Application : public seq :: Application
{
public:
    virtual ~Application() {}
    virtual seq :: Renderer* createRenderer() { return new Renderer( *this ); }
};

typedef lunchbox :: RefPtr< Application > ApplicationPtr;
```

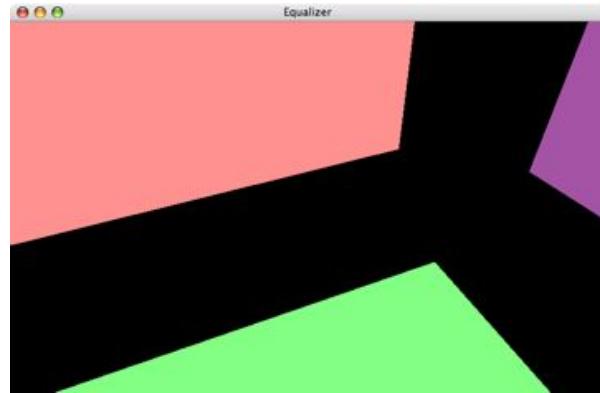


Figure 23: Hello, World!

6. Programming Interface

The renderer is responsible for executing the application's rendering code. One instance for each GPU is used. All calls to a single renderer are executed serially and therefore thread-safe.

In `eqHello`, the renderer draws six colored quads. The only change from a standard OpenGL application is the usage of the rendering context provided by Equalizer, most notably the frustum and viewport. The rendering context is described in detail in Section 8.8.2, and `eqHello` simply calls `applyRenderingContext` which will execute the appropriate OpenGL calls.

After setting up lighting, the model is positioned using `applyModelMatrix`. For convenience, Sequel maintains one camera per view. The usage of this camera is purely optional, an application can implement its own camera model.

The actual OpenGL code, rendering six colored quads, is omitted here for brevity:

```
void eqHello :: Renderer :: draw( co :: Object* frameData )
{
    applyRenderingContext(); // set up OpenGL State

    const float lightPos[] = { 0.0f, 0.0f, 1.0f, 0.0f };
    glLightfv( GL_LIGHT0, GL_POSITION, lightPos );

    const float lightAmbient[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    glLightfv( GL_LIGHT0, GL_AMBIENT, lightAmbient );

    applyModelMatrix(); // global camera

    // render six axis-aligned colored quads around the origin
```

6. Programming Interface

Equalizer uses a C++ programming interface. The API is minimally invasive. Equalizer imposes only a minimal, natural execution framework upon the application. It does not provide a scene graph, or interferes in any other way with the application's rendering code. The restructuring work enforced by Equalizer is the minimal refactoring needed to parallelize the application for scalable, distributed rendering.

The API documentation is available on the website or in the header files, and provides a comprehensive documentation on individual methods, types and other elements of the API. Methods marked with a specific version are part of the official, public API and have been introduced by this Equalizer version. Reasonable care is taken to not break API compatibility or change the semantics of these methods within future Equalizer versions of the same major revision. Any eventual changes to the public API are documented in the release notes and the file `CHANGES.txt`.

In addition the official, public API Equalizer exposes a number of unstable methods and, where unavoidable, internal APIs. These are clearly marked in the API documentation. Unstable methods may be used by the programmer, but their interface or functionality may change in any future Equalizer version. The usage of internal methods is discouraged. Undocumented or unversioned methods should be considered as part of the unstable API.

6.1. Namespaces

The Equalizer software stack is modularized, layering gradually more powerful, but less generic APIs on top of each other. It furthermore relies on a number of required and optional libraries. Application developers are exposed to the following namespaces:

6. Programming Interface

seq The core namespace for Sequel, the simple interface to the Equalizer client library.

eq The core namespace for the Equalizer client library. The classes and their relationship in this namespace closely model the configuration file format. The classes in the **eq** namespace are the main subject of this Programming Guide. Figure 25 provides an overview map of the most important classes in the Equalizer namespace, grouped by functionality.

eq::util The **eq::util** namespace provides common utility classes, which often simplify the usage of OpenGL functions. Most classes in this namespace are used by the Equalizer client library, but are usable independently from Equalizer for application development.

eq::admin The **eq::admin** namespace implements an administrative API to change the configurations of a running server. This admin API is not yet finalized and will very likely change in the future.

eq::fabric The **eq::fabric** namespace is the common data management and transport layer between the client library, the server and the administrative API. Most Equalizer client classes inherit from base classes in this namespace and have all their data access methods in these base classes.

co Collage is the network library used by Equalizer. It provides basic functionality for network communication, such as **Connection** and **ConnectionSet**, as well as higher-level functionality such as **Node**, **LocalNode**, **Object** and **Serializable**. Please refer to Section 10 for an introduction into the network layer, and to Section 8.3 for distributed objects.

lunchbox The lunchbox library provides C++ classes to abstract the underlying operating system and to implement common helper functionality for multi-threaded applications. Examples are **lunchbox::Clock** providing a high-resolution timer, or **lunchbox::MTQueue** providing a thread-safe, blocking fifo. Classes in this namespace are fully documented in the API documentation on the Equalizer website, and are not subject of this Programming Guide.

hwloc, boost, vmmib, gpu-sd External libraries providing manual and automatic thread affinity, serialization and the foundation for RSP multicast, vector and matrix mathematics as well as local and remote GPU discovery, respectively. The hwloc and gpu-sd libraries are used only internally and not exposed through the API.

eq::server The server namespace, implementing the functionality of the Equalizer server, which is an internal namespace not to be used by application developers. The **eq::admin** namespace enables run-time administration of Equalizer servers. The server does not yet expose a stable API.

The Equalizer examples are implemented in their own namespaces, e.g., **eqPly** or **eVolve**. They rely mostly on subclassing from the **eq** namespace, with the occasional usage of functionality from the **eq::util**, **eq::fabric**, **co** and **lunchbox** namespace. Figure 24 shows the namespaces and their layering.

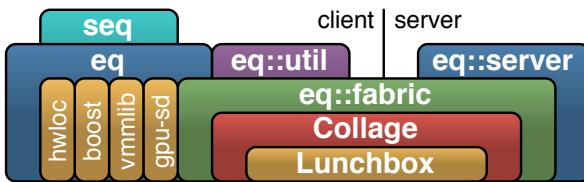


Figure 24: Namespaces

6. Programming Interface

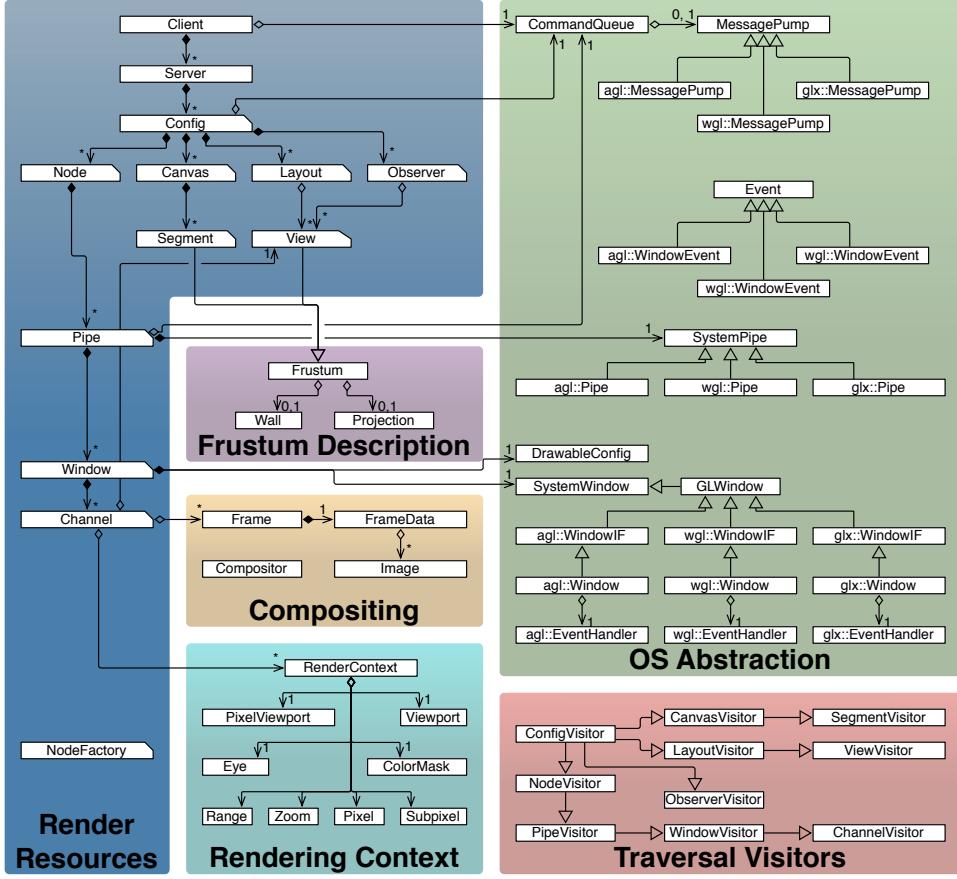


Figure 25: Equalizer client UML map

6.2. Task Methods

Methods called by the application have the form **verb[Noun]**, whereas methods called by Equalizer ('Task Methods') have the form **nounVerb**. For example, the application calls `Config::startFrame` to render a new frame, which causes, among many other things, `Node::frameStart` to be called in all active render clients.

The application inherits from Equalizer classes and overrides virtual functions to implement certain functionality, e.g., the application's OpenGL rendering in `eq::Channel::frameDraw`. These task methods are similar in concept to C function callbacks. Section 8 will discuss the important task methods. A full list can be found on the website¹⁶.

6.3. Execution Model and Thread Safety

Using threading correctly in OpenGL-based applications is easy with Equalizer. Equalizer creates one rendering thread for each graphics card. All task methods for a pipe, and therefore all OpenGL commands, are executed from this thread. This threading model follows the OpenGL 'threading model', which maintains a current context for each thread. If structured correctly, the application rarely has to take care of thread synchronization or protection of shared data.

¹⁶see <http://www.equalizergraphics.com/documents/design/taskMethods.html>

7. Sequel: The seqPly Polygonal Renderer

The main thread is responsible for maintaining the application logic. It reacts on user events, updates the data model and requests new frames to be rendered. It drives the whole application, as shown in Figure 26.

The rendering threads concurrently render the application’s database. The database should be accessed in a read-only fashion during rendering to avoid threading problems. This is normally the case, for example all modern scene graphs use read-only render traversals, writing the GPU-specific information into a separate per-GPU data structure.

All rendering threads in the configuration run asynchronously to the application’s main thread. Depending on the configuration’s latency, they can fall n frames behind the last frame finished by the application thread. A latency of one frame is usually not perceived by the user, but can increase rendering performance substantially since operations can be better pipelined.

Rendering threads on a single node are synchronized when using the default thread model `draw_sync`. When a frame is finished, all local rendering threads are done drawing. Therefore the application can safely modify the data between the end of a frame and the beginning of a new frame. Furthermore, only one instance of the scene data has to be maintained within a process, since all rendering threads are guaranteed to draw the same frame.

This per-node frame synchronization does not inhibit latency across rendering nodes. Furthermore, advanced rendering software which multi-buffers the dynamic parts of the database can disable the per-node frame synchronization, as explained in Section 9.3.2. Some scene graphs implement multi-buffered data, and can profit from relaxing the local frame synchronization.

7. Sequel: The seqPly Polygonal Renderer

In this section the source code of seqPly is used to introduce the Sequel API in detail, and relevant design decisions, caveats and other remarks are discussed. Sequel is conceived to lower the entry barrier into creating parallel rendering applications without sacrificing major functionality. It implements proven design patterns of Equalizer applications. Sequel does not prohibit the usage of Equalizer or Collage functionality, it simply makes it easier to use for the common use cases.

The seqPly parallel renderer provides a subset of features found in its cousin application, eqPly, introduced in the next section. It is more approachable, using approximately one tenth of the source code needed by eqPly due to the use of the Sequel abstraction layer and reduced functionality.

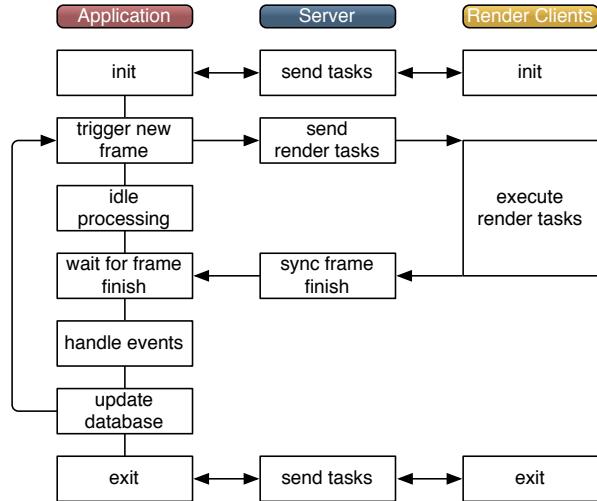


Figure 26: Simplified Execution Model

7. Sequel: The seqPly Polygonal Renderer

7.1. main Function

The main function is almost identical to `eqHello`. It instantiates an application object, initializes it, starts the main loop and finally de-initializes the application. The source code is not reproduced here due to its similarity with Section 5.

7.2. Application

The application object in Sequel represents one process in the cluster. The main application instance has the rendering loop and controls all execution. All other instances used for render client processes are passive and driven by Equalizer.

Sequel applications derive their application object from `seq::Application` and selectively override functionality. They have to implement `createRenderer`, as explained below. The `seqPly` application overrides init, exit, run and implements `createRenderer`.

The initialization and exit routines are overwritten to parse `seqPly`-specific command line options and to load and unload the requested model:

```
bool Application::init( const int argc, char** argv )
{
    const eq::Strings& models = _parseArguments( argc, argv );
    if( !seq::Application::init( argc, argv, 0 ) )
        return false;

    _loadModel( models );
    return true;
}

bool Application::exit()
{
    _unloadModel();
    return seq::Application::exit();
}
```

Sequel manages distributed objects, simplifying their use. This includes registration, automatic creation and mapping as well as commit and synchronization of objects. One special object for initialization (not used in `seqPly`) and one for per-frame data are managed by Sequel, in addition to an arbitrary number of application-specific objects.

The objects passed to `seq::Application::init` and `seq::Application::run` are automatically distributed and instantiated on the render clients, and then passed to the respective task callback methods. The application may pass a 0 pointer if it does not need an object for initialization or per-frame data. Objects are registered with a type, and when automatically created, the `createObject` method on the application or renderer is used to create an instance based on this type.

The run method is overloaded to pass the frame data object to the Sequel application run loop. The object will be distributed and synchronized to all renderers:

```
bool Application::run()
{
    return seq::Application::run( &_frameData );
}
```

The application is responsible for creating renderers. Sequel will request one renderer for each GPU rendering thread. Sequel also provides automatic mapping and synchronization of distributed objects, for which the application has to provide a creation callback:

```
seq::Renderer* Application::createRenderer()
{
    return new Renderer( *this );
```

7. Sequel: The seqPly Polygonal Renderer

```

}

co::Object* Application::createObject( const uint32_t type )
{
    switch( type )
    {
        case seq::OBJECTTYPEFRAMEDATA:
            return new eqPly::FrameData;

        default:
            return seq::Application::createObject( type );
    }
}

```

7.3. Renderer

The renderer is responsible for executing the application's rendering code. One instance for each GPU is used. All calls to a single renderer are executed serially and therefore thread-safe.

The seqPly rendering code uses the same data structure and algorithm as eqPly, described in Section 8.8.2. This renderer captures the GPU-specific data in a State object, which is created and destroyed during init and exit. The state captures also the OpenGL function table, which is available when init is called, but not yet during the constructor of the renderer:

```

bool Renderer::init( co::Object* initData )
{
    _state = new State( glewGetContext( ) );
    return seq::Renderer::init( initData );
}

bool Renderer::exit()
{
    _state->deleteAll();
    delete _state;
    _state = 0;
    return seq::Renderer::exit();
}

```

The rendering code is similar to the typical OpenGL rendering code, except for a few modifications to configure the rendering. First, the render context is applied and lighting is set up. The render context, described in detail in Section 8.8.2, sets up the stereo buffer, 3D viewport as well as the projection and view matrices using the appropriate OpenGL calls. Applications can also retrieve the render context and apply the settings themselves:

```

applyRenderContext();

glLightfv( GL_LIGHT0, GL_POSITION, lightPosition );
glLightfv( GL_LIGHT0, GL_AMBIENT, lightAmbient );
glLightfv( GL_LIGHT0, GL_DIFFUSE, lightDiffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, lightSpecular );

glMaterialfv( GL_FRONT, GL_AMBIENT, materialAmbient );
glMaterialfv( GL_FRONT, GL_DIFFUSE, materialDiffuse );
glMaterialfv( GL_FRONT, GL_SPECULAR, materialSpecular );
glMateriali( GL_FRONT, GL_SHININESS, materialShininess );

```

After the static light setup, the model matrix is applied to the existing view matrix, completing the modelview matrix and positioning the model. Sequel maintains a per-view camera, which is modified through mouse and keyboard events and determines the model matrix. Applications can overwrite this event handling and maintain their own camera model. Afterwards, the state is set up with the

8. Equalizer: The eqPly Polygonal Renderer

projection-modelview matrix for view frustum culling, the DB range for sort-last rendering and the cull-draw traversal is executed, as described in Section 8.8.2:

```
applyModelMatrix();  
  
glColor3f( .75f, .75f, .75f );  
  
// Compute cull matrix  
const eq::Matrix4f& modelM = getModelMatrix();  
const eq::Matrix4f& view = getViewMatrix();  
const eq::Frustumf& frustum = getFrustum();  
const eq::Matrix4f projection = frustum.compute_matrix();  
const eq::Matrix4f pmv = projection * view * modelM;  
const seq::RenderContext& context = getRenderContext();  
  
_state->setProjectionModelViewMatrix( pmv );  
_state->setRange( &context.range.start );  
_state->setColors( model->hasColors() );  
  
model->cullDraw( *_state );
```

8. Equalizer: The eqPly Polygonal Renderer

In this section the source code of `eqPly` is used to introduce the Equalizer API in detail, and relevant design decisions, caveats and other remarks are discussed.

`eqPly` is a parallel renderer for polygonal data in the ply file format. It supports all relevant Equalizer features, and can be used to render on large-scale displays, immersive environments with head tracking and to render massive data sets using all scalable rendering features of Equalizer. It is a superset of `seqPly`, introduced in Section 7.

The `eqPly` example is shipped with the Equalizer distribution and serves as a reference implementation of an Equalizer-based application of medium complexity. It focuses on the example usage of core Equalizer features, not on advanced rendering features or visual quality.

Figure 27 shows how the most important Equalizer classes are used through inheritance by the `eqPly` example. All classes in the example are in the `eqPly` namespace to avoid type name ambiguities, in particular for the `Window` class which is frequently used as a type in the global namespace by windowing systems.

The `eqPly` classes fall into two categories: Subclasses of the rendering entities introduced in Section 3, and classes for distributing data.

The function and typical usage for each of the rendering entities is discussed in this section. Each of these classes inherits from a base class in the `eq:fabric` namespace, which implements data distribution for the entity. The fabric base classes are omitted in Figure 27.

The distributed data classes are helper classes based on `co:Serializable` or its base class `co:Object`. They illustrate the typical usage of distributed objects for static as well as dynamic, frame-specific data. Furthermore they are used for a basic scene graph distribution of the model data to the render client processes.

8.1. main Function

The main function starts off with initializing the Equalizer library. The command line arguments are passed on to Equalizer. They are used to set certain default values based on Equalizer-specific options¹⁷, e.g., the default server address. Furthermore, a `NodeFactory` is provided. The `EQERROR` macro, and its counterparts

¹⁷Equalizer-specific options always start with `--eq-`

8. Equalizer: The eqPly Polygonal Renderer

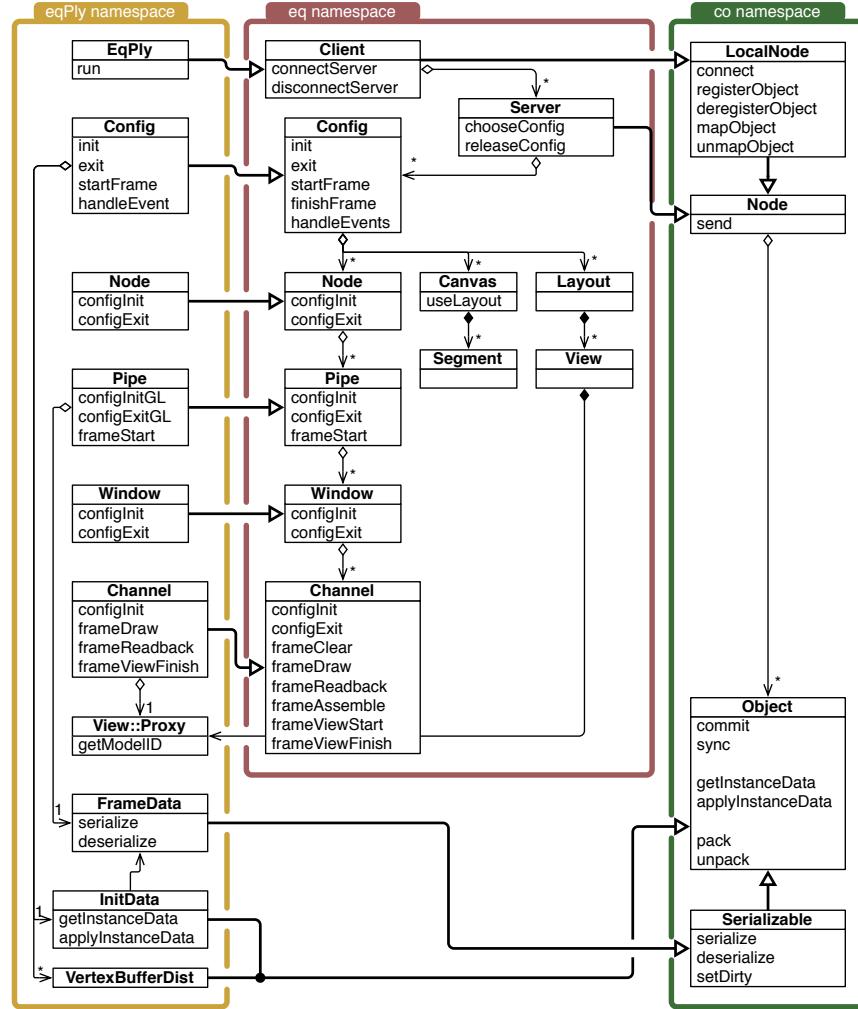


Figure 27: UML Diagram eqPly and relevant Equalizer Classes

EQWARN, EQINFO and EQVERB allow selective debugging outputs with various logging levels:

```
int main( const int argc, char** argv )
{
    // 1. Equalizer initialization
    NodeFactory nodeFactory;
    eqPly::initErrors();

    if( !eq::init( argc, argv, &nodeFactory ) )
    {
        LBERROR << "Equalizer_init_failed" << std::endl;
        return EXIT_FAILURE;
    }
}
```

The node factory is used by Equalizer to create the object instances of the configured rendering entities. Each of the classes inherits from the same type provided by Equalizer in the eq namespace. The provided `eq::NodeFactory` base class instantiates 'plain' Equalizer objects, thus making it possible to selectively subclass individual entity types, as it is done by `eqHello`. For each rendering resources used in the configuration, one C++ object will be created during initialization. Config, node

8. Equalizer: The eqPly Polygonal Renderer

and pipe objects are created and destroyed in the node thread, whereas window and channel objects are created and destroyed in the pipe thread:

```
class NodeFactory : public eq::NodeFactory
{
public:
    virtual eq::Config* createConfig( eq::ServerPtr parent )
    {
        return new eqPly::Config( parent );
    }
    virtual eq::Node* createNode( eq::Config* parent )
    {
        return new eqPly::Node( parent );
    }
    virtual eq::Pipe* createPipe( eq::Node* parent )
    {
        return new eqPly::Pipe( parent );
    }
    virtual eq::Window* createWindow( eq::Pipe* parent )
    {
        return new eqPly::Window( parent );
    }
    virtual eq::Channel* createChannel( eq::Window* parent )
    {
        return new eqPly::Channel( parent );
    }
    virtual eq::View* createView( eq::Layout* parent )
    {
        return new eqPly::View( parent );
    }
};
```

The second step is to parse the command line into the `LocalInitData` data structure. A part of it, the base class `InitData`, will be distributed to all render client nodes. The command line parsing is done by the `LocalInitData` class, which is discussed in Section 8.3.1:

```
// 2. parse arguments
eqPly::LocalInitData initData;
initData.parseArguments( argc, argv );
```

The third step is to create an instance of the application and to initialize it locally. The application is a subclass of `eq::Client`, which in turn is an `co::LocalNode`. The underlying Collage network library, discussed in Section 10, is a peer-to-peer network of `co::LocalNodes`. The client-server concept is implemented the higher-level `eq` client namespace.

The local initialization of a node creates at least one local listening socket, which allows the `eq::Client` to communicate over the network with other nodes, such as the server and the rendering clients. The listening socket(s) can be configured using the `-eq-listen` command line parameter, by adding connections to the `appNode` in the configuration file, or by programmatically adding connection descriptions to the client before the local initialization:

```
// 3. initialization of local client node
lunchbox::RefPtr< eqPly::EqPly > client = new eqPly::EqPly( initData );
if( !client->initLocal( argc, argv ) )
{
    LBERROR << "Can't init client" << std::endl;
    eq::exit();
    return EXIT_FAILURE;
}
```

Finally everything is set up, and the `eqPly` application is executed:

```
// 4. run client
const int ret = client->run();
```

After the application has finished, it is de-initialized and the `main` function returns:

```
// 5. cleanup and exit
client->exitLocal();

LBASSERTINFO( client->getRefCount() == 1, client );
client = 0;
```

8. Equalizer: The eqPly Polygonal Renderer

```

eq::exit();
eqPly::exitErrors();
return ret;
}

```

8.2. Application

In the case of `eqPly`, the application is also the render client. The `eqPly` executable has three runtime behaviors:

1. **Application:** The executable started by the user, the controlling entity of the rendering session.
2. **Auto-launched render client:** The typical render client, started by the server. The server starts the executable with special parameters, which cause `Client::initLocal` to never return. During exit, the server terminates the process. By default, the server starts the render client using `ssh`. The launch command can be used to configure another program to auto-launch render clients.
3. **Resident render client:** Manually prestarted render client, listening on a specified port for server commands. This mode is selected using the command-line option `-eq-client` and `-eq-listen <address>` to specify a well-defined listening address, and potentially `-r` to keep the client running across multiple runs¹⁸.

8.2.1. Main Loop

The application's main loop starts by connecting the application to an Equalizer server. If no server is specified, `Client::connectServer` tries first to connect to a server on the local machine using the default port. If that fails, it will create a server running within the application process using auto-configuration as described in Section 3.1.2. The command line parameter `-eq-config` can be used to specify a gpu-sd session or configuration file, and `-eq-server` to explicitly specify a server address.

```

int EqPly::run()
{
    // 1. connect to server
    eq::ServerPtr server = new eq::Server;
    if( !connectServer( server ) )
    {
        LBERROR << "Can't open server" << std::endl;
        return EXIT_FAILURE;
    }
}

```

The second step is to ask the server for a configuration. The `ConfigParams` are a placeholder for later Equalizer implementations to provide additional hints and information to the server for auto-configuration. The configuration chosen by the server is created locally using `NodeFactory::createConfig`. Therefore it is of type `eqPly::Config`, but the return value is `eq::Config`, making the `static_cast` necessary:

```

// 2. choose config
eq::ConfigParams configParams;
Config* config = static_cast<Config*>(server->chooseConfig( configParams ));

if( !config )
{
    LBERROR << "No matching config on server" << std::endl;
    disconnectServer( server );
}

```

¹⁸see <http://www.equalizergraphics.com/documents/design/residentNodes.html>

8. Equalizer: The eqPly Polygonal Renderer

```
    return EXIT_FAILURE;
}
```

Finally it is time to initialize the configuration. For statistics, the time for this operation is measured and printed. During initialization the server launches and connects all render client nodes, and calls the appropriate initialization task methods, as explained in later sections. `Config::init` returns after all nodes, pipes, windows and channels are initialized.

The return value of `Config::init` depends on the configuration robustness attribute. This attribute is set by default, allowing configurations to launch even when some entities failed to initialize. If set, `Config::init` always returns true. If deactivated, it returns `true` only if all initialization task methods were successful. In any case, `Config::getError` only returns `ERROR_NONE` if all entities have initialized successfully.

The `EQLOG` macro allows topic-specific logging. The numeric topic values are specified in the respective `log.h` header files, and logging for various topics is enabled using the environment variable `EQ_LOG_TOPICS`:

```
// 3. init config
lunchbox::Clock clock;

config->setInitData( _initData );
if( !config->init( ) )
{
    LBWARN << "Error_during_initialization:" << config->getError()
        << std::endl;
    server->releaseConfig( config );
    disconnectServer( server );
    return EXIT_FAILURE;
}
if( config->getError( ) )
    LBWARN << "Error_during_initialization:" << config->getError()
        << std::endl;

LBLOG( LOG_STATS ) << "Config_init_took_" << clock.getTimef() << ".ms"
    << std::endl;
```

When the configuration was successfully initialized, the main rendering loop is executed. It runs until the user exits the configuration, or when a maximum number of frames has been rendered, specified by a command-line argument. The latter is useful for benchmarks. The `Clock` is reused for measuring the overall performance. A new frame is started using `Config::startFrame` and a frame is finished using `Config::finishFrame`.

When a new frame is started, the server computes all rendering tasks and sends them to the appropriate render client nodes. The render client nodes dispatch the tasks to the correct node or pipe thread, where they are executed in order of arrival.

`Config::finishFrame` blocks on the completion of the frame `current - latency`. The latency is specified in the configuration file, and allows several outstanding frames. This allows overlapping execution in the node processes and pipe threads and minimizes idle times.

By default, `Config::finishFrame` also synchronizes the completion of all local rendering tasks for the current frame. This facilitates porting of existing rendering codes, since the database does not have to be multi-buffered. Applications such as `eqPly`, which do not need this per-node frame synchronization, can disable it as explained in Section 9.3.2:

```
// 4. run main loop
uint32_t maxFrames = _initData.getMaxFrames();
int lastFrame = 0;

clock.reset();
while( config->isRunning( ) && maxFrames-- )
```

8. Equalizer: The eqPly Polygonal Renderer

```
{
    config->startFrame();
    if( config->getError() )
        LBWARN << "Error_during_frame_start:" << config->getError()
        << std::endl;
    config->finishFrame();
```

Figure 28 shows the execution of the rendering tasks of a 2-node 2D compound without latency and with a latency of one frame. The asynchronous execution pipelines certain rendering operations and hides imbalances in the load distribution, resulting in an improved framerate. For example, we have observed a speedup of 15% on a five-node rendering cluster when using a latency of one frame instead of no latency¹⁹. A latency of one or two frames is normally not perceived by the user. The statistics overlay is explained in detail in Section 9.11.2.

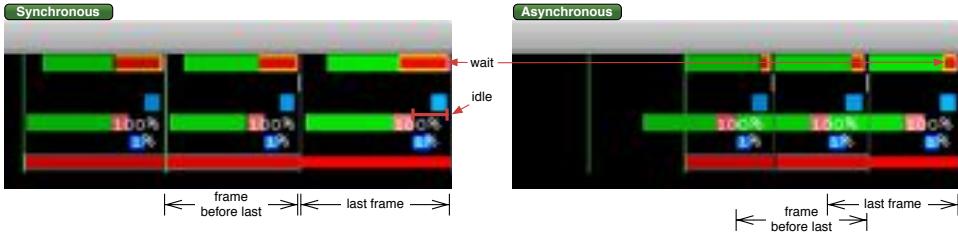


Figure 28: Synchronous and Asynchronous Execution

When playing a camera animation, eqPly prints the rendering performance once per animation loop for benchmarking purposes:

```
if( config->getAnimationFrame() == 1 )
{
    const float time = clock.resetTimef();
    const size_t nFrames = config->getFinishedFrame() - lastFrame;
    lastFrame = config->getFinishedFrame();

    LBLOG( LOG_STATS ) << time << "ms_for_" << nFrames << "frames@"
        << ( nFrames / time * 1000.f ) << "FPS"
        << std::endl;
}
```

eqPly uses event-driven execution, that is, it only request new rendering frames if an event or animation requires an update. The eqPly::Config maintains a dirty state, which is cleared after a frame has been started, and set when an event causes a redraw. Furthermore, when an animation is running or head tracking is active, the config always signals the need for a new frame.

If the application detects that it is currently idle, all pending commands are gradually flushed, while still looking for a redraw event. Then it waits and handles one event at a time, until a redraw is needed:

```
while( !config->needRedraw() ) // wait for an event requiring redraw
{
    if( hasCommands() ) // execute non-critical pending commands
    {
        processCommand();
        config->handleEvents(); // non-blocking
    }
    else // no pending commands, block on user event
    {
        const eq::ConfigEvent* event = config->nextEvent();
        if( !config->handleEvent( event ) )
            LBVERB << "Unhandled_" << event << std::endl;
```

¹⁹<http://www.equalizergraphics.com/scalability.html>

8. Equalizer: The eqPly Polygonal Renderer

```

        }
    config->handleEvents(); // process all pending events
}
```

When the main rendering loop has finished, Config::finishAllFrames is called to catch up with the latency. It returns after all outstanding frames have been rendered, and is needed to provide an accurate measurement of the framerate:

```

const uint32_t frame = config->finishAllFrames();
const float time = clock.resetTimef();
const size_t nFrames = frame - lastFrame;
LBLOG( LOG_STATS ) << time << "ms for " << nFrames << "frames @ "
    << ( nFrames / time * 1000.f ) << "FPS" << std::endl;
```

The remainder of the application code cleans up in the reverse order of initialization. The config is exited, released and the connection to the server is closed:

```

// 5. exit config
clock.reset();
config->exit();
LBLOG( LOG_STATS ) << "Exit took " << clock.getTimef() << "ms" << std::endl;

// 6. cleanup and exit
server->releaseConfig( config );
if( !disconnectServer( server ) )
    LBERROR << "Client::disconnectServer failed" << std::endl;

return EXIT_SUCCESS;
}
```

8.2.2. Render Clients

In the second and third use case of the eqPly, when the executable is used as a render client, Client::initLocal never returns. Therefore the application's main loop is never executed. To keep the client resident, the eqPly example overrides the client loop to keep it running beyond one configuration run:

```

void EqPly::clientLoop()
{
    do
    {
        eq::Client::clientLoop();
        LBINFO << "Configuration run successfully executed" << std::endl;
    }
    while( _ initData.isResident() ); // execute at least one config run
}
```

8.3. Distributed Objects

Equalizer provides distributed objects which 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. Section 10.4 covers distributed objects in detail.

Distributed objects can be static (immutable) or dynamic. Dynamic objects are versioned. The eqPly example uses static distributed objects to provide initial data and the model to all rendering nodes, as well as a versioned object to provide frame-specific data such as the camera position to the rendering methods.

8. Equalizer: The eqPly Polygonal Renderer

8.3.1. InitData - a Static Distributed Object

The `InitData` class holds a couple of parameters needed during initialization. These parameters never change during one configuration run, and are therefore static.

On the application side, the class `Localln initData` subclasses `InitData` to provide the command line parsing and to set the default values. The render nodes only instantiate the distributed part in `InitData`.

A static distributed object has 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, i.e., multicast where available, for data transport. They perform no type checking or transformation on the data. It is the application's responsibility to exactly match the order and types of variables during serialization and de-serialization.

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 its pointer.

For `InitData`, serialization in `getInstanceData` and de-serialization in `applyInstanceData` is performed by streaming all member variables to or from the provided data streams:

```
void InitData :: getInstanceData( co :: DataOStream& os )
{
    os << _frameDataID << _windowSystem << _renderMode << _useGLSL << _invFaces
    << _logo << _roi;
}

void InitData :: applyInstanceData( co :: DataIStream& is )
{
    is >> _frameDataID >> _windowSystem >> _renderMode >> _useGLSL >> _invFaces
    >> _logo >> _roi;
    LBASSERT( _frameDataID != eq :: UUID :: ZERO );
}
```

8.3.2. FrameData - a Versioned Distributed Object

Versioned objects have to override `getChangeType` to indicate how they want to have changes to be handled. All types of versioned objects currently implemented have the following characteristics:

- The master instance of the object generates new versions for all slaves. These versions are continuous, starting at `co::VERSION_FIRST`. It is possible to commit on slave instances, but special care has to be taken to handle possible conflicts. Section 10.4.4 covers slave object commits in detail.
- 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`.

Upon `commit` the delta data from the previous version is sent 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.

8. Equalizer: The eqPly Polygonal Renderer

Not syncing a mapped, versioned object creates a memory leak. The method `Object::notifyNewHeadVersion` is called whenever a new version is received by the node. The notification is send from the command thread, which is different from the node main thread. The object should not be synced from this method, but instead a message may be send to the application, which then takes the appropriate action. The default implementation asserts when too many versions have been queued to detect memory leaks during development.

Besides the instance data (de-)serialization methods used to map an object, versioned objects may implement `pack` and `unpack` to serialize or de-serialize the changes since the last version. If these methods are not implemented, their default implementation forwards the (de-)serialization request to `getInstanceData` and `applyInstanceData`, respectively.

The creation of distributed, versioned objects is simplified when using `co::Serializable`, which implements one common way of tracking data changes in versioned objects. The concept of a dirty bit mask is used to mark parts of the object for serialization, while preserving the capability to inherit objects. Other ways of implementing change tracking, e.g., using incarnation counters, can still be implemented by using `co::Object` which leaves all flexibility to the developer. Figure 29 shows the relationship between `co::Serializable` and `co::Object`.

The `FrameData` is sub-classed from `Serializable`, and consequently tracks its changes by setting the appropriate dirty bit whenever it is changed. The serialization methods are called by the `co::Serializable` with the dirty bit mask needed to serialize all data, or with the dirty bit mask of the changes since the last `commit`. The `FrameData` only defines its own dirty bits and serialization code:

```
/** The changed parts of the data since the last pack(). */
enum DirtyBits
{
    DIRTY_CAMERA = co::Serializable::DIRTY_CUSTOM << 0,
    DIRTY_FLAGS = co::Serializable::DIRTY_CUSTOM << 1,
    DIRTY_VIEW = co::Serializable::DIRTY_CUSTOM << 2,
    DIRTY_MESSAGE = co::Serializable::DIRTY_CUSTOM << 3,
};

void FrameData::serialize( co::DataStream& os, const uint64_t dirtyBits )
{
    co::Serializable::serialize( os, dirtyBits );
    if( dirtyBits & DIRTY_CAMERA )
        os << _position << _rotation << _modelRotation;
    if( dirtyBits & DIRTY_FLAGS )
        os << _modelID << _renderMode << _colorMode << _quality << _ortho
            << _statistics << _help << _wireframe << _pilotMode << _idle
            << _compression;
    if( dirtyBits & DIRTY_VIEW )
        os << _currentViewID;
    if( dirtyBits & DIRTY_MESSAGE )
        os << _message;
}

void FrameData::deserialize( co::DataStream& is, const uint64_t dirtyBits )
{
    co::Serializable::deserialize( is, dirtyBits );
    if( dirtyBits & DIRTY_CAMERA )
        is >> _position >> _rotation >> _modelRotation;
```

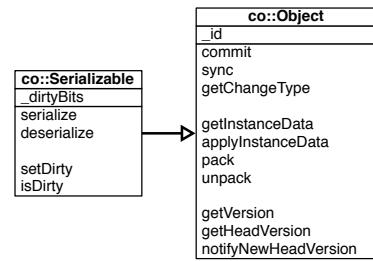


Figure 29: `co::Serializable` and `co::Object`

8. Equalizer: The eqPly Polygonal Renderer

```

if( dirtyBits & DIRTY_FLAGS )
    is >> _modelID >> _renderMode >> _colorMode >> _quality >> _ortho
        >> _statistics >> _help >> _wireframe >> _pilotMode >> _idle
        >> _compression ;
if( dirtyBits & DIRTY_VIEW )
    is >> _currentViewID ;
if( dirtyBits & DIRTY_MESSAGE )
    is >> _message ;
}

```

8.3.3. Scene Data

Some applications rely on a shared filesystem to access the data, for example when out-of-core algorithms are used. Other applications prefer to load the data only on the application process, and use distributed objects to synchronize the scene data with the render clients.

eqPly chooses the second approach, using static distributed objects to distribute the model loaded by the application. It can be easily extended to versioned objects to support dynamic data modifications.

The kD-tree data structure and rendering code for the model is strongly separated from Equalizer, and kept in the separate namespace `mesh`. It can also be used in other rendering software, for example in a GLUT application. To keep this separation while implementing data distribution, an external 'mirror' hierarchy is constructed aside the kD-tree. This hierarchy of `VertexBufferDist` nodes is responsible for cloning the model data on the remote render clients.

The identifier of the model's root object of this distributed hierarchy is passed as part of the `InitData` for the default model, or as part of the `View` for each logical view. It is used on the render clients to map the model when it is needed for rendering.

Figure 30 shows the UML hierarchy of the model and distribution classes. Section 10.4 illustrates other approaches to employ distributed objects for data distribution and synchronization.

Each `VertexBufferDist` object corresponds to one node of the model's data tree. It is serializing the data for this node. Furthermore, it mirrors the kD-tree by having a `VertexBufferDist` child for each child of its corresponding tree node. During serialization, the identifier of these children is sent to the remote nodes, which reconstruct the mirror distribution hierarchy and model data tree based on this data.

The serialization function `getInstanceData` sends all the data needed to reconstruct the model tree: the object identifiers of its children, vertex data for the tree root and vertex indices for the leaf nodes, as well as the bounding sphere and database range of each node. The deserialization function `applyInstanceIdata` retrieves the data in multiple steps, and constructs the model tree on the fly based on this information. It is omitted here for brevity:

```

void VertexBufferDist :: getInstanceData( co :: DataOStream& os )
{
    LBASSERT( _node );
    os << _isRoot ;

    if( _left && _right )
    {

```

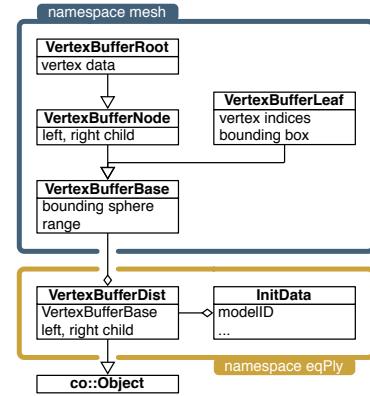


Figure 30: Scene Data in eqPly

8. Equalizer: The eqPly Polygonal Renderer

```

os << _left->getID() << _right->getID();

if( _isRoot )
{
    LBASSERT( _root );
    const mesh::VertexBufferData& data = _root->_data;

    os << data.vertices << data.colors << data.normals << data.indices
        << _root->_name;
}
else
{
    os << lunchbox::UUID::ZERO << lunchbox::UUID::ZERO;

    LBASSERT( dynamic_cast< const mesh::VertexBufferLeaf* >(_node) );
    const mesh::VertexBufferLeaf* leaf =
        static_cast< const mesh::VertexBufferLeaf* >(_node);

    os << leaf->_boundingBox[0] << leaf->_boundingBox[1]
        << uint64_t( leaf->_vertexStart ) << uint64_t( leaf->_indexStart )
        << uint64_t( leaf->_indexLength ) << leaf->_vertexLength;
}

os << _node->_boundingSphere << _node->_range;
}

```

Applications distributing a dynamic scene graph use the frame data instead of the init data as the entry point to their scene graph data structure. Figure 31 shows one possible implementation, where the identifier and version of the scene graph root are transported using the frame data. The scene graph root then serializes and de-serializes his immediate children by transferring their identifier and current version, similar to the static distribution done by eqPly.

The objects are still created by the application, and then registered or mapped with the session to distribute them. When mapping objects in a hierarchical data structure, their type often has to be known to create them. Equalizer does not currently provide object typing, this has to be done by the application, either implicitly in the current implementation context, or by transferring a type identifier. In eqPly, object typing is implicit since it is well-defined which object is mapped in which context.

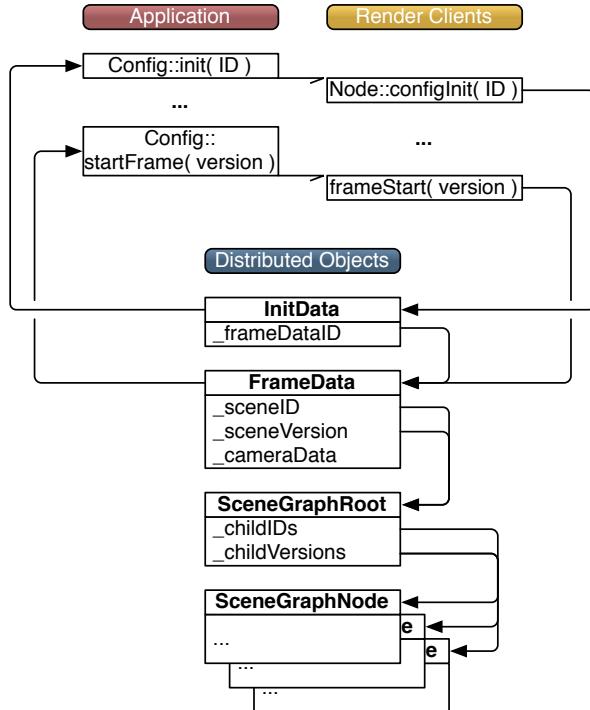


Figure 31: Scene Graph Distribution

8. Equalizer: The eqPly Polygonal Renderer

8.4. Config

The `eq::Config` class is driving the application's rendering, that is, it is responsible for updating the data based on received events, requesting new frames to be rendered and to provide the render clients with the necessary data.

8.4.1. Initialization and Exit

The config initialization happens in parallel, that is, all config initialization tasks are transmitted by the server at once and their completion is synchronized afterwards.

The tasks are executed by the node and pipe threads in parallel. The parent's initialization methods are always executed before any child initialization method. This parallelization allows a speedy startup on large-scale graphics clusters. On the other hand, it means that initialization functions are called even if the parent's initialization has failed. Figure 32 shows a sequence diagram of the config initialization.

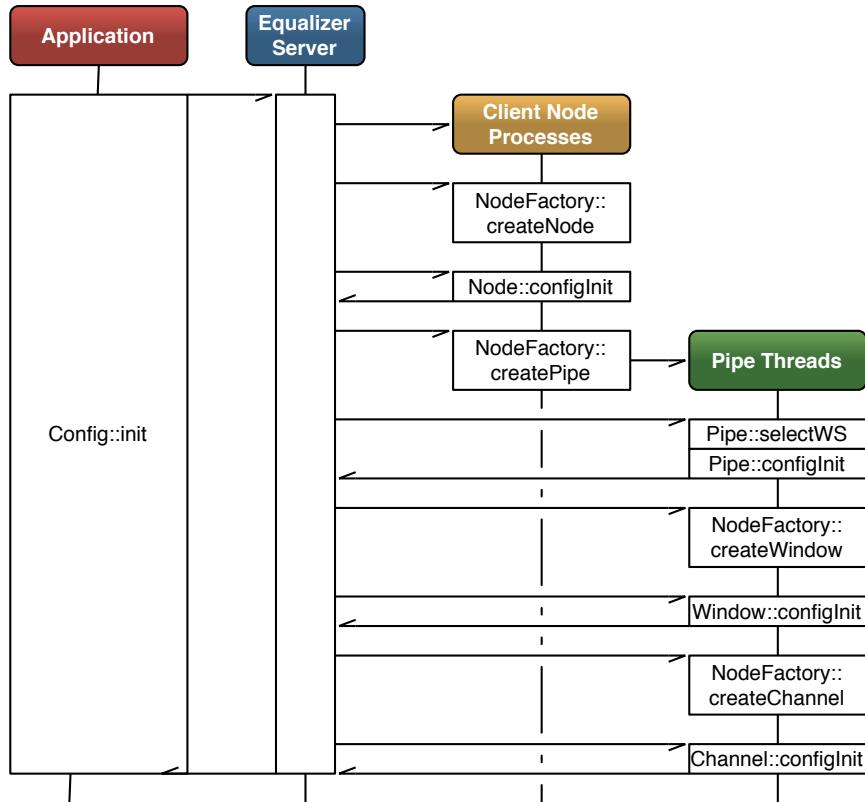


Figure 32: Config Initialization Sequence

The `eqPly::Config` class holds the master versions of the initialization and frame data objects. Both are registered with the `eq::Config`. The configuration forwards the registration to the local client node and augments the object registration for buffered objects.

First, it configures the objects to retain its data `latency+1` commits, which corresponds to the typical use case where objects are committed once per frame. This allows render clients, which often are behind the application process, to map objects with an old version. This does not necessarily translate into increased memory usage, since new versions are only created when the object was dirty during commit.

8. Equalizer: The eqPly Polygonal Renderer

Second, it retains the data for buffered objects data for `latency` frames after their deregistration. This allows to map the object on a render client, even after it has been deregistered on the application node. This does delay the deallocation of the buffered object data by `latency` frames.

The identifier of the initialization data is transmitted to the render client nodes using the `initID` parameter of `eq::Config::init`. The identifier of the frame data is transmitted using the `InitData`.

Equalizer will pass this identifier to all `configInit` calls of the respective objects:

```
bool Config::init()
{
    if( !_animation.isValid( ) )
        _animation.loadAnimation( _initData.getPathFilename( ) );

    // init distributed objects
    if( !_initData.useColor( ) )
        _frameData.setColorMode( COLOR_WHITE );

    _frameData.setRenderMode( _initData.getRenderMode( ) );
    registerObject( &_frameData );
    _frameData.setAutoObsolete( getLatency( ) );

    _initData.setFrameDataID( _frameData.getID( ) );
    registerObject( &_initData );

    // init config
    if( !eq::Config::init( _initData.getID( ) ) )

```

After a successful initialization, the models are loaded and registered for data distribution. When idle, Equalizer will predistribute object data during registration to accelerate the mapping of slave instances. Registering the models after `Config::init` ensures that the render clients are running and can cache the data:

```
_loadModels();
_registerModels();
```

The configuration tries to set up a tracking device for head tracking. Equalizer does not provide extensive support for tracking devices, as this is an orthogonal problem to parallel rendering. Tracking device support has already been solved by a number of implementations, such as VRCO Trackd, VRPN and others, which have proven to be easily integrated with Equalizer. The example code in `eqPly` provides a reference implementation for the integration of such a tracking library. Section 9.6 provides more background on head tracking.

```
if( !_initData.getTrackerPort().empty( ) )
{
    if( !_tracker.init( _initData.getTrackerPort() ) )
        LBWARN << "Failed to initialize _tracker" << std::endl;
    else
    {
        // Set up position of tracking system wrt world space
        // Note: this depends on the physical installation.
        eq::Matrix4f m( eq::Matrix4f::IDENTITY );
        m.scale( 1.f, 1.f, -1.f );
        _tracker.setWorldToEmitter( m );

        m = eq::Matrix4f::IDENTITY;
        m.rotate_z( -M_PI_2 );
        _tracker.setSensorToObject( m );
        LBINFO << "Tracker initialized" << std::endl;
    }
}
```

8. Equalizer: The eqPly Polygonal Renderer

The exit function of the configuration stops the render clients by calling `eq::Config::exit`, and then de-registers the initialization and frame data objects with the session:

```
bool Config :: exit ()
{
    const bool ret = eq::Config :: exit ();
    _deregisterData ();
    _closeAdminServer ();

    // retain model & distributors for possible other config runs, dtor deletes
    return ret;
}
```

8.4.2. Frame Control

The rendering frames are issued by the application. The `eqPly::Config` overrides `startFrame` to update its data, commit a new version of the frame data object, and then requests the rendering of a new frame using the current frame data version. This version is passed to the rendering callbacks and will be used by the rendering threads to synchronize the frame data to the state belonging to the current frame:

```
uint32_t Config :: startFrame ()
{
    _updateData ();
    const eq::uint128_t& version = _frameData.commit ();

    _redraw = false;
    return eq::Config :: startFrame( version );
}
```

Updating the shared data is decomposed in different steps. If a tracker is used, the current head position and orientation is retrieved and passed to Equalizer, which uses the head matrix together with the wall or projection description to compute the view frusta, as explained in Section 9.6:

```
void Config :: _updateData ()
{
    // update head position
    if( _tracker.isRunning() )
    {
        _tracker.update();
        const eq::Matrix4f& headMatrix = _tracker.getMatrix();
        _setHeadMatrix( headMatrix );
    }
}
```

The camera position is recomputed based on the current navigation mode, and the idle state for rendering is set. When idle, `eqPly` performs anti-aliasing to gradually reduce aliasing effects in the rendering. The idle state is tracked by the application and used by the rendering callbacks to jitter the frusta, accumulate and display the results:

```
// update camera
if( _animation.isValid( ) )
{
    const eq::Vector3f& modelRotation = _animation.getModelRotation();
    const CameraAnimation::Step& curStep = _animation.getNextStep();

    _frameData.setModelRotation( modelRotation );
    _frameData.setRotation( curStep.rotation );
    _frameData.setCameraPosition( curStep.position );
}
else
{
    if( _frameData.usePilotMode() )
```

8. Equalizer: The eqPly Polygonal Renderer

```

        _frameData.spinCamera( -0.001f * _spinX, -0.001f * _spinY );
    else
        _frameData.spinModel( -0.001f * _spinX, -0.001f * _spinY, 0.f );
    _frameData.moveCamera( 0.0f, 0.0f, 0.001f * _advance );
}

// idle mode
if( isIdleAA( ) )
{
    LBASSERT( _numFramesAA > 0 );
    _frameData.setIdle( true );
}
else
    _frameData.setIdle( false );
_numFramesAA = 0;
}

```

8.4.3. Event Handling

Events are sent by the render clients to the application using `eq::Config::sendEvent`. At the end of the frame, `Config::finishFrame` calls `Config::handleEvents` to do the event handling. The default implementation processes all pending events by calling `Config::handleEvent` for each of them.

Since `eqPly` uses event-driven execution, the config maintains a dirty state to allow the application to know when a redraw is needed.

The `eqPly` example implements `Config::handleEvent` to provide the various reactions to user input, most importantly camera updates based on mouse events. The camera position has to be handled correctly regarding latency, and is therefore saved in the frame data.

The event handling code reproduced here is just showing the handling of one type of event. A detailed description on how to customize event handling can be found in Section 9.1:

```

case eq::Event::WINDOW_POINTER_WHEEL:
    _frameData.moveCamera( -0.05f * event->data.pointerWheel.yAxis,
                           0.f,
                           0.05f * event->data.pointerWheel.xAxis );
    _redraw = true;
    return true;
}

```

8.4.4. Model Handling

Model data is handled by the config. Models in `eqPly` are static, and therefore the render clients only need to map one instance of the model per node.

Multiple models can be loaded in `eqPly`. A configuration has a default model, stored in `InitData`, and one model per view, stored and distributed using the `View`. The loaded models are evenly distributed over the available views of the configuration, as shown in Figure 16.

The channel acquires the model during rendering from the config, using the model identifier from its current view, or from the frame data if no view is configured.

The per-process config instance maintains the mapped models, and lazily maps new models, which are registered by the application process. Since the model loading may be called concurrently from different pipe render threads, it is protected by a `mutex`:

```

const Model* Config::getModel( const eq::uint128_t& modelID )
{
    if( modelID == eq::UUID::ZERO )

```

8. Equalizer: The eqPly Polygonal Renderer

```

    return 0;

// Protect if accessed concurrently from multiple pipe threads
const eq::Node* node = getNodes().front();
const bool needModelLock = (node->getPipes().size() > 1);
lunchbox::ScopedWrite _mutex( needModelLock ? &_modelLock : 0 );

const size_t nModels = _models.size();
LBASSERT( _modelDist.size() == nModels );

for( size_t i = 0; i < nModels; ++i )
{
    const ModelDist* dist = _modelDist[ i ];
    if( dist->getID() == modelID )
        return _models[ i ];
}

_modelDist.push_back( new ModelDist );
Model* model = _modelDist.back()->loadModel( getApplicationNode(),
                                              getClient(), modelID );
LBASSERT( model );
_models.push_back( model );

return model;
}

```

8.4.5. Layout and View Handling

For layout and model selection, eqPly maintains an active view and canvas. The identifier of the active view is stored in the frame data, which is used by the render client to highlight it using a different background color. The active view can be selected by clicking into a view, or by cycling through all the views using a keyboard shortcut.

The model of the active view can be changed using a keyboard shortcut. The model is view-specific, and therefore the model identifier for each view is stored on the view, which is used to retrieve the model on the render clients.

View-specific data is not limited to a model. Applications can choose to make any application-specific data view-specific, e.g., cameras, rendering modes or annotations. A view is a generic concept for an application-specific view on data, eqPly is simply using different models to illustrate the concept:

```

void Config::switchCanvas()
{
    const eq::Canvases& canvases = getCanvases();
    if( canvases.empty( ) )
        return;

    _frameData.setCurrentViewID( eq::UUID::ZERO );

    if( !_currentCanvas )
    {
        _currentCanvas = canvases.front();
        return;
    }

    eq::CanvasesCIter i = std::find( canvases, _currentCanvas );
    LBASSERT( i != canvases.end( ) );

    ++i;
    if( i == canvases.end( ) )
        _currentCanvas = canvases.front();
    else
        _currentCanvas = *i;
    switchView(); // activate first view on canvas
}

```

8. Equalizer: The eqPly Polygonal Renderer

```

}

void Config :: _switchView()
{
    const eq :: Canvases& canvases = getCanvases();
    if( !_currentCanvas && !canvases.empty( ))
        _currentCanvas = canvases.front();

    if( !_currentCanvas )
        return;

    const eq :: Layout* layout = _currentCanvas->getActiveLayout();
    if( !layout )
        return;

    const View* view = _getCurrentView();
    const eq :: Views& views = layout->getViews();
    LBASSERT( !views.empty( ));

    if( !view )
    {
        _frameData.setCurrentViewID( views.front()->getID( ) );
        return;
    }

    eq :: ViewsCIter i = std :: find( views.begin(), views.end(), view );
    if( i != views.end( ) )
        ++i;
    if( i == views.end( ) )
        _frameData.setCurrentViewID( eq :: UUID :: ZERO );
    else
        _frameData.setCurrentViewID( (*i)->getID( ) );
}

```

The layout of the canvas with the active view can also be dynamically switched using a keyboard shortcut. The first canvas using the layout is found, and then the next layout of the configuration is set on this canvas.

Switching a layout causes the initialization and de-initialization task methods to be called on the involved channels, and potentially windows, pipes and nodes. This operation might fail, which may cause the config to stop running.

Layout switching is typically used to change the presentation of views at runtime. The source code omitted for brevity.

8.5. Node

For each active render client, one `eq::Node` instance is created on the appropriate machine. Nodes are only instantiated on their render client processes, i.e., each process will only have one instance of the `eq::Node` class. The application process might also have a node class, which is handled in exactly the same way as the render client nodes. The application and render clients might use a different node factory, instantiating a different type of `eq::Node` and children.

All dynamic data is multi-buffered in `eqPly`. During initialization, the node relaxes the thread synchronization between the node and pipe threads, unless the configuration file overrides this. Section 9.3 provides a detailed explanation of thread synchronization in Equalizer.

During node initialization the static, per-config data is mapped to a local instance using the identifier passed from `Config::init`. No pipe, window or channel tasks methods are executed before `Node::configInit` has returned:

```

bool Node :: configInit( const eq :: uint128_t& initID )
{
    // All render data is static or multi-buffered, we can run asynchronously

```

8. Equalizer: The eqPly Polygonal Renderer

```

if( getIAttribute( IATTR_THREAD_MODEL ) == eq::UNDEFINED )
    setIAttribute( IATTR_THREAD_MODEL, eq::ASYNC );

if( !eq::Node::configInit( initID ) )
    return false;

Config* config = static_cast< Config*>( getConfig( ) );
if( !config->loadData( initID ) )
{
    setError( ERROR_EQPLY_MAPOBJECT_FAILED );
    return false;
}
return true;
}

```

The actual mapping of the static data is done by the config. The config retrieves the distributed `InitData`. The object is directly unmapped since it is static, and therefore all data has been retrieved:

```

bool Config::loadData( const eq::uint128_t& initDataID )
{
    if( !_initData.isAttached( ) )
    {
        const uint32_t request = mapObjectNB( &_initData, initDataID,
                                             co::VERSION_OLDEST,
                                             getApplicationNode( ) );
        if( !mapObjectSync( request ) )
            return false;
        unmapObject( &_initData ); // data was retrieved, unmap immediately
    }
    else // appNode, _initData is registered already
    {
        LBASSERT( _initData.getID() == initDataID );
    }
    return true;
}

```

8.6. Pipe

All task methods for a pipe and its children are executed in a separate thread. This approach optimizes GPU usage, since all tasks are executed serially and therefore do not compete for resources or cause OpenGL context switches. Multiple GPU run in parallel with each other.

The pipe uses an `eq::SystemPipe`, which abstracts and manages window-system-specific code for the GPU, e.g., an X11 Display connection for the glX pipe system.

8.6.1. Initialization and Exit

Pipe threads are not explicitly synchronized with each other in `eqPly` due to the use of the async thread model. Pipes might be rendering different frames at any given time. Therefore frame-specific data has to be allocated for each pipe thread, which affect only to the frame data.

The frame data is a member variable of the `eqPly::Pipe`, and is mapped to the identifier provided by the initialization data. The initialization in `eq::Pipe` does the GPU-specific initialization, which is window-system-dependent. On AGL the display ID is determined, and on glX the display connection is opened:

```

bool Pipe::configInit( const eq::uint128_t& initID )
{
    if( !eq::Pipe::configInit( initID ) )
        return false;

    Config* config = static_cast<Config*>( getConfig( ) );

```

8. Equalizer: The eqPly Polygonal Renderer

```
const InitData& initData      = config->getInitData();
const eq::uint128_t& frameDataID = initData.getFrameDataID();

return config->mapObject( &_frameData , frameDataID );
}
```

The config exit function is similar to the config initialization. The frame data is unmapped and GPU-specific data is de-initialized by `eq::Config::exit`:

```
bool Pipe::configExit()
{
    eq::Config* config = getConfig();
    config->unmapObject( &_frameData );

    return eq::Pipe::configExit();
}
```

8.6.2. Window System

Equalizer supports multiple window system interfaces, at the moment glX/X11, WGL and AGL/Carbon. Some operating systems, and therefore some Equalizer versions, support multiple window systems concurrently.

Each pipe might use a different window system for rendering, which is determined before `Pipe::configInit` by `Pipe::selectWindowSystem`. The default implementation of `selectWindowSystem` uses the first supported window system.

The `eqPly` examples allows selecting the window system using a command line option. Therefore the implementation of `selectWindowSystem` is overwritten and returns the specified window system, if supported:

```
eq::WindowSystem Pipe::selectWindowSystem() const
{
    const Config* config = static_cast<const Config*>( getConfig() );
    return config->getInitData().getWindowSystem();
}
```

8.6.3. Carbon/AGL Thread Safety

Parts of the Carbon API used for window and event handling in the AGL window system are not thread safe. The application has to call `eq::Global::enterCarbon` before any thread-unsafe Carbon call, and `eq::Global::leaveCarbon` afterwards. These functions should be used only during window initialization and exit, not during rendering. For implementation reasons `enterCarbon` might block up to 50 milliseconds. Carbon calls in the window event handling routine `Window::processEvent` are thread-safe, since the global carbon lock is set in this method. Please contact the Equalizer developer mailing list if you need to use Carbon calls on a per-frame basis.

8.6.4. Frame Control

All task methods for a given frame of the pipe, window and channel entities belonging to the thread are executed in one block, starting with `Pipe::frameStart` and finished by `Pipe::finishFrame`. The frame start callback is therefore the natural place to update all frame-specific data to the version belonging to the frame.

In `eqPly`, the version of the only frame-specific object `FrameData` is passed as the per-frame id from `Config::startFrame` to the frame task methods. The pipe uses this version to update its instance of the frame data to the current version, and unlocks its child entities by calling `startFrame`:

```
void Pipe::frameStart( const eq::uint128_t& frameID , const uint32_t frameNumber )
{
    eq::Pipe::frameStart( frameID , frameNumber );
}
```

8. Equalizer: The eqPly Polygonal Renderer

```
_frameData.sync( frameID );  
}
```

8.7. Window

The Equalizer window abstracts an OpenGL drawable and a rendering context. When using the default window initialization functions, all windows of a pipe share the OpenGL context. This allows reuse of OpenGL objects such as display lists and textures between all windows of one pipe.

The window uses an `eq::SystemWindow`, which abstracts and manages window-system-specific handles to the drawable and context, e.g., an X11 window XID and `GLXContext` for the glX window system.

The window class is the natural place for the application to maintain all data specific to the OpenGL context.

8.7.1. Window System Interface

The particulars of creating a window and OpenGL context depend on the window system used. One can either use the implementation provided by the operating system, e.g., AGL, WGL or glX, or some higher-level toolkit, e.g., Qt.

All window-system specific functionality is implemented by a specialization of `eq::SystemWindow`. The `SystemWindow` class defines the minimal interface to be implemented for a new window system. Each `Window` uses one `SystemWindow` during execution. This separation allows an easy implementation and adaption to another window system or application.

Equalizer provides a generic interface and implementation for the three most common window systems through the OpenGL specific `eq::GLWindow` class: AGL, WGL and glX. The interfaces define the minimal functionality needed to reuse other window system specific classes, for example the AGL, WGL and glX event handlers. The separation of the OpenGL calls from the `SystemWindow` permits to implement window systems which don't use any OpenGL context and therefore to use a different renderer. The implementation derived from these interfaces provides a sample implementation which honors all configurable window attributes.

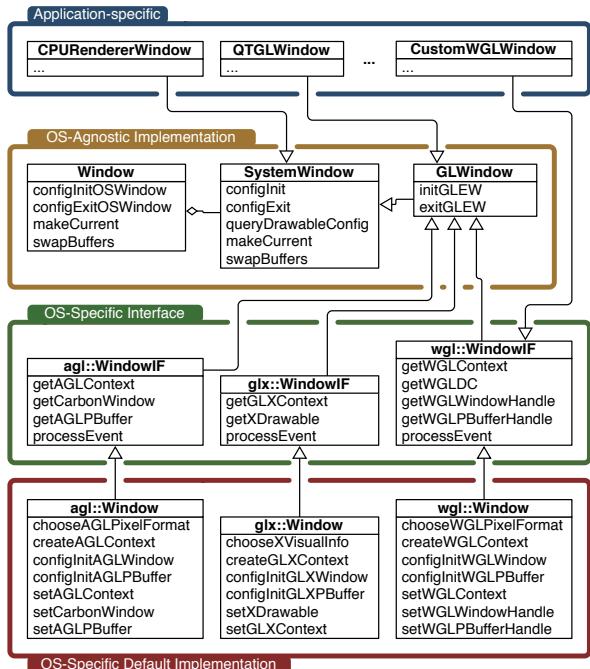


Figure 33: SystemWindow UML Class Hierarchy

8. Equalizer: The eqPly Polygonal Renderer

8.7.2. Initialization and Exit

The initialization sequence uses multiple, override-able task methods. The main task method `configInit` calls first `configInitSystemWindow`, which creates and initializes the `SystemWindow` for this window. The `SystemWindow` initialization code is implementation specific. If the `SystemWindow` was initialized successfully, `configInit` calls `configInitGL`, which performs the generic OpenGL state initialization. The default implementation sets up some typical OpenGL state, e.g., it enables the depth test. Most nontrivial applications do override this task method.

The `SystemWindow` initialization takes into account various attributes set in the configuration file. Attributes include the size of the various frame buffer planes (color, alpha, depth, stencil) as well as other framebuffer attributes, such as quad-buffered stereo, doublebuffering, fullscreen mode and window decorations. Some of the attributes, such as stereo, doublebuffer and stencil can be set to `eq::AUTO`, in which case the Equalizer default implementation will test for their availability and enable them if possible.

For the window-system specific initialization, `eqPly` uses the default Equalizer implementation. The `eqPly` window initialization only overrides the OpenGL-specific initialization function `configInitGL` to initialize a state object and an overlay logo. This function is only called if an OpenGL context was created and made current:

```
bool Window::configInitGL( const eq::uint128_t& initID )
{
    if( !eq::Window::configInitGL( initID ) )
        return false;

    glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, 1 );
    glEnable( GL_CULL_FACE ); // OPT - produces sparser images in DB mode
    glCullFace( GLBACK );

    LBASSERT( !_state );
    _state = new VertexBufferState( getObjectManager( ) );

    const Config* config = static_cast<const Config*>( getConfig( ) );
    const InitData& initData = config->getInitData();

    if( initData.showLogo( ) )
        _loadLogo();

    if( initData.useGLSL( ) )
        _loadShaders();

    return true;
}
```

The state object is used to handle the creation of OpenGL objects in a multipipe, multithreaded execution environment. It uses the object manager of the `eq::Window`, which is described in detail in Section 8.7.3.

The logo texture is loaded from the file system and bound to a texture ID used later by the channel for rendering. A code listing is omitted, since the code consists of standard OpenGL calls and is not Equalizer-specific.

The window exit happens in the reverse order of the initialization. First, `configExitGL` is called to de-initialize OpenGL, followed by `configExitSystemWindow` which de-initializes the drawable and context and deletes the `SystemWindow` allocated in `configInitSystemWindow`.

The window OpenGL exit function of `eqPly` de-allocates all OpenGL objects. The object manager does not delete the object in its destructor, since it does not know if an OpenGL context is still current.

```
bool Window::configExitGL()
{
```

8. Equalizer: The eqPly Polygonal Renderer

```
if( _state && !_state->isShared( ) )
    _state->deleteAll();

delete _state;
_state = 0;

return eq::Window::configExitGL();
}
```

8.7.3. Object Manager

The object manager is, strictly speaking, not a part of the window. It is mentioned here since the `eqPly` window uses an object manager.

The state object in `eqPly` gathers all rendering state, which includes an object manager for OpenGL object allocation.

The object manager (OM) is a utility class and can be used to manage OpenGL objects across shared contexts. Typically one OM is used for each set of shared contexts of a single GPU.

Each `eq::Window` has an object manager with the key type `const void*`, for as long as it is initialized. Each window can have a shared context window. The OM is shared with this shared context window. The shared context window is set by default to the first window of each pipe, and therefore the OM will be shared between all windows of a pipe. The same key is used by all contexts to get the OpenGL name of an object, thus reusing of the same object within the same share group. The method `eq::Window::setSharedContextWindow` can be used to set up a different context sharing.

`eqPly` uses the window's object manager in the rendering code to obtain the OpenGL objects for a given data item. The address of the data item to be rendered is used as the key.

For the currently supported types of OpenGL objects please refer to the API documentation on the Equalizer website. For each object, the following functions are available:

supportsObjects() returns true if the usage for this particular type of objects is supported. For objects available in OpenGL 1.1 or earlier, this function is not implemented.

getObject(key) returns the object associated with the given key, or FAILED.

newObject(key) allocates a new object for the given key. Returns FAILED if the object already exists or if the allocation failed.

obtainObject(key) convenience function which gets or obtains the object associated with the given key. Returns FAILED only if the object allocation failed.

deleteObject(key) deletes the object.

8.8. Channel

The channel is the heart of the application's rendering code, it executes all task methods needed to update the configured views. It performs the various rendering operations for the compounds. Each channel has a set of task methods to execute the clear, draw, readback and assemble stages needed to render a frame.

8. Equalizer: The eqPly Polygonal Renderer

8.8.1. Initialization and Exit

During channel initialization, the near and far planes are set to reasonable values to contain the whole model. During rendering, the near and far planes are adjusted dynamically to the current model position:

```
bool Channel::configInit( const eq::uint128_t& initID )
{
    if( !eq::Channel::configInit( initID ) )
        return false;

    setNearFar( 0.1f, 10.0f );
    _model = 0;
    _modelID = lunchbox::UUID::ZERO;
    return true;
}
```

8.8.2. Rendering

The central rendering routine is `Channel::frameDraw`. This routine contains the application's OpenGL rendering code, which uses the contextual information provided by Equalizer. As most of the other task methods, `frameDraw` is called in parallel by Equalizer on all pipe threads in the configuration. Therefore the rendering must not write to shared data, which is the case for all major scene graph implementations.

In `eqPly`, the OpenGL context is first set up using various `apply` convenience methods from the base Equalizer channel class. Each of the `apply` methods uses the corresponding `get` methods and then calls the appropriate OpenGL functions. It is also possible to just query the values from Equalizer using the `get` methods, and use them to set up the OpenGL state appropriately, for example by passing the parameters to the renderer used by the application.

For example, the implementation for `eq::Channel::applyBuffer` does set up the correct rendering buffer and color mask, which depends on the current eye pass and possible anaglyphic stereo parameters:

```
void eq::Channel::applyBuffer()
{
    glBindBuffer( getReadBuffer() );
    glBindBuffer( getDrawBuffer() );

    const ColorMask& colorMask = getDrawBufferMask();
    glColorMask( colorMask.red, colorMask.green, colorMask.blue, true );
}
```

The contextual information has to be used to render the view as expected by Equalizer. Failure to use certain information will result in incorrect rendering for some or all configurations. The channel render context consist of:

Buffer The OpenGL read and draw buffer as well as color mask. These parameters are influenced by the current eye pass, eye separation and anaglyphic stereo settings.

Viewport The two-dimensional pixel viewport restricting the rendering area within the channel. For correct operations, both `glViewport` and `glScissor` have to be used. The pixel viewport is influenced by the destination channel's viewport definition and compound viewports set for sort-first/2D decompositions.

Frustum The same frustum parameters as defined by `glFrustum`. Typically the frustum used to set up the OpenGL projection matrix. The frustum is influenced by the destination channel's view definition, compound viewports, head matrix and the current eye pass. If the channel has a subpixel parameter, the

8. Equalizer: The eqPly Polygonal Renderer

frustum will be jittered before it is applied. Please refer to Section 9.10 for more information.

Head Transformation A transformation matrix positioning the frustum. This is typically an identity matrix and is used for off-axis frusta in immersive rendering. It is normally used to set up the ‘view’ part of the modelview matrix, before static light sources are defined.

Range A one-dimensional range with the interval [0..1]. This parameter is optional and should be used by the application to render only the appropriate subset of its data for sort-last rendering. It is influenced by the compound range attribute.

The rendering first checks a number of preconditions, such as if the rendering was interrupted by a reset and if the idle anti-aliasing is finished. Then the near and far planes are re-computed, before the rendering context is applied:

```
void Channel::frameDraw( const eq::uint128_t& frameID )
{
    if( stopRendering() )
        return;

    _initJitter();
    if( _isDone() )
        return;

    Window* window = static_cast<Window*>( getWindow() );
    VertexBufferState& state = window->getState();
    const Model* oldModel = _model;
    const Model* model = _getModel();

    if( oldModel != model )
        state.setFrustumCulling( false ); // create all display lists/VBOs

    if( model )
        _updateNearFar( model->getBoundingSphere() );

    eq::Channel::frameDraw( frameID ); // Setup OpenGL state
```

The `frameDraw` method in `eqPly` calls the `frameDraw` method from the parent class, the Equalizer channel. The default `frameDraw` method uses the `apply` convenience functions to setup the OpenGL state for all render context information, except for the range which will be used later during rendering:

```
void eq::Channel::frameDraw( const uint128_t& frameID )
{
    applyBuffer();
    applyViewport();

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
    applyFrustum();

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    applyHeadTransform();
}
```

After the basic view setup, a directional light is configured, and the model is positioned using the camera parameters from the frame data. The camera parameters are transported using the frame data to ensure that all channels render a given frame using the same position.

Three different ways of coloring the object are possible: Using the colors of the mode, using a unique per-channel color to demonstrate the decomposition as shown

8. Equalizer: The eqPly Polygonal Renderer

in Figure 34, or using solid white for anaglyphic stereo. The model colors are per-vertex and are set during rendering, whereas the unique per-channel color is set in `frameDraw` for the whole model:

```

glLightfv( GL_LIGHT0, GL_POSITION, lightPosition );
glLightfv( GL_LIGHT0, GL_AMBIENT, lightAmbient );
glLightfv( GL_LIGHT0, GL_DIFFUSE, lightDiffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, lightSpecular );

glMaterialfv( GL_FRONT, GL_AMBIENT, materialAmbient );
glMaterialfv( GL_FRONT, GL_DIFFUSE, materialDiffuse );
glMaterialfv( GL_FRONT, GL_SPECULAR, materialSpecular );
glMateriali( GL_FRONT, GL_SHININESS, materialShininess );

const FrameData& frameData = _getFrameData();
glPolygonMode( GL_FRONT_AND_BACK,
                frameData.useWireframe() ? GL_LINE : GL_FILL );

const eq::Vector3f& position = frameData.getCameraPosition();

glMultMatrixf( frameData.getCameraRotation().array );
glTranslatef( position.x(), position.y(), position.z() );
glMultMatrixf( frameData.getModelRotation().array );

if( frameData.getColorMode() == COLOR_DEMO )
{
    const eq::Vector3ub color = getUniqueColor();
    glColor3ub( color.r(), color.g(), color.b() );
}
else
    glColor3f( .75f, .75f, .75f );

```

Finally the model is rendered. If the model was not loaded during node initialization, a quad is drawn in its place:

```

if( model )
    _drawModel( model );
else
{
    glNormal3f( 0.f, -1.f, 0.f );
    glBegin( GL_TRIANGLE_STRIP );
        glVertex3f( .25f, 0.f, .25f );
        glVertex3f( -.25f, 0.f, .25f );
        glVertex3f( -.25f, 0.f, -.25f );
        glVertex3f( .25f, 0.f, -.25f );
    glEnd();
}

```

To draw the model, a helper class for view frustum culling is set up, using the view frustum from Equalizer (projection and view matrix) and the camera position (model matrix) from the frame data. The frustum helper computes the six frustum planes from the projection and modelView matrices. During rendering, the bounding spheres of the model are tested against these planes to determine the visibility with the frustum.

Furthermore, the render state from the window and the database range from the channel is obtained. The render state manages display list or VBO allocation:

```

void Channel::_drawModel( const Model* scene )
{

```

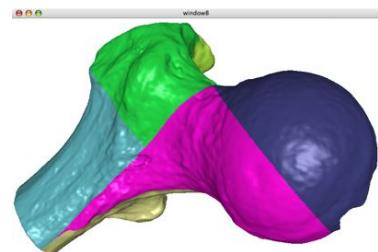


Figure 34: Destination View of a DB Compound using Demonstrative Coloring

8. Equalizer: The eqPly Polygonal Renderer

```

Window* window = static_cast<Window*>( getWindow( ) );
VertexBufferState& state = window->getState();
const FrameData& frameData = _getFrameData();

if( frameData.getColorMode() == COLOR_MODEL && scene->hasColors( ) )
    state.setColors( true );
else
    state.setColors( false );
state.setChannel( this );

// Compute cull matrix
const eq::Matrix4f& rotation = frameData.getCameraRotation();
const eq::Matrix4f& modelRotation = frameData.getModelRotation();
eq::Matrix4f position = eq::Matrix4f::IDENTITY;
position.setTranslation( frameData.getCameraPosition() );

const eq::Frustumf& frustum = getFrustum();
const eq::Matrix4f projection = useOrtho() ? frustum.compute_ortho_matrix():
frustum.compute_matrix();
const eq::Matrix4f& view = getHeadTransform();
const eq::Matrix4f model = rotation * position * modelRotation;

state.setProjectionModelViewMatrix( projection * view * model );
state.setRange( &getRange().start );

const eq::Pipe* pipe = getPipe();
const GLuint program = state.getProgram( pipe );
if( program != VertexBufferState::INVALID )
    glUseProgram( program );

scene->cullDraw( state );

```

The model data is spatially organized in a 3-dimensional kD-tree²⁰ for efficient view frustum culling. When the model is loaded by `Node::configInit`, it is preprocessed into the kD-tree. During this preprocessing step, each node of the tree gets a database range assigned. The root node has the range $[0, 1]$, its left child $[0, 0.5]$ and its right child $[0.5, 1]$, and so on for all nodes in the tree. The preprocessed model is saved in a binary format for accelerating subsequent loading.

The rendering loop maintains a list of candidates to render, which initially contains the root node. Each candidate of this list is tested for full visibility against the frustum and range, and rendered if visible. It is dropped if it is fully invisible or fully out of range. If it is partially visible or partially in range, the children of the node are added to the candidate list.

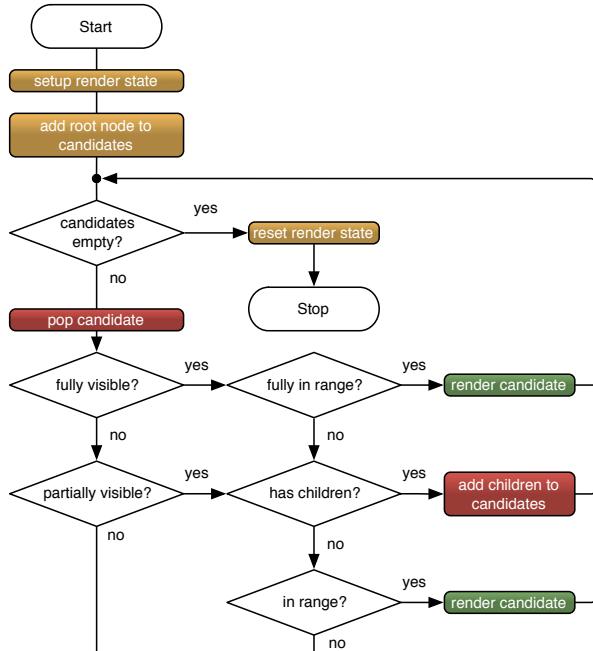


Figure 35: Main Render Loop

²⁰See also <http://en.wikipedia.org/wiki/Kd-tree>

9. Advanced Features

Figure 35 shows a flow chart of the rendering algorithm, which performs efficient view frustum and range culling.

The actual rendering uses display lists or vertex buffer objects. These OpenGL objects are allocated using the object manager. The rendering is done by the leaf nodes, which are small enough to store the vertex indices in a `short` value for optimal performance with VBO's. The leaf nodes reuse the objects stored in the object manager, or create and set up new objects if it was not yet set up. Since one object manager is used per thread (pipe), this allows a thread-safe sharing of the compiled display lists or VBO's across all windows of a pipe.

The main rendering loop is implemented in `VertexBufferRoot::cullDraw()`, and not duplicated here.

8.8.3. Assembly

Like most applications, eqPly uses most of the default implementation of the `frameReadback` and `frameAssemble` task methods. To implement an optimization and various customizations, `frameReadback` is overwritten. eqPly does not need the alpha channel on the destination view. The output frames are flagged to ignore alpha, which allows the compressor to drop 25% of the data during image transfer. Furthermore, compression can be disabled and the compression quality can be changed at runtime to demonstrate the impact of compression on scalable rendering:

```
const FrameData& frameData = _getFrameData();
const eq::Frames& frames = getOutputFrames();
for( eq::FramesCIter i = frames.begin(); i != frames.end(); ++i )
{
    eq::Frame* frame = *i;
    // OPT: Drop alpha channel from all frames during network transport
    frame->setAlphaUsage( false );

    if( frameData.isIdle() )
        frame->setQuality( eq::Frame::BUFFER_COLOR, 1.f );
    else
        frame->setQuality( eq::Frame::BUFFER_COLOR, frameData.getQuality() );

    if( frameData.useCompression() )
        frame->useCompressor( eq::Frame::BUFFER_COLOR, EQ_COMPRESSOR_AUTO );
    else
        frame->useCompressor( eq::Frame::BUFFER_COLOR, EQ_COMPRESSOR_NONE );
}

eq::Channel::frameReadback( frameID );
```

The `frameAssemble` method is overwritten for the use of Subpixel compound with idle software anti-aliasing, as described in Section 9.10.

9. Advanced Features

This section discusses important features not covered by the previous eqPly section. Where possible, code examples from the Equalizer distribution are used to illustrate usage of the specific feature. Its purpose is to provide information on how to address a typical problem or use case when developing an Equalizer-based application.

9.1. Event Handling

Event handling requires flexibility. On one hand, the implementation differs slightly for each operating and window system due to conceptual differences in the specific implementation. On the other hand, each application and widget set has its own model on how events are to be handled. Therefore, event handling in Equalizer is

9. Advanced Features

customizable at any stage of the processing, to the extreme of making it possible to disable all event handling code in Equalizer. In this aspect, Equalizer substantially differs from GLUT, which imposes an event model and hides most of the event handling in `glutMainLoop`.

The default implementation provides a convenient, easily accessible event framework, while allowing all necessary customizations. It gathers all events from all node processes in the main thread of the application, so that the developer only has to implement `Config::processEvent` to update its data based on the preprocessed, generic keyboard and mouse events. It is very easy to use and similar to a GLUT-based implementation.

9.1.1. Threading

Events are received and processed by the pipe thread a window belongs to. For AGL, Equalizer internally forwards the event from the main thread, where it is received, to the appropriate pipe thread. This model allows window and channel modifications which are naturally thread-safe, since they are executed from the corresponding render thread.

9.1.2. Message Pump

To dispatch events, Equalizer ‘pumps’ the native events. On WGL and GLX, this happens on each thread with windows, whereas on AGL it happens on the main thread and on each pipe thread. By default, Equalizer pumps these events automatically for the application in-between executing task methods.

The method `Pipe::createMessagePump` is called by Equalizer during application and pipe thread initialization (before `Pipe::configInit!`) to create a message pump for the given pipe. For AGL, this affects the node thread and pipe threads, since the node thread message pump needs to dispatch the events to the pipe thread event queue. Custom message pumps may be implemented, and it is valid to return no message pump to disable message dispatch for the respective pipe.

If the application disables message pumping in Equalizer, it has to make sure the events are pumped externally, as it often done within external widget sets such as Qt.

9.1.3. Event Data Flow

Events are received by an event handler. The event handler finds the `eq::SystemWindow` associated to the event. It then creates a generic `Event`, which holds the event data in an independent format. The original native event and this generic `Event` form the `SystemWindowEvent`, which is passed to the concrete `SystemWindow` for processing.

The purpose of the `SystemWindow` processing method, `processEvent`, is to perform window-system specific event processing. For example, `AGLWindow::processEvent` calls `aglUpdateContext` whenever a resize event is received. For further, generic processing, the `Event` is passed on to `Window::processEvent`. This `Event` no longer contains the native event.

`Window::processEvent` is responsible for handling the event locally and for translating it into

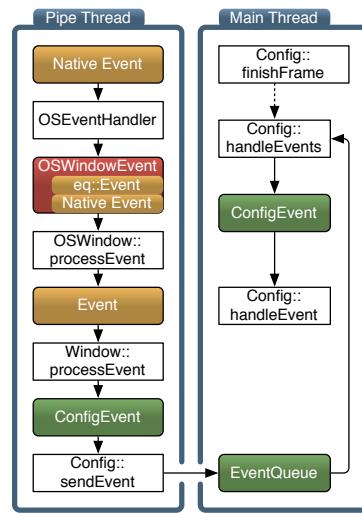


Figure 36: Event Processing

9. Advanced Features

a generic `ConfigEvent`. The config events are send to the application thread using `Config::sendEvent`.

Events send using `Config::sendEvent` are queued in the application thread. After a frame has been finished, `Config::finishFrame` calls `Config::handleEvents`. The default implementation of this method provides non-blocking event processing, that is, it calls `Config::handleEvent` for each queued event. By overriding `handleEvents`, event-driven execution can easily be implemented.

If the event was processed, `processEvent` has to return `true`. If `false` is returned to the event handler, the event will be passed to the previously installed, window-system-specific event handling function.

Figure 36 shows the overall data flow of an event.

9.1.4. Default Implementation

Equalizer provides an `AGLEventHandler`, `GLXEventHandler` and `WGLEventHandler`, which handle events for an `AGLWindowIF`, `GLXWindowIF` and `WGLWindowIF`, respectively. Figure 37 illustrates the class hierarchy for event processing.

The concrete implementation of these window interfaces is responsible for setting up and de-initializing event handling.

Carbon events issued for AGL windows are initially received by the node main thread, and automatically dispatched to the pipe thread. The pipe thread will dispatch the event to the appropriate event handler. One `AGLEventHandler` per window is used, which is created during `AGLWindow::configInit`. The event handler installs a Carbon event handler for all important events. The event handler uses an `AGLWindowEvent` to pass the Carbon `EventRef` to `AGLWindowIF::processEvent`. During window exit, the installed Carbon handler is removed when the window's event handler is deleted. No event handling is set up for PBuffers.

For each GLX window, one `GLXEventHandler` is allocated. FBO's and PBuffers are handled in the same way as window drawables. The event dispatch finds the corresponding event handler for each received event, which is then used to process the event in a fashion similar to AGL. The `GLXWindowEvent` passes the `XEvent` to `GLXWindowIF::processEvent`.

Each `WGLWindow` allocates one `WGLEventHandler` when the window handle is set. The `WGLEventHandler` passes the native event parameters `uMsg`, `wParam` and `lParam` to `WGLWindowIF::processEvent` as part of the `WGLWindowEvent`. No event handling is set up for PBuffers.

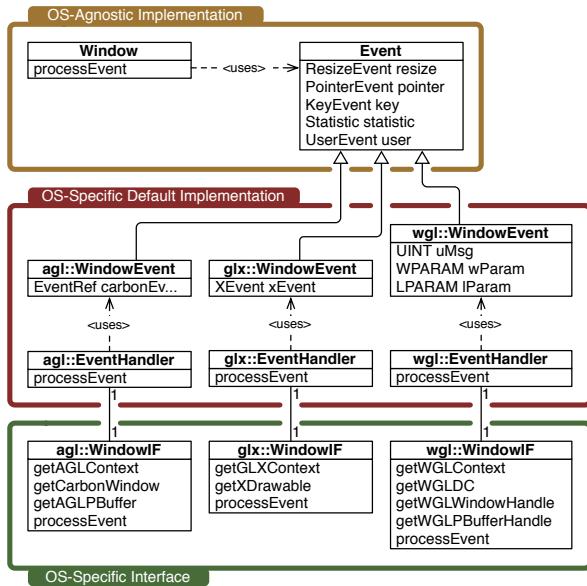


Figure 37: UML Class Diagram for Event Handling

9. Advanced Features

9.1.5. Custom Events in eqPixelBench

The `eqPixelBench` example is a benchmark program to measure the pixel transfer rates from and to the framebuffer of all channels within a configuration. It uses custom config events to send the gathered data to the application. It is much simpler than the `eqPly` example since it does not provide any useful rendering or user interaction.

The rendering routine of `eqPixelBench` in `Channel::frameDraw` loops through a number of pixel formats and types. For each of them, it measures the time to readback and assemble a full-channel image. The format, type, size and time is recorded in a config event, which is sent to the application.

The `ConfigEvent` derives from the `eq::ConfigEvent` structure and has the following definition:

```
struct ConfigEvent : public eq::ConfigEvent
{
public:
    enum Type
    {
        READBACK = eq::Event::USER,
        ASSEMBLE,
        START_LATENCY,
    };

    ConfigEvent()
    {
        size = sizeof( ConfigEvent );
    }

    // channel name is in user event data
    char          formatType[32];
    eq::Vector2i   area;
    uint64_t      dataSizeGPU;
    uint64_t      dataSizeCPU;
    float         msec;
};
```

The `Config::sendEvent` method transmits an `eq::ConfigEvent` or derived class to the application. The `ConfigEvent` has to be a C-type structure, and its `size` member has to be set to the full size of the event to be transmitted. Each event has a type which is used to identify it by the config processing function.

User-defined types start at `eq::ConfigEvent::USER`, and the member variable `ConfigEvent::user` can be used to store up to `EQ_USER_EVENT_SIZE21` bytes. In this space, the channel's name is stored. Additional variables are used to transport the pixel format and type, the size and the time it took for rendering.

On the application end, `Config::handleEvent` uses the `ostream` operator for the derived config event to output these events in a nicely formatted way:

```
bool Config::handleEvent( const eq::ConfigEvent* event )
{
    switch( event->data.type )
    {
        case ConfigEvent::READBACK:
        case ConfigEvent::ASSEMBLE:
        case ConfigEvent::START_LATENCY:
            cout << static_cast< const ConfigEvent*>( event ) << endl;
            return true;

        default:
            return eq::Config::handleEvent( event );
    }
}
```

²¹currently 32 bytes

9.2. Error Handling

All Equalizer entities use an error code to report error conditions during various operations. This error code is reported back from the render processes to the application config instance. The last error is propagated from the failed resource to the config object, where it can be queried by the application using `Config::getError`.

Each error emitted by Equalizer code has a textual description. This string is automatically used for printing an `Error` on an `std::ostream`, and can be queried from the `base::ErrorRegistry`, which is accessible from `base::Global`.

Applications can register additional error codes and strings. The `eVolve` example uses this to report if required OpenGL extensions are missing. The registration of new error codes is not thread-safe, that is, no other thread should use the error registry when it is modified. It is therefore strongly advised to register application-specific errors before `eq::init` and clear them after `eq::exit`:

```
int main( const int argc, char** argv )
{
    // 1. Equalizer initialization
    NodeFactory nodeFactory;
    eVolve::initErrors();

    if( !eq::init( argc, argv, &nodeFactory ) )
    ...
    eq::exit();
    eVolve::exitErrors();
    return ret;
}
```

When the application returns false from a `configInit` task method due to an application-specific error, it should set an error code using `setError`. Application-specific errors can have any value equal or greater than `eq::ERROR_CUSTOM`:

```
/** Defines errors produced by eVolve. */
enum Error
{
    ERROR_EVOLVE_ARB_SHADER_OBJECTS_MISSING = eq::ERROR_CUSTOM,
    ERROR_EVOLVE_EXT_BLEND_FUNC_SEPARATE_MISSING,
    ERROR_EVOLVE_ARB_MULTITEXTURE_MISSING,
    ERROR_EVOLVELOADSHADERS_FAILED,
    ERROR_EVOLVELOADMODEL_FAILED,
    ERROR_EVOLVE_MAPOBJECT_FAILED
};
```

During initialization, `eVolve` may use these error codes:

```
bool Window::configInitGL( const eq::uint128_t& initID )
{
    Pipe*      pipe      = static_cast<Pipe*>( getPipe() );
    Renderer* renderer = pipe->getRenderer();

    if( !renderer )
        return false;

    if( !GLEW_ARB_shader_objects )
    {
        setError( ERROR_EVOLVE_ARB_SHADER_OBJECTS_MISSING );
        return false;
    }
}
```

It is not required to register an error string for each application-specific error. Registering a string will however cause error printing to use this string instead of using only the numerical error value. Applications may also redefine existing error strings to their convenience, e.g., for internationalization purposes:

9. Advanced Features

```

namespace
{
struct ErrorData
{
    const uint32_t code;
    const std::string text;
};

ErrorData _errors[] = {
    { ERROR_EVOLVE_ARB_SHADER_OBJECTS_MISSING,
        "GL_ARB_shader_objects_extension_missing" },
    { ERROR_EVOLVE_EXT_BLEND_FUNC_SEPARATE_MISSING,
        "GL_ARB_shader_objects_extension_missing" },
    { ERROR_EVOLVE_ARB_MULTITEXTURE_MISSING,
        "GL_ARB_shader_objects_extension_missing" },
    { ERROR_EVOLVE_LOADSHADERS_FAILED,
        "Can't_load_shaders" },
    { ERROR_EVOLVE_LOADMODEL_FAILED,
        "Can't_load_model" },
    { ERROR_EVOLVE_MAPOBJECT_FAILED,
        "Mapping_data_from_application_process_failed" },
    { 0, "" } // last!
};
}

void initErrors()
{
    co::ErrorRegistry& registry = co::Global::getErrorRegistry();

    for( size_t i=0; _errors[i].code != 0; ++i )
        registry.setString( _errors[i].code, _errors[i].text );
}

void exitErrors()
{
    co::ErrorRegistry& registry = co::Global::getErrorRegistry();

    for( size_t i=0; _errors[i].code != 0; ++i )
        registry.eraseString( _errors[i].code );
}

```

9.3. Thread Synchronization

Equalizer applications use multiple, potentially asynchronous execution threads. The default execution model is focused on making the porting of existing applications as easy as possible, as described in Section 6.3. The default, per-node thread synchronization provided by Equalizer can be relaxed by advanced applications to gain better performance through higher asynchronicity.

9.3.1. Threads

The application or node main thread is the primary thread of each process and executes the `main` function. The application and render clients initialize the local node for communications with other nodes, including the server, using `Client::initLocal`.

During this initialization, Equalizer creates and manages two threads for communication, the receiver thread and the command thread. Normally no application code is executed from these two threads.

The receiver thread manages the network connections to other nodes and receives data. It dispatches the received data either to the application threads, or to the command thread.

The command thread processes internal Equalizer request from other nodes, for example during `co::Object` mapping. In some special cases the command thread exe-

9. Advanced Features

cutes application code, for example when a remote node maps a static or unbuffered object, `Object::getInstanceData` is called from the command thread.

The receiver and command thread are terminated when the application stops network communications using `Client::exitLocal`.

During config initialization, one pipe thread is created for each pipe. The pipe threads execute all render task methods for this pipe, and therefore executes the application's rendering code. All pipe threads are terminated during `Config::exit`.

Since a layout switch on a canvas may involve different resources, pipe threads may be started and stopped dynamically at runtime. Before such an update, Equalizer will always finish all pending rendering operations to ensure that all resources are idle. The rest of this section discusses the thread synchronization between the main thread and the pipe threads.

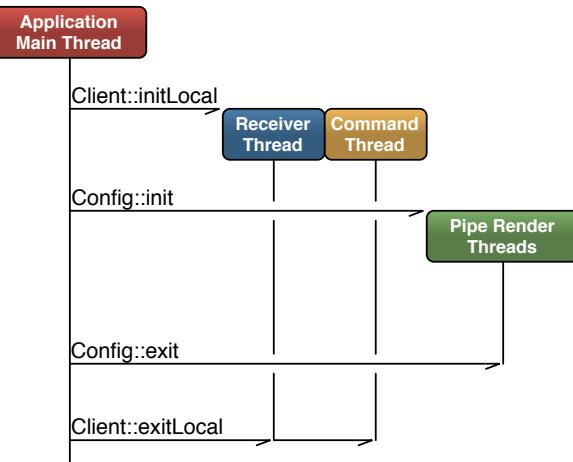


Figure 38: Threads within one Node Process

9.3.2. Thread Synchronization Models

Equalizer supports three threading models, which can be set programmatically by the application or through the configuration file format. Applications typically hard-code their threading model. The file format is commonly used to change the threading model for benchmarking and experimentation.

The following thread synchronization models are implemented:

ASYNC: No synchronization happens between render threads, except for synchronizing the finish of frame `current-latency`. The `eqPly` and `eVolve` Equalizer examples use this threading model, by setting it in `Node::configInit`. This synchronization model provides the best performance and should be used by all applications which multi-buffer all non-static, frame-specific data.

DRAW_SYNC: Additionally to the synchronization of the async thread model, all local render threads are synchronized, so that the draw operations happen synchronously with the node main loop. Compositing, swap barriers and buffer swaps happen asynchronously. This model allows to use the same database for rendering, and safe modifications of this database are possible from the node thread, since the pipe threads do not execute any rendering tasks between frames. This is the default threading model. This threading model should be used by applications which keep one copy of the scene graph per node.

LOCAL_SYNC: Additionally to the synchronization of the async thread model, all local frame operations, including readback, assemble and swap buffer are synchronized with the node main loop. This threading model should be used by applications which need to access non-buffered, frame-specific data after rendering, e.g., during `Channel::frameAssemble`.

9. Advanced Features

Figure 39 illustrates the synchronization and task execution for the thread synchronization models. Please note that the thread synchronization synchronizes all pipe render threads on a **single** node with the node's main thread. The per-node frame synchronization does not break the asynchronous execution across nodes.

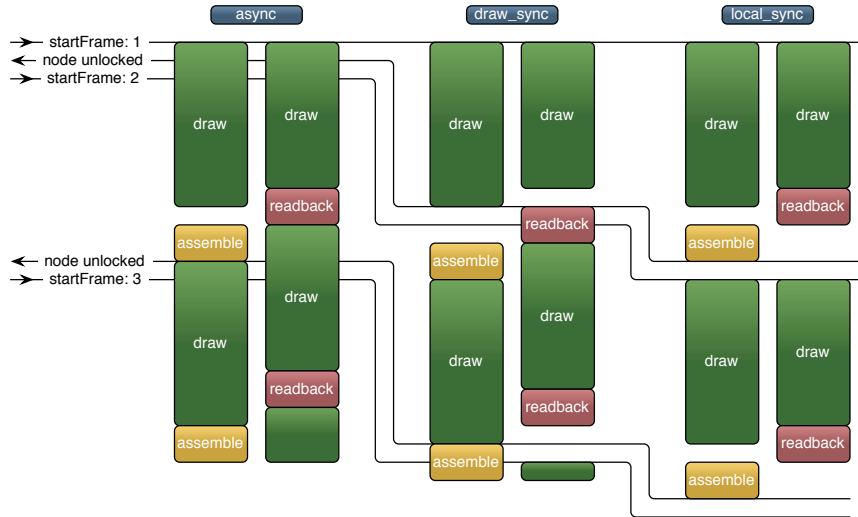


Figure 39: Async, draw_sync and local_sync Thread Synchronization Models

An implication of the **draw_sync** and **local_sync** models is that the application node cannot issue a new frame until it has completed all its draw tasks. For larger cluster configurations it is therefore advisable to assign only assemble operations to the application node. If the machine running the application process should also contribute for rendering, a second node may be used to perform off-screen draw and readback operations for the application node process.

9.3.3. DPlex Compounds

DPlex decomposition requires multiple frames to be rendered concurrently. Applications using the thread synchronization models **draw_sync** (the default) or **local_sync** have to use one render client process per GPU to benefit from DPlex task decomposition.

Non-threaded pipes should not be used for DPlex source and destination channels. Applications using the **local_sync** thread model cannot benefit from DPlex if the application node uses a DPlex source or destination channel.

Applications using the **async** thread synchronization model can fully profit from DPlex using multiple render threads on a multi-GPU system. For these applications, all render threads run asynchronously and can render different frames at the same time.

Synchronizing the draw operations between multiple pipe render threads, and potentially the application thread, breaks DPlex decomposition. At any given time, only one frame can be rendered from the same process. The speedup of DPlex relies however on the capability to render different frames concurrently.

If one process per GPU is configured, draw-synchronous applications can scale the performance using DPlex compounds. The processes are not synchronized with each other, since each process keeps its own version of the scene data.

9.3.4. Thread Synchronization in Detail

The application has extended control over the task synchronization during a frame. Upon `Config::startFrame`, Equalizer invokes the `frameStart` task methods of the various entities. The entities unlock all their children by calling `startFrame`, e.g., `Node::frameStart` has to call `Node::startFrame` to unlock the pipe threads. Note that certain `startFrame` calls, e.g., `Window::startFrame`, are currently empty since the synchronization is implicit due to the sequential execution within the thread.

Figure 40 illustrates the local frame synchronization. Each entity uses `waitForFrameStarted` to block on the parent's `startFrame`, e.g., `Pipe::frameStart` calls `Node::waitForFrameStarted` to wait for the corresponding `Node::startFrame`. This explicit synchronization allows to update non-critical data before synchronizing with `waitForFrameStarted`, or after unlocking using `startFrame`. Figure 40 illustrates this synchronization model.

At the end of the frame, two similar sets of synchronization methods are used. The first set synchronizes the local execution, while the second set synchronizes the global execution.

The local synchronization consists of `releaseFrameLocal` to unlock the local frame, and of `waitForFrameLocal` to wait for the unlock. For the default synchronization model `sync_draw`, Equalizer uses the task method `frameDrawFinish` which is called on each resource after the last `Channel::frameDraw` invocation for this frame. Consequently, `Pipe::frameDrawFinish` calls `Pipe::releaseFrameLocal` to signal that it is done drawing the current frame, and `Node::frameDrawFinish` calls `Pipe::waitForFrameLocal` for each of its pipes to block the node thread until the current frame has been drawn.

The second, global synchronization is used for the frame completion during `Config::finishFrame`, which causes `frameFinish` to be called on all entities, passing the oldest frame number, i.e., frame `current-latency`. The `frameFinish` task methods have to call `releaseFrame` to signal that the entity is done with the frame. The release causes the parent's `frameFinish` to be invoked, which is synchronized internally. Once all `Node::releaseFrame` have been called, `Config::finishFrame` returns.

Figure 41 outlines the synchronization for the application, node and pipe classes for an application node and one render client when using the default `draw_sync` thread model. Please note that `Config::finishFrame` does block until the current frame has been released locally and until the frame `current - latency` has been released by all nodes. The window and channel synchronization are similar and omitted for simplicity.

It is absolutely vital for the execution that `Node::startFrame` and `Node::releaseFrame` are called, respectively. The default implementations of the node task methods do take care of that.

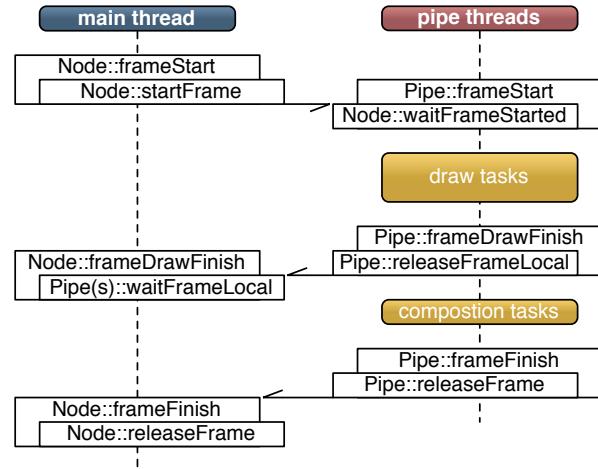


Figure 40: Per-Node Frame Synchronization

9. Advanced Features

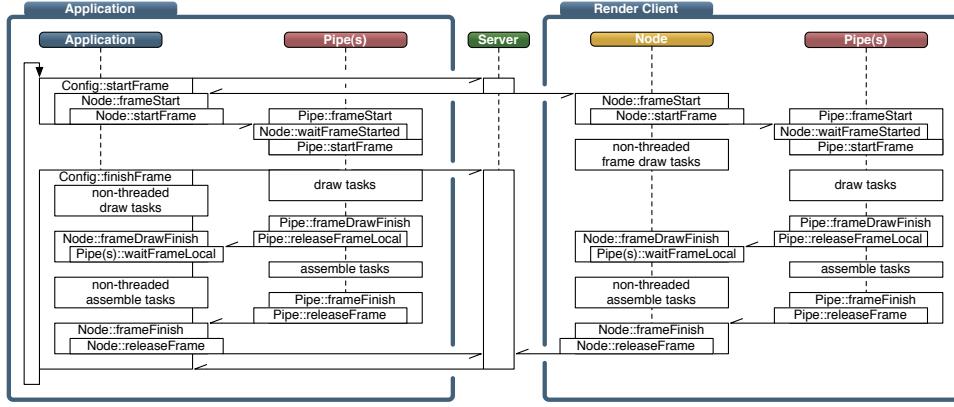


Figure 41: Synchronization of Frame Tasks

9.4. OpenGL Extension Handling

Equalizer uses GLEW²² for OpenGL extension handling, particularly the GLEW_MX implementation providing multi-context support.

GLEW uses a function table called `GLEWContext` to store function pointers for non-standard OpenGL functions. This table is initialized once per OpenGL context by retrieving the pointers through a system-specific method. OpenGL functions are then redefined using a macro which retrieves the `GLEWContext` using `glewGetContext`, looking up the corresponding function pointer in this table and calling the function. Equalizer applications can disable these GLEW macros by defining `EQ_IGNORE_GLEW` before including `eq/eq.h`.

Each `eq::GLWindow`, which is the `SystemWindow` base class for all OpenGL system windows, has a `GLEWContext`. This context can be obtained by using `glewGetContext` on the `eq::Window` or `eq::Channel`. Equalizer (re-)initializes the GLEW context whenever a new OpenGL context is set.

Extended OpenGL functions called from a `GLWindow`, window or channel instance can be called directly. GLEW will call the object's `glewGetContext` to obtain the correct context:

```
void eqPly::Channel::_drawModel( const Model* model )
{
    ...
    glUseProgram( program );
    ...
}
```

Functions called from another place need to define a macro or function `glewGetContext` that returns the pointer to the `GLEWContext` of the appropriate window:

```
// state has GLEWContext* from window
#define glewGetContext state.glewGetContext

/* Set up rendering of the leaf nodes. */
void VertexBufferLeaf::setupRendering( VertexBufferState& state,
                                         GLuint* data ) const
{
    ...
    glBindBuffer( GL_ARRAY_BUFFER, data[VERTEX_OBJECT] );
    glBufferData( GL_ARRAY_BUFFER, _vertexLength * sizeof( Normal ),
                  &_globalData.normals[_vertexStart], GL_STATIC_DRAW );
    ...
}
```

²²<http://glew.sourceforge.net>

9. Advanced Features

The WGL and GLX pipe manage a WGLEWContext and GLXEWContext, respectively. These context are useful if extended wgl or glX functions are used during window initialization. The GLEW context structures are initialized using a temporary OpenGL context, created using the proper display device of the pipe.

9.5. Advanced Window Initialization

This section explains window initialization in detail. It discusses in detail the handling of the different window systems. Window systems are integrated through the `SystemWindow` interface, which externalizes the details of the windowing API from the `eq::Window` implementation and facilitates the integration with new windowing API's and custom applications.

Equalizer provides sample implementations for the supported window systems. These sample implementations are intended to be sub-classed and various steps in the window initialization can be overwritten and customized.

An application typically chooses to subclass the sample implementation if only minor tweaks are needed for integration. For major changes or new window systems, it is often easier to subclass directly from `SystemWindow` or `GLWindow` and implement the abstract methods of this interface class.

The method `Window::configInitSystemWindow` is used to instantiate and initialize the `SystemWindow` implementation during config initialization. After a successful `SystemWindow` initialization, `Window::configInitGL` is called for the generic OpenGL state setup.

Since window initialization is notoriously error-prone and hard to debug in an distributed application, the sample implementation propagates the reason for errors from the render clients back to the application. The Pipe and Window classes have a `setError` method, which is used to set an error code. This string is passed to the Config instance on the application node, where it can be retrieved using `getError`. The Section 9.2 explains error handling in more detail.

The sample implementations `agl::Window`, `glx::Window` and `wgl::Window` all have similar, override-able methods for all sub-tasks. This allows partial customization, without the need of rewriting tedious window initialization code, e.g., the OpenGL pixel format selection.

9.5.1. Drawable Configuration

OpenGL drawables have a multitude of buffer modes. A drawable might be single-buffered, double-buffered or quad-buffered, have auxiliary image planes such as stencil, accumulation and depth buffer or multisampling.

The OpenGL drawable is configured using window attributes. These attributes are used by the method choosing the pixel format (or visual in X11 speak) to select the correct drawable configuration.

Window attributes can either be configured through the configuration file (see Appendix B), or programmatically. In the configuration file, modes are selected which are not application-specific, for example stereo formats for active stereo displays.

Applications which require certain drawable attributes can set the corresponding window attribute hint during window initialization. The Equalizer volume rendering example, `eVolve`, is such an example. It does need alpha planes for rendering and compositing. The window initialization of `eVolve` sets the attribute before calling the default initialization method of Equalizer:

```
bool Window::configInit( const eq::uint128_t& initID )
{
    // Enforce alpha channel, since we need one for rendering
```

9. Advanced Features

```
    setIAttribute( IATTR_PLANES_ALPHA, 8 );

    return eq::Window::configInit( initID );
}
```

9.5.2. AGL Window Initialization

AGL initialization happens in three steps: choosing a pixel format, creating the context and creating a drawable.

Most AGL and Carbon calls are not thread-safe. The Equalizer methods calling these functions use `Global::enterCarbon` and `Global::leaveCarbon` to protect the API calls. Please refer to Section 8.6.3 for more details.

The pixel format is chosen based on the window's attributes. Some attributes set to auto, e.g., stereo, cause the method first to request the feature and then to back off and retry if it is not available. The pixel format returned by `chooseAGLPixelFormat` has to be destroyed using `destroyAGLPixelFormat`. When no matching pixel format is found, `chooseAGLPixelFormat` returns 0 and the AGL window initialization returns with a failure.

The context creation also uses the global Carbon lock. Furthermore, it sets up the swap buffer synchronization with the vertical retrace, if enabled by the corresponding window attribute hint. Again the window initialization fails if the context could not be created.

The drawable creation method `configInitAGLDrawable` calls either `configInitAGLFullscreen`, `configInitAGLWindow` or `configInitAGLPBuffer`

The top-level AGL window initialization code therefore looks as follows:

```
bool Window::configInit()
{
    AGLPixelFormat pixelFormat = chooseAGLPixelFormat();
    if( !pixelFormat )
        return false;

    AGLContext context = createAGLContext( pixelFormat );
    destroyAGLPixelFormat( pixelFormat );
    setAGLContext( context );

    if( !context )
        return false;

    makeCurrent();
    initGLEW();
    return configInitAGLDrawable();
}
```

9.5.3. GLX Window Initialization

GLX initialization is very similar to AGL initialization. Again the steps are: choose frame buffer configuration (pixel format), create OpenGL context and then create drawable. The only difference is that the data returned by `chooseGLXFBConfig` has to be freed using `XFree`:

```
bool Window::configInit()
{
    GLXFBConfig* fbConfig = chooseGLXFBConfig();
    if( !fbConfig )
    {
        setError( ERROR_SYSTEMWINDOW_PIXELFORMAT_NOTFOUND );
        return false;
    }

    GLXContext context = createGLXContext( fbConfig );
```

9. Advanced Features

```
setGLXContext( context );
if( !context )
{
    XFree( fbConfig );
    return false;
}

const bool success = configInitGLXDrawable( fbConfig );
XFree( fbConfig );

if( !success || !_xDrawable )
{
    if( getError() == ERROR_NONE )
        setError( ERROR_GLXWINDOW_NO_DRAWABLE );
    return false;
}

makeCurrent();
initGLEW();
_initSwapSync();
if( getIAtribute( eq::Window::IATTR_HINT_DRAWABLE ) == FBO )
    configInitFBO();

return success;
}
```

9.5.4. WGL Window Initialization

The WGL initialization requires another order of operations compared to AGL or GLX. The following functions are used to initialize a WGL window:

1. `initWGLAffinityDC` is used to set up an affinity device context, which might be needed for window creation. The WGL window tracks potentially two device context handles, one for OpenGL context creation (the affinity DC), and one for `swapBuffers` (the window's DC).
2. `chooseWGLPixelFormat` chooses a pixel format based on the window attributes. If no device context is given, it uses the system device context. The chosen pixel format is set on the passed device context.
3. `configInitWGLDrawable` creates the drawable. The device context passed to `configInitWGLDrawable` is used to query the pixel format and is used as the device context for creating a PBuffer. If no device context is given, the display device context is used. On success, it sets the window handle. Setting a window handle also sets the window's device context.
4. `createWGLContext` creates an OpenGL rendering context using the given device context. If no device context is given, the window's device context is used. This function does not set the window's OpenGL context.

The full `configInitWGL` task method, including error handling and cleanup, looks as follows:

```
bool Window::configInit()
{
    if( !initWGLAffinityDC( ) )
    {
        setError( ERROR_WGL_CREATEAFFINITYDC_FAILED );
        return false;
    }

    int pixelFormat = chooseWGLPixelFormat();
    if( pixelFormat == 0 )
```

9. Advanced Features

```
{  
    exitWGLAffinityDC();  
    return false;  
}  
  
if( !configInitWGLDrawable( pixelFormat ))  
{  
    exitWGLAffinityDC();  
    return false;  
}  
  
if( !_wglDC )  
{  
    exitWGLAffinityDC();  
    setWGLDC( 0, WGLDC_NONE );  
    setError( ERROR_WGLWINDOWNODRAWABLE );  
    return false;  
}  
  
HGLRC context = createWGLContext();  
if( !context )  
{  
    configExit();  
    return false;  
}  
  
setWGLContext( context );  
makeCurrent();  
initGlew();  
_initSwapSync();  
if( getIAttribute( eq::Window::IATTR_HINT_DRAWABLE ) == FBO )  
    return configInitFBO();  
  
return true;  
}
```

9.6. Stereo and Immersive Rendering

Equalizer fully supports immersive rendering for Virtual Reality, supporting multiple tracked observers, head-mounted displays (HMD) and flexible focus distance support. Equalizer does not support tracking devices directly. Other open source and commercial implementations cover this functionality well and are easily integrated with Equalizer.

An Equalizer configuration contains a number of observers. Each observer represents one tracked entity. Most immersive configurations only have one observer. It is possible to support multiple tracked viewers in the same configuration, e.g., to use two HMDs.

Each observer has its own head matrix, eye separation and focus information for tracking. In `eqPly`, all observers are updated with the same data. A typical application would assign different tracking data to different observers.

The `eqPly` example contains basic support for head tracking to illustrate how applications use these features.

9.6.1. Stereo Rendering

Figure 42(a) illustrates a monoscopic view frustum. The viewer is positioned at the origin of the global coordinate system, and the frustum is completely symmetric. This is the typical view frustum for non-stereoscopic applications.

In stereo rendering, the scene is rendered twice, with the two frusta 'moved' by the distance between the eyes, as shown in Figure 42(b). The stereo frusta are asymmetric, offset by half of the eye base. The stereo convergence plane is the

9. Advanced Features

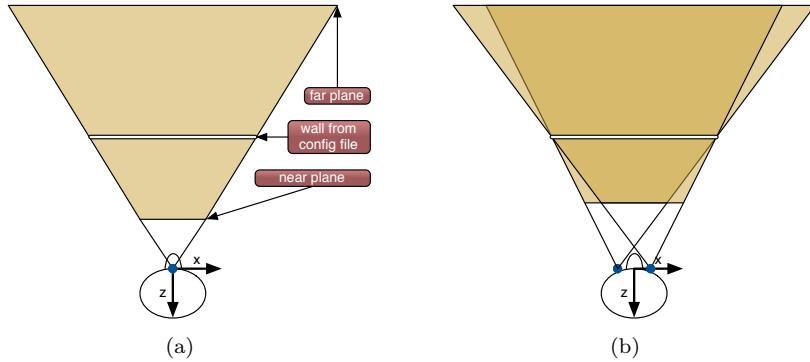


Figure 42: Monoscopic (a) and Stereoscopic (b) Frusta

same as the projection surface, unless specified otherwise using the focus distance API (see Section 9.6.3).

Note that while stereo rendering is often implemented using the simpler toe-in method, Equalizer implements the correct approach using asymmetric frusta.

9.6.2. Immersive Rendering

In immersive visualization, the observer is tracked in and the view frusta are adapted to the viewer's position and orientation, as shown in Figure 43(a). The transformation *origin* → *viewer* is set by the application using `Observer::setHeadMatrix`, which is used by the server to compute the eye positions, and consequently the view frusta.

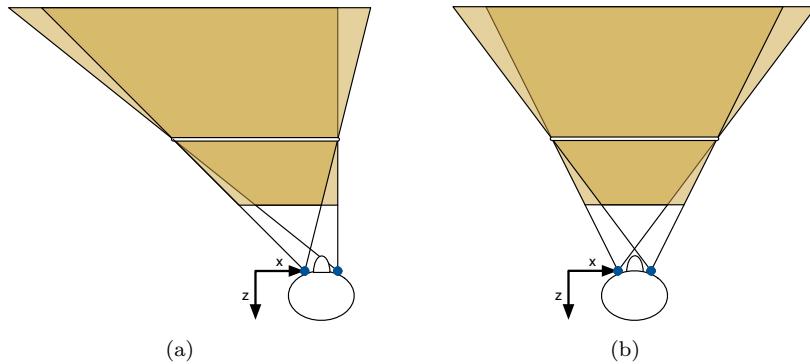


Figure 43: Tracked (a) and HMD (b) Immersive Frusta

Projection surfaces which are not X/Y-planar create frusta which are not oriented along the Z axis, as shown in Figure 44(a). These frusta are positioned using the channel's head transformation, which can be retrieved using `Channel::getHeadTransform`.

For head-mounted displays (HMD), the tracking information is used to move the frusta with the observer, as shown in Figure 43(b). This results in different projections compared to normal tracking with fixed projection screens. This difference is transparent to Equalizer applications, only the configuration file uses a different wall type for HMD's.

9. Advanced Features

9.6.3. Focus Distance

The focus distance, also called stereo convergence, is the Z distance in which the left and right eye frusta converge. A plane parallel to the near plane positioned in the focus distance creates the same 2D image for both eyes.

The frustum calculation up to Equalizer version 1.0 places the stereo convergence on the projection surface, as shown in Figure 44(a). The focus distance is therefore the distance between the origin and the middle of the projection surface. This is how almost all Virtual Reality software handles the focal plane.

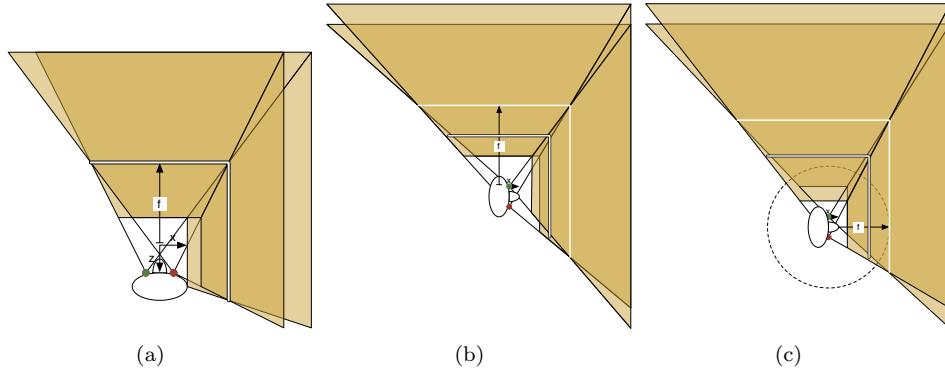


Figure 44: Fixed(a) and dynamic focus distance relative to origin(b) and observer(c)

In Equalizer 1.2 and later the focus distance and mode can be configured and changed at runtime for each observer and on the config for untracked views. This allows applications to expose the focal plane to the user, or to automatically calculate it based on the scene and view direction.

The default focus mode `fixed` implements the algorithm used by Equalizer 1.0. This mode ignores the focus distance setting. The focus modes `relative_to_origin` and `relative_to_observer` use the focus distance to dynamically change the stereo convergence.

The focus distance calculation relative to the origin allows to change the focus independent of the observer position by separating the projection surface from the stereo convergence plane upon which the observer focuses itself. The convergence plane of the first wall in the negative Z direction is moved to be at the given focus distance, as shown in Figure 44(b). All other walls are moved by the same relative amount. The movement is made from the view of the central eye, thus leaving the mono frustum unchanged. Figure 44(b) shows the new logical 'walls' used for frustum calculations in white, while the physical projection from Figure 44(a) is still visible.

The focus distance calculation relative to the observer is similar to the origin algorithm, but it keeps the closest wall in the observer's view direction at the given focus distance, as shown in Figure 44(c). When the observer moves forward, the focal plane moves forward as well. Consequently, when the observer looks in a different direction, a different object in the scene is focused, as indicated by the dotted circle in Figure 44(c).

Figure 45 shows an example for the three focus distance modes. The configured wall is one meter behind the origin, the model two meters behind and the observer is half a meter in front of the origin. The focus distance was set to two meters.

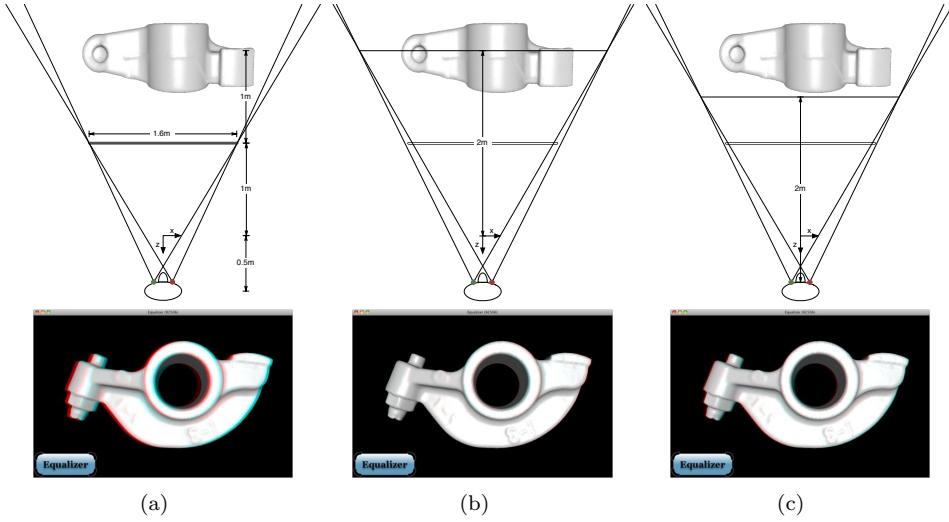


Figure 45: Fixed(a), relative to origin(b) and observer(c) focus distance examples

9.6.4. Application-specific Scaling

All Equalizer units in the configuration file and API are configured in meters. This convention allows the reuse of configurations across multiple applications. When visualizing real-world objects, e.g., for architectural visualizations, it guarantees that they appear realistic and full immersion into the visualization can be achieved.

Certain applications want to visualize objects in immersive environments in a scale different from their natural size, e.g., astronomical simulations. Each model, and therefore each view, might have a different scale. Applications can declare this scale as part of the `eq::View`, which will be applied to the virtual environment by Equalizer. Common metric model scales are provided as constants.

9.7. Layout API

The Layout API provides an abstraction for render surfaces (Canvas and Segment) and the arrangement of rendering areas on them (Layout and View). Its functionality has been described in Section 3.8 and Section 3.9. This section focuses on how to use the Layout API programmatically.

The application has read access to all canvases, segment, layouts and views of the configuration. The render client has access to the current view in the channel task methods. The layout entities can be sub-classed using the `NodeFactory`. Currently the layout of a canvas, the frustum and stereo mode of a view as well as the view's user data object can be changed at runtime.

9.7.1. Subclassing and Data Distribution

Layout API entities (Canvas, Segment, Layout, View) are sub-classed like all other Equalizer entities using the `NodeFactory`. Equalizer registers the master instance of these entities on the server node. Mutable parameters, e.g., the active layout of a canvas, are distributed using slave object commits (cf. Section 10.4.4). Application-specific data can be attached to a view using a distributable `UserData` object. Figure 46 shows the UML class hierarchy for the `eqPly::View`.

The render clients can access a slave instance of the view using `Channel::getView`. When called from one of the frame task methods, this method will return the view

9. Advanced Features

of the current destination channel for which the task method is executed. Otherwise it returns the channel's native view, if it has one. Only destination channels of an active canvas have a native view.

Equalizer commits dirty layout entities at the beginning of each `Config::startFrame`, and synchronizes the slave instances on the render clients correctly with the current frame.

The most common entity to subclass is the View, since the application often amends it with view-specific application data. A view might have a distributable user data object, which has to inherit from `co::Object` to be committed and synchronized with the associated view.

Externalizing the data distribution in a `UserData` object instance of a view. The server can then handle the identifier and version information for each slave instance.

In `eqPly`, the application-specific data is the model identifier and the number of anti-aliasing steps to be rendered when idle. This data is distributed by a `View::Proxy` object which serves as the view's user data. The proxy defines the necessary dirty bits and serializes the data:

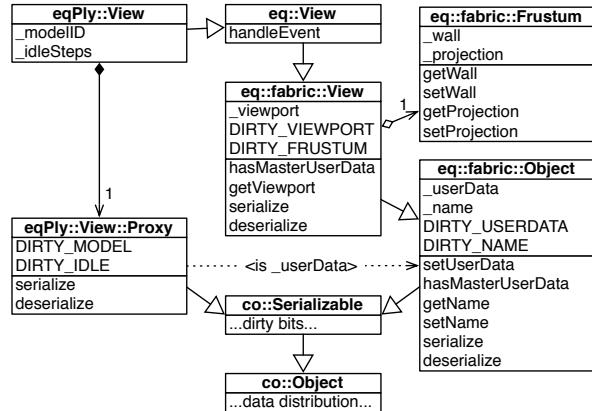


Figure 46: UML Hierarchy of `eqPly::View`

In eqPly, the application-specific data is the model identifier and the number of anti-aliasing steps to be rendered when idle. This data is distributed by a View::Proxy object which serves as the view's user data. The proxy defines the necessary dirty bits and serializes the data:

```

/** The changed parts of the view. */
enum DirtyBits
{
    DIRTY_MODEL = co::Serializable::DIRTY_CUSTOM << 0,
    DIRTY_IDLE   = co::Serializable::DIRTY_CUSTOM << 1
};

void View::Proxy::serialize( co::DataOutputStream& os, const uint64_t dirtyBits )
{
    if( dirtyBits & DIRTY_MODEL )
        os << _view->.modelID;
    if( dirtyBits & DIRTY_IDLE )
        os << _view->.idleSteps;
}

void View::Proxy::deserialize( co::DataStream& is, const uint64_t dirtyBits )
{
    if( dirtyBits & DIRTY_MODEL )
        is >> _view->.modelID;
    if( dirtyBits & DIRTY_IDLE )
    {
        is >> _view->.idleSteps;
        if( isMaster( ) )
            setDirty( DIRTY_IDLE ); // redistribute slave settings
    }
}

```

The `eqPly::View` sets its `Proxy` as the user data object in the constructor. By default, the master instance of the view's user data is on the application instance of

9. Advanced Features

the view. This may be changed by overriding `hasMasterUserData`. The proxy object registration, mapping and synchronization is fully handled by the fabric layer, no further handling has to be done by the application:

```
View::View( eq::Layout* parent )
    : eq::View( parent )
    , _proxy( this )
    , _modelID( lunchbox::UUID::ZERO )
    , _idleSteps( 0 )
{
    setUserData( &_proxy );
}

void View::setModelID( const lunchbox::uint128_t& id )
{
    if( _modelID == id )
        return;

    _modelID = id;
    _proxy.setDirty( Proxy::DIRTY_MODEL );
}
```

9.7.2. Run-time Layout Switch

The application can use a different layout on a canvas. This will cause the running entities to be updated on the next frame. At a minimum, this means the channels involved in the last layout on the canvas are de-initialized, that is `configExit` and `NodeFactory::releaseChannel` is called, and channels involved in the new layout are initialized. If a layout does not cover fully a canvas, the layout switch can also cause the (de-)initialization of windows, pipes and nodes.

Due to the entity (de-)initialization and the potential need to initialize view-specific data, e.g., a model, a layout switch is relatively expensive and will stall `eqPly` for about a second.

Initializing an entity can fail. If that happens, the server will exit the whole configuration, and send an `EXIT` event to the application node. The exit event will cause the application to exit, since it resets the config's running state.

9.7.3. Run-time Stereo Switch

Similar to switch a layout, the stereo mode of each view can be switched at runtime. This causes all destination channels of the view to update the cyclop eye or the left and right eye.

The configuration contains stereo information in three places: the view, the segment and the compound.

The view defines which stereo mode is active: mono or stereo. This setting can be done in the configuration file or programmatically.

The segment has an `eye` attribute in which defines which eyes are displayed by this segment. This is typically all eyes for active stereo projections and the left or right eye for passive stereo. The cyclop eye for passive stereo can either be set on a single segment or both, depending if the second projector

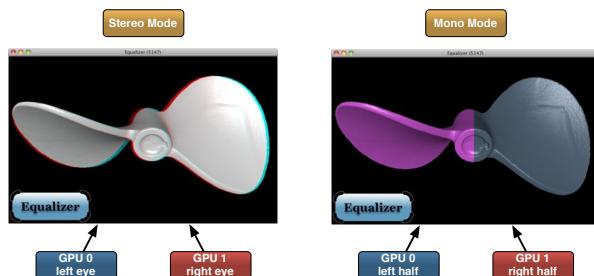


Figure 47: Using two different decompositions during stereo and mono rendering

9. Advanced Features

is active during monoscopic rendering. The default setting enables all eyes for a segment.

The compound has an eye attribute defining which eyes it is capable of updating. This allows to specify different compounds for the same channel, depending on the stereo mode. One use case is to use one GPU each for updating a stereo destination channel in stereo mode, and using a 2D compound with the two GPUs in mono mode. Figure 47 illustrates this example. The coloring in mono mode is to illustrate the 2D decomposition. The default setting for the compound eye attributes is all eyes.

9.7.4. View Frustum Update

Changing view frustum parameters is typically done for non-fullscreen application windows and multi-view layouts, where the rendering is not meant to be viewed in real-world size in an immersive environment. A typical use case is changing the field-of-view of the rendering.

A view is derived from `eq::Frustum` (Figure 46), and the application process can set the wall or projection parameters at runtime. For a description of wall and projection parameters please refer to Section 3.11.3. The new data will be effective for the next frame.

The frustum of a view overrides the underlying frustum of the segments. The frustum of a view does not typically correspond to a physical projection, and can therefore be changed at runtime. The frustum of a segment does however correspond to a physical projection, and is therefore immutable.

The default Equalizer event handling is using the view API to maintain the aspect ratio of destination channels after a window resize. Without updating the wall or projection description, the rendering would become distorted.

When a window is resized, a `CHANNEL_RESIZE` event is eventually generated. If the corresponding channel has a view, `Channel::processEvent` sends a `VIEW_RESIZE` event to the application. This event contains the identifier of the view.

The config event is dispatched to `View::handleEvent`. Using the original size and wall or projection description of the view, a new wall or projection is computed, keeping the aspect ratio and the height of the frustum constant. This new frustum is automatically applied by Equalizer at the next config frame.

Figure 48 shows a sequence diagram of such a view update.

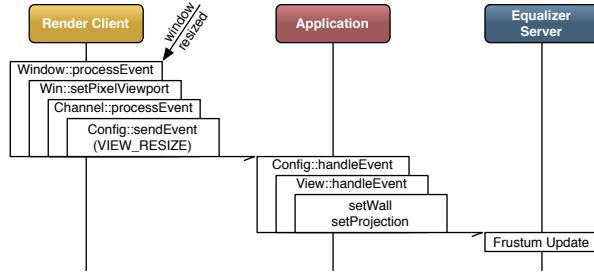


Figure 48: Event Flow during a View Update

9.8. Region of Interest

9.8.1. Motivation

Regions of interest (ROI) define which part of the framebuffer was updated by the application. They allow Equalizer to perform important optimizations during scalable rendering.

The first optimization concerns the compositing operation. The declared ROI allows Equalizer to restrict the readback, compression, network transmission and assembly to only the relevant pixels.

9. Advanced Features

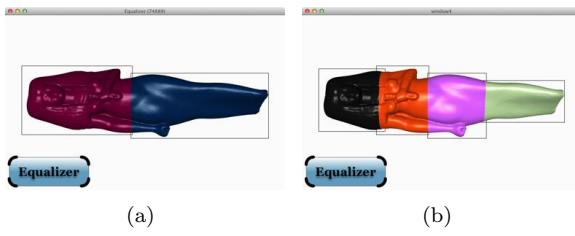


Figure 49: ROI for a two-way (a) and four-way DB (b) compound

For 2D compounds, the same optimization is applied, but has a smaller impact on the compositing performance. Figure 50 illustrates the ROI for a four-way 2D compound.

The second optimization is important for load-balanced 2D compounds. The 2D load equalizer uses internal timing statistics to form a 2D load grid for computing the split of the next frame. Without ROI, an even load within each channel has to be assumed. With ROI, an even load within the declared, smaller region of interest can be used. This causes a smaller error in the load estimation, and therefore a more accurate load prediction.

Figure 50 illustrates this load grid for a four-way 2D compound with ROI (top) and without ROI (bottom).

9.8.2. ROI in eqPly

The application should declare its regions of interest during rendering. This declaration can be very fine-grained, e.g., on each leaf node of a scene graph. Equalizer will track and optimize multiple regions automatically. The current implementation of Equalizer merges all regions of a single channel into one region for compositing and load-balancing. Later versions may use different, more optimal, heuristics based on the application-declared regions.

Each leaf node of the kd-tree used in eqPly declares its region of interest when it is rendered. The region is calculated by projecting the bounding box into screen space and normalizing the resulting screen space rectangle:

```
void VertexBufferLeaf::draw( VertexBufferState& state ) const
{
    if( state.stopRendering( ) )
        return;

    state.updateRegion( _boundingBox );

    ...

void VertexBufferState::updateRegion( const BoundingBox& box )
{
    const Vertex corners[8] = { Vertex( box[0][0], box[0][1], box[0][2] ),
                                Vertex( box[1][0], box[0][1], box[0][2] ),
                                Vertex( box[0][0], box[1][1], box[0][2] ),
                                Vertex( box[1][0], box[1][1], box[0][2] ),
                                Vertex( box[0][0], box[0][1], box[1][2] ),
                                Vertex( box[1][0], box[0][1], box[1][2] ),
                                Vertex( box[0][0], box[0][1], box[1][2] ),
                                Vertex( box[0][0], box[1][1], box[1][2] ) };

```

This is particularly important for DB compounds, which typically cover less screen space on each channel as resources are added to the decomposition. Figure 49 illustrates this for a two-way and four-way decomposition. Declaring the ROI alleviates the increasing compositing cost as resources are added to a DB compound.

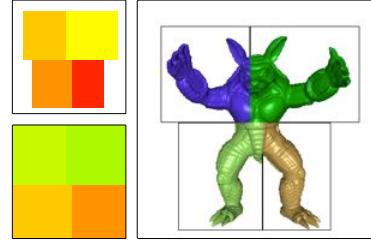


Figure 50: ROI for 2D load balancing

9. Advanced Features

```
Vertex( box[1][0], box[1][1], box[1][2] ) };
```

```
Vector4f region( std::numeric_limits< float >::max(),
                  std::numeric_limits< float >::max(),
                  -std::numeric_limits< float >::max(),
                  -std::numeric_limits< float >::max() );
```

```
for( size_t i = 0; i < 8; ++i )
{
    const Vertex corner = _pmvMatrix * corners[i];
    region[0] = std::min( corner[0], region[0] );
    region[1] = std::min( corner[1], region[1] );
    region[2] = std::max( corner[0], region[2] );
    region[3] = std::max( corner[1], region[3] );
}

// transform region of interest from [ -1 -1 1 1 ] to normalized viewport
const Vector4f normalized( region[0] * .5f + .5f,
                            region[1] * .5f + .5f,
                            ( region[2] - region[0] ) * .5f,
                            ( region[3] - region[1] ) * .5f );
```

```
declareRegion( normalized );
```

```
...
```

The declareRegion is eventually forwarded to eq::Channel::declareRegion.

9.9. Image Compositing for Scalable Rendering

Two task methods are responsible for collecting and compositing the result image during scalable rendering. Scalable rendering is a use case of parallel rendering, when multiple channels are contributing to a single view. This requires reading back the pixel data from the source GPU and assembling it on the destination GPU.

Channels producing one or more `outputFrames` use `Channel::frameReadback` to read the pixel data from the frame buffer. The channels receiving one or multiple `inputFrames` use `Channel::frameAssemble` to assemble the pixel data into the frame-buffer. Equalizer takes care of the network transport of frame buffer data between nodes.

Normally the programmer does not need to interfere with the image compositing. Changes are sometimes required at a high level, for example to order the input frames or to optimize the readback. The following sections describe the image compositing API in Equalizer.

9.9.1. Compression Plugins

Compression plugins allow the creation of runtime-loadable modules for image compression. Equalizer will search predefined directories during `eq::init` for dynamic shared objects (DSO) containing compression libraries (EqualizerCompressor*.dll on Windows, libEqualizerCompressor*.dylib on Mac OS X, libEqualizerCompressor*.so on Linux).

The interface to a compression DSO is a C API, which allows to maintain binary compatibility across Equalizer versions. Furthermore, the definition of an interface facilitates the creation of new compression codecs for developers.

Please refer to the Equalizer API documentation on the website for the full specification for compression plugins. The Equalizer DSO doubles as a compression plugin and implements a set of compression engines, which can be used as a reference implementation.

9. Advanced Features

Each compression DSO may contain multiple compression engines. The number of compressors in the DSO is queried by Equalizer using `EqCompressGetNumCompressors`.

For each compressor, `EqCompressorGetInfo` is called to retrieve the information about the compressor. The information contains the API version the DSO was written against, a unique name of the compressor, the type of input data accepted as well as information about the compressor's speed, quality and compression ratio.

Each image transported over the network allocates its own compressor or decompressor instance. This allows compressor implementations to maintain information in a thread-safe manner. The handle to a compressor or decompressor instance is a void pointer, which typically hides a C++ object instantiated by the compression DSO.

A unit test is delivered with Equalizer which runs all compressors against a set of images and provides performance information to calculate the compressor characteristics.

9.9.2. Parallel Direct Send Compositing

To provide a motivation for the design of the image compositing API, the direct send parallel compositing algorithm is introduced in this section. Other parallel compositing algorithms, e.g. binary-swap, can also be expressed through an Equalizer configuration file.

Direct send has two important properties: an algorithmic complexity of $O(1)$ for each node, that is, the compositing cost per node is constant as resources are added, and the capability to perform total ordering during compositing, e.g, to back-to-front sort all contributions of a 3D volume rendering correctly.

The main idea behind direct send is to parallelize the costly recomposition for database (sort-last) decomposition. With each additional source channel, the amount of pixel data to be composited grows linearly. When using the simple approach of compositing all frames on the destination channel, this channel quickly becomes the bottleneck in the system. Direct send distributes this workload evenly across all source channels, and thereby keeps the compositing work per channel constant.

In direct send compositing, each rendering channel is also responsible for the sort-last composition of one screen-space tile. He receives the framebuffer pixels for his tile from all the other channels. The size of one tile decreases linearly with the number of source channels, which keeps the total amount of pixel data per channel constant.

After performing the sort-last compositing, the color information is transferred to the destination channel, similarly to a 2D (sort-first) compound. The amount

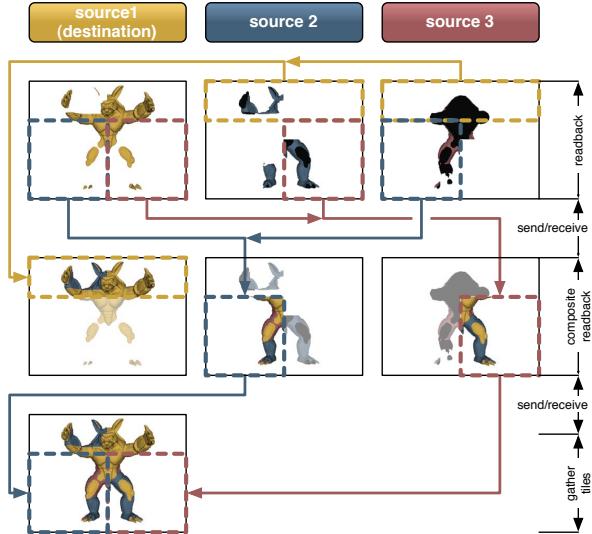


Figure 51: Direct Send Compositing

9. Advanced Features

of pixel data for this part of the compositing pipeline also approaches a constant value, i.e., the full frame buffer.

Figure 51 illustrates this algorithm for three channels. The Equalizer website contains a presentation²³ explaining and comparing this algorithm with the binary-swap algorithm.

The following operations have to be possible to perform this algorithm:

- Selection of color and/or depth frame buffer attachments
- Restricting the read-back area to a part of the rendered area
- Positioning the pixel data correctly on the receiving channels

Furthermore it has to be possible for the application to implement a read back of only the relevant region of interest, that is, the 2D area of the framebuffer updated during rendering. This optimization will be fully supported by later versions of Equalizer.

9.9.3. Frame, Frame Data and Images

An `eq::Frame` references an `eq::FrameData`. The frame data is the object connecting output with input frames. Output and input frames with the same name within the same compound tree will reference the same frame data.

The frame data is a holder for images and additional information, such as output frame attributes and pixel data availability.

An `eq::Image` holds a two-dimensional snapshot of the framebuffer and can contain color and/or depth information.

The frame synchronization through the frame data allows the input frame to wait for the pixel data to become ready, which is signaled by the output frame after readback.

Furthermore, the frame data transports the inherited range of the output frame's compound. The range can be used to compute the assembly order of multiple input frames, e.g., for sorted-blend compositing in volume rendering applications.

Readback and assemble operations on frames and images are designed to be asynchronous. They have a start and finish method for both readback and assemble to allow the initiation and synchronization of the operation. Currently, only synchronous readback and assembly using `glReadPixels` and `glDrawPixels` is implemented in the respective start method of the image. Later versions of Equalizer will implement asynchronous pixel transfers.

The offset of input and output frames characterizes the position of the frame data relative to the framebuffer, that is, the **window**'s lower-left corner. For output frames this is the position of the channel relative to the window.

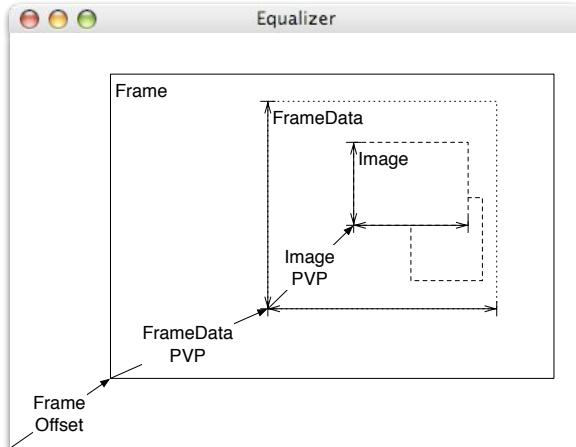


Figure 52: Hierarchy of Assembly Classes

²³<http://www.equalizergraphics.com/documents/EGPGV07.pdf>

9. Advanced Features

For output frames, the frame data's pixel viewport is the area of the frame buffer to read back. It will transport the offset from the source to the destination channel, that is, the frame data pixel viewport for input frames position the pixel data on the destination. This has the effect that a partial framebuffer readback will end up in the same place in the destination channels.

The image pixel viewport signifies the region of interest that will be read back. The default readback operation reads back one image using the full pixel viewport of the frame data.

Figure 52 illustrates the relationship between frames, frame data and images.

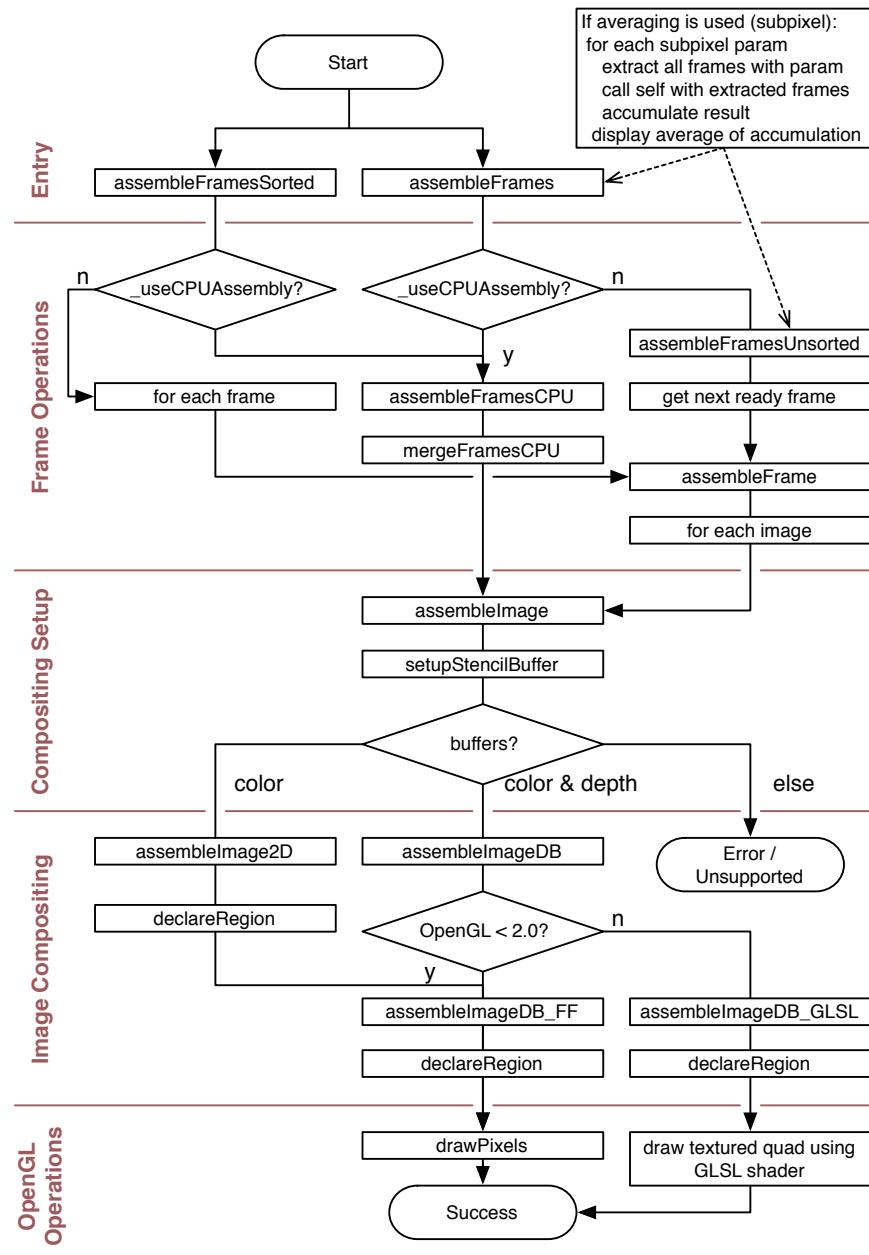


Figure 53: Functional Diagram of the Compositor

9.9.4. The Compositor

The **Compositor** class gathers a set of static functions which implement the various compositing algorithms and low-level optimizations. Figure 53 provides a top-down functional overview of the various compositor functions.

On a high level, the compositor combines multiple input frames using 2D tiling, depth-compositing for polygonal data or sorted, alpha-blended compositing for semitransparent volumetric data. These operations composite either directly all images on the GPU, or use a CPU-based compositor and then transfer the preintegrated result to the GPU. The high-level entry points automatically select the best algorithm. The CPU-based compositor uses, based on availability, OpenMP and the Paracomp compositing library to accelerate its operation.

On the next lower level, the compositor provides functionality to composite a single frame, either using 2D tiling (possibly with blending for alpha-blended compositing) or depth-based compositing.

The per-frame compositing in turn relies on the per-image compositing functionality, which automatically decides on the algorithm to be used (2D or depth-based). The concrete per-image assembly operation uses OpenGL operations to composite the pixel data into the framebuffer, potentially using GLSL for better performance.

9.9.5. Custom Assembly in eVolve

The **eVolve** example is a scalable volume renderer. It uses 3D texture-based volume rendering, where the 3D texture is intersected by view-aligned slices. The slices are rendered back-to-front and blended to produce the final image, as shown in Figure 54²⁴.

When using 2D (sort-first) or stereo decompositions, no special programming is needed to achieve good scalability, as **eVolve** is mostly fill-limited and therefore scales nicely in these modes.

The full power of scalable volume rendering is however in DB (sort-last) compounds, where the full volume is divided into separate bricks. Each of the bricks is rendered like a separate volume. For recomposition, the RGBA frame buffer data resulting from these render passes then has to be assembled correctly.

Conceptually, the individual volume bricks of each of the source channels produces pixel data which can be handled like one big 'slice' through the full texture. Therefore they have to be blended back-to-front in the same way as the slice planes are blended during rendering.

Database decomposition has the advantage of scaling any part of the volume rendering pipeline: texture and main memory (smaller bricks for each channel), fill rate (less samples per channel) and IO bandwidth for time-dependent data (less data per time step and channel). Since the amount of texture memory needed for each node decreases linearly, they make it possible to render data sets which are not feasible to visualize with any other approach.

For recomposition, the 2D frame buffer contents are blended to form a seamless picture. For correct blending, the frames are ordered in the same back-to-front order as the slices used for rendering, and use the same blending parameters. Sim-

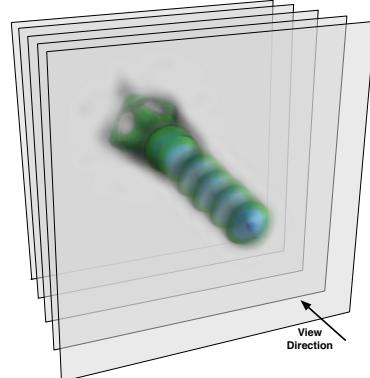


Figure 54: Volume Rendering using 3D Textures

²⁴Volume Data Set courtesy of: SFB-382 of the German Research Council (DFG)

9. Advanced Features

plified, the frame buffer images are ‘thick’ slices which are ‘rendered’ by writing their content to the destination frame buffer using the correct order.

For orthographic rendering, determining the compositing order of the input frames is trivial. The screen-space orientation of the volume bricks determines the order in which they have to be composited. The bricks in eVolve are created by slicing the volume along one dimension. Therefore the range of the resulting frame buffer images, together with the sorting order, is used to arrange the frames during compositing. Figure 56(a) shows this composition for one view.

Finding the correct assembly order for perspective frusta is more complex. The perspective distortion invalidates a simple orientation criteria like the one used for orthographic frusta. For the view and frustum setup shown in Figure 56(b)²⁵ the correct compositing order is 4-3-1-2 or 1-4-3-2.

To compute the assembly order, eVolve uses the angle between the *origin → slice* vector and the near plane, as shown in Figure 56(b). When the angle becomes greater than 90°, the compositing order of the remaining frames has to be changed. The result image of this composition naturally looks the same as the volume rendering would when rendered on a single channel. Figure 55 shows the result of the composition from Figure 56(b).



Figure 55: Result of Figure 56(b)

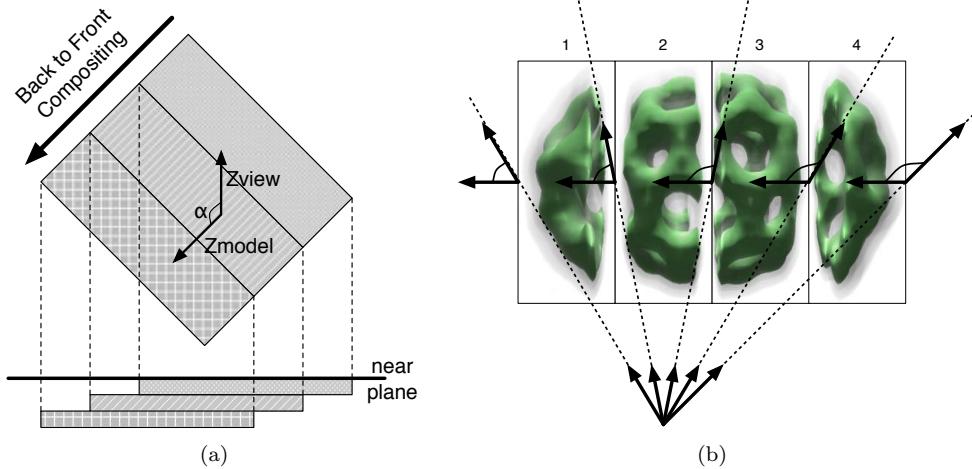


Figure 56: Back-to-Front Compositing for Orthogonal and Perspective Frusta

The assembly algorithm described in this section also works with parallel compositing algorithms such as direct-send.

9.10. Subpixel Processing

In eqPly, the image quality is gradually refined when the camera is not moving. Up to 256 samples per pixel are accumulated to implement idle antialiasing. This

²⁵Volume Data Set courtesy of: AVS, USA

9. Advanced Features

feature is compatible with subpixel compounds, which use a fixed non-idle anti-aliasing and then accelerate the generation of the remaining samples. The same basic algorithm might be applied to other multisampling effects, e.g., depth-of-field.

9.10.1. Transparent Software Anti-Aliasing

Any Equalizer applications can benefit without modification from subpixel compounds. Applications performing their own, configurable idle or non-idle anti-aliasing can easily be integrated with subpixel compounds, as described in the next sections.

The default implementation of `applyFrustum` and `applyOrtho` in `eq::Channel` apply jittering when a subpixel parameter is set. Furthermore, the Compositor uses an `eq::Accum` buffer to accumulate and display the results of a subpixel decomposition in `assembleFramesUnsorted` and `assembleFramesSorted`.

9.10.2. Software Anti-Aliasing in eqPly

The supersampling algorithm used by `eqPly` performs anti-aliasing, using a technique which computes randomized samples within a pixel by jittering the frustum by a subpixel amount. The pixel is split into an 16x16 subpixel grid, and a sample is taken randomly within each sub-pixel. The samples are accumulated and displayed after each step.

To accumulate the result, each `eqPly::Channel` maintains an accumulation buffer for each rendered eye pass. This accumulation buffer is lazily allocated and resized at the beginning of each frame:

```
// set up accumulation buffer
accum.buffer = new eq::util::Accum( glewGetContext() );
const eq::PixelViewport& pvp = getPixelViewport();
LBASSERT( pvp.isValid() );

if( !accum.buffer->init( pvp, getWindow()->getColorFormat() ) ||
    accum.buffer->getMaxSteps() < 256 )
{
    LBWARN <<" Accumulation_buffer_initialization_failed , "
           << "idle_AA_not_available." << std::endl;
    delete accum.buffer;
    accum.buffer = 0;
    accum.step = -1;
    return false;
}

// else
LBVERB << " Initialized "
       << (accum.buffer->usesFBO() ? "FBO_accum" : "glAccum")
       << " buffer_for_ " << getName() << " " << getEye()
       << std::endl;

view->setIdleSteps( accum.buffer ? 256 : 0 );
return true;
}
```

In idle mode, the results are accumulated at the end of the frame and displayed after each iteration by `frameViewFinish`. Since `frameViewFinish` is only called on destination channels this operation is done only on the final rendering result. When all the steps are done the config will stop rendering new frames.

If the current pixel viewport is different from the one saved in `frameViewStart`, the accumulation buffer needs also to be resized and the idle anti-aliasing is reset:

```
Accum& accum = _accum[ lunchbox::getIndexOfLastBit( getEye() ) ];
if( accum.buffer )
```

9. Advanced Features

```

{
    const eq::PixelViewport& pvp = getPixelViewport();
    const bool isResized = accum.buffer->resize( pvp.w, pvp.h );

    if( isResized )
    {
        const View* view = static_cast< const View*>( getView() );
        accum.buffer->clear();
        accum.step = view->getIdleSteps();
        accum.stepsDone = 0;
    }
    else if( frameData.isIdle() )
    {
        setupAssemblyState();

        if( !_isDone() && accum.transfer )
            accum.buffer->accum();
        accum.buffer->display();

        resetAssemblyState();
    }
}

```

The subpixel area is a function of the current jitter step, the channel's subpixel description and the idle state. Each source channel is responsible for filling a subset of the sampling grid. To quickly converge to a good anti-aliasing, each channel selects its samples using a pseudo-random approach, using a precomputed prime number table to find the subpixel for the current step:

```

eq::Vector2i Channel::_getJitterStep() const
{
    const eq::SubPixel& subPixel = getSubPixel();
    const uint32_t channelID = subPixel.index;
    const View* view = static_cast< const View*>( getView() );
    if( !view )
        return eq::Vector2i::ZERO;

    const uint32_t totalSteps = uint32_t( view->getIdleSteps() );
    if( totalSteps != 256 )
        return eq::Vector2i::ZERO;

    const Accum& accum = _accum[ lunchbox::getIndexOfLastBit( getEye() ) ];
    const uint32_t subset = totalSteps / getSubPixel().size;
    const uint32_t index = ( accum.step * _primes[ channelID % 100 ] )%subset +
        ( channelID * subset );
    const uint32_t sampleSize = 16;
    const int dx = index % sampleSize;
    const int dy = index / sampleSize;

    return eq::Vector2i( dx, dy );
}

```

The `FrameData` class holds the information whether the application is in idle mode. The `Config` updates the idle mode information depending on the application's state. Each `Channel` will perform anti-aliasing when no user event requires a redraw.

When the rendering is not in idle mode, the jitter is queried from Equalizer which returns an optimal subpixel offset for the given subpixel decomposition. This is used during normal rendering of subpixel compounds:

```

eq::Vector2f Channel::getJitter() const
{
    const FrameData& frameData = _getFrameData();
    const Accum& accum = _accum[ lunchbox::getIndexOfLastBit( getEye() ) ];

    if( !frameData.isIdle() || accum.step <= 0 )
        return eq::Channel::getJitter();
}

```

9. Advanced Features

During idle rendering of any decomposition, the jitter for the frustum is computed using the normalized subpixel center point and the size of a pixel on the near plane. A random position within the sub-pixel is set as a sample position, which will be used to move the frustum. The `getJitter` method will return the computed jitter vector for the current frustum. This method has a default implementation in `eq::Channel` for subpixel compounds, but is overwritten in `eqPly` to perform idle anti-aliasing:

```
const eq::Vector2i jitterStep = _getJitterStep();
if( jitterStep == eq::Vector2i::ZERO )
    return eq::Vector2f::ZERO;

const eq::PixelViewport& pvp = getPixelViewport();
const float pvp_w = float( pvp.w );
const float pvp_h = float( pvp.h );
const float frustum_w = float(( getFrustum().get_width( ) ));
const float frustum_h = float(( getFrustum().get_height( ) ));

const float pixel_w = frustum_w / pvp_w;
const float pixel_h = frustum_h / pvp_h;

const float sampleSize = 16.f; // sqrt( 256 )
const float subpixel_w = pixel_w / sampleSize;
const float subpixel_h = pixel_h / sampleSize;

// Sample value randomly computed within the subpixel
lunchbox::RNG rng;
const eq::Pixel& pixel = getPixel();

const float i = ( rng.get< float >() * subpixel_w +
                  float( jitterStep.x( ) ) * subpixel_w ) / float( pixel.w );
const float j = ( rng.get< float >() * subpixel_h +
                  float( jitterStep.y( ) ) * subpixel_h ) / float( pixel.h );

return eq::Vector2f( i, j );
```

9.10.3. Subpixel Compounds in `eqPly`

Subpixel compounds accelerate the idle anti-aliasing computation in `eqPly`. The accumulation buffer used in the Compositor in `frameAssemble` is the same accumulation buffer used by `eqPly` in `frameViewFinish` for collecting the idle anti-aliasing results.

When an application is passing an `eq::Accum` object to the compositor, the compositor adds subpixel decomposition result into this accumulation buffer. Otherwise it uses its own accumulation buffer, which it clears in the beginning of the compositing process.

Since `eqPly` accumulates the results over multiple rendering frames, each `Channel` manages its own accumulation buffer, which is passed to the compositor in `frameAssemble` and used in `frameViewFinish` to display the results accumulated so far.

The number of performed jitter steps is tracked in `frameDraw` and `frameAssemble`, and is used in `frameFinish` to decrement the outstanding anti-aliasing samples. Therefore, the idle anti-aliasing is finished much faster if the current configuration uses a subpixel compound.

9.11. Statistics

9.11.1. Statistics Gathering

Statistics are measured in milliseconds since the configuration was initialized. The server synchronizes the per-configuration clock on each node automatically. Each

9. Advanced Features

statistic event records the originator's (channel, window, pipe, node, view or config) unique identifier.

Statistics are enabled per entity using an attribute hint. The hint determines how precise the gathered statistics are. When set to `fastest`, the per-frame clock is sampled directly when the event occurs. When set to `nicest`, all OpenGL commands will be finished before sampling the event. This may incur a performance penalty, but gives more correct results. The default setting is `fastest` in release builds, and `nicest` in debug builds. The `fastest` setting often attributes times to the operation causing an OpenGL synchronization instead of the operation submitting the OpenGL commands, e.g., the readback time contains operations from the preceding draw operation.

The events are processed by the channel's and window's `processEvent` method. The default implementation sends these events to the config using `Config::sendEvent`, as explained in Section 9.1. When the default implementation of `Config::handleEvent` receives the statistics event, it sorts the event per frame and per originator. When a frame has been finished, the events are pushed to the local (app-)node for visualization. Furthermore, the server also receives these event which will be used to implement the various runtime adjustments by the equalizers.

Figure 57 shows the visualization of statistics events in an overlay²⁶.

9.11.2. Statistics Overlay

The `eq::Channel` provides the method `drawStatistics` which renders a statistics overlay using the gathered statistics events. Statistics rendering is toggled on and off using the 's' key in the shipped Equalizer examples.

Figure 58 shows a detailed view of Figure 57. The statistics shown are for a two-node 2D compound. The destination channel is on the `appNode` and contributes to the rendering.



Figure 57: Statistics for a two node 2D compound

²⁶3D model courtesy Cyberware, Inc.

9. Advanced Features

This configuration executes two `Channel::frameDraw` tasks, one `Channel::frameReadback` task on the remote node, one `Channel::frameAssemble` task on the local node, as well as frame transmission and compression.

The X axis is the time, the right-most pixel is the most current time. One pixel on the screen corresponds to a given time unit, here one millisecond per pixel. The scale is zoomed dynamically to powers-of-ten milliseconds to fit the statistics into the available viewport. This allows easy and accurate evaluations of bottlenecks or misconfigurations in the rendering pipeline. The scale of the statistics is printed directly above the legend.

On the Y axis are the entities: channels, windows, nodes and the config. The top-most channel is the local channel since it executes `frameAssemble`, and the lower channel is the remote channel, executing `frameReadback`.

To facilitate the understanding, older frames are gradually grayed out. The right-most, current frame is brighter than the frame before it.

The configuration uses the default latency of one frame. Consequently, the execution of two frames overlaps. This can be observed in the early execution of the remote channel's `frameDraw`, which starts while the local channel is still drawing and assembling the previous frame.

The asynchronous execution allows operations to be pipelined, i.e., the compression, network transfer and assembly with the actual rendering and readback. This increases performance by minimizing idle and wait times. In this example, the remote `channel2` has no idle times, and executes the compression and network transfer of its output frame in parallel with rendering and readback. Likewise, the application node receives and decompresses the frame in parallel to its rendering thread.

In the above example, the local channel finishes drawing the frame early, and therefore spends a couple of milliseconds waiting for the input frame from the remote channel. These wait events, rendered red, are a sub-event of the yellow `frameAssemble` task. Using a `load_equalizer` instead of a static 2D decomposition would balance the rendering in this example better and minimize or even eliminate this wait time.

The white `Window::swapBuffers` task might take a longer time, since the execution of the swap buffer is locked to the vertical retrace of the display. Note that the remote source window does not execute `swapBuffers` in this configuration, since it is a single-buffered FBO.

The beginning of a frame is marked by a vertical green line, and the end of a frame by a vertical gray line. These lines are also attenuated. The brightness and color matches the event for `Config::startFrame` and `Config::finishFrame`, respectively. The event for `startFrame` is typically not visible, since it takes less than one millisecond to execute. If no idle processing is done by the application, the event for `finishFrame` occupies a full frame, since the config is blocked here waiting for the frame `current - latency` to complete.

A legend directly below the statistics facilitates understanding. It lists the per-entity operations with its associated color. Furthermore, some other textual information is overlayed with the statistics. The total compression ratio is printed with each readback and compression statistic. In this case the image has been not been compressed during download. For network transfer it has been compressed to 1% of its original size since it contains many black background pixels. For readback the plugin with the name 0x101, `EQ_COMPRESSOR_TRANSFER_RGBA_TO_BGRA` has been used, and for compression the plugin 0x11, `EQ_COMPRESSOR_RLE_DIFF_BGRA` was used. The compressor names are listed in the associated plugin header `compressorTypes.h`.

9.12. GPU Computing with CUDA

The Equalizer parallel rendering framework can be used to scale GPU computing applications (also known as GPGPU) based on CUDA. The necessary steps to write a distributed GPGPU application using Equalizer are described in the following.

Please note that the support for CUDA is only enabled if the CUDA libraries are found in their default locations. Otherwise please adapt the build system accordingly.

9.12.1. CUDA Initialization

Equalizer automatically initializes a CUDA context on each pipe (using the specified device number) whenever `hint_cuda_GL_interop` is set. The initialization is currently done using the CUDA runtime API to support device emulation. On the other side, if this hint is not set the OpenGL interop feature is not enabled, and furthermore, the GPU which executes CUDA code defaults to device 0.

Once CUDA is initialized correctly, C for CUDA can be used as any other GPU programming interface.

9.12.2. CUDA Memory Management

Equalizer does not provide any facility to perform memory transfer from and to the CUDA devices. This is entirely left to the programmer.

10. The Collage Network Library

The Collage networking library provides a peer-to-peer communication infrastructure. It is used by Equalizer to communicate between the application node, the server and the render clients. It can also be used by applications to implement distributed processing independently or complementary to the Equalizer functionality.

The network library is implemented in the `co` namespace. It 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. Different implementations of a connection exist.

Node/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. A Node is a proxy for a remote LocalNode.

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

Figure 59 provides an overview of the major collage classes and their relationship, as discussed in the following sections.

10.1. Connections

The `co::Connection` is the basic primitive used for communication between processes in Equalizer. It provides a stream-oriented communication between two endpoints.

A connection is either closed, connected or listening. A closed connection cannot be used for communications. A connected connection can be used to read or write data to the communication peer. A listening connection can accept connection requests.

10. The Collage Network Library

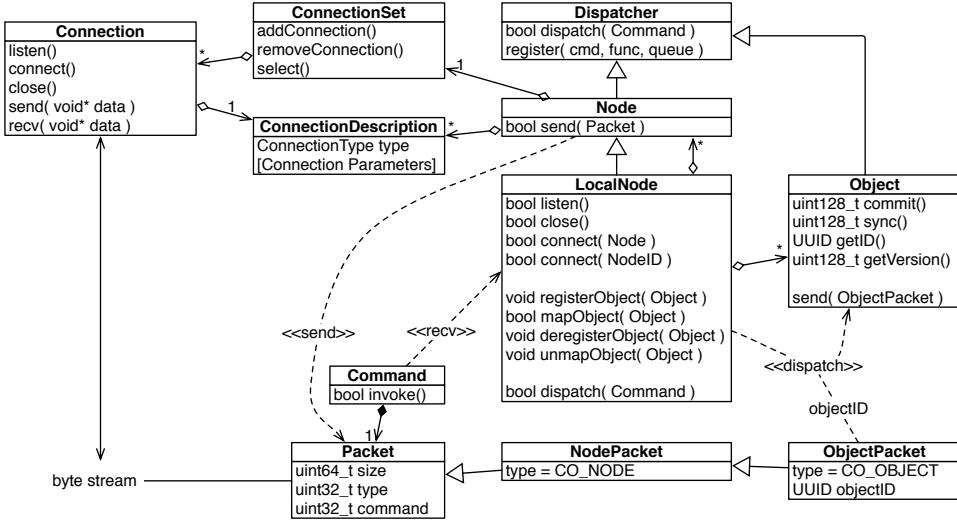


Figure 59: UML class diagram of the major Collage classes

A `co::ConnectionSet` is used to manage multiple connections. The typical use case is to have one or more listening connections for the local process, and a number of connected connections for communicating with other processes.

The connection set is used to select one connection which requires some action. This can be a connection request on a listening connection, pending data on a connected connection or the notification of a disconnect.

The connection and connection set can be used by applications to implement other network-related functionality, e.g., to communicate with a sound server on a different machine.

10.2. Packet Handling

Nodes and objects communicate using packets derived from `co::Packet`. The basic packet dispatch is implemented in the `co::Dispatcher` class, from which `co::Node` and `co::Object` is sub-classed.

The dispatcher allows the registration of a packets with a dispatch queue and an invocation method. Each packet has a type and command identifier, which is used to identify the receiver, registered queue and method. The method `dispatchCommand` pushes the packet to the registered queue and sets the registered command function on the packet command. When the commands are dequeued by the processing thread the registered method is executed using `Command::invoke`.

A command function groups the method and `this` pointer, allowing to call a C++ method on a concrete instance. If no queue is registered for a certain command, `dispatchCommand` directly calls the registered command function.

This dispatch and invocation functionality is used within Equalizer to dispatch commands from the receiver thread to the appropriate node or pipe thread, and then to invoke the command when it is processed by these threads.

10.3. Nodes

The `co::Node` is the abstraction of one process in the cluster. Each node has a universally unique identifier. This identifier is used to address nodes, e.g., to query connection information to connect to the node. Nodes use connections to communicate with each other by sending packets sub-classed from `co::NodePacket`.

The `co::LocalNode` is the specialization of the node for the given process. It encapsulates the communication logic for connecting remote nodes, as well as object registration and mapping. Local nodes are set up in the listening state during initialization.

A remote Node can either be connected explicitly by the application or due to a connection from the given remote node. The explicit connection can be done by programmatically creating a node, adding the necessary `ConnectionDescriptions` and connecting it to the local node. It may also be done by connecting the remote node to the local node by using its `NodeID`. This will cause Collage to query connection information for this node from the already-connected nodes, instantiating the node and connecting it. Both operations may fail.

10.3.1. Zeroconf Discovery

Each `LocalNode` provides a `co::Zeroconf` communicator, which allows node and resource discovery. This requires compiling using a Lunchbox library supporting `dns_sd`. The service "`_collage._tcp`" is used to announce the presence of a listening `LocalNode` using the zeroconf protocol to the network, unless the local node has no listening connections. The node identifier and all listening connection descriptions are announced using keys starting with "`co_`". Internally Collage uses this information to connect unknown nodes by using the node identifier alone.

Applications may use the zeroconf communicator to add additional key-value pairs to announce application-specific data, and to retrieve a snapshot of all key-value pairs of all discovered nodes on the network.

10.3.2. Communication between Nodes

Figure 60 shows the communication between two nodes. The nodes in the listening state also have a listening connection, which is not shown in the Figure for complexity reasons.

When the remote node sends a packet, the listening node receives the packet and dispatches it from the receiver thread using the method `dispatchCommand`. The default implementation knows how to dispatch packets of type `node`, `session` or `object`. Applications can define custom data types for packets, and then have to extend `dispatchCommand` to handle these custom data types.

Node packets are directly dispatched using `Dispatcher::dispatchCommand`. For object packets, the appropriate object is found on the local node and `LocalNode::dispatchCommand` will dispatch the packet.

If `dispatchCommand` returns false, the packet will be redispersed later. This is used if an object has not been mapped locally, and therefore the packet could not be dispatched.

If an application wants to extend communication on the node level, it can either define its own datatype for packets, or to define custom node packets. Packets with a custom datatype are handled by overriding `LocalNode::dispatchCommand`. Custom node packets can use any command greater than `CMD_NODE_CUSTOM` as the packet command. By registering a `co::CommandFunc` for these commands, the `co::Dispatcher` dispatch and invoke mechanism can be used automatically.

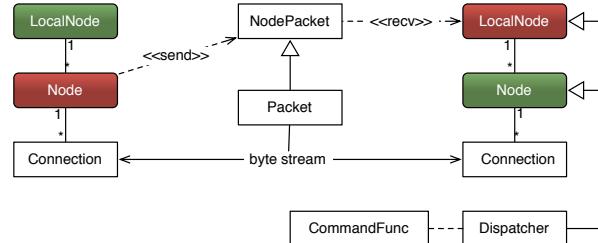


Figure 60: Communication between two Nodes

10.4. Objects

Distributed objects provide powerful, object-oriented data distribution for C++ objects. They facilitate the implementation of data distribution in a cluster environment. Their functionality and an example use case for parallel rendering has been described in Section 8.3.

Distributed objects subclass from `co::Serializable` or `co::Object`. The application programmer implements serialization and deserialization of the distributed data. Objects are dynamically attached to a listening local node, which manages the network communication and packet dispatch between different instances of the same distributed object.

Objects are addressed using a universally unique identifier. The identifier is automatically created in the object constructor. The master version of a distributed object is registered with the `co::LocalNode`. The identifier of the master instance can be used by other nodes to map their instance of the object, thus synchronizing the object's data and identifier with the remotely registered master version.

Distributed objects can be static (immutable) or dynamic. Dynamic objects are versioned. New versions are committed from the master instance, which sends the delta between the previous and current version to all mapped slave objects. The slave objects sync the queued deltas when they need a version.

Objects may have a maximum number of unapplied versions, which will cause the commit on the master instance to block if any slave instance has reached the maximum number of queued versions. By default, slave instances can queue an unlimited amount of unapplied versions.

Objects use compression plugins to reduce the amount of data sent over the network. By default, the plugin with the highest compression ratio and lossless compression for `EQ_COMPRESSOR_DATATYPE_BYTE` tokens is chosen. The application may override `Object::chooseCompressor` to deactivate compression by returning `EQ_COMPRESSOR_NONE` or to select object-specific compressors.

10.4.1. Common Usage for Parallel Rendering

Distributed objects are addressed using universally unique identifiers, because pointers to other objects cannot be distributed directly; they have no meaning on remote nodes.

The entry point for shared data on a render client is the identifier passed by the application to `Config::init`. This identifier typically contains the identifier of a static distributed object, and is passed by Equalizer to all `configInit` task methods. Normally this initial object is mapped by the render clients in `Node::configInit`. It normally contains identifiers of other shared data objects.

The distributed data objects referenced by the initial data object are often versioned objects, to keep them in sync with the rendered frames. Similar to the initial identifier passed to `Config::init`, an object identifier or object version can be passed to `Config::startFrame`. Equalizer will pass this identifier to all `frameStart` task methods. In `eqPly`, the frame-specific data, e.g., the global camera data, is versioned. The frame data identifier is passed in the initial data, and the frame data version is passed with each new frame request.

There are multiple ways of implementing data distribution for an existing C++ class hierarchy, such as a scene graph:

Subclassing The classes to be distributed inherit from `co::Object` and implement the serialization methods. This approach is recommended if the source code of existing classes can be modified. It is used for `eqPly::InitData` and `eqPly::FrameData`. (Figure 61(a))

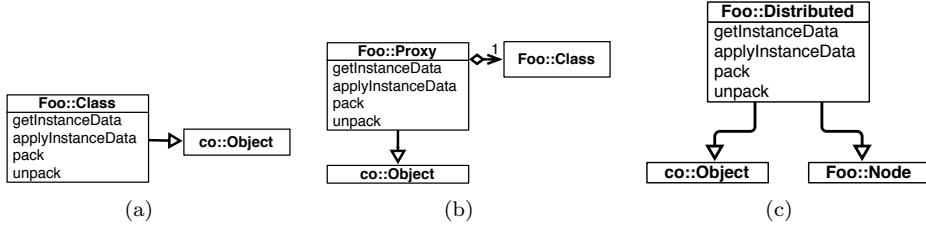


Figure 61: Object Distribution using Subclassing, Proxies or Multiple Inheritance

Proxies For each object to be distributed, a proxy object is created which manages data distribution for its associated object. This requires the application to track changes on the object separately from the object itself. The model data distribution of eqPly is using this pattern. (Figure 61(b))

Multiple Inheritance A new class inheriting from the class to be distributed and from co::Object implements the data distribution. This requires the application to instantiate a different type of object instead of the existing object, and to create wrapper methods in the superclass calling the original method and setting the appropriate dirty flags. This pattern is not used in eqPly. (Figure 61(c))

10.4.2. Change Handling and Serialization

Equalizer determines the way changes are to be handled by calling Object::getChangeType during the registration of the master version of a distributed object. The change type determines the memory footprint and the contract for the serialization methods. The deserialization always happens in the application thread causing the deserialization. The following change types are possible:

STATIC The object is not versioned. The instance data is serialized whenever a new slave instance is mapped. The serialization happens in the command thread on the master node, i.e., in parallel to the application threads. Since the object's distributed data is supposed to be static, this should not create any race conditions. No additional data is stored.

INSTANCE The object is versioned, and the instance and delta data is identical, that is, only instance data is serialized. The serialization always happens before LocalNode::registerObject or Object::commit returns. Previous instance data is saved to be able to map old versions.

DELTA The object is versioned, and the delta data is typically smaller than the instance data. Both the delta and instance data are serialized before Object::commit returns. Previous instance data is saved to be able to map old versions.

UNBUFFERED The object is versioned. No data is stored, and no previous versions can be mapped. The instance data is serialized from the command thread whenever a new slave instance is mapped, i.e., in parallel to application threads. The application has to ensure that this does not create any thread conflicts. The delta data is serialized before Object::commit returns. The application may choose to use a different, more optimal implementation to pack deltas by using a different implementation for getInstanceData and pack.

10.4.3. co::Serializable

co::Serializable implements one typical usage pattern for data distribution based on co::Object. The co::Serializable data distribution is founded on the concept of dirty bits, allowing inheritance with data distribution. Dirty bits form a 64-bit mask which marks which parts of the object have to be distributed during the next commit. It is easier to use, but imposes one typical way to implement data distribution.

For serialization, the default co::Object serialization functions are implemented by co::Serializable, which (de-)serializes and resets the dirty bits, and calls `serialize` or `deserialize` 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.

To use co::Serializable, the following steps have to be taken:

Inherit from co::Serializable: The base class will provide the dirty bit management and call `serialize` and `deserialize` appropriately. By overriding `getChangeType`, the default versioning strategy might be changed.

Define new dirty bits: Define dirty bits for each data item by starting at Serializable::DIRTY_CUSTOM, shifting this value consecutively for each new dirty bit.

Implement serialize and deserialize: For each object-specific dirty bit which is set, stream the corresponding data item to or from the provided stream. Call the parent method first in both functions. For application-specific objects, write a (de-)serialization function.

Mark dirty data: In each 'setter' method, call `setDirty` with the corresponding dirty bit.

The registration and mapping of co::Serializables is done in the same way as for co::Objects, which has been described in Section 10.4.

10.4.4. Slave Object Commit

Versioned slave objects may also commit data to the master instance. However, this requires the implementation to take special care to resolve conflicts during deserialization. The algorithm and constraints for slave object commits are explained in this section.

When using `commit` on slave object instances, multiple modifications on the same database may happen simultaneously. To avoid excessive synchronization in the network layer, Equalizer implements a lightweight approach which requires the application to avoid or handle conflicting changes on the same object.

When a slave object instance commits data, `pack` is called on the object. The data serialized by `pack` is sent to the master instance of the object. On the master instance, `notifyNewVersion` is called from the command thread to signal the data availability. The slave instance generates a unique, random version. When no data was serialized, `VERSION_NONE` is returned.

The master instance can `sync` delta data received from slaves. When using the version generated by the slave object commit, only the delta for this version is applied. When using `VERSION_NEXT` as the version parameter to `sync`, one slave commit is unpacked. This call may block if no data is pending. When using `VERSION_HEAD` all pending versions are unpacked. The data is synchronized in the order it has been received from the slaves. If a specific order is needed, it is the application's responsibility to serialize the slave commits accordingly.

10. The Collage Network Library

Synchronizing slave data on a master instance does not create a new version. The application has to explicitly call `commit` to pack the changed data, which will create a new version send to all slaves.

This approach requires the application to handle or avoid, if possible, the following situations:

- A slave instance will always receive its own changes when the master commits the slave change.
- A slave instance commits data which does not correspond to the last version of the master instance.
- Two different slave instances commit changes simultaneously which may result in conflicts.
- A master instance applies data to an already changed database.
- The master instance commits a new version while a slave instance is modifying its instance.

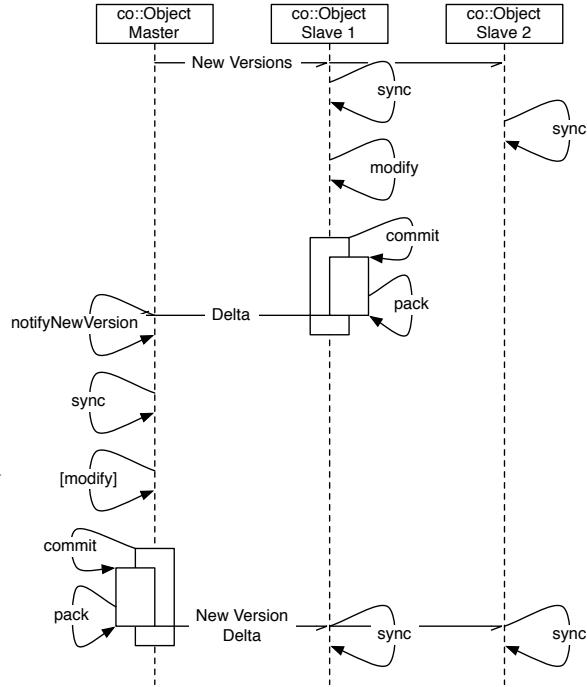


Figure 62: Slave Commit Communication Sequence

Figure 62 depicts one possible communication sequence. Where possible, it is best to adhere to the following rules:

- Handle the case that a slave instance will receive its own changes again.
- Set the appropriate dirty bits when unpacking slave delta data on the master object instance to redistribute the changes.
- Write your application code so that only one object instance is modified at any given time.
- When multiple slave instances have to modify the same object at the same time, try to modify and serialize disjunct data of the object.
- Before modifying a slave instance, `sync` it to the head version.
- Write serialization code defensively, e.g.:
 - When adding a new child to an object, check that it has not been added already.
 - When removing a child, transmit the child's identifier instead of its index for removal.
 - Modify a variable by serializing its value (instead of an operation on it) so that the last change will win.

10.4.5. Communication between Object Instances

Objects can send command packets to each other by subclassing `co::ObjectPacket`, which is derived from `NodePacket`. The object packet extends the node packet by the object identifier and an object instance identifier.

Multiple instances of the same object (identifier) can be mapped on the same node and session simultaneously. Each object instance has a node-unique instance identifier. When the instance ID is set to `EQ_INSTANCE_ALL` (the default value), a command is dispatch to all object instances registered to the session with the correct object ID. Multiple object instance per node are used when different threads on the node need to process a different version of the same object.

When the packet is intended for a specific instance, set the instance identifier of the object packet to the corresponding value.

Distributed objects provide data synchronization functionality, as described above. Additionally, applications can send command packets by creating custom object packets with a command greater than `CMD_OBJECT_CUSTOM`.

Distributed object communication relies on the lower node communication layers, as shown in Figure 63.

10.4.6. Usage in Equalizer

The Equalizer client library and server are built on top of the network layer. They influence the design of it, and serve as a sample implementation on how to use classes in the `co` namespace.

Equalizer uses the common base class `eq::fabric::Object`, which derives from `co::Serializable` for implementing data distribution between the server, application and render clients. This class manages common entity data, and is not intended to be subclassed directly. The inheritance chain `eq::View` → `eq::fabric::View` → `eq::fabric::Object` → `co::Serializable` → `co::Object` serves as sample implementation for this process.

Equalizer objects derived from `co::Serializable`, e.g., `eq::View`, are registered, mapped and synchronized by Equalizer. To execute the rendering tasks, the server send the task command packet to the render clients, which dispatch and execute them in the received order.

10.5. Barrier

The `co::Barrier` provides a networked barrier primitive. It is an `co::Object` used by Equalizer to implement software swap barriers, but it can be used as a generic barrier in application code.

The barrier uses both the data distribution for synchronizing its data, as well as custom command packets to implement the barrier logics. A barrier is a versioned object. Each version can have a different height, and enter requests are automatically grouped together by version.

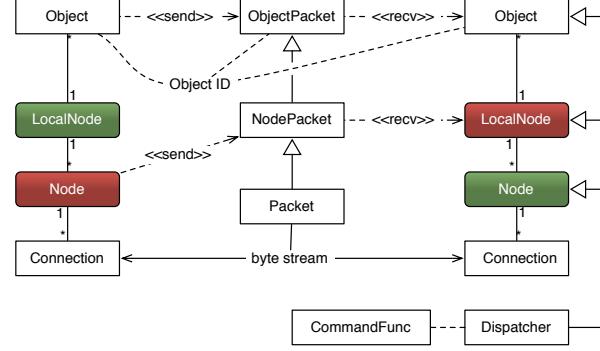


Figure 63: Communication between two Objects

A. Command Line Options

A. Command Line Options

Equalizer recognizes a number of command line options. To not pollute the application namespace for argument, all Equalizer command line options start with `-eq-`. The following options are recognized:

- `-eq-client`** is used to start render client processes. Starts a resident render client when used without additional arguments, as described in Section 4.3.1. The server appends an undocumented argument to this option which is used by the `eq::Client` to connect to the server and identify itself.
- `-eq-config <sessionName|filename.eqc>`** is used to configure an application-local server, as described in Section 3.1.2.
- `-eq-layout <layoutName>`** activates all layouts of the given name on all respective canvases during initialization. This option may be used multiple times.
- `-eq-listen <connection>`** configures the local node to listen on the given connections. The formation of the connection argument is documented in `co::ConnectionDescription::fromString()`. This option may be used multiple times.
- `-eq-logfile <filename>`** redirects all Equalizer output to the given file.
- `-eq-render-client <filename>`** provides an alternate executable name for starting render clients. This option is useful if the client machines use a different directory layout from the application machine.
- `-eq-server <connection>`** provides a connection description when using a separate server process.

B. File Format

The current Equalizer file format is a one-to-one representation of the server's internal data structures. Its purpose is intermediate, that is, it will gradually be replaced by automatic resource detection and automatic configuration. However the core scalability engine will always use a similar structure as currently exposed by the file format.

The file format represents an ASCII deserialization of the server. Streaming an `eq::server::Server` to an `eq::base::Log` ostream produces a valid configuration file. Likewise, loading a configuration file produces an `eq::server::Server`.

The file format uses the same syntactical structure as VRML. If your editor supports syntax highlighting and formatting for VRML, you can use this mode for editing .eqc files.

The configuration file consist of an optional global section and a server configuration. The global section defines default values for various attributes. The server section represents an `eq::server::Server`.

B.1. File Format Version

B.2. Global Section

The global section defines default values for attributes used by the individual entities in the server section. The naming convention for attributes is:

`EQ_<ENTITY>_<DATATYPE>ATTR_<ATTR_NAME>`

B. File Format

The entity is the capitalized name of the corresponding section later in the configuration file: connection, config, pipe, window, channel or compound. The connection is used by the server and nodes.

The datatype is one capital letter for the type of the attribute's value: S for strings, C for a character, I for an integer and F for floating-point values. Enumeration values are handled as integers. Strings have always to be surrounded by double quotes ””. A character has to be surrounded by single quotes ””.

The attribute name is the capitalized name of the entities attribute, as discussed in the following sections.

Global attribute values have useful default parameters, which can be overridden with an environment variable of the same name. For enumeration values the corresponding integer value has to be used. The global values in the config file override environment variables, and are in turn overridden by the corresponding attributes sections of the specific entities.

The globals section starts with the token `global` and an open curly brace '{', and is terminated with a closing curly brace '}'. Within the braces, globals are set using the attribute's name followed by its value. The following attributes are available:

| | |
|-----------------|--|
| Name | EQ_CONNECTION_SATTR_HOSTNAME |
| Value | string |
| Default | ”localhost” |
| Details | The hostname or IP address used to connect the server or node. When used for the server, the listening port of the server is bound to this address. When used for a node, the server first tries to connect to the render client node using this hostname, and then tries to launch the render client executable on this host. |
| See also | <code>EQ_NODE_SATTR_LAUNCH_COMMAND</code> <code>EQ_CONNECTION_IATTR_PORT</code> <code>EQ_CONNECTION_IATTR_TYPE</code> |

| | |
|----------------|---|
| Name | EQ_CONNECTION_IATTR_TYPE |
| Value | TCPIP SDP RSP PIPE [Win32 only] |
| Default | TCPIP |
| Details | The protocol for connections. SDP programmatically selects the socket direct protocol (AF_INET_SDP) provided by most InfiniBand protocol stacks, TCPIP uses normal TCP sockets (AF_INET). RSP provides reliable UDP multicast. PIPE uses a named pipe to communicate between two processes on the same machine. |

| | |
|----------------|---|
| Name | EQ_CONNECTION_IATTR_PORT |
| Value | unsigned |
| Default | 0 |
| Details | The listening port used by the server or node. For nodes, the port can be used to contact prestarted, resident render client nodes or to use a specific port for the node. If 0 is specified, a random port is chosen. Note that a server with no connections automatically creates a default connection using the server's default port. |

| | |
|----------------|---|
| Name | EQ_CONNECTION_IATTR_FILENAME |
| Value | string |
| Default | none |
| Details | The filename of the named pipe used by the server or node. The filename has to be unique on the local host. |

B. File Format

| | |
|----------------|---|
| Name | EQ_CONFIG_IATTR_ROBUSTNESS |
| Value | ON OFF |
| Default | ON |
| Details | Handle initialization failures robustly by deactivating the failed resources but keeping the configuration running. |

| | |
|-----------------|--|
| Name | EQ_CONFIG_FATTR_EYE_BASE |
| Value | float |
| Default | 0.05 |
| Details | The default distance in meters between the left and the right eye, i.e., the eye separation. The eye base influences the frustum during stereo rendering. See Section 9.6 for details. |
| See also | EQ_WINDOW_IATTR_HINT_STEREO EQ_COMPOUND_IATTR_STEREO_MODE EQ_CONFIG_FATTR_FOCUS_DISTANCE EQ_CONFIG_IATTR_FOCUS_MODE |

| | |
|-----------------|---|
| Name | EQ_CONFIG_FATTR_FOCUS_DISTANCE |
| Value | float |
| Default | 1.0 |
| Details | The default distance in meters to the focal plane. The focus distance and mode influence the calculation of stereo frusta. See Section 9.6.3 for details. |
| See also | EQ_CONFIG_IATTR_FOCUS_MODE EQ_CONFIG_FATTR_EYE_BASE |

| | |
|-----------------|--|
| Name | EQ_CONFIG_IATTR_FOCUS_MODE |
| Value | fixed relative_to_origin relative_to_observer |
| Default | fixed |
| Details | The mode how to use the focus distance. The focus mode and distance influence the calculation of stereo frusta. See Section 9.6.3 for details. |
| See also | EQ_CONFIG_IATTR_FOCUS_DISTANCE EQ_CONFIG_FATTR_EYE_BASE |

| | |
|-----------------|---|
| Name | EQ_NODE_SATTR_LAUNCH_COMMAND |
| Value | string |
| Default | ssh -n %h %c >& %h.%n.log [POSIX] ssh -n %h %c [WIN32] |
| Details | The command used by the server to auto-launch nodes which could not be connected. The launch command is executed from a process forked from the server process. The % tokens are replaced by the server at runtime with concrete data: %h is replaced by the hostname, %c by the render client command to launch, including command line arguments and %n by a node-unique identifier. Each command line argument is surrounded by launch command quotes. |
| See also | EQ_NODE_SATTR_LAUNCH_COMMAND_QUOTE EQ_NODE_IATTR_LAUNCH_TIMEOUT |

B. File Format

| | |
|----------------|---|
| Name | EQ_NODE_CATTR_LAUNCH_COMMAND_QUOTE |
| Value | character |
| Default | ' [POSIX] " [WIN32] |
| Details | The server uses command line arguments to launch render client nodes correctly. Certain launch commands or shells use different conventions to separate command line arguments. These arguments might contain white spaces, and therefore have to be surrounded by quotes to identify their borders. This option is mostly used on Windows. |
| Name | EQ_NODE_IATTR_LAUNCH_TIMEOUT |
| Value | unsigned |
| Default | 60'000 (1 minute) |
| Details | Defines the timeout in milliseconds to wait for an auto-launched node. If the render client process did not contact the server within that time, the node is considered to be unreachable and the initialization of the configuration fails. |
| Name | EQ_NODE_IATTR_THREAD_MODEL |
| Value | ASYNC DRAW_SYNC LOCAL_SYNC |
| Default | DRAW_SYNC |
| Details | The threading model for node synchronization. See Section 9.3.2 for details. |
| Name | EQ_PIPE_IATTR_HINT_THREAD |
| Value | OFF ON |
| Default | ON |
| Details | Determines if all task methods for a pipe and its children are executed from a separate operating system thread (default) or from the node main thread. Non-threaded pipes have certain performance limitations and should only be used where necessary. |
| Name | EQ_PIPE_IATTR_HINT_AFFINITY |
| Value | OFF AUTO CORE unsigned SOCKET unsigned |
| Default | AUTO |
| Details | Determines how all pipe threads are bound to processing cores. When set to OFF, threads are unbound and will be scheduled by the operation system on any core. If set to AUTO, threads are bound to all cores of the processor which is closest to the GPU used by the pipe, that is, which have the lowest latency to access the GPU device. This feature requires that Lunchbox and Equalizer have been compiled with the hwloc library version 1.5 or later. When set to CORE, the pipe threads will be bound to the given core. When set to SOCKET, the pipe threads will be bound to all cores of the given processor (socket). The CORE and SOCKET settings require that Lunchbox and Equalizer have been compiled with the hwloc library version 1.2 or later. |
| Name | EQ_PIPE_IATTR_HINT_CUDA_GL_INTEROP |
| Value | OFF ON |
| Default | OFF |
| Details | Determines if the pipe thread enables interoperability between CUDA and OpenGL or not. When set to ON, the CUDA device is configured on which the thread executes the CUDA device code. If no graphics adapter is specified in the config file, the one providing the highest compute performances on the particular node is chosen. When set to OFF, no CUDA device is configured at all. |

B. File Format

| | |
|-----------------|---|
| Name | EQ_WINDOW_IATTR_HINT_STEREO |
| Value | OFF ON AUTO |
| Default | AUTO |
| Details | Determines if the window selects a quad-buffered stereo visual. When set to AUTO, the default window initialization methods try to allocate a stereo visual for windows, but fall back to a mono visual if allocation fails. For PBuffers, AUTO selects a mono visual. |
| See also | EQ_COMPOUND_IATTR_STEREO_MODE |
| Name | EQ_WINDOW_IATTR_HINT_DOUBLEBUFFER |
| Value | OFF ON AUTO |
| Default | AUTO |
| Details | Determines if the window selects a double-buffered stereo visual. When set to AUTO, the default window initialization methods try to allocate a double-buffered visual for windows, but fall back to a single-buffered visual if allocation fails. For PBuffers, AUTO selects a single-buffered visual. |
| Name | EQ_WINDOW_IATTR_HINT_DECORATION |
| Value | OFF ON |
| Default | ON |
| Details | When set to OFF, window borders and other decorations are disabled, and typically the window cannot be moved or resized. This option is useful for source windows during decomposition. The implementation is window-system specific. |
| Name | EQ_WINDOW_IATTR_HINT_FULLSCREEN |
| Value | OFF ON |
| Default | OFF |
| Details | When set to ON, the window displays in fullscreen. This option forces window decorations to be OFF. The implementation is window-system specific. |
| Name | EQ_WINDOW_IATTR_HINT_SWAPSYNC |
| Value | OFF ON |
| Default | ON |
| Details | Determines if the buffer swap is synchronized with the vertical retrace of the display. This option is currently not implemented for GLX. For WGL, the WGL_EXT_swap_control extension is required. For optimal performance, set swap synchronization to OFF for source-only windows. This option has no effect on single-buffered windows. |
| Name | EQ_WINDOW_IATTR_HINT_DRAWABLE |
| Value | window pbuffer FBO OFF |
| Default | window |
| Details | Selects the window's drawable type. A window is an on-screen, window system-dependent window with a full-window OpenGL drawable. PBuffers are off-screen drawables created using window system-dependent PBuffer API's. FBO are off-screen frame buffer objects. A disabled drawable creates a system window without a frame buffer for context sharing, e.g., for asynchronous operations in a separate thread. To calculate the PBuffer or FBO size on unconnected devices, a pipe viewport size of 4096x4096 is assumed, unless specified otherwise using the pipe's viewport parameter. |

B. File Format

| | |
|-----------------|---|
| Name | EQ_WINDOW_IATTR_HINT_STATISTICS |
| Value | OFF FASTEST [ON] NICEST |
| Default | FASTEST [Release Build] NICEST [Debug Build] |
| Details | Determines how statistics are gathered. OpenGL buffers commands, which causes the rendering to be executed at an arbitrary point in time. Nicest statistics gathering executes a <code>Window::finish</code> , which calls by default <code>glFinish</code> , in order to accurately account the rendering operations to the sampled task method. However, calling <code>glFinish</code> has a performance impact. Therefore, the fastest statistics gathering samples the task statistics directly, without finishing the OpenGL commands first. Some operations, e.g., frame buffer readback, inherently finish all previous OpenGL commands. |
| See also | EQ_NODE_IATTR_HINT_STATISTICS EQ_CHANNEL_IATTR_HINT_STATISTICS |
| Name | EQ_WINDOW_IATTR_PLANES_COLOR |
| Value | unsigned RGBA16F RGBA32F |
| Default | AUTO |
| Details | Determines the number of color planes for the window. The interpretation of this value is window system-specific, as some window systems select a visual with the closest match to this value, and some select a visual with at least the number of color planes specified. RGBA16F and RGBA32F select floating point framebuffers with 16 or 32 bit precision per component, respectively. AUTO selects a visual with a reasonable quality, typically eight bits per color. |
| Name | EQ_WINDOW_IATTR_PLANES_ALPHA |
| Value | unsigned |
| Default | UNDEFINED |
| Details | Determines the number of alpha planes for the window. The interpretation of this value is window system-specific, as some window systems select a visual with the closest match to this value, and some select a visual with at least the number of alpha planes specified. By default no alpha planes are requested. |
| Name | EQ_WINDOW_IATTR_PLANES_DEPTH |
| Value | unsigned |
| Default | AUTO |
| Details | Determines the precision of the depth buffer. The interpretation of this value is window system-specific, as some window systems select a visual with the closest match to this value, and some select a visual with at least the number of depth bits specified. AUTO select a visual with a reasonable depth precision, typically 24 bits. |
| Name | EQ_WINDOW_IATTR_PLANES_STENCIL |
| Value | unsigned |
| Default | AUTO |
| Details | Determines the number of stencil planes for the window. The interpretation of this value is window system-specific, as some window systems select a visual with the closest match to this value, and some select a visual with at least the number of stencil planes specified. AUTO tries to select a visual with at least one stencil plane, but falls back to no stencil planes if allocation fails. Note that for depth-compositing and pixel-compositing at least one stencil plane is needed. |

B. File Format

| | |
|-----------------|---|
| Name | EQ_WINDOW_IATTR_PLANES_ACCUM |
| Value | unsigned |
| Default | UNDEFINED |
| Details | Determines the number of color accumulation buffer planes for the window. The interpretation of this value is window system-specific, as some window systems select a visual with the closest match to this value, and some select a visual with at least the number of accumulation buffer planes specified. |
| Name | EQ_WINDOW_IATTR_PLANES_ACCUM_ALPHA |
| Value | unsigned |
| Default | UNDEFINED |
| Details | Determines the number of alpha accumulation buffer planes for the window. The interpretation of this value is window system-specific, as some window systems select a visual with the closest match to this value, and some select a visual with at least the number of accumulation buffer planes specified. If this attribute is undefined, the value of EQ_WINDOW_IATTR_PLANES_ACCUM is used to determine the number of alpha accumulation buffer planes. |
| Name | EQ_WINDOW_IATTR_PLANES_SAMPLES |
| Value | unsigned |
| Default | UNDEFINED |
| Details | Determines the number of samples used for multisampling. |
| Name | EQ_CHANNEL_IATTR_HINT_STATISTICS |
| Value | OFF FASTEST [ON] NICEST |
| Default | FASTEST [Release Build] |
| Details | NICEST [Debug Build] |
| See also | See EQ_WINDOW_IATTR_HINT_STATISTICS. EQ_NODE_IATTR_HINT_STATISTICS EQ_WINDOW_IATTR_HINT_STATISTICS |
| Name | EQ_COMPOUND_IATTR_STEREO_MODE |
| Value | AUTO QUAD ANAGLYPH PASSIVE |
| Default | AUTO |
| Details | Selects the algorithm used for stereo rendering. QUAD-buffered stereo uses the left and right buffers of a stereo window (active stereo). Anaglyphic stereo uses glColorMask to mask colors for individual eye passes, used in conjunction with colored glasses. PASSIVE stereo never selects a stereo buffer of a quad-buffered drawable. AUTO selects PASSIVE if the left or right eye pass is not used, QUAD if the drawable is capable of active stereo rendering, and ANAGLYPH in all other cases. |
| Name | EQ_COMPOUND_IATTR_STEREO_ANAGLYPH_LEFT_MASK |
| Value | [RED GREEN BLUE] |
| Default | [RED] |
| Details | Select the color mask for the left eye pass during anaglyphic stereo rendering. |
| Name | EQ_COMPOUND_IATTR_STEREO_ANAGLYPH_RIGHT_MASK |
| Value | [RED GREEN BLUE] |
| Default | [GREEN BLUE] |
| Details | Select the color mask for the right eye pass during anaglyphic stereo rendering. |

B. File Format

B.3. Server Section

The server section consists of connection description parameters for the server listening sockets, and a number of configurations for this server. Currently only the first configuration is used.

B.3.1. Connection Description

A connection description defines the network parameters of an Equalizer process. Currently TCP/IP, SDP, RDMA, RSP and PIPE connection types are supported. TCP/IP creates a TCP socket. SDP is very similar, except that the address family AF_INET_SDP instead of AF_INET is used to enforce a SDP connection. RDMA implements native connections over the InfiniBand verbs protocol on Linux. RSP provides reliable UDP multicast. PIPE uses a named pipe for fast interprocess communication on Windows.

On Linux, RDMA connections provide the fastest implementation for InfiniBand fabrics. SDP is slower than RDMA and faster than IP over InfiniBand, and is also supported on Windows. Note that you can also use the transparent mode provided by most InfiniBand implementations to use SDP with TCP connections by preloading the SDP shared library.

Furthermore, a port for the socket can be specified. When no port is specified for the server, the default port 4242 (+UID on Posix systems) is used. When no port is specified for a node, a random port will be chosen by the operating system. For prelaunched render clients, a port has to be specified for the server to find the client node.

The hostname is the IP address or resolvable host name. A server or node may have multiple connection descriptions, for example to use a named pipe for local communications and TCP/IP for remote nodes.

The interface is the IP address or resolvable host name of the adapter to which multicast traffic is send.

A server listens on all provided connection descriptions. If no hostname is specified for a server connection description, it listens to INADDR_ANY, and is therefore reachable on all network interfaces. If the server's hostname is specified, the listening socket is bound only to this address. If any of the given hostnames is not resolvable, or any port cannot be used, server initialization will fail.

For a node, all connection descriptions are used while trying to establish a connection to the node. When auto-launched by the server, all connection descriptions of the node are passed to the launched node process, which will cause it to bind to all provided descriptions.

```
server
{
    connection # 0-n times , listening connections of the server
    {
        type      TCPIP | SDP | RDMA | PIPE | RSP
        port     unsigned # TCPIP, SDP
        filename string   # PIPE
        hostname string
        interface string  # RSP
    }
}
```

B.3.2. Config Section

A configuration has a number of parameters, nodes, observers, layouts, canvases and compounds.

The nodes and their children describe the rendering resources in a natural, hierarchical way. Observers, layouts and canvases describe the properties of the physical

B. File Format

projection system. Compounds use rendering resources (channels) to execute rendering tasks.

For an introduction to writing configurations and the concepts of the configuration entities please refer to Section 3.

The latency of a config defines the maximum number of frames the slowest operation may fall behind the application thread. A latency of 0 synchronizes all rendering tasks started by `Config::startFrame` in `Config::finishFrame`. A latency of one synchronizes all rendering tasks started one frame ago in `finishFrame`.

For a description of config attributes please refer to Section B.2.

```
config # 1-n times, currently only the first one is used by the server
{
    latency int      # Number of frames nodes may fall behind app, default 1
    attributes
    {
        eye_base     float      # distance between left and right eye
        focus_distance float
        focus_mode    fixed | relative_to_origin | relative_to_observer
        robustness   OFF | ON  # tolerate resource failures (init only)
    }
}
```

B.3.3. Node Section

A node represents a machine in the cluster, and is one process. It has a name, a hostname, a number of connection descriptions and at least one pipe. The name of the node can be used for debugging, it has no influence on the execution of Equalizer. The host is used to automatically launch remote render clients. For a description of node and connection attributes please refer to Section B.2.

```
(node|appNode) # 1-n times, a system in the cluster
                # 0|1 appNode: launches render thread within app process
{
    name      string
    host      string # Used to auto-launch render nodes
    connection # 0-n times, possible connections to this node
    {
        type      TCPIP | SDP | PIPE
        port      unsigned
        hostname string
        filename  string
    }
    attributes
    {
        thread_model ASYNC | DRAWSYNC | LOCALSYNC
        launch_command      string      # render client launch command
        launch_command_quote 'character' # command argument quote char
        launch_timeout       unsigned   # timeout in milliseconds
    }
}
```

B.3.4. Pipe Section

A pipe represents a graphics card (GPU), and is one execution thread. It has a name, GPU settings and attributes. The name of a pipe can be used for debugging, it has no influence on the execution of Equalizer.

The GPU is identified by two parameters, a port and a device. The port is only used for the GLX window system, and identifies the port number of the X server, i.e., the number after the colon in the DISPLAY description ('**:0.1**').

The device identifies the graphics adapter. For the GLX window system this is the screen number, i.e., the number after the dot in the DISPLAY description (**:0.1**). The OpenGL output is always restricted by glX to the GPU attached to selected screen.

B. File Format

For the AGL window system, the device selects the *n*th display in the list of online displays. The OpenGL output is optimized for the selected display, but not restricted to the attached GPU.

For the WGL window system, the device selects the *n*th GPU in the system. The GPU can be offline, in this case only PBuffer windows can be used. To restrict the OpenGL output to the GPU, the `WGL_NV_gpu_affinity` extension is used when available. If the extension is not present, the window is opened on the *n*th monitor, but OpenGL commands may be sent to a different or all GPU's, depending on the driver implementation.

The viewport of the pipe can be used to override the pipe resolution. The viewport is defined in pixels. The x and y parameter of the viewport are currently ignored. The default viewport is automatically detected. For offline GPU's, a default of 4096x4096 is used.

For a description of pipe attributes please refer to Section B.2.

```
pipe # 1-n times
{
    name      string
    port      unsigned    # X server number or ignored
    device    unsigned    # graphics adapter number
    viewport [ viewport ] # default: autodetect
    attributes
    {
        hint_thread     OFF | ON   # default ON
        hint_affinity   AUTO | OFF | CORE unsigned | SOCKET unsigned
    }
}
```

B.3.5. Window Section

A window represents an OpenGL drawable and holds an OpenGL context. It has a name, a viewport and attributes. The name of a window can be used for debugging, it has no influence on the execution of Equalizer, other then it being used as the window title by the default window creation methods.

The viewport of the window is relative to the pipe. It can be specified in relative or absolute coordinates. Relative coordinates are normalized coordinates relative to the pipe, e.g., a viewport of [0.25 0.25 0.5 0.5] creates a window in the middle of the screen, using 50% of the pipe's size. Absolute coordinates are integer pixel values, e.g., a viewport of [50 50 800 600] creates a window 50 pixels from the upper-left corner, sized 800x600 pixels, regardless of the pipe's size. The default viewport is [0 0 1 1], i.e., a full-screen window.

For a description of window attributes please refer to Section B.2.

```
window # 1-n times
{
    name      string
    viewport [ viewport ] # wrt pipe, default full screen

    attributes
    {
        hint_stereo          OFF | ON | AUTO
        hint_doublebuffer     OFF | ON | AUTO
        hint_decoration       OFF | ON
        hint_fullscreen       OFF | ON
        hint_swapsync         OFF | ON           # AGL, WGL only
        hint_drawable         window | pbuffer | FBO | OFF
        hint_statistics       OFF | FASTEST [ON] | NICEST
        planes_color          unsigned | RGBA16F | RGBA32F
        planes_alpha           unsigned
        planes_depth           unsigned
        planes_stencil         unsigned
        planes_accum           unsigned
    }
}
```

B. File Format

```
planes_accum_alpha  unsigned
planes_samples      unsigned
```

B.3.6. Channel Section

A channel is a two-dimensional area within a window. It has a name, viewport and attributes. The name of the channel is used to identify the channel in the respective segments or compounds. It should be unique within the config.

Output channels are referenced in their respective segments. Source channels are directly referenced by their respective source compounds.

The viewport of the channel is relative to the window. As for windows, it can be specified in relative or absolute coordinates. The default viewport is [0 0 1 1], i.e., fully covering its window.

The channel can have an alternate drawable description. Currently, the window's framebuffer can be replaced by framebuffer objects bound to the window's OpenGL context. The window's default framebuffer can be partially overwritten with framebuffer objects.

For a description of channel attributes please refer to Section B.2.

```
channel # 1-n times
{
    name      string
    viewport [ viewport ] #wrt window, default full window
    drawable [ FBO_COLOR FBO_DEPTH FBO_STENCIL ]
    attributes
    {
        hint_statistics     OFF | FASTEST [ON] | NICEST
    }
}
```

B.3.7. Observer Section

An observer represents a tracked entity, i.e, one user. It has a name, eye separation and focal plane paramters. The name of an observer can be used for debugging, it has no influence on the execution of Equalizer. It can be used to reference the observer in views, in which case the name should be unique. Not all views have to be tracked by an observer.

```
observer # 0...n times
{
    name      string
    eye_base float
    focus_distance float
    focus_mode   fixed | relative_to_origin | relative_to_observer
}
```

B.3.8. Layout Section

A layout represent a set of logical views on one or more canvases. It has a name and child views. The name of a layout can be used for debugging, it has no influence on the execution of Equalizer. It can be used to reference the layout, in which case the name should be unique.

A layout is applied to a canvas. If no layout is applied to a canvas, nothing is rendered on this canvas, i.e, the canvas is inactive.

The layout assignment can be changed at runtime by the application. The intersection between views and segments defines which output (sub-)channels are available. These output channels are typically used as destination channels in a compound. They are automatically created during configuration loading or creation.

B. File Format

```

layout # 0...n times
{
    name string
    view # 1...n times

```

B.3.9. View Section

A view represents a 2D area on a canvas. It has a name, viewport, observer and frustum. The name of a view can be used for debugging, it has no influence on the execution of Equalizer. It can be used to reference the view, in which case the name should be unique. The viewport specifies which 2D area of the parent layout is covered by this view in normalized coordinates.

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

A view has a stereo mode, which defines if the corresponding destination channel update the cyclop or left and right eye. The stereo mode can be changed at runtime by the application.

A view is a view on the application's model, in the sense used by the Model-View-Controller pattern. It can be a scene, viewing mode, viewing position, or any other representation of the application's data.

```

view # 1...n times
{
    name      string
    observer  observer-ref
    viewport [viewport]
    mode MONO | STEREO

    wall      # frustum description
    {
        bottom_left [float float float]
        bottom_right [float float float]
        top_left    [float float float]
        type        fixed | HMD
    }
    projection # alternate frustum description, last one wins
    {
        origin      [float float float]
        distance    float
        fov         [float float]
        hpr         [float float float]
    }
}

```

B.3.10. Canvas Section

A canvas represents a logical projection surface of multiple segments. It has a name, frustum, layouts, and segments. The name of a canvas can be used for debugging, it has no influence on the execution of Equalizer. It can be used to reference the canvas, in which case the name should be unique.

Each canvas consists of one or more segments. Segments can be planar or non-planar to each other, and can overlap or have gaps between each other. A canvas can define a frustum, which will create default planar sub-frusta for its segments.

The layouts referenced by the canvas can be applied to it by the application at runtime. One layout can be referenced by multiple canvases. The first layout is

B. File Format

the layout active by default, unless the command line option `-eq-layout` was used to select another default layout.

A canvas may have a swap barrier, which becomes the default swap barrier for all its subsequent segments. A swap barrier is used to synchronize the output of multiple windows. For software swap synchronization, all windows using a swap barriers of the same name are synchronized. Hardware swap synchronization is used when a `NV_group` is specified. All windows using the same `NV_group` on a single system are synchronized with each other using hardware synchronization. All groups using the same `NV_barrier` across systems are synchronized with each other using hardware synchronization. When using hardware synchronization, the barrier name is ignored.

```

canvas # 0...n times
{
    name      string
    layout    layout-ref | OFF   # 1...n times

    wall
    {
        bottom_left [ float float float ]
        bottom_right [ float float float ]
        top_left    [ float float float ]
        type        fixed | HMD
    }
    projection
    {
        origin      [ float float float ]
        distance   float
        fov         [ float float ]
        hpr        [ float float float ]
    }
    swapbarrier # default swap barrier for all segments of canvas
    {
        name string
        NV_group OFF | ON | unsigned
        NV_barrier OFF | ON | unsigned
    }
}
```

B.3.11. Segment Section

A segment represents a single display, i.e., a projector or monitor. It references a channel, has a name, viewport, frustum and potentially a swap barrier. The name of a segment can be used for debugging, it has no influence on the execution of Equalizer. It can be used to reference the segment, in which case the name should be unique.

The channel referenced by the segment defines the output channel. The viewport of the segment defines the 2D area covered by the channel on the canvas. Segments can overlap each other, e.g., when edge-blended projectors or passive stereo is used. The intersections of a segment with all views of all layouts create destination channels for rendering. The destination channels are copies of the segment's output channel with a viewport smaller or equal to the output channel viewport.

The segment eyes define which eyes are displayed by this segment. For active stereo outputs the default setting 'all' is normally used, while passive stereo segments define the left or right eye, and potentially the cyclop eye.

A segment can define a frustum, in which case it overrides the default frustum calculated from the canvas frustum and segment viewport. A segment can have a swap barrier, which is used as the swap barrier on the destination compounds of all its destination channels.

```
segment # 1...n times
```

B. File Format

```
{
    channel  string
    name     string
    viewport [ viewport ]
    eye     [ CYCLOP LEFT RIGHT ] # eye passes , default all

    wall      # frustum description
    {
        bottom_left   [ float float float ]
        bottom_right  [ float float float ]
        top_left      [ float float float ]
        type          fixed | HMD
    }
    projection # alternate frustum description , last one wins
    {
        origin       [ float float float ]
        distance     float
        fov          [ float float ]
        hpr          [ float float float ]
    }
    swapbarrier { ... } # set as barrier on all dest compounds
}
```

B.3.12. Compound Section

Compounds are the basic data structure describing the rendering setup. They use channels for rendering. Please refer to Section 3.11 for a description of compound operation logics.

The name of the compound is used for the default names of swap barriers and output frames.

A channel reference is either the name of the channel in the resource section if no canvases are used, or the destination channel reference of a view/segment intersection. Channel segment references are delimited by braces, in which the canvas, segment, layout and view describing the channel are named, i.e, 'channel (canvas "PowerWall" segment 0 layout "Simple" view 0)'.

Compound tasks describe the operations the compound executes. The default is all tasks for compounds with no children (leaf compounds) and CLEAR READBACK ASSEMBLE for all others. A leaf compound using the same channel as its parent compound does not have a default clear task, since this has been executed by the parent already. The readback and assemble tasks are only executed if the compound has output frames or input frames, respectively. Tasks are not inherited by the children of a compound.

The buffer defines the default frame buffer attachments read back by output frames. Output frames may change the buffer attachments used.

The eye attribute defines which eyes are handled by this compound. This attribute can be used to write one compound for monoscopic rendering and another for stereoscopic rendering, as illustrated in Figure 47.

The viewport restricts the rendering to the area relative to the parent compound. The range restricts the database range, relative to the parent. The pixel setting selects the pixel decomposition kernel, relative to the parent. The subpixel defines the jittering applied to the frustum, relative to the parent. The zoom scales the parent pixel viewport resolution. The DPlex period defines that $\frac{1}{period}$ frames are rendered, and the phase defines when in the period the rendering starts. All these attributes are inherited by the children of a compound. Viewport, range, pixel and period parameters are cumulative.

Equalizers are used to automatically optimize the decomposition. A 2D, horizontal or vertical load equalizer adjusts the viewport of all direct children of the compound each frame. A DB load equalizer adjusts the range of all direct children.

B. File Format

A dynamic framerate (DFR) equalizer adjusts the zoom for a constant framerate. A framerate equalizer smoothens the framerate of the compound's window to produce a steady output framerate, typically for DPlex compounds. A monitor equalizer adjusts the output image zoom to monitor another canvas.

For a description of compound attributes please refer to Section B.2.

A wall or projection description is used to define the view frustum of the compound. The frustum is inherited and typically only defined on the topmost compound. The last specified frustum description is used. Sizes are specified in meters. Figure 14 illustrates the frustum parameters. Setting a frustum on a compound is discouraged, a proper view and segment description should be used instead. View frusta override segment frusta which override compound frusta.

Output frames transport frame buffer contents to input frames of the same name. If the compound has a name, the default frame name is `frame.compoundName`, otherwise the default name is `frame.channelName`. The frame buffer attachments to read back are inherited from the compound, but can be overridden by output frames. Frames of type `texture` copy the framebuffer contents to a texture, and can only be used to composite frames between windows of the same pipe.

```

compound # 1-n times
{
    name      string
    channel   channel-ref          # see below

    task      [ CLEAR DRAW READBACK ASSEMBLE ] # CULL later
    buffer   [ COLOR DEPTH ]           # default COLOR

    eye      [ CYCLOP LEFT RIGHT ] # eyes handled, default all

    viewport [ viewport ]          # wrt parent compound, sort-first
    range    [ float float ]        # DB-range for sort-last
    pixel    [ int int int int ]    # pixel decomposition (x y w h)
    subpixel [ int int ]           # subpixel decomposition (index size)
    zoom     [ float float ]        # up/downscale of parent pvp
    period   int                  # DPlex period
    phase    int                  # DPlex phase

    view_equalizer {} # assign resources to child load_equalizers
    load_equalizer # adapt 2D tiling or DB range of children
    {
        mode 2D | DB | VERTICAL | HORIZONTAL
        damping float # 0: no damping, 1: no changes
        boundary [ x y ]          # 2D tile boundary
        boundary float            # DB range granularity
        assemble_only_limit float # limit for using dest as src
    }
    DFR_equalizer # adapt ZOOM to achieve constant framerate
    {
        framerate float # target framerate
        damping float   # 0: no damping, 1: no changes
    }
    framerate_equalizer {} # smoothen window swapbuffer rate (DPlex)
    monitor_equalizer {}   # set frame zoom when monitoring other views

    attributes
    {
        stereo_mode AUTO | QUAD | ANAGLYPH | PASSIVE # default AUTO
        stereo_anaglyph_left_mask [ RED GREEN BLUE ] # default red
        stereo_anaglyph_right_mask [ RED GREEN BLUE ] # df green blue
    }

    wall # frustum description, deprecated by view and segment frustum
    {
        bottom_left [ float float float ]
        bottom_right [ float float float ]
    }
}
```

B. File Format

```
    top_left      [ float float float ]
    type          fixed | HMD
}
projection # alternate frustum description , last one wins
{
    origin        [ float float float ]
    distance      float
    fov           [ float float ]
    hpr           [ float float float ]
}

swapbarrier {...} # compounds with the same name sync swap

child-compounds

outputframe
{
    name  string
    buffer [ COLOR DEPTH ]
    type texture | memory
}
inputframe
{
    name string # corresponding output frame
}
}
}
```

