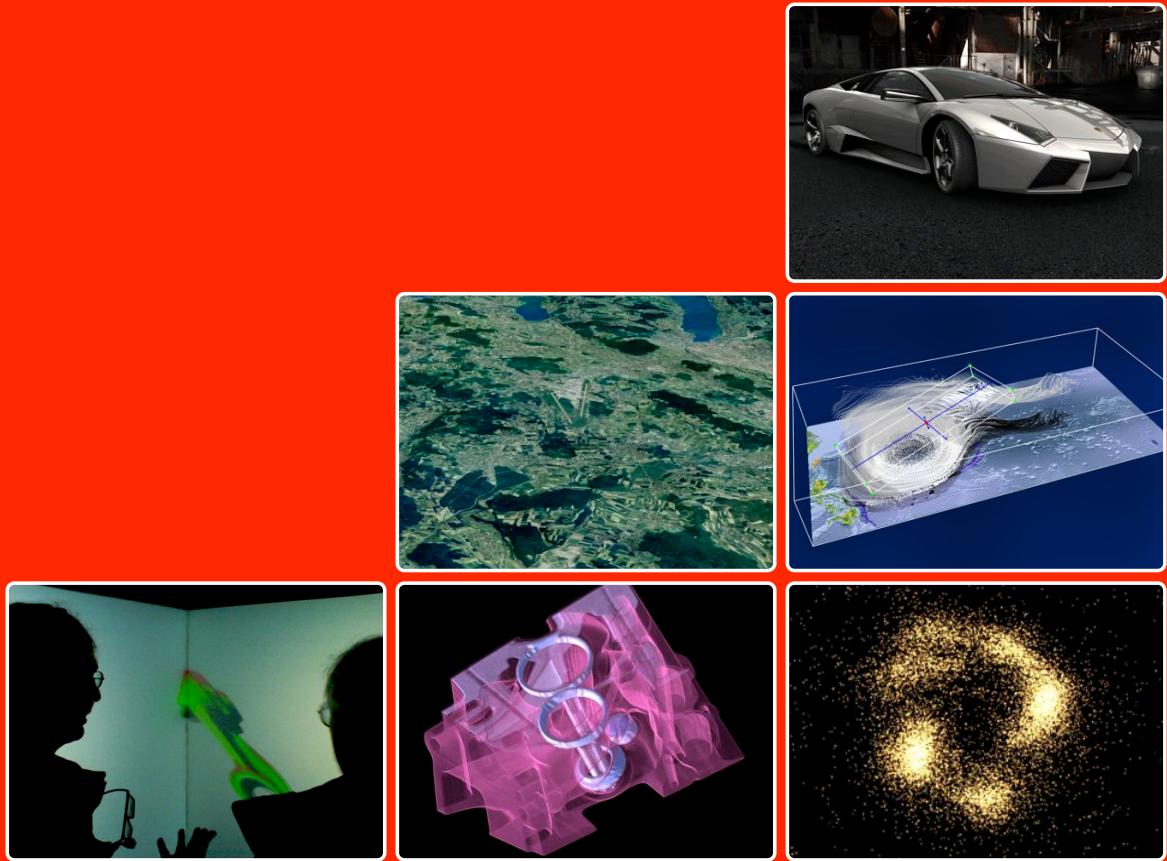


Equalizer

Programming and User Guide

Eyescale Software GmbH



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

Version 1.6 for Equalizer 0.9

Equalizer 0.9 Programming and User Guide

August 7, 2009

Contributors

Written by Stefan Eilemann.

Engineering contributions by Maxim Makhinya, Jonas Bösch, Christian Marten and Patrick Bouchaud.

Copyright

©2007-2009 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], eqRaster, a scalable terrain visualization² [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 eqNBody gravitational n-body simulation, based on the N-Body CUDA example [bottom right].

The images on the back page show the following scalable rendering modes: Combined 2D/DB [top left], DPlex [middle left], Pixel [middle center], 2D [bottom left], DB [bottom center] and stereo [bottom right].

¹Image copyright Realtime Technology AG, 2008

²Image courtesy of Visualization and Multimedia Lab, University of Zurich

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

⁴Data set courtesy of General Electric, USA

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
1.3.1. Server	2
1.3.2. Application	2
1.3.3. Render Clients	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	4
2.4. DPlex Compounds	5
2.5. Pixel Compounds	5
2.6. Automatic Runtime Adjustments	7
3. Writing Configurations	7
3.1. Preparation	7
3.2. Overview	7
3.3. Node	9
3.4. Pipe	9
3.5. Window	9
3.6. Channel	9
3.7. Canvases	10
3.7.1. Segments	11
3.8. Layouts	11
3.8.1. Views	11
3.9. Observers	12
3.10. Compounds	12
3.10.1. Writing Compounds	13
3.10.2. Compound Channels	14
3.10.3. Frustum	14
3.10.4. Compound Classification	14
3.10.5. Tasks	14
3.10.6. Decomposition - Attributes	14
3.10.7. Recomposition - Frames	15
3.10.8. Synchronization - Swapbarriers	15
3.10.9. Adjustments - Equalizers	15
II. Programming Guide	18
4. Hello, World!	18
5. The Programming Interface	18
5.1. Namespaces	19
5.2. Task Methods	19
5.3. Execution Model and Thread Safety	19

Contents

6. The eqPly polygonal renderer	20
6.1. The main Function	21
6.2. Application	23
6.2.1. Main Loop	23
6.2.2. Render Clients	26
6.3. Distributed Objects	27
6.3.1. Common Usage	27
6.3.2. Change Handling	28
6.3.3. InitData - a Static Distributed Object	28
6.3.4. FrameData - a Versioned Distributed Object	29
6.3.5. Scene Data	30
6.4. Config	32
6.4.1. Initialization and Exit	32
6.4.2. Frame Control	34
6.4.3. Event Handling	35
6.4.4. Model Handling	35
6.4.5. Layout and View Handling	36
6.5. Node	37
6.6. Pipe	38
6.6.1. Initialization and Exit	38
6.6.2. Window System	39
6.6.3. Carbon/AGL Thread Safety	39
6.6.4. Frame Control	39
6.7. Window	40
6.7.1. Window System Interface	40
6.7.2. Initialization and Exit	40
6.7.3. Object Manager	41
6.8. Channel	42
6.8.1. Initialization and Exit	42
6.8.2. Rendering	43
6.8.3. Assembly	47
7. Advanced Features	47
7.1. Event Handling	47
7.1.1. Threading	48
7.1.2. Message Pump	48
7.1.3. Event Data Flow	48
7.1.4. Default Implementation	49
7.1.5. Custom Events in eqPixelBench	50
7.2. Thread Synchronization	51
7.2.1. Threads	51
7.2.2. Thread Synchronization Models	51
7.2.3. DPlex Compounds	52
7.2.4. Thread Synchronization in Detail	53
7.3. OpenGL Extension Handling	54
7.4. Advanced Window Initialization	55
7.4.1. Drawable Configuration	55
7.4.2. AGL Window Initialization	56
7.4.3. GLX Window Initialization	56
7.4.4. WGL Window Initialization	57
7.5. Head Tracking	58
7.6. Layout API	59
7.6.1. Subclassing and Data Distribution	59
7.6.2. Dynamic Layout Switch	60

List of Figures

7.6.3. View Frustum Update	60
7.7. Image Compositing for Scalable Rendering	61
7.7.1. Compression Plugins	61
7.7.2. Parallel Direct Send Compositing	62
7.7.3. Frame, Frame Data and Images	63
7.7.4. The Compositor	64
7.7.5. Custom Assembly in eVolve	64
7.8. Statistics	66
7.8.1. Statistics Gathering	66
7.8.2. Statistics Overlay	66
8. The Network Layer	68
8.1. Connections	68
8.2. Packet Handling	69
8.3. Nodes	69
8.4. Session	70
8.5. Object	70
8.6. Barrier	71
8.7. Usage in Equalizer	71
8.7.1. eq::Object	71
A. File Format	73
A.1. File Format Version	73
A.2. Global Section	73
A.3. Server Section	78
A.3.1. Connection Description	78
A.3.2. Config Section	79
A.3.3. Node Section	79
A.3.4. Pipe Section	80
A.3.5. Window Section	80
A.3.6. Channel Section	81
A.3.7. Observer Section	81
A.3.8. Layout Section	82
A.3.9. View Section	82
A.3.10. Canvas Section	83
A.3.11. Segment Section	83
A.3.12. Compound Section	84

List of Figures

1. Parallel Rendering	1
2. Equalizer Processes	2
3. A 2D compound	3
4. A database compound	4
5. A stereo compound	5
6. A DPlex compound	5
7. A pixel compound	6
8. Pixel compound kernel	6
9. Example pixel kernels for a four-to-one pixel compound	6
10. An example configuration	8
11. Wall and projection parameters	10
12. A canvas using four, non-consecutive segments	11
13. A layout with four views	11

List of Figures

14.	A display wall using a six-segment canvas with a two-view layout	12
15.	2D load-balancing	15
16.	Cross-segment load-balancing for two segments using eight GPU's	16
17.	Cross-segment load-balancing for a CAVE	16
18.	Dynamic Frame Resolution	17
19.	Monitoring a projection wall	17
20.	Hello, World!	18
21.	Namespace layering for eqPly and eqServer	19
22.	Simplified execution model	20
23.	UML diagram eqPly and relevant Equalizer classes	21
24.	Synchronous and asynchronous execution	25
25.	Object distribution using subclassing, proxies or multiple inheritance	27
26.	eq::Object and eq::net::Object	30
27.	Scene Data in eqPly	31
28.	Scene Graph Distribution	32
29.	Config Initialization Sequence	33
30.	eq::OSWindow UML class hierarchy	40
31.	Destination view of a DB compound using demonstrative coloring	44
32.	Main Render Loop	46
33.	Event Processing	48
34.	UML Diagram for Event Handling Classes	49
35.	Threads within one node process	51
36.	Async, draw_sync and local_sync thread synchronization models	52
37.	Per-node frame synchronization	53
38.	Synchronization of frame tasks	54
39.	Monoscopic, Stereoscopic, Tracked and HMD frusta	59
40.	UML hierarchy of eqPly::View	60
41.	Event Flow during a View Update	61
42.	Direct Send Compositing	62
43.	Hierarchy of assembly classes	63
44.	Functional diagram of the compositor	64
45.	Volume Rendering using 3D Textures	65
46.	Result of Figure 47(b)	65
47.	Back-to-Front Compositing for Orthogonal and Perspective Frusta	66
48.	Statistics for a two-node 2D compound	66
49.	Detail of the Statistics from Figure 48.	67
50.	Communication between two nodes	69
51.	Communication between two Sessions	70
52.	Communication between two objects	71

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

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 scale 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 OpenGL 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 know-how and abstracts configuration, threading, synchronization, windowing and event handling. It is a ‘GLUT on steroids’, providing parallel and distributed execution, scalable rendering features, network data distribution 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 list⁵.

1.1. Parallel Rendering

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

A parallel rendering application uses the same basic execution model and extends 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 as easy as possible.

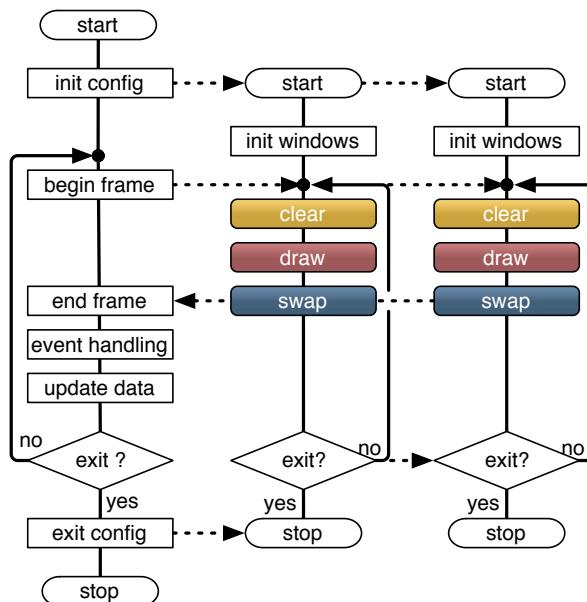


Figure 1: Parallel Rendering

⁵<http://www.equalizergraphics.com/lists.html>

1. Introduction

1.2. Installing Equalizer and running eqPly

Equalizer can be installed by downloading the distribution⁶ and compiling the source code. After installing Equalizer, please take a look at the Quickstart Guide⁷ to get familiar with the capabilities of Equalizer and the eqPly example.

Compiling Equalizer is as simple as running `make` on Linux or Mac OS X, building the provided XCode project on Mac OS X 10.5, or building the Equalizer Visual Studio 2005 solution on Windows. Note that on Mac OS X 10.4 (Tiger), some prerequisites have to be installed before running `make`, as explained in `README.Darwin`.

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. The underlying network layer uses a peer-to-peer structure, allowing render clients to communicate directly with each other, e.g., during parallel compositing.

1.3.1. Server

Each Equalizer server is responsible for managing one visualization system, i.e., a shared memory system or graphics cluster. It controls and launches the application's rendering clients. Currently, Equalizer only supports one application per server, but it will provide concurrent and efficient multi-application support in the future.

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 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. It executes the rendering tasks received from the server by calling the appropriate task methods (see Section 5.2) 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

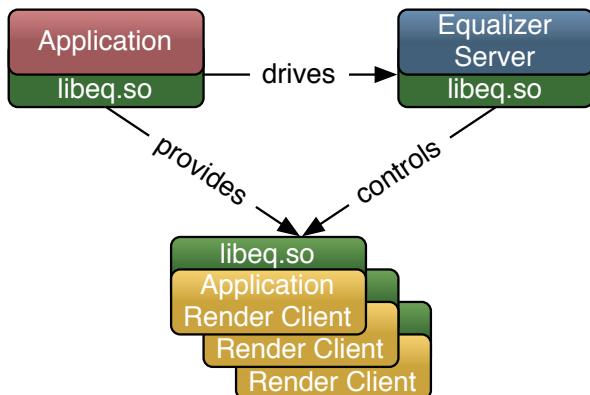


Figure 2: Equalizer Processes

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

⁷<http://www.equalizergraphics.com/documents/EqualizerGuide.html>

2. Scalable Rendering

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.

2. Scalable Rendering

Real-time visualization is an inherently parallel problem. Unfortunately, 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 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 `loadBalancer` 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.

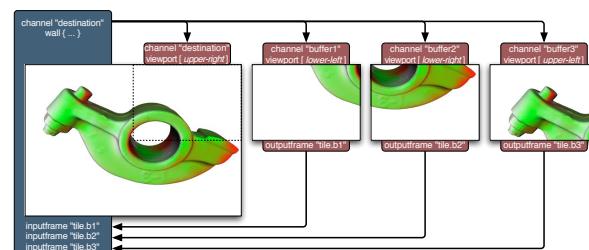


Figure 3: A 2D compound

2. Scalable Rendering

2.2. DB or Sort-Last Compounds

DB, as shown in Figure 4⁸, 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 recomposition algorithms, such as direct-send address this problem by keeping the per-node IO constant (see Figure 42).

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).

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 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 7.7.2.

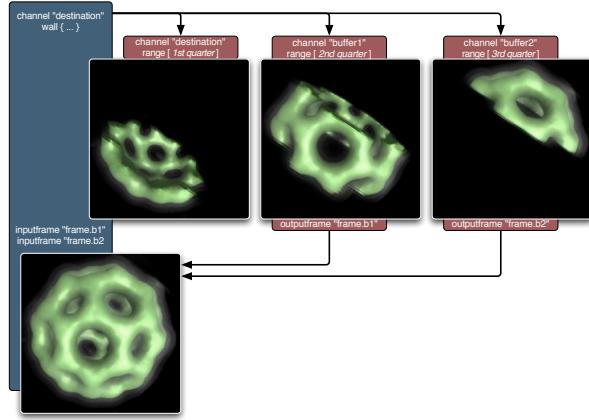


Figure 4: A database compound

2.3. Stereo Compounds

Stereo compounds, as shown in Figure 5⁹, 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.

⁸3D model courtesy of AVS, USA.

⁹3D model courtesy of Stereolithography Archive at Clemson University.

2. Scalable Rendering

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.

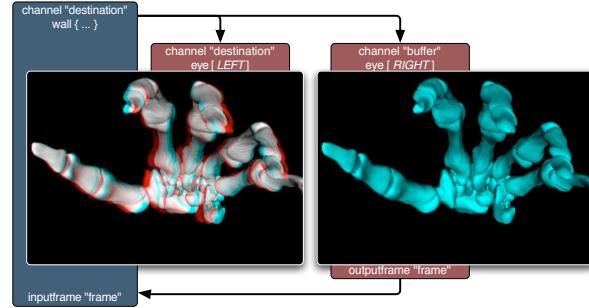


Figure 5: A stereo compound

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 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 DPlex load-balancer to smoothen the framerate.

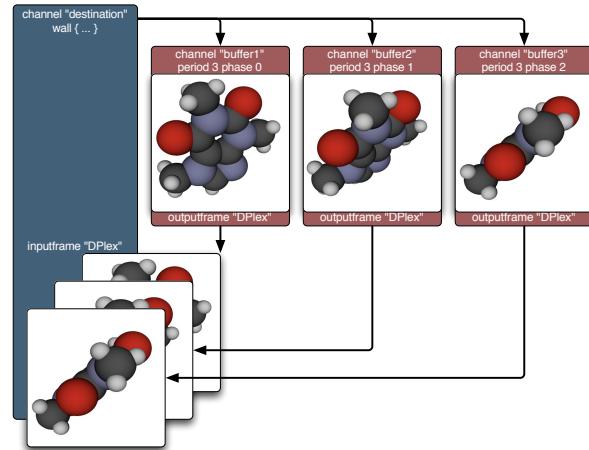


Figure 6: A DPlex compound

2.5. Pixel Compounds

Pixel compounds are similar to 2D compounds. The frusta of the source rendering units are modified so that each unit renders an evenly distributed subset of pixels, as shown in Figure 7¹⁰

As 2D compounds, pixels compounds have low, constant IO requirements for the pixel transfers during recomposition.

¹⁰3D model courtesy of AVS, USA.

2. Scalable Rendering

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.

Pixel compounds work well for purely fill-limited applications. Techniques like frustum culling do not reduce the rendered data for the source

rendering resources. Pixel compounds are ideal for ray-tracing, which is highly fill-limited and needs the full database for rendering anyway. Volume rendering applications are also well suited for this mode, and should choose it over 2D compounds.

Pixel compounds in Equalizer are configured using the `pixel` parameter, using the values [`x` `y` `width` `height`] to configure the size and offset of the sampling kernel. The width and height of the sampling kernel define how many pixels are skipped in the `x` and `y` direction, respectively. The `x` and `y` offset define the index of the source channel within the kernel. They have to be smaller than the size of the kernel. Figure 8 illustrates these parameters, and Figure 9 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.

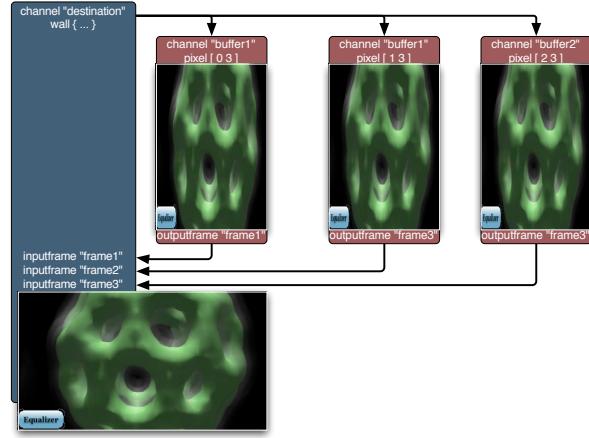


Figure 7: A pixel compound

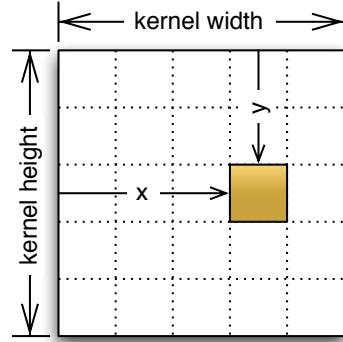


Figure 8: Pixel compound kernel

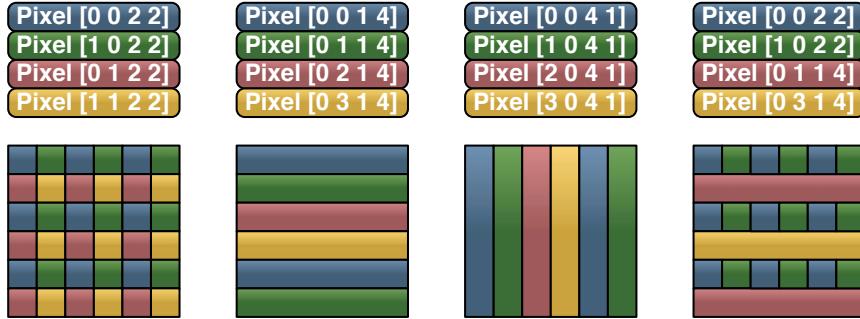


Figure 9: Example pixel kernels for a four-to-one pixel compound

3. Writing Configurations

2.6. 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 segment projections 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 3.10.9.

3. Writing Configurations

3.1. 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.
- 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.

3.2. Overview

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 A. This section gives an introduction on how to write configuration files.

A configuration consists of the declaration of the rendering resources and the usage description for these resources. It can optionally contain the description of the physical layout of the projection system and logical layouts on the projection canvases.

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.

3. Writing Configurations

The resource usage is configured using a compound tree, which is a hierarchical representation of the rendering decomposition and recomposition across the resources. It is explained in Section 3.10.

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

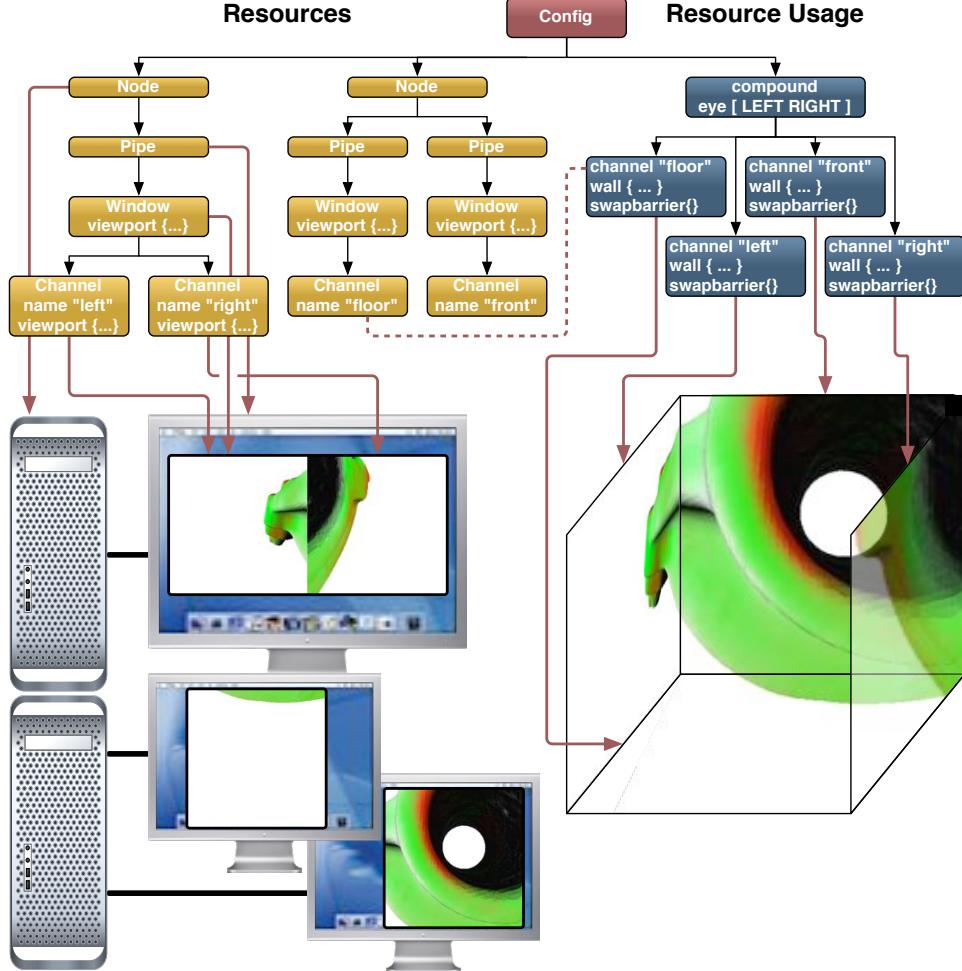


Figure 10: An example configuration

Figure 10 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. The compound description is only used by the server to compute the rendering tasks. The application is not aware of compounds, and does not need to concern itself with the parallel rendering logics of a configuration.

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. Writing Configurations

3.3. Node

For each machine in your cluster, create one `node`. Create one `appNode` if your application process should also render, which is often the case. 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. Make sure the configured `hostname` is reachable from all nodes. An IP address may be used as the hostname.

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.

3.4. 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). In the current implementation it is also one operating system thread. 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.

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 thread¹¹.

Further versions of Equalizer might introduce threaded windows, where all window-related task methods are executed in a separate operating system thread.

3.5. 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`). The window uses an `eq::OSWindow`, 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.

3.6. 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'.

¹¹see <http://www.equalizergraphics.com/documents/design/nonthreaded.html>

3. Writing Configurations

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.

3.7. Canvases

If you are writing a configuration for workstation usage you can skip the following sections and restart with Section 3.10.

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.

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 11 illustrates the wall and projection frustum parameters.

Canvases provide a convenient way to configure projection surfaces. A canvas uses layouts, which describe logical views. The usage of canvases and layouts is optional. Typically, each desktop window uses one canvas.

A canvas represents one logical 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 3.10.3), 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.

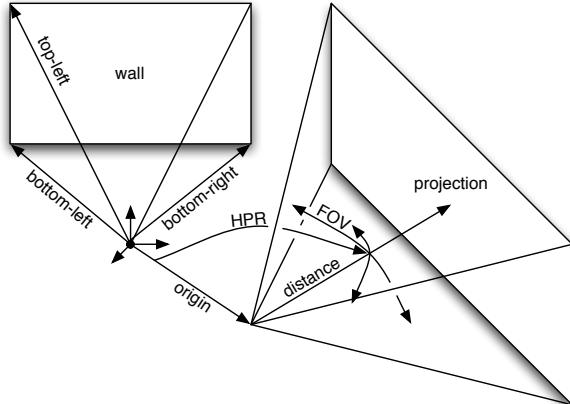


Figure 11: Wall and projection parameters

3. Writing Configurations

3.7.1. 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 3.7. For passive stereo installations, configure one segment per eye pass, where the segment for the left and right eye have the same viewport.

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 with respect to the canvas. Segments might overlap (edge-blended projectors) or have gaps between each other (display walls, Figure 12¹²). The viewport is used to configure the segment’s default frustum from the canvas frustum description, and to place layout views correctly.

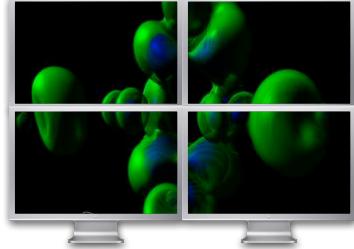


Figure 12: A canvas using four, non-consecutive segments

3.8. 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.

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.

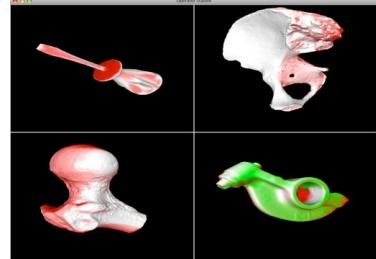


Figure 13: A layout with four views

3.8.1. Views

Configure one **view** for each logical view in each layout. Set the **viewport** to position the view.

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.

¹²Dataset courtesy of VolVis distribution of SUNY Stony Brook, NY, USA.

3. Writing Configurations



Figure 14: A display wall using a six-segment canvas with a two-view layout

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.

3.9. 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, and an eye separation. 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.

3.10. Compounds

Compound trees are used to describe how the rendering resources are combined to produce the desired output, and how they 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 command line program `configTool`, shipped with the Equalizer distribution, creates some standard configurations automatically.

3. Writing Configurations

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.

3.10.1. 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.

Standalone channels Set the channel by using the channel name from the resource section. Define a frustum as explained in Section 3.7.

View/Segment channels Set the channel by using the canvas, segment, layout and view name. 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 activate at runtime.

Software swap lock When using software swap synchronization, swap-lock the destination compounds using a swapbarrier. All windows with a swapbarrier of the same name synchronize their swapbuffers.

Hardware swap lock When using hardware swap synchronization, use swapbarriers for the destination compounds, 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. Please refer to your OpenGL driver documentation for details.

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`, `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

3. Writing Configurations

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.

3.10.2. 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. Unused channels, windows, pipes and nodes are not instantiated during config initialization. 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.

3.10.3. Frustum

Compounds have a frustum description to define the physical layout of the display environment. The frustum specification is described in Section 3.7.

The frustum description is inherited by the children, therefore the frustum is typically defined on the topmost compound.

3.10.4. 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.

3.10.5. 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 never executes 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.

3.10.6. 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. 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. Writing Configurations

3.10.7. 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 7.7. 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.

3.10.8. Synchronization - Swapbarriers

Compounds may have a software or hardware swapbarrier to synchronize the buffer swap of multiple channels. A software swapbarrier is configured by giving it a name. Windows with a swapbarrier 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.

Note that a software swap barrier is not accurate enough for edge-blended projection system or immersive installations. Such installations typically use a hardware synchronization, e.g., nVidia G-Sync cards. It is however sufficient for display walls made out of LCD's and for some non-immersive projection systems.

Swapbarrier 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 swapbarrier 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.

3.10.9. 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 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 on any application for 2D, and on 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.

2D load-balancing increases the framerate over a static decomposition in virtually all cases. It works best if the application data is relatively uni-

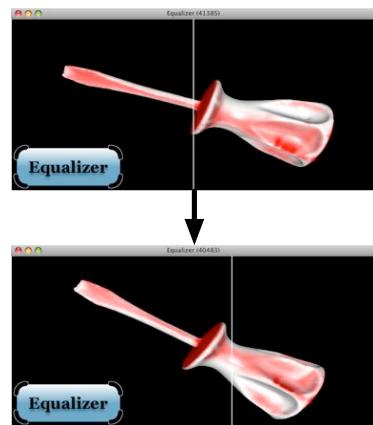


Figure 15: 2D load-balancing

3. Writing Configurations

formly 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 16¹³ 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 left 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.

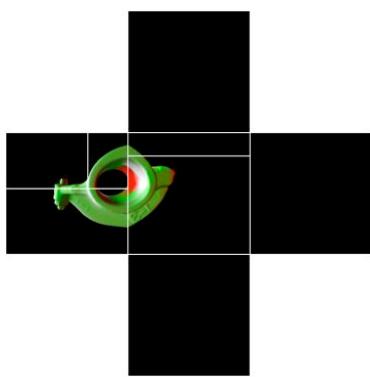


Figure 17: Cross-segment load balancing for a CAVE

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

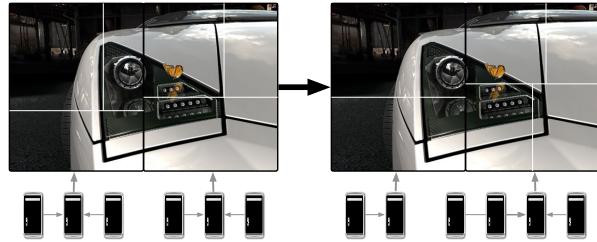


Figure 16: Cross-segment load-balancing for two segments using eight GPU's

Figure 17 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

¹³Image Copyright Realtime Technology AG, 2008

3. Writing Configurations

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 18¹⁴ 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 downscaling the rendering resolution, it will also upscale the resolution if the parameters allow for it. 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 19 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.

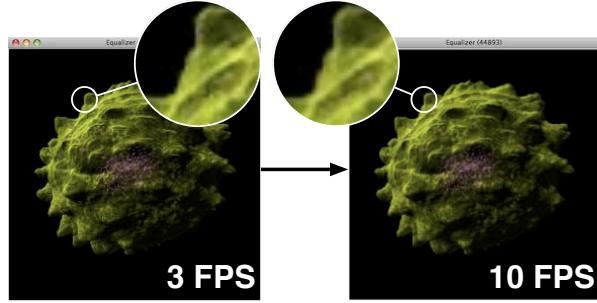


Figure 18: Dynamic Frame Resolution

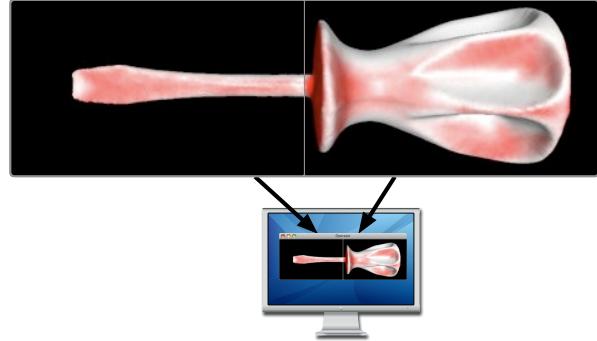


Figure 19: Monitoring a projection wall

¹⁴Data set courtesy of Olaf Ronneberger, Computer Science Institute, University of Freiburg, Germany

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.

4. Hello, World!

The `eqHello` example is a minimal application to illustrate the basic principle of an Equalizer application: The application developer has to implement the rendering method `Channel::frameDraw`, similar to the `glutDisplayFunc` in GLUT applications. It can be run as a stand-alone application from the command line.

The `eqHello` redraw function renders six colored quads, rotating around the origin. The `frameDraw` method provided by the `eq::Channel` can be used as a convenience function to setup the frustum and other OpenGL state. After setting up some lighting parameters, `eqHello` rotates the scene and renders the quads using immediate mode:

```
void Channel::frameDraw( const uint32_t spin )
{
    // setup OpenGL State
    eq::Channel::frameDraw( spin );

    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 );

    // rotate scene around the origin
    glRotatef( static_cast<float>( spin ) * 0.5f, 1.0f, 0.5f, 0.25f );

    // render six axis-aligned colored quads around the origin
    [...]
}
```

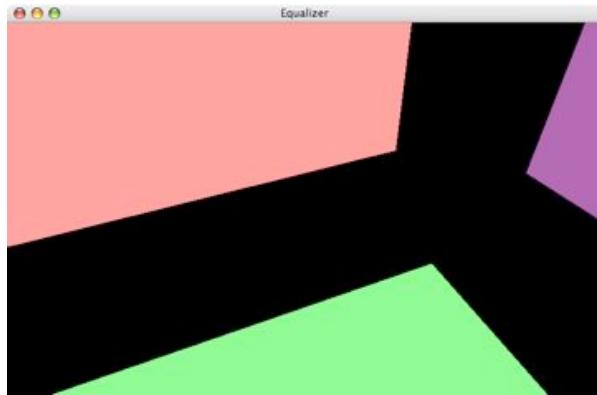


Figure 20: Hello, World!

The `eqHello` main function sets up the communication with the server, initializes and drives the rendering. The details of this setup are explained in Section 6.

5. The Programming Interface

Equalizer uses a C++ programming interface. The API is minimally invasive, so Equalizer imposes only a minimal, natural execution framework upon the applica-

5. The Programming Interface

tion. 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 rendering.

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.

5.1. Namespaces

Application developers are exposed to the following namespaces:

eq::base The `eq::base` namespace provides C++ classes to abstract the underlying operating system and implements common helper functionality. One example is the `eq::base::Clock` class providing a high-resolution timer. Classes in this namespace are fully documented in the API documentation on the Equalizer website, and are not subject of this Programming Guide.

eq::net The Equalizer network layer provides basic functionality for network communication, such as `Connection` and `ConnectionSet`, as well as higher-level functionality such as `Node`, `Session` and `Object`. Please refer to Section 8 for an introduction into the network layer, and to Section 6.3 for distributed objects.

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.

Examples shipped with Equalizer are implemented in their own namespaces, e.g., `eqPly` or `eVolve`. They rely mostly on subclassing classes from the `eq` namespace, with the occasional usage of functionality from the `eq::net` and `eq::base` namespace.

The functionality of the Equalizer server is implemented in the `eq::server` namespace. This namespace does not yet expose an API intended to be used by developers.

Figure 21 shows the namespace layering for `eqPly` and `eqServer`.

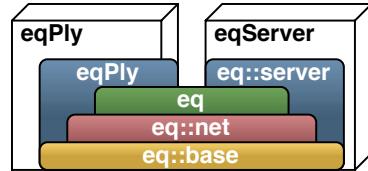


Figure 21: Namespace layering for `eqPly` and `eqServer`

5.2. Task Methods

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 6 will discuss the important task methods. A full list can be found on the website¹⁵.

5.3. 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 is the OpenGL ‘threading model’, which maintains a current

¹⁵see <http://www.equalizergraphics.com/documents/design/taskMethods.html>

6. The eqPly polygonal renderer

context for each thread. If structured correctly, the application rarely has to take care of thread synchronization or protection of shared data.

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 22.

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.

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 are pipelined better.

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 7.2.2. Some scene graphs implement multi-buffered data, and can profit from relaxing the frame synchronization.

6. The eqPly polygonal renderer

In this section the source code of eqPly is explained 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.

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.

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

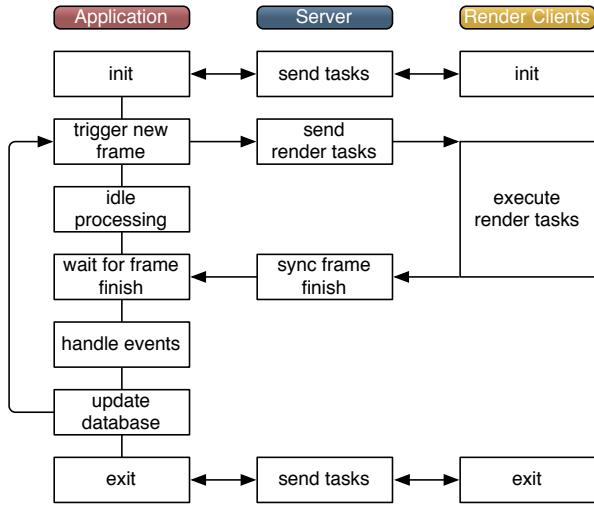


Figure 22: Simplified execution model

6. The eqPly polygonal renderer

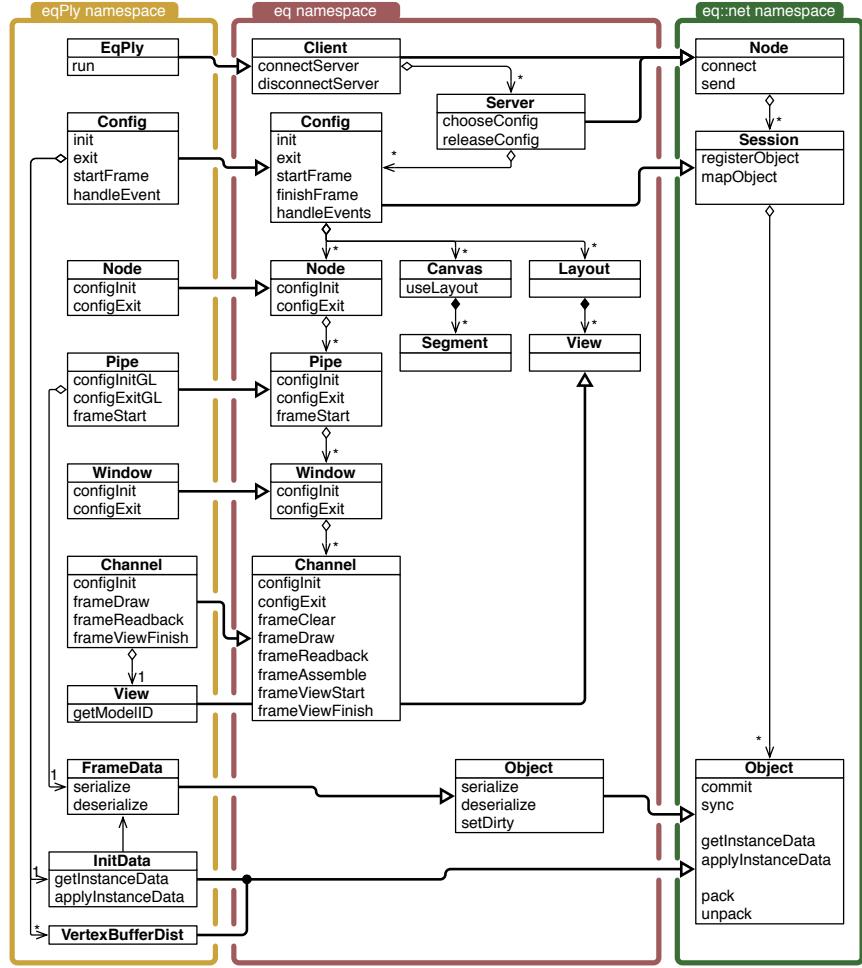


Figure 23: UML diagram eqPly and relevant Equalizer classes

global namespace by windowing systems. Figure 23 shows how the most important Equalizer classes are used through inheritance by the eqPly example.

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.

The distributed data classes are helper classes based on eq::Object or its parent class eq::net::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.

6.1. The main Function

The main function starts off with parsing 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 6.3.3:

```
int main( int argc, char** argv )
{
    // 1. parse arguments
    eqPly::LocalInitData initData;
```

6. The eqPly polygonal renderer

```
 initData.parseArguments( argc, argv );
```

The second step is to redirect all Equalizer output to a file if this was requested through the command line. This should happen before initializing Equalizer, so that output during initialization is redirected and threads launched by Equalizer will use the given log stream. Any `std::ostream` can be used as log output:

```
// 2. Redirect log output to file, if requested
std::ofstream* logFile = 0;
const std::string& logFilename = initData.getLogFilename();
if( !logFilename.empty() )
{
    logFile = new ofstream( logFilename.c_str() );
    eq::base::Log::setOutput( *logFile );
}
```

The third step is to initialize the Equalizer library. The initialization function of Equalizer also parses the command line, which is used to set certain default values based on Equalizer-specific options¹⁶, e.g., the default server address. Furthermore, a `NodeFactory` is provided. The `EQERROR` macro, and its counterparts `EQWARN`, `EQINFO` and `EQVERB` allow selective debugging outputs with various logging levels:

```
// 2. Equalizer initialization
NodeFactory nodeFactory;
if( !eq::init( argc, argv, &nodeFactory ) )
{
    EQERROR << "Equalizer_init_failed" << 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 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() { return new eqPly::View(); }
};
```

The fourth 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 `eq::net::Node`. The underlying Equalizer network layer, discussed in Section 8, is a peer-to-peer network of `eq::net::Nodes`. 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

¹⁶Equalizer-specific options always start with `--eq-`

6. The eqPly polygonal renderer

server and the rendering clients. The listening socket(s) can be configured using the `-eq-listen` command line parameter, or by programmatically adding connection descriptions to the client before the local initialization.

```
// 4. initialization of local client node
RefPtr< eqPly::Application > client = new eqPly::Application( initData );
if( !client->initLocal( argc, argv ) )
{
    EQERROR << "Can't_init_client" << endl;
    eq::exit();
    return EXIT_FAILURE;
}
```

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

```
// 5. run client
const int ret = client->run();
```

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

```
// 6. cleanup and exit
client->exitLocal();
client = 0;

eq::exit();

if( logFile )
    logFile->close();
delete logFile;

return ret;
}
```

6.2. 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 pre-started 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 runs¹⁷.

6.2.1. Main Loop

The application's main loop starts by connecting the application to an Equalizer server. The command line parameter `-eq-server` explicitly specifies a server address. If no server was specified, `Client::connectServer` tries first to connect to a server on

¹⁷see <http://www.equalizergraphics.com/documents/design/residentNodes.html>

6. The eqPly polygonal renderer

the local machine using the default port. If that fails, it will create a server running within the application process with a default one-channel configuration¹⁸.

Another configuration can be selected using the command-line parameter `-eq-config`. This configuration may only contain an `appNode`, i.e., no distributed configuration is possible since the process-local server does not create a listening socket.

```
int Application :: run()
{
    // 1. connect to server
    eq::ServerPtr server = new eq::Server;
    if( !connectServer( server ) )
    {
        EQERROR << "Can't_open_server" << 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 choosing the 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 cast necessary:

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

if( !config )
{
    EQERROR << "No_matching_config_on_server" << endl;
    disconnectServer( server );
    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. It returns `true` only if all initialization task methods were successful.

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
eq::base::Clock clock;

config->setInitData( _initData );
if( !config->init( ) )
{
    EQERROR << "Config_initialization_failed:" <<
        config->getErrorMessage() << endl;
    server->releaseConfig( config );
    disconnectServer( server );
    return EXIT_FAILURE;
}

EQLOG( eq::LOG_STATS ) << "Config_init_took_" << clock.getTimef() << "ms"
    << 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

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

6. The eqPly polygonal renderer

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 7.2.2.

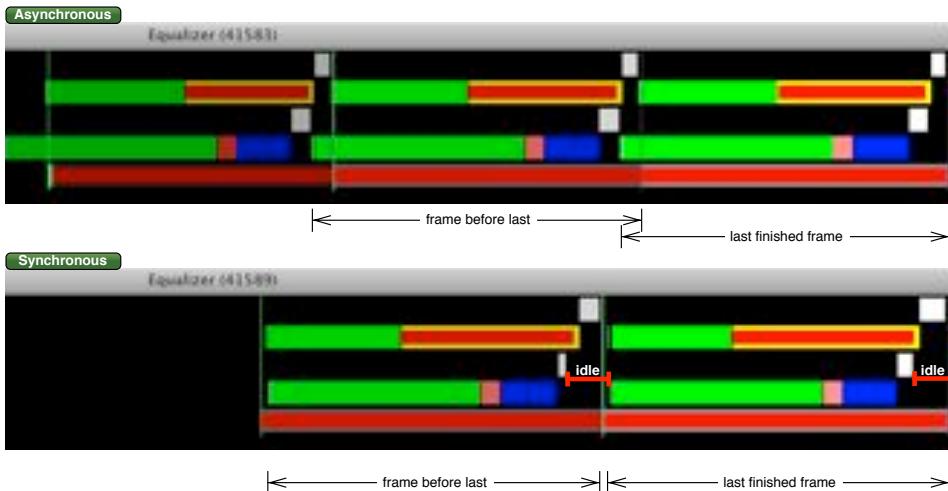


Figure 24: Synchronous and asynchronous execution

Figure 24 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.

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:

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

clock.reset();
```

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

6. The eqPly polygonal renderer

```

while( config->isRunning( ) && maxFrames— )
{
    config->startFrame();
    // config->renderData(...);
    config->finishFrame();

    while( !config->needsRedraw( ) ) // 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 ) )
                EQVERB << "Unhandled_" << event << endl;
        }
        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.getTimef();
EQLOG( eq::LOG_STATS ) << "Rendering_took_" << time << "ms(" << frame
    << "frames@_" << ( frame / time * 1000.f )
    << "FPS)" << 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();
EQLOG( eq::LOG_STATS ) << "Exit_took_" << clock.getTimef() << "ms" << endl;

// 6. cleanup and exit
server->releaseConfig( config );
if( !disconnectServer( server ) )
    EQERROR << "Client::disconnectServer failed" << endl;
server = 0;
return EXIT_SUCCESS;
}

```

6.2.2. 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:

```

bool Application::clientLoop()
{
    if( !_initData.isResident( ) ) // execute only one config run
        return eq::Client::clientLoop();

    // else execute client loops 'forever'
    while( true ) // TODO: implement SIGHUP handler to exit?
    {
        if( !eq::Client::clientLoop( ) )
            return false;
    }
}

```

6. The eqPly polygonal renderer

```

    EQINFO << "One_configuration_run_successfully_executed" << endl;
}
return true;
}

```

6.3. Distributed Objects

Equalizer provides distributed objects which facilitate the implementation of data distribution in a cluster environment. Distributed objects are created by subclassing from `eq::Object` or `eq::net::Object`. The application programmer implements serialization and deserialization of the distributed data.

Objects are dynamically assigned to a session, which assigns session-unique identifiers to objects and manages the network communication between different instances of the same distributed object.

The master version of a distributed object is registered with a `eq::net::Session`, which assigns a session-unique identifier to the object. This identifier can be used by other nodes to map their instance of the object, thus synchronizing the object's data with the remotely registered master version.

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.

6.3.1. Common Usage

Distributed objects are addressed using session-unique identifiers, because pointers to other objects cannot be distributed directly; they have no meaning on remote nodes. The session used to register distributed objects is normally the `eq::Config`, which is sub-classed from `eq::net::Session`.

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 typically 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.

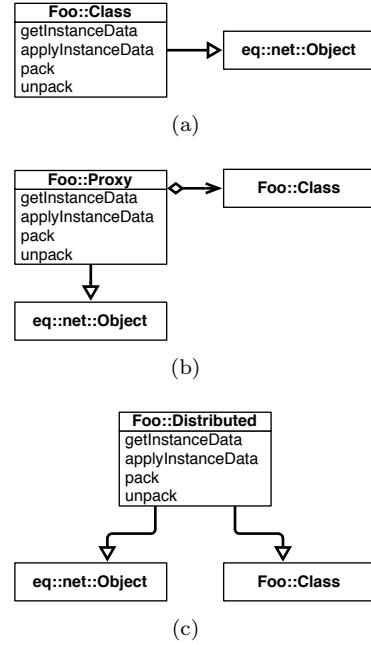


Figure 25: Object distribution using subclassing, proxies or multiple inheritance

6. The eqPly polygonal renderer

There are multiple ways of implementing data distribution for an existing C++ class hierarchy:

Subclassing The classes to be distributed inherit from `eq::net::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 25(a))

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 25(b))

Multiple Inheritance A new class inheriting from the class to be distributed and from `eq::net::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 25(c))

6.3.2. Change Handling

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 following change types are possible:

STATIC The object is not versioned. The instance data is serialized whenever a new slave instance is mapped. 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. 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 and saved 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 whenever a new slave instance is mapped. The application can choose to use a different, more optimal implementation to pack deltas.

6.3.3. 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 method 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 data transport between nodes. They perform no type checking or transformation on the data. It is the application's

6. The eqPly polygonal renderer

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( eq::net::DataOStream& os )
{
    os << _frameDataID << _windowSystem << _renderMode << _useGLSL << _invFaces
    << _logo;
}

void InitData :: applyInstanceData( eq::net::DataIStream& is )
{
    is >> _frameDataID >> _windowSystem >> _renderMode >> _useGLSL >> _invFaces
    >> _logo;

EQASSERT( _frameDataID != EQ_ID_INVALID );
EQINFO << "New_InitData_instance" << endl;
}
```

6.3.4. 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:

- Only the master instance of the object is writable, that is, `eq::net::Object::commit` can be called only on the master instance to generate a new version.
- Slave instance versions can only be advanced, that is, `eq::net::Object::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 `Session::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.

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.

6. The eqPly polygonal renderer

The creation of distributed, versioned objects is further facilitated by `eq::Object`, which implements the most 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 `eq::net::Object` which leaves this flexibility to the developer. Figure 26 shows the relationship between `eq::Object` and `eq::net::Object`.

The `FrameData` is sub-classed from `eq::Object`, and consequently tracks its changes by setting the appropriate dirty bit whenever it is changed. The serialization methods are called by the `eq::Object` 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 = eq::Object::DIRTY_CUSTOM << 0,
    DIRTY_FLAGS = eq::Object::DIRTY_CUSTOM << 1,
    DIRTY_VIEW = eq::Object::DIRTY_CUSTOM << 2,
    DIRTY_MESSAGE = eq::Object::DIRTY_CUSTOM << 3,
};

void FrameData::serialize( eq::net::DataOStream& os, const uint64_t dirtyBits )
{
    eq::Object::serialize( os, dirtyBits );
    if( dirtyBits & DIRTY_CAMERA )
        os << _translation << _rotation << _modelRotation;
    if( dirtyBits & DIRTY_FLAGS )
        os << _modelID << _renderMode << _colorMode << _ortho << _statistics
            << _help << _wireframe << _pilotMode;
    if( dirtyBits & DIRTY_VIEW )
        os << _currentViewID;
    if( dirtyBits & DIRTY_MESSAGE )
        os << _message;
}

void FrameData::deserialize( eq::net::DataIStream& is,
                            const uint64_t dirtyBits )
{
    eq::Object::deserialize( is, dirtyBits );
    if( dirtyBits & DIRTY_CAMERA )
        is >> _translation >> _rotation >> _modelRotation;
    if( dirtyBits & DIRTY_FLAGS )
        is >> _modelID >> _renderMode >> _colorMode >> _ortho >> _statistics
            >> _help >> _wireframe >> _pilotMode;
    if( dirtyBits & DIRTY_VIEW )
        is >> _currentViewID;
    if( dirtyBits & DIRTY_MESSAGE )
        is >> _message;
}
```

6.3.5. Scene Data

Some applications might 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.

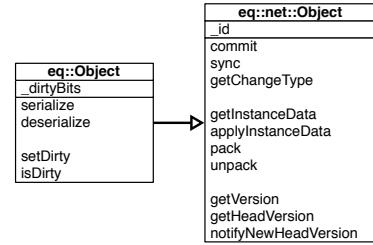


Figure 26: `eq::Object` and `eq::net::Object`

6. The eqPly polygonal renderer

eqPly uses static distributed objects to distribute the model loaded by the application. This approach 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, an external 'mirror' hierarchy is constructed around the 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 27 shows the UML hierarchy of the model and distribution classes.

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 `applyInstanceData` 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( eq::net::DataOStream& os )
{
    EQASSERT( _node );
    os << _isRoot;

    if( _left && _right )
    {
        os << _left->getID() << _right->getID();

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

            os << data.vertices << data.colors << data.normals << data.indices;
        }
    }
    else
    {
        os << EQ_ID_INVALID << EQ_ID_INVALID;

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

        os << leaf->_vertexStart << leaf->_vertexLength << leaf->_indexStart
            << leaf->_indexLength;
    }
}
```

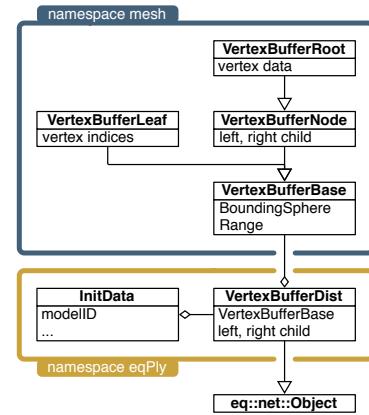


Figure 27: Scene Data in eqPly

6. The eqPly polygonal renderer

```
    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 28 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 deserializes 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.

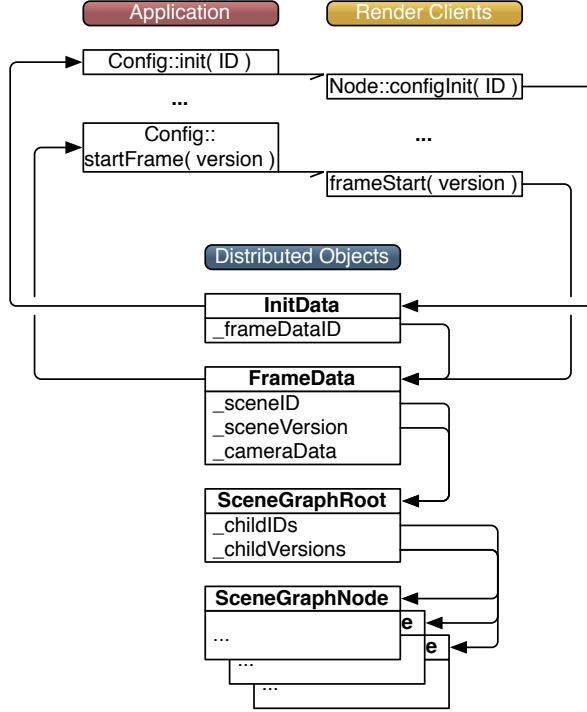


Figure 28: Scene Graph Distribution

6.4. Config

The configuration 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.

6.4.1. 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 is necessary to allow a speedy startup of the configuration 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 29 shows a sequence diagram of the config initialization.

The `eqPly::Config` class holds the master versions of the initialization and frame data. Both objects are registered with the `eq::Config`, which is the `eq::net::Session` used for rendering. Equalizer takes care of the session setup and exit in `Client::chooseConfig` and `Client::releaseConfig`, respectively.

6. The eqPly polygonal renderer

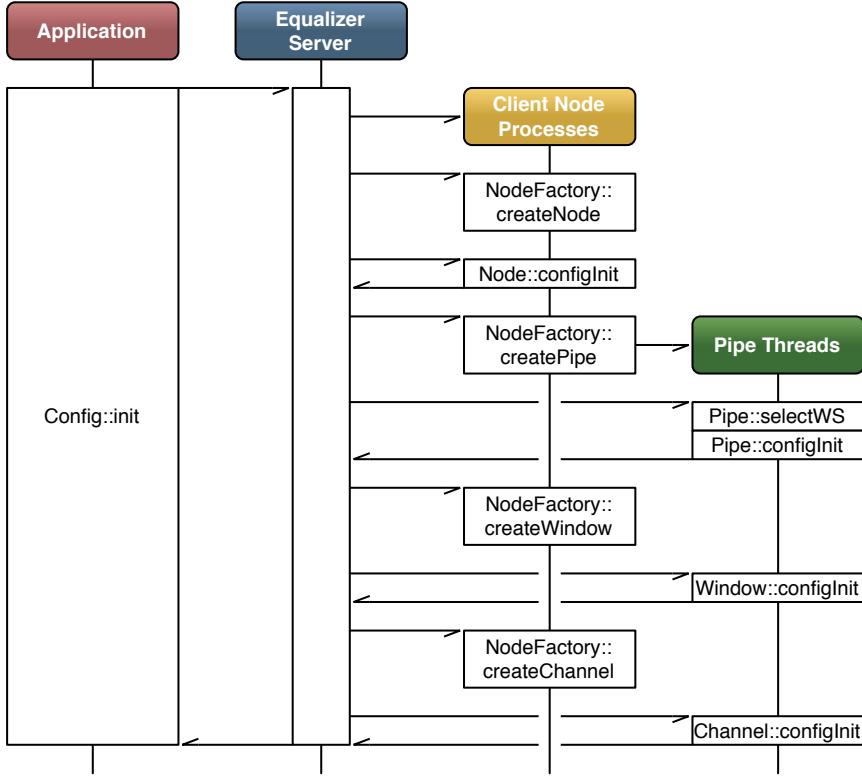


Figure 29: Config Initialization Sequence

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

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

```
bool Config :: init ()
{
    // init distributed objects
    _frameData.data.color      = _ initData.useColor();
    _frameData.data.renderMode = _ initData.getRenderMode();
    registerObject( &_frameData );

    _ initData.setFrameDataID( _frameData.getID( ) );
    registerObject( &_ initData );

    // init config
    _running = eq::Config :: init( _ initData.getID( ) );
    if( !_running )
        return false;
}
```

If the configuration was initialized correctly, 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 implementations²⁰, which can and have been easily integrated with Equalizer. The example code in `eqPly` provides a reference implementation for the integration of such a tracking library. Section 7.5 provides more background on head tracking.

```
// init tracker
if( !_ initData.getTrackerPort().empty( ) )
{
    if( !_ tracker.init( _ initData.getTrackerPort() ) )
```

²⁰VRCO Trackd, VRPN, etc.

6. The eqPly polygonal renderer

```

EQWARN << "Failed_to_initialise_tracker" << endl;
else
{
    // Set up position of tracking system wrt world space
    // Note: this depends on the physical installation.
    vmm::Matrix4f m( vmm::Matrix4f::IDENTITY );
    m.scale( 1.f, 1.f, -1.f );
    _tracker.setWorldToEmitter( m );

    m = vmm::Matrix4f::IDENTITY;
    m.rotateZ( -M_PI_2 );
    _tracker.setSensorToObject( m );
    EQINFO << "Tracker_initialized" << endl;
}
}

return true;
}

```

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()
{
    _running = false;
    const bool ret = eq::Config::exit();

    _initData.setFrameDataID( EQ_ID_INVALID );
    deregisterObject( &_initData );
    deregisterObject( &_frameData );

    return ret;
}

```

6.4.2. Frame Control

The rendering frames are issued by the application. The `eqPly::Config` only overrides `startFrame` to update its data before forwarding the start frame request to the `eq::Config`.

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 7.5.

The camera position is updated and the frame data is committed, which generates a new version of this object. 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()
{
    // update head position
    if( _tracker.isRunning() )
    {
        _tracker.update();
        const vmm::Matrix4f& headMatrix = _tracker.getMatrix();
        setHeadMatrix( headMatrix );
    }

    // update database
    _frameData.data.rotation.preRotateX( -0.001f * _spinX );
    _frameData.data.rotation.preRotateY( -0.001f * _spinY );
    const uint32_t version = _frameData.commit();

    return eq::Config::startFrame( version );
}

```

6. The eqPly polygonal renderer

6.4.3. 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:

```
bool Config::handleEvent( const eq::ConfigEvent* event )
{
    switch( event->type )
    {
        [...]
        case eq::ConfigEvent::POINTER_MOTION:
            if( event->pointerMotion.buttons == eq::PTR_BUTTON_NONE )
                return true;

            if( event->pointerMotion.buttons == eq::PTR_BUTTON1 )
            {
                _spinX = 0;
                _spinY = 0;

                _frameData.data.rotation.preRotateX(
                    -0.005f * event->pointerMotion.dx );
                _frameData.data.rotation.preRotateY(
                    -0.005f * event->pointerMotion.dy );
                _redraw = true;
            }
            else if( event->pointerMotion.buttons == eq::PTR_BUTTON2 ||
                      event->pointerMotion.buttons == ( eq::PTR_BUTTON1 |
                      eq::PTR_BUTTON3 ) )
            {
                _frameData.data.translation.z +=
                    .005f * event->pointerMotion.dy;
                _redraw = true;
            }
            else if( event->pointerMotion.buttons == eq::PTR_BUTTON3 )
            {
                _frameData.data.translation.x +=
                    .0005f * event->pointerMotion.dx;
                _frameData.data.translation.y ==
                    .0005f * event->pointerMotion.dy;
                _redraw = true;
            }
            return true;

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

6.4.4. 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 `View`. The

6. The eqPly polygonal renderer

loaded models are evenly distributed over the available views of the configuration, as shown in Figure 13.

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 config maintains the mapped models, and lazily maps new models, which are registered by the application process:

```
const Model* Config::getModel( const uint32_t modelID )
{
    if( modelID == EQ_ID_INVALID ) // no model loaded by application
        return 0;

    // Accessed concurrently from pipe threads
    eq::base::ScopedMutex< eq::base::SpinLock > _mutex( _modelLock );

    const size_t nModels = _models.size();
    EQASSERT( _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()->mapModel( this, modelID );
    EQASSERT( model );
    _models.push_back( model );

    return model;
}
```

6.4.5. 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 using different models to illustrate the concept:

```
bool Config::_handleKeyEvent( const eq::KeyEvent& event )
{
    switch( event.key )
    {
        ...
        case 'm':
        case 'M':
        {
            if( _modelDist.empty( ) ) // no models
                return true;

            // current model
            const uint32_t viewID = _frameData.getCurrentViewID();
            View* view = static_cast< View* >( findView( viewID ) );
            const uint32_t currentID = view ?
```

6. The eqPly polygonal renderer

```

view->getModelID() : _frameData.getModelID();

// next model
ModelDistVector::const_iterator i;
for( i = _modelDist.begin(); i != _modelDist.end(); ++i )
{
    if( (*i)->getID() != currentID )
        continue;

    ++i;
    break;
}
if( i == _modelDist.end() )
    i = _modelDist.begin(); // wrap around

// set identifier on view or frame data (default model)
const uint32_t modelID = (*i)->getID();
if( view )
    view->setModelID( modelID );
else
    _frameData.setModelID( modelID );
return true;
}
...
}

```

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 will 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.

6.5. 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`.

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 7.2 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 uint32_t initID )
{
    if( !eq::Node::configInit( initID ) )
        return false;

    // All render data is static or multi-buffered, we can run asynchronously
    if( getIAttribute( IATTR_THREAD_MODEL ) == eq::UNDEFINED )
        setIAttribute( IATTR_THREAD_MODEL, eq::ASYNC );

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

```

6. The eqPly polygonal renderer

```

    return config->mapData( initData );
}

```

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 the data has been retrieved:

```

bool Config::mapData( const uint32_t initDataID )
{
    if( _initData.getID() == EQ_ID_INVALID )
    {
        EQCHECK( mapObject( &_initData , initDataID ) );
        unmapObject( &_initData ); // data was retrieved, unmap immediately
    }
    else // appNode, _initData is registered already
        EQASSERT( _initData.getID() == initDataID );
}

```

6.6. Pipe

All task methods for a pipe and its children are executed in a separate thread. This approach optimizes usage of the GPU, since all tasks are executed serially and therefore do not compete for resources or cause OpenGL context switches. Later versions of Equalizer might introduce threaded windows to allow the parallel and independent execution of rendering tasks on a single pipe.

6.6.1. Initialization and Exit

Pipe threads are not explicitly synchronized with each other, that is, pipes might be rendering different frames at any given time. Therefore frame-specific data has to be allocated for each pipe thread, which in the case of `eqPly` is 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 uint32_t initID )
{
    const Config* config = static_cast<Config*>( getConfig( ) );
    const InitData& initData = config->getInitData();
    const uint32_t frameDataID = initData.getFrameDataID();
    eq::Config* config = getConfig();

    const bool mapped = config->mapObject( &_frameData , frameDataID );
    EQASSERT( mapped );

    return eq::Pipe::configInit( initID );
}

```

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();
}

```

6. The eqPly polygonal renderer

6.6.2. 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 concurrently²¹.

Each pipe might use a different window system for rendering, which is determined before Pipe::configInit by Pipe::selectWindowSystem. The default implementation of selectWindowSystem loops over all window systems and returns the first supported window system, determined by using supportsWindowSystem.

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<Config*>( getConfig( ) );
    const InitData& initData = config->getInitData();
    const eq :: WindowSystem ws = initData.getWindowSystem();

    if( ws == eq :: WINDOW_SYSTEM_NONE )
        return eq :: Pipe :: selectWindowSystem();
    if( !supportsWindowSystem( ws ) )
    {
        EQWARN << "Window_system_" << ws
            << "_not_supported,_using_default_window_system" << endl;
        return eq :: Pipe :: selectWindowSystem();
    }

    return ws;
}
```

6.6.3. 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 various 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.

6.6.4. 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 then unlocks its child entities by calling startFrame:

```
void Pipe :: frameStart( const uint32_t frameID, const uint32_t frameNumber )
{
    node->waitFrameStarted( frameNumber );

    _frameData.sync( frameID );
    startFrame( frameNumber );
```

²¹see <http://www.equalizergraphics.com/compatibility.html>

6. The eqPly polygonal renderer

}

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

6.7.1. 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::OSWindow`. The `OSWindow` class defines the minimal interface to be implemented for a new window system. Each `Window` uses one `OSWindow` 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: 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 implementation derived from these interfaces provides a sample implementation which honors all configurable window attributes.

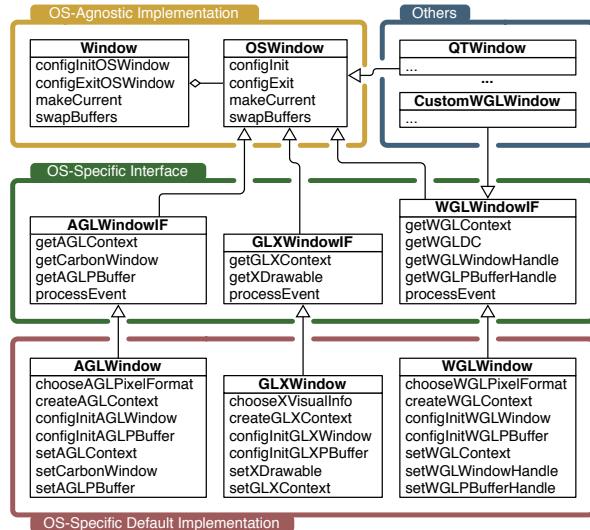


Figure 30: `eq::OSWindow` UML class hierarchy

The initialization sequence uses multiple, override-able task methods. The main task method `configInit` calls first `configInitOSWindow`, which creates and initializes the `OSWindow` for this window. The `OSWindow` initialization code is implementation specific. If the `OSWindow` 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 `OSWindow` 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

6. The eqPly polygonal renderer

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 uint32_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( GL_BACK );

    EQASSERT( !_state );
    _state = new VertexBufferState( getObjectManager() );
    _loadLogo();

    const Config* config = static_cast< const Config* >( getConfig() );
    const InitData& initData = config->getInitData();
    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 6.7.3.

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 `configExitOSWindow` which de-initializes the drawable and context and deletes the `OSWindow` allocated in `configInitOSWindow`.

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()
{
    if( _state )
        _state->deleteAll();

    delete _state;
    _state = 0;

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

6.7.3. 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.

6. The eqPly polygonal renderer

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 and spawns all contexts of a single GPU²².

The OM is a template class. The template type is the key used to identify objects. The same key is used by all contexts to get the OpenGL name of an object.

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 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.

6.8. Channel

The channel is the heart of the application, it contains the actual rendering code. The channel is used to perform 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.

6.8.1. 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 uint32_t initID )
{
    setNearFar( 0.1f, 10.0f );
    return true;
}
```

²²<http://www.equalizergraphics.com/documents/design/objectManager.html>

6. The eqPly polygonal renderer

6.8.2. Rendering

The central rendering routine is `Channel::frameDraw`. This routine contains the application's OpenGL rendering code, which is being rendered using 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` method(s) and then calls the appropriate OpenGL function(s). 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()
{
    glReadBuffer( getReadBuffer( ) );
    glDrawBuffer( 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.

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. It is influenced by the compound range attribute.

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:

6. The eqPly polygonal renderer

```

void eq::Channel::frameDraw( const uint32_t frameID )
{
    applyBuffer();
    applyViewport();

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

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

void Channel::frameDraw( const uint32_t frameID )
{
    // Setup OpenGL state
    eq::Channel::frameDraw( frameID );
}

```

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 in Figure 31, 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& translation = frameData.getCameraTranslation();

glMultMatrixf( frameData.getCameraRotation().array );
glTranslatef( translation.x(), translation.y(), translation.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:

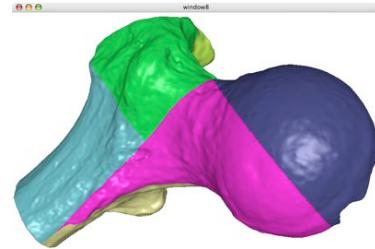


Figure 31: Destination view of a DB compound using demonstrative coloring

6. The eqPly polygonal renderer

```

const Model* model = _getModel();
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 and the camera position 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* model )
{
    Window* window = static_cast<Window*>( getWindow() );
    mesh::VertexBufferState& state = window->getState();

    const Pipe* pipe = static_cast<Pipe*>( getPipe() );
    const FrameData& frameData = pipe->getFrameData();

    const eq::Range& range = getRange();
    vmml::FrustumCuller& culler;

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

    _initFrustum( culler, model->getBoundingSphere() );
    model->beginRendering( state );
}

```

The model data is spatially organized in a 3-dimensional kD-tree²³ 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 32 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

²³See also <http://en.wikipedia.org/wiki/Kd-tree>

6. The eqPly polygonal renderer

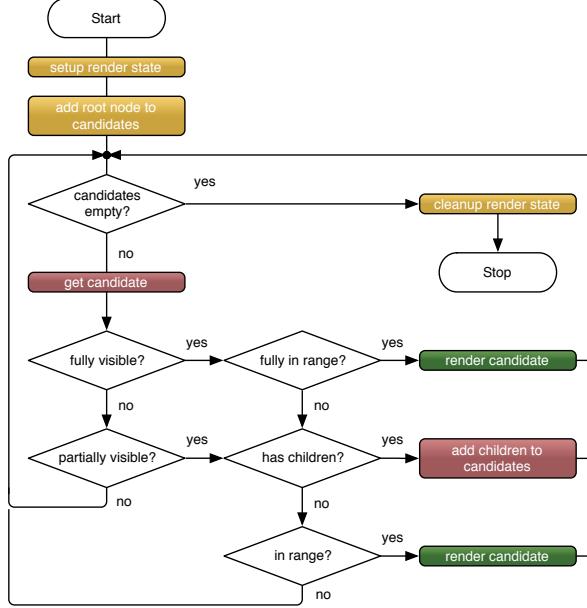


Figure 32: Main Render Loop

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 in `eqPly` looks like this:

```

model->beginRendering( state );

// start with root node
vector< const VertexBufferBase* > candidates;
candidates.push_back( model );

while( !candidates.empty() )
{
    const VertexBufferBase* treeNode = candidates.back();
    candidates.pop_back();

    // completely out of range check
    if( treeNode->getRange()[0] >= range.end ||
        treeNode->getRange()[1] < range.start )
        continue;

    // bounding sphere view frustum culling
    switch( culler.testSphere( treeNode->getBoundingSphere() ) )
    {
        case vmml::VISIBILITY_FULL:
            // if fully visible and fully in range, render it
            if( treeNode->getRange()[0] >= range.start &&
                treeNode->getRange()[1] < range.end )
            {
                treeNode->render( state );
                break;
            }
            // partial range, fall through to partial visibility
        case vmml::VISIBILITY_PARTIAL:
        {
            const VertexBufferBase* left = treeNode->getLeft();
            const VertexBufferBase* right = treeNode->getRight();

            if( !left && !right )
            {
```

7. Advanced Features

```
        if( treeNode->getRange()[0] >= range.start )
            treeNode->render( state );
            // else drop, to be drawn by 'previous' channel
    }
else
{
    if( left )
        candidates.push_back( left );
    if( right )
        candidates.push_back( right );
}
break;
}
case vmml::VISIBILITY_NONE:
    // do nothing
    break;
}
model->endRendering( state );
}
```

6.8.3. Assembly

Like most applications, eqPly can use the default implementation of the `frameReadback` and `frameAssemble` task methods. Only `frameReadback` is overwritten for an optimization. Since eqPly does not need the alpha channel on the destination view, the output frames are flagged to ignore it, which allows the compressor to drop 25% of the data during image transfer:

```
void Channel::frameReadback( const uint32_t frameID )
{
    // OPT: Drop alpha channel from all frames during network transport
    const eq::FrameVector& frames = getOutputFrames();
    for( eq::FrameVector::const_iterator i = frames.begin();
          i != frames.end(); ++i )
    {
        (*i)->setAlphaUsage( false );
    }
    eq::Channel::frameReadback( frameID );
}
```

7. 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 feature. Its purpose is to provide information on how to address a typical problem or use case when developing an Equalizer-based application.

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

7. Advanced Features

handling in `glutMainLoop`. More information on event handling can be found on the Equalizer website²⁴.

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.

7.1.1. 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.

7.1.2. 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 each pipe thread. By default, Equalizer pumps these events automatically for the application in-between executing task methods.

The method `Pipe::useMessagePump` can be overridden to return `false` to not enable message dispatch for the respective thread.

The method `useMessagePump` is called by Equalizer during application and pipe thread initialization (before `configInit!`) to determine if the pipe's thread shall automatically dispatch OS events. For AGL, this affects the node threads and pipe thread, since the node thread message pump needs to dispatch the events to the pipe thread event queue.

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.

7.1.3. Event Data Flow

Events are received by an event handler. The event handler finds the `eq::OSWindow` 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 `OSWindowEvent`, which is passed to the concrete `OSWindow` for processing.

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

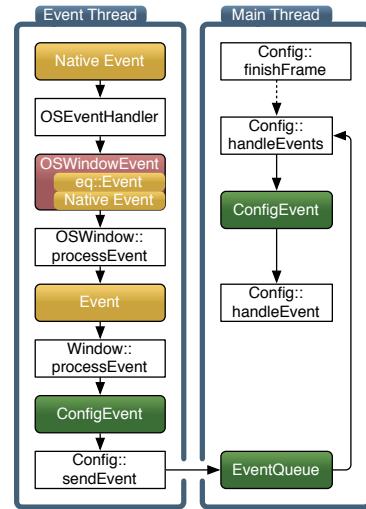


Figure 33: Event Processing

²⁴see <http://www.equalizergraphics.com/documents/design/eventHandling.html>

7. Advanced Features

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 33 shows the overall data flow of an event.

7.1.4. Default Implementation

Equalizer provides an `AGLEventHandler`, `GLXEventHandler` and `WGLEventHandler`, which handle events for an `AGLWindowIF`, `GLXWindowIF` and `WGLWindowIF`, respectively. Figure 34 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.

GLX uses one X11 Display connection for all windows on a GPU, which is managed by the `eq::Pipe`. For each thread, one `GLXEventHandler` is allocated. The pipe registers the display connection with this per-thread event handler. Events for windows created using this display connection are automatically received by the event handler. No per-window event handling setup needs to be done. PBuffers are handled in the same way as window drawables.

The glX event handler iterates over all windows of the registered pipe to find the originating `GLXWindowIF`, 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

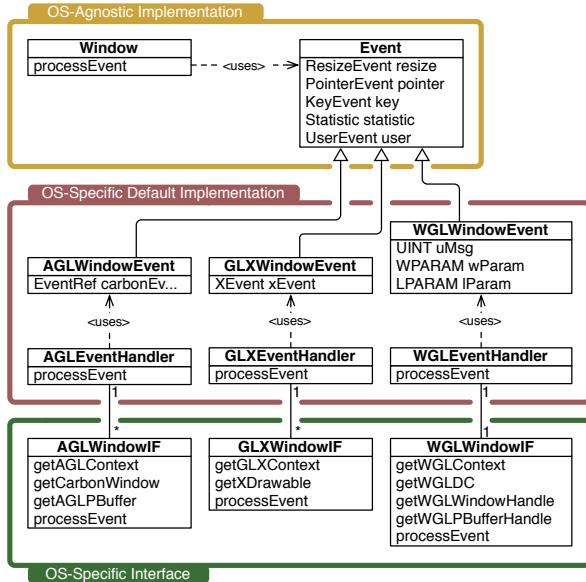


Figure 34: UML Diagram for Event Handling Classes

7. Advanced Features

IParam to `WGLWindowIF::processEvent` as part of the `WGLWindowEvent`. No event handling is set up for PBuffers.

Later Equalizer versions will introduce `Pipe::processEvent` and `PipeEvent` to communicate pipe-specific events, e.g., monitor resolution changes. It is also likely that an `OSPipe` abstraction, similar to the `OSWindow`, will be created to separate the window system specific code from the `eq::Pipe`.

7.1.5. 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::ConfigEvent::USER,
        ASSEMBLE
    };

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

    // channel name is in user event data
    char          formatType[64];
    vmmr::Vector2i area;
    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_SIZE`²⁵ 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:

```
std::ostream& operator << ( std::ostream& os, const ConfigEvent* event );
...
bool Config::handleEvent( const eq::ConfigEvent* event )
{
    switch( event->type )
    {
        case ConfigEvent::READBACK:
        case ConfigEvent::ASSEMBLE:
```

²⁵currently 32 bytes

7. Advanced Features

```

cout << static_cast< const ConfigEvent* >( event ) << endl;
return true;

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

}

```

7.2. 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 5.3. The default, per-node thread synchronization provided by Equalizer can be relaxed by advanced applications to gain better performance through higher asynchronicity.

7.2.1. 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 `eq::net::Object`

mapping. In some special cases the command thread executes 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. The pipe threads are terminated during `Config::exit`.

The rest of this section discusses the thread synchronization between the main thread and the pipe threads.

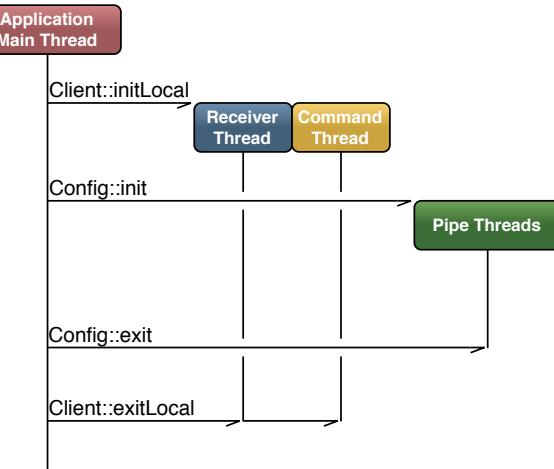


Figure 35: Threads within one node process

7.2.2. Thread Synchronization Models

Equalizer supports three threading models, which can be set programmatically by the application or through the configuration file format. Applications typically

7. Advanced Features

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: Additionaly 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. 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: Additionaly 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`.

Figure 36 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.

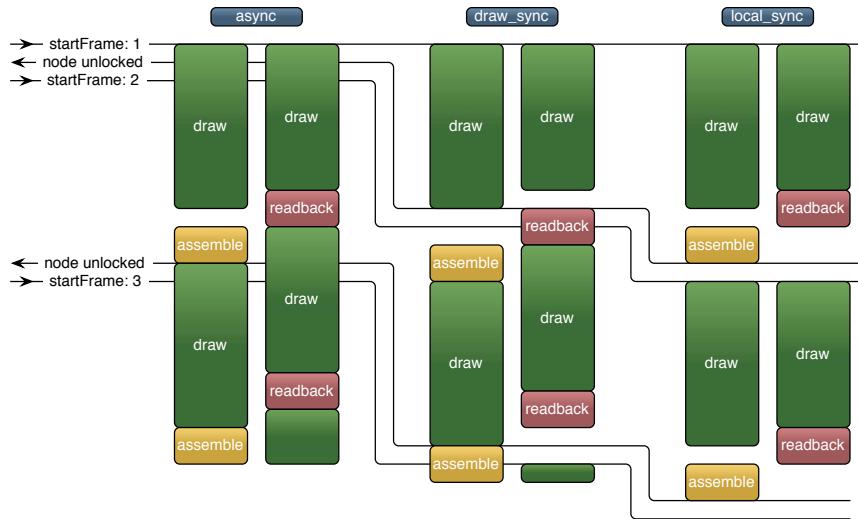


Figure 36: Async, draw_sync and local_sync thread synchronization models

7.2.3. 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.

7. Advanced Features

Non-threaded pipes should not be used for DPlex source and destination channels. Applications using the `local.sync` thread model can not 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.

7.2.4. 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.

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 37 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.

Figure 37 illustrates the local frame synchronization.

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.currentLatency`. The `frameFinish` task methods

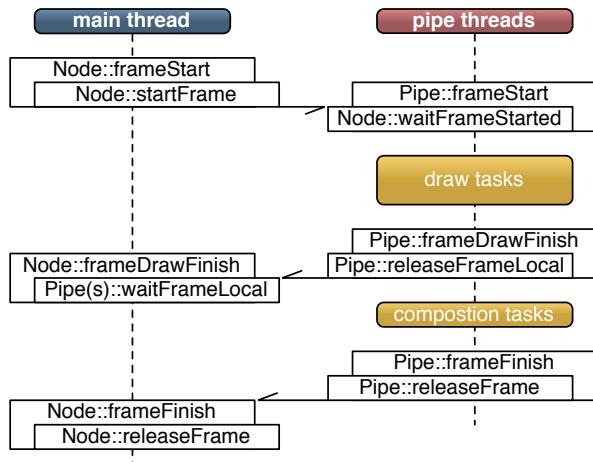


Figure 37: Per-node frame synchronization

7. Advanced Features

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.

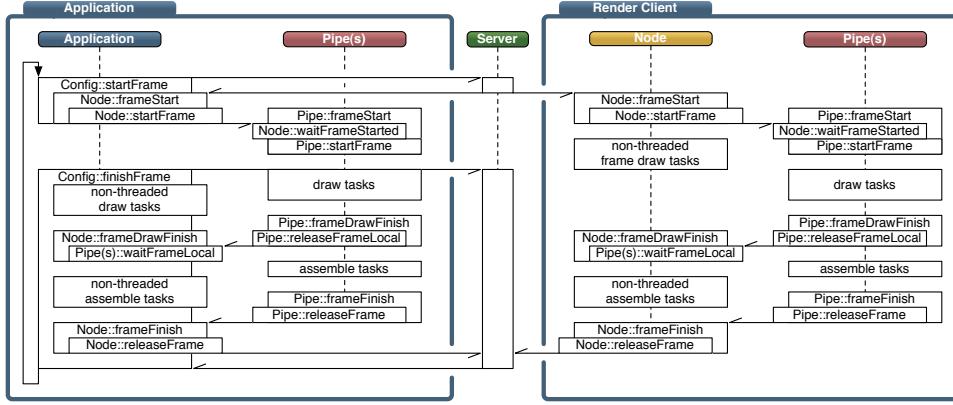


Figure 38: Synchronization of frame tasks

Figure 38 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.

7.3. OpenGL Extension Handling

Equalizer uses GLEW²⁶ for OpenGL extension handling, particularly the GLEW MX implementation providing multi-context support.

Each `eq::Window` has a `GLEWContext`. This context can be obtained by using `glewGetContext` on the window or channel. GLEW MX uses this function to dispatch the functions to the correct context. Equalizer (re-)initializes the GLEW context whenever a new OpenGL context is set on the window.

Extended OpenGL functions called from a 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. */

```

²⁶<http://glew.sourceforge.net>

7. Advanced Features

```
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 );
    ...
}
```

7.4. 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 `OSWindow` 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 `OSWindow` and implement the abstract methods of this interface class.

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

Since window initialization is notoriously error-prone and hard to debug, the sample implementation propagates the reason for errors from the render clients back to the application. The `Pipe` and `Window` classes have a `setErrorString` method, which is used to set an error string. This string is passed to the `Config` instance on the application node, where it can be retrieved using `getErrorMessage`.

The sample implementations `AGLWindow`, `GLXWindow` and `WGLWindow` 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.

7.4.1. 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 A), 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:

7. Advanced Features

```
bool eVolve::Window::configInit( const uint32_t initID )
{
    // Enforce alpha channel, since we need one for rendering
    setIAAttribute( IATTR_PLANES_ALPHA, 8 );

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

7.4.2. 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 6.6.3 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 AGLWindow::configInit( )
{
    AGLPixelFormat pixelFormat = chooseAGLPixelFormat();
    if( !pixelFormat )
        return false;

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

    if( !context )
        return false;

    return configInitAGLDrawable();
}
```

7.4.3. GLX Window Initialization

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

```
bool GLXWindow::configInit( )
{
    XVisualInfo* visualInfo = chooseXVisualInfo();
    if( !visualInfo )
        return false;

    GLXContext context = createGLXContext( visualInfo );
    setGLXContext( context );

    if( !context )

```

7. Advanced Features

```
    return false;

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

if( success && !_xDrawable )
{
    _window->setErrorMessage(
        "configInitGLXDrawable_did_not_set_a_X11_drawable" );
    return false;
}

return success;
}
```

7.4.4. 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 WGLWindow::configInit()
{
    if( !initWGLAffinityDC( ) )
    {
        _window->setErrorMessage( "Can't_create_affinity_dc" );
        return false;
    }

    int pixelFormat = chooseWGLPixelFormat();
    if( pixelFormat == 0 )
    {
        exitWGLAffinityDC();
        return false;
    }

    if( !configInitWGLDrawable( pixelFormat ) )
    {
        exitWGLAffinityDC();
        return false;
    }
}
```

7. Advanced Features

```

if( !_wglIDC )
{
    exitWGLAffinityDC();
    setWGLDC( 0, WGLDC_NONE );
    _window->setErrorMessage(
        " configInitWGLDrawable_did_not_set_a_WGL_drawable" );
    return false;
}

HGLRC context = createWGLContext();
if( !context )
{
    configExit();
    return false;
}

setWGLContext( context );
makeCurrent();
_initGlew();

if( getIAttribute( Window::IATTR_HINT_SWAPSYNC ) != AUTO )
{
    if( WGLEW_EXT_swap_control )
    {
        // set vsync on/off
        const GLint vsync =
            ( getIAttribute( Window::IATTR_HINT_SWAPSYNC )==OFF ) ? 0 : 1;
        wglSwapIntervalEXT( vsync );
    }
    else
        EQWARN << "WGLEW_EXT_swap_control_not_supported, ignoring window"
            << "swapSync_hint" << std::endl;
}

if( !joinNVSwapBarrier( ) )
{
    _window->setErrorMessage( "Joining_NV_swap_group_failed" );
    return false;
}

if( getIAttribute( Window::IATTR_HINT_DRAWABLE ) == FBO )
    configInitFBO();

return true;
}

```

7.5. Head Tracking

The `eqPly` example contains rudimentary support for head tracking to show how head tracking can be integrated with Equalizer. Support for a wide range of tracking devices is not within the scope of Equalizer. Other open source and commercial implementations cover this functionality sufficiently and can easily be integrated with Equalizer.

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

Figure 39(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 39(b).

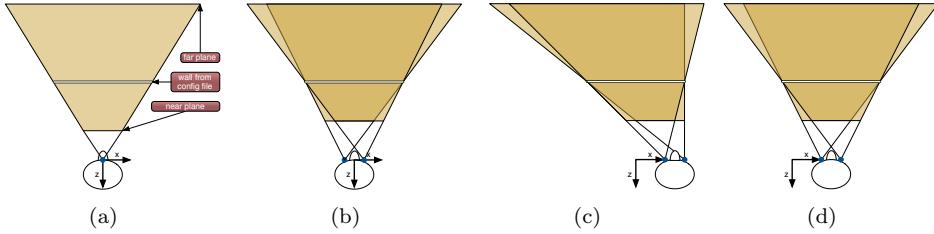


Figure 39: Monoscopic, Stereoscopic, Tracked and HMD frusta

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 39(c). The transformation *origin* → *viewer* is set by the application using `Config::setHeadMatrix`, which is used by the server to compute the frusta. The resulting off-axis 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 39(d). This results in different projections compared to normal tracking with fixed projection screens.

7.6. 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 3.7 and Section 3.8. 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 and the frustum of a view can be changed at runtime.

7.6.1. 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 application node, so that the application can change mutable parameters and distribute its own data to the render clients. Figure 40 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 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 amends it by view-specific data. The view, like all other layout entities, is indirectly sub-classed from `eq::Object` through `eq::Frustum`. The design and usage of `eq::Object` is explained in Section 8.7.1.

In `eqPly`, the application-specific data is the model identifier. A new dirty bit for the model is defined, and the model identifier is serialized when requested:

7. Advanced Features

```
/** The changed parts of the view. */
enum DirtyBits
{
    DIRTY_MODEL      = eq::View::DIRTY_CUSTOM << 0,
};

void View::serialize( eq::net::DataStream& os, const uint64_t dirtyBits )
{
    eq::View::serialize( os, dirtyBits );
    if( dirtyBits & DIRTY_MODEL )
        os << _modelID;
}

void View::deserialize( eq::net::DataStream& is, const uint64_t dirtyBits )
{
    eq::View::deserialize( is, dirtyBits );
    if( dirtyBits & DIRTY_MODEL )
        is >> _modelID;
}

void View::setModelID( const uint32_t id )
{
    _modelID = id;
    setDirty( DIRTY_MODEL );
}
```

7.6.2. Dynamic 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::releaseEntity` 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.

7.6.3. 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 40), 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 3.10.3. 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 pro-

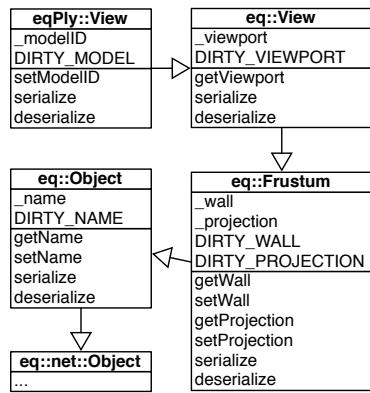


Figure 40: UML hierarchy of `eqPly::View`

7. Advanced Features

jection, 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.

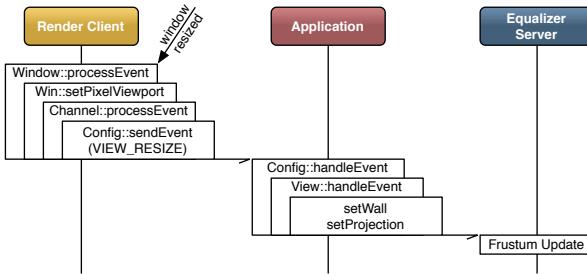


Figure 41: Event Flow during a View Update

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. Based on 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 41 shows a sequence diagram of such a view update.

7.7. 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.

The source 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 framebuffer. 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.

7.7.1. 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, libeqCompressor*.dylib on Mac OS X, libeqCompressor*.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.

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

7. Advanced Features

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.

7.7.2. 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.

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

Figure 42 illustrates this algorithm for three channels. The Equalizer website contains a presentation²⁷ explaining and comparing this algorithm with the binary-swap algorithm.

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

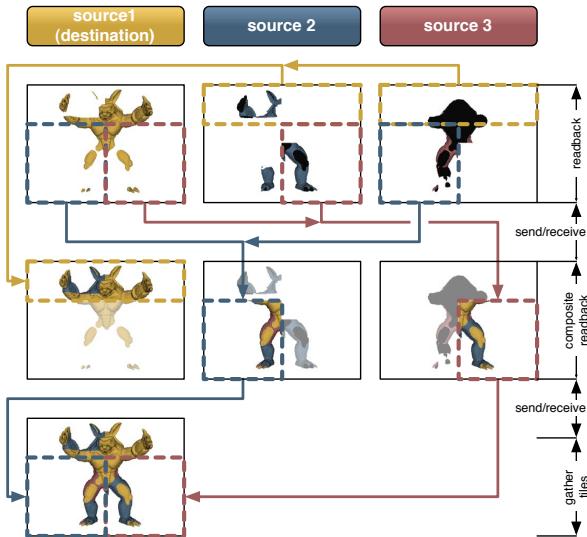


Figure 42: Direct Send Compositing

²⁷<http://www.equalizergraphics.com/documents/EGPGV07.pdf>

7. Advanced Features

- 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 should 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 actually updated during rendering. This optimization will be fully supported by later versions of Equalizer.

7.7.3. 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 read-back.

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.

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 43 illustrates the relationship between frames, frame data and images.

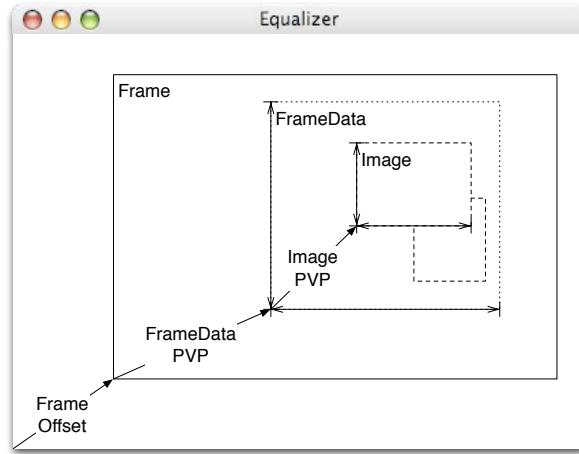


Figure 43: Hierarchy of assembly classes

7. Advanced Features

7.7.4. The Compositor

The Compositor class gathers a set of static functions which implement the various compositing algorithms and low-level optimizations. Figure 44 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 semi-transparent volumetric data. These operations composite either directly all images on the GPU, or use a CPU-based compositor and then transfer the pre-integrated 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.

7.7.5. 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 45²⁸.

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.

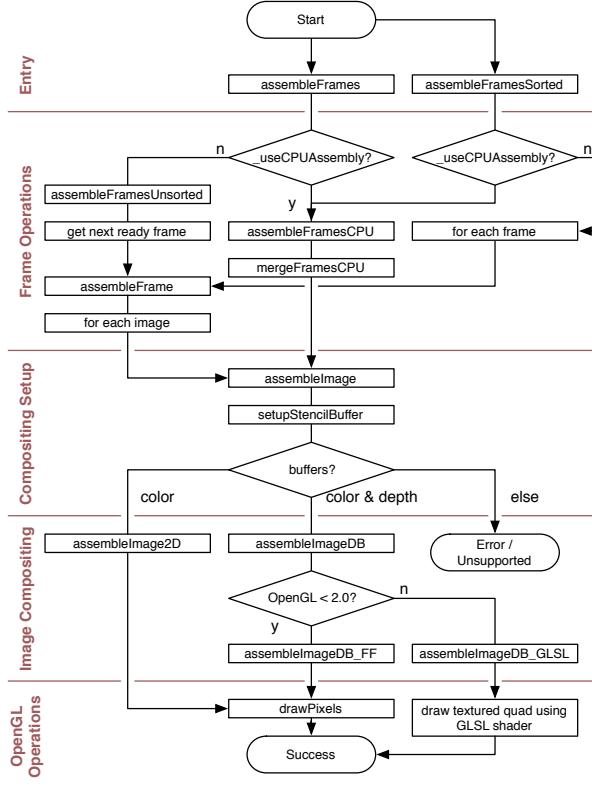


Figure 44: Functional diagram of the compositor

²⁸Volume Data Set courtesy of: SFB-382 of the German Research Council (DFG)

7. Advanced Features

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.

DB compounds have 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. Simplified, 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 47(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 47(b)²⁹ 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 47(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 46 shows the result of the composition from Figure 47(b).

The assembly algorithm described in this section also works with parallel compositing algorithms such as direct-send.

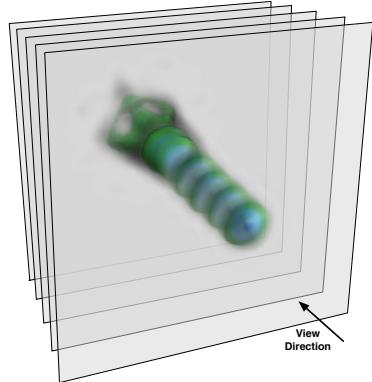


Figure 45: Volume Rendering using 3D Textures



Figure 46: Result of Figure 47(b)

²⁹Volume Data Set courtesy of: AVS, USA

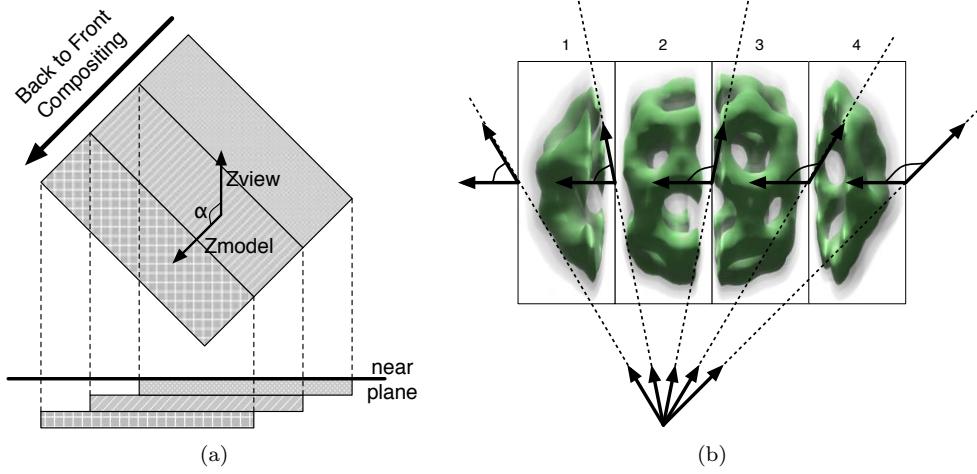


Figure 47: Back-to-Front Compositing for Orthogonal and Perspective Frusta

7.8. Statistics

7.8.1. Statistics Gathering

Statistics are measured in milliseconds since the configuration was initialized. On each node, the global configuration clock is synchronized with the server clock. Each statistic event records the originator's (channel, window, frame 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 incurs a performance penalty, but gives more correct results. The default setting is `fastest` in release builds, and `nicest` in debug builds.

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 7.1. 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.

Figure 48 shows the visualization of statistics events in an overlay³⁰.

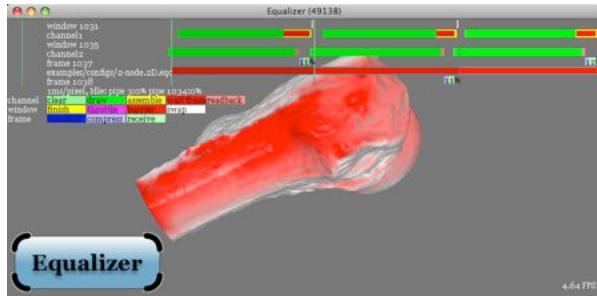


Figure 48: Statistics for a two-node 2D compound

7.8.2. Statistics Overlay

The Equalizer examples render a statistics overlay using the gathered statistics events directly before `Window::swapBuffers`. Statistics are toggled on and off using

³⁰3D model courtesy Cyberware, Inc.

7. Advanced Features

the 's' key. The overlay rendering method, `Channel::drawStatistics`, can be called directly by any application to render the overlay. It can also be overwritten by a custom implementation.

Figure 49 shows a detail view of Figure 48. The statistics shown are for a two-node 2D compound. The destination channel is on the `appNode` and contributes to the rendering.



Figure 49: Detail of the Statistics from Figure 48.

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. 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.

The rightmost pixel is the most current time. On the Y axis are the entities: channels, windows, frames and the config. The upper channel is the local channel since it executes `frameAssemble`, and the lower channel is the remote channel, executing `frameReadback`.

The draw time has been artificially inflated by using immediate mode rendering to provide more expressive statistics.

To facilitate the understanding, older frames are gradually grayed out. The rightmost, current frame is brighter than the frame-before-last.

The configuration used has a 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 assembling the previous frame.

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 to complete.

In the above example, the local channel finishes drawing the frame early, and therefore spends a considerable amount of time waiting for the input frame from the remote channel. These wait events, rendered red, are a sub-event of the yellow `frameAssemble` task.

The frame statistics provide information on the image transmission for scalable rendering. They are grouped per frame, not per node, that is, the transmit and

8. The Network Layer

compress part are executed by one node and the receive by another, even though the statistics are drawn in the same line. The decompression time is part of the receive statistic. The total compression ratio is printed with each compression statistic, in this case the image has been compressed to 11% of its original size.

The white `Window::swapBuffers` task might also take also a long time, since the execution of the swap buffer is locked to the vertical retrace of the display, and enforced by `glFinish` before sampling the end time of the event. In release builds, this does not happen and all OpenGL commands are pipelined with the execution on the CPU.

Note that the remote source window does not execute the `swapBuffers` in this configuration, since it is a single-buffered PBuffer.

8. The Network Layer

The Equalizer networking layer 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 core Equalizer functionality.

The network layer is implemented in the `eq::net` namespace. It provides layered functionality, which provide higher level functionality for the programmer. The main primitives in the network layer are:

Connection A stream-oriented point-to-point communication line. Different implementations of a connection exist.

Node The abstraction of a process in the cluster. Nodes communicate with each other using connections.

Session Provides unique identifiers and object mapping to a set of nodes.

Object Provides object-oriented data distribution of C++ objects between nodes within a session.

8.1. Connections

The `eq::net::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.

An `eq::net::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.

8.2. Packet Handling

Nodes, sessions and objects communicate using packets derived from `eq::net::Packet`. The basic packet dispatch and handling is implemented in the `eq::net::Base` class, from which `eq::net::Node`, `eq::net::Session` and `eq::net::Object` is sub-classed.

The network base class allows the registration of a packets with a dispatch queue and an invocation method. Each packet has a command identifier, which is used to identify the registered queue and method. The method `dispatchCommand` pushes the packet to the registered queue, whereas the method `invokeCommand` calls the registered command function.

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.

8.3. Nodes

The `eq::net::Node` is the abstraction of one process in the cluster. Each node has a unique identifier, based on the UUID standard³¹. 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 `eq::net::NodePacket`.

Nodes, like connections, are either in the closed, listening or connected state. A listening node uses one or more listening connections in conjunction with a connection set to wait for data or connection requests. For each connection request, a connection and node are created to represent this node on the local machine. This node is in the connected state, and its connection is added to the connection set of the listening node.

Figure 50 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 `Base::dispatchCommand`. For session and object packets, the appropriate session is found and `Session::dispatchCommand` will dispatch the packet.

If `dispatchCommand` returns false, the packet will be re-dispatched 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

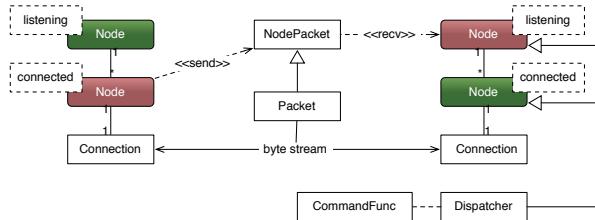


Figure 50: Communication between two nodes

³¹http://en.wikipedia.org/wiki/Universally_Unique_Identifier

8. The Network Layer

a custom datatype are handled by overriding `Node::dispatchCommand`. Custom node packets can use any command greater than `CMD_NODE_CUSTOM` as the packet command. By registering a `eq::net::CommandFunc` for these commands, the `eq::net::Base` dispatch and invoke mechanism can be used automatically.

8.4. Session

The `eq::net::Session` gathers provides higher-level functionality to a set of nodes. The master instance of a session is registered to a node, which makes this node the session server and assigns a node-unique identifier to the session. All other nodes map their instance of the session to their local listening node using the session's identifier. One given node can participate in any number of sessions, both as master and slave. This allows the flexible usage of sessions to group different aspects of an application.

A session provides the facilitate to allocate session-unique identifiers. These identifiers can be used by the application to uniquely address data within the cluster. They are also used by the session to address distributed objects, which are described in the following section.

Sessions instances can communicate to each other directly using an `eq::net::SessionPacket`, which is derived from the `NodePacket` and extends it by the session identifier. The session ID is used by the node to dispatch the packet to the session.

The session, similar to the node, uses the methods `dispatchCommand` and `invokeCommand` to dispatch and execute packets using either the `eq::net::Base` class for session packets or to dispatch and invoke object packets on the appropriate `eq::net::Object`. Application developers can use this mechanism by creating custom session packets with a command greater than `CMD_SESSION_CUSTOM` and registering a command function for these commands.

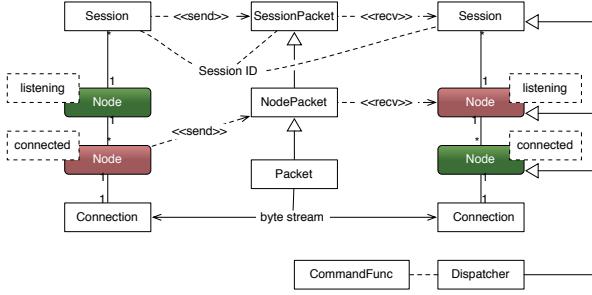


Figure 51: Communication between two Sessions

8.5. Object

Distributed objects provide powerful, object-oriented data distribution for C++ objects. Their functionality and an example use case for parallel rendering has been described in Section 6.3.

The master instance of an `eq::net::Object` is registered with a session, which assigns a session-unique identifier to the object. Using this identifier, other nodes can map their slave instance of the object to their local proxy of the session.

Objects can send command packets to each other by subclassing `eq::net::ObjectPacket`, which is derived from `SessionPacket`. The object packet extends the session 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_ID_ANY` (the default value), a command is dispatch to all object instance 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.

8. The Network Layer

Distributed objects provide data synchronization functionality, as described in Section 6.3. 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 and session communication layers, as shown in Figure 52.

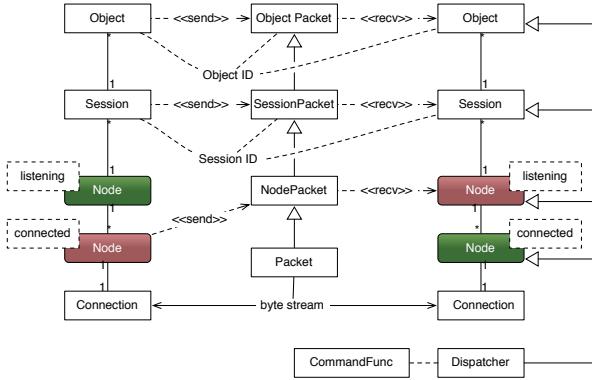


Figure 52: Communication between two objects

8.6. Barrier

The `eq::net::Barrier` provides a networked barrier primitive. It is an `eq::net::Object` used by Equalizer for software swap barrier support, 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 request are automatically grouped together by version.

8.7. 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 `eq::net` namespace.

8.7.1. eq::Object

Equalizer uses the `eq::net::Object` data distribution mechanism for implementing data distribution between the server, application and render clients. To do this, `eq::Object` implements one usage pattern for the `eq::net::Object`, allowing inheritance with data distribution. It is easier to use, but imposes one typical way to implement data distribution.

The `eq::Object` data distribution is based on the concept of dirty bits. Dirty bits form a 64-bit mask which marks which parts of the object have to be distributed during the next commit.

For serialization, the default `eq::net::Object` serialization functions are implemented by `eq::Object`, 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 `eq::Object`, the following steps have to be taken:

Inherit from `eq::Object`: 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 `Object::DIRTY_CUSTOM`, shifting this value consecutively by each new dirty bit.

8. The Network Layer

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 inheritance chain `eq::View` → `eq::Frustum` → `eq::Object` → `eq::net::Object` serves as sample implementation for this process. The `eq::Object` base class provides the facility to set a name on the object, which is automatically distributed.

The registration and mapping of `eq::Objects` is done in the same way as for `eq::net::Objects`, which has been described in Section 8.5. Equalizer objects derived from `eq::Object`, e.g., `eq::View`, are typically registered and mapped by Equalizer.

A. 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`.

A.1. File Format Version

A.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>`

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	<code>EQ_CONNECTION_SATTR_HOSTNAME</code>
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_CONNECTION_SATTR_LAUNCH_COMMAND</code> <code>EQ_CONNECTION_IATTR_TCPIP_PORT</code> <code>EQ_CONNECTION_IATTR_TYPE</code>

A. File Format

Name	EQ_CONNECTION_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_CONNECTION_SATTR_LAUNCH_COMMAND_QUOTE EQ_CONNECTION_IATTR_LAUNCH_TIMEOUT
Name	EQ_CONNECTION_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_CONNECTION_IATTR_TYPE
Value	TCPIP SDP 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). PIPE uses a named pipe to communicate between two processes on the same machine.
Name	EQ_CONNECTION_IATTR_TCPIP_PORT
Value	unsigned
Default	0
Details	The listening port used by the server or node. For nodes, the port can be used to contact pre-started, 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_PIPE_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.
Name	EQ_CONNECTION_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.

A. File Format

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 7.5 for details.
See also	EQ_WINDOW_IATTR_HINT_STEREO EQ_COMPOUND_IATTR_STEREO_MODE
Name	EQ_NODE_IATTR_THREAD_MODEL
Value	ASYNC DRAW_SYNC LOCAL_SYNC
Default	DRAW_SYNC
Details	The threading model for node synchronization. See Section 7.2.2 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_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.

A. File Format

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
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. 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.
Name	EQ_WINDOW_IATTR_HINT_STATISTICS
Value	OFF FASTEST [ON] NICEST
Default	FASTEST [Release Build]
Details	NICEST [Debug Build]
See also	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 Window::finish, which calls by default glFinish, in order to accurately account the rendering operations to the sampled task method. However, calling glFinish 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.
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.

A. File Format

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.
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

A. File Format

Name	EQ_COMPOUND_IATTR_STEREO_MODE
Value	QUAD ANAGLYPH
Default	QUAD
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.
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.

A.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.

A.3.1. Connection Description

A connection description defines the network parameters of an Equalizer process. Currently TCP/IP, SDP 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. PIPE uses a named pipe for fast interprocess communication on Windows.

Note that you can also use the transparent mode provided by most InfiniBand implementations to use SDP with TCP connections.

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 pre-launched 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.

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
{
```

A. File Format

```

type      TCPIP | SDP | PIPE [Win32 only]
TCPIP_port unsigned
PIPE_filename string
hostname   string
}

config  {...} # 1-n times, currently only the first one is used by the server
}

```

A.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 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 A.2.

```

config
{
    latency int      # Nr. of frames nodes may fall behind application, default 1
    attributes
    {
        eye_base float # distance between left and right eye
    }

    appNode {...}  # 0|1 times, launches render threads within app process
    node {...}     # 1-n times, a system in the cluster
    observer {...} # 0-n times, one tracked entity
    layout {...}   # 0-n times, logical views on canvases
    canvas {...}   # 0-n times, physical segment layout of a projection surface
    compound {...} # 0-n times, rendering description
}

```

A.3.3. Node Section

A node represents a machine in the cluster, and is one process. It has a name, 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. For a description of node and connection attributes please refer to Section A.2.

```

node
{
    name      string
    connection # 0-n times, possible connections to this node
    {
        type      TCPIP | SDP | PIPE
        TCPIP_port unsigned
        PIPE_filename string
        hostname   string
        command    string      # render client launch command
        command_quote 'character' # launch command argument quote char
        timeout    unsigned     # timeout in milliseconds for launch
    }
    attributes
}

```

A. File Format

```
{
    thread_model      ASYNC | DRAWSYNC | LOCALSYNC
}
pipe { ... } # 1-n times
}
```

A.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.

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. If the extension is not present, the window is opened on the *n*th monitor, but OpenGL commands are sent to all GPU's.

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 A.2.

```
pipe
{
    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
    }
    window { ... } # 1-n times
}
```

A.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 with respect 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

A. File Format

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 A.2.

```
window
{
    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
        hint_statistics   off | fastest [on] | nicest
        planes_color      unsigned | RGBA16F | RGBA32F
        planes_alpha      unsigned
        planes_depth      unsigned
        planes_stencil    unsigned
        planes_accum      unsigned
        planes_accum_alpha unsigned
        planes_samples    unsigned
    }

    channel { ... } # 1-n times
}
```

A.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.

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 A.2.

```
channel
{
    name      string
    viewport [ viewport ] # wrt window, default full window
    drawable [ FBO.COLOR FBO.DEPTH FBO.STENCIL ]

    attributes
    {
        hint_statistics      OFF | FASTEST [ON] | NICEST
    }
}
```

A.3.7. Observer Section

An observer represents a tracked entity, i.e, one user. It has a name and an eye separation. 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,

A. File Format

in which case the name should be unique. Not all views have to be tracked by an observer.

```
observer # 0...n times
{
    name      string
    eyeBase   float
}
```

A.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.

```
layout # 0...n times
{
    name string
    view {...} # 1...n times
}
```

A.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.

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 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 ]

    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 ]
    }
}
```

A. File Format

```

        distance      float
        fov           [ float float ]
        hpr           [ float float float ]
    }
}

```

A.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 is made 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 the layout active by default.

```

canvas # 0...n times
{
    name   string
    layout layout-ref # 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 ]
    }
    segment { ... } # 1...n times
}

```

A.3.11. Segment Section

A segment represents a single display, i.e., a projector or monitor. It references a channel, has a name, viewport and frustum. 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 section 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.

A segment can define a frustum, in which case it overrides the default frustum calculated from the canvas frustum and segment viewport.

```

segment # 1...n times
{
    channel string
    name   string
    viewport [ viewport ]

    wall      // frustum description
}

```

A. File Format

```
{
    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 ]
}
}
```

A.3.12. Compound Section

Compounds are the basic data structure describing the rendering setup. They use channels for rendering. Please refer to Section 3.10 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 READBACK ASSEMBLE for all others. 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 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 eye pass select the mono, left or right eye passes rendered by the compound. 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. 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 A.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 11 illustrates the frustum parameters.

A swap barrier is used to synchronize the output of multiple compounds. All swap barriers of the same name are synchronized. The default name is barrier.root-

A. File Format

`CompoundName]`. 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.

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
{
    name      string
    channel   channel-ref # where the compound tasks are executed

    task      [ CLEAR DRAW READBACK ASSEMBLE ]
    buffer   [ COLOR DEPTH ]           # default COLOR

    viewport [ viewport ]            # wrt parent compound, sort-first
    range    [ float float ]        # DB-range for sort-last
    pixel    [ int int ]           # pixel decomposition (step size)
    eye      [ CYCLOP LEFT RIGHT ] # monoscopic or stereo view
    zoom     [ float float ]        # up/downscale of parent pvp
    period   int                   # DPlex period, every nth frame rendered
    phase    int                   # DPlex phase, when to start rendering

    load_equalizer # adjust 2D tiling or DB range of children
    {
        mode 2D | DB | VERTICAL | HORIZONTAL
        damping float # 0: no damping, 1: no changes
    }
    DFR_equalizer # adjust 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          QUAD | ANAGLYPH # default QUAD
        stereo_anaglyph_left_mask [ RED GREEN BLUE ] # default red
        stereo_anaglyph_right_mask [ RED GREEN BLUE ] # df green blue
    }

    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 ]
    }

    child-compounds
}

```

A. File Format

```
swapbarrier # compounds with the same barriername sync swap
{
    name string
    NV_group OFF | ON | unsigned
    NV_barrier OFF | ON | unsigned
}
outputframe
{
    name string
    buffer [ COLOR DEPTH ]
    type texture | memory
}
inputframe
{
    name string # corresponding output frame
}
}

channel-ref: 'string' | '(' channel-segment-ref ')'
channel-segment-ref: ( canvas-ref ) segment-ref ( layout-ref ) view-ref
canvas-ref: 'string' | 'index'
segment-ref: 'string' | 'index'
layout-ref: 'string' | 'index'
view-ref: 'string' | 'index'
observer-ref: 'string' | 'index'
```

