Bern University of Applied Sciences
Engineering and Information Technology

CAVE Rendering Framework

Technical Report

Division	CPVR
Authors	Brigitte Hulliger
	Jonas Walti
	Stefan Broder
Supervisors	Prof. U. Künzler
	Michael Luggen

Version	1.0

Date	Version	Name	Comment
05.06.2009	0.8	all	First draft for expert H. van der Kleij
09.06.2009	0.9	all	Draft for Supervisors
12.06.2009	1.0	all	Last corrections for delivery

Table 0.1: History

Contents

List of Figures

List of Tables

Listings

Abstract

There are numerous parallel rendering frameworks and libraries applicable in visualisation systems. A relatively new approach is provided by the Equalizer API which, compared to Chromium for example, runs an application itself in parallel instead of only the generated OpenGL stream. Unfortunately, Equalizer requires knowledge about parallel rendering and synchronisation in order to get a running application.

The goal of this project was to find a solution that allows both the flexible parallel rendering of an application with Equalizer and the unlimited development of OpenGL applications without any knowledge of the Equalizer API. OpenSceneGraph (OSG) is a high performance 3D graphics toolkit that allows the development of such feature-rich OpenGL applications and therefore provides a solution for the second requirement.

The CAVE Rendering Framework (CRF) provides all functionality needed to render OSG applications with Equalizer without further knowledge about parallel rendering or Equalizer. It is compatible with most Equalizer configurations and supports most of the OSG features. The purpose of the CRF is a running setup for a four sided Cave Automatic Virtual Environment (CAVE) used at the Berner Fachhochschule - Technik und Informatik (BFH-TI), but it is possible to use it in any visualisation system configurable with Equalizer.

1 Introduction

This document describes the work of integrating an existing scene graph technology into Equalizer; Starting with the preceding evaluation of eligible scene graph technologies followed by Equalizer specific configurations and ends with a detailed description of the implementation of the developed rendering framework. This document shall advise programmers how to make use of the CRF.

The use of Equalizer as a parallel rendering framework was a requirement. Advantages as well as disadvantages are discussed later in this document. The group of available scene graph technologies was reduced to two in a former evaluation project. These two were OSG and Object-Oriented Graphics Rendering Engine (OGRE). Both of them are discussed in the section **??**.

The purpose of the integration of OSG into Equalizer is to run any scene graph application in a Virtual Reality (VR) installation without any (bigger) changes of the application code.

All examples and configurations described in this document refer to and were tested in a real virtual reality environment.

2 Evaluation

Due to the fact that the use of Equalizer was required for the project, the only technologies that needed to be evaluated were the two scene graph frameworks OSG and OGRE.

A scene graph is a collection of nodes in a tree structure where a node may have many children. It arranges the logical and spatial representation of a graphical scene. The advantage of such a representation lies in the fact that an operation applied to a node in the tree automatically propagates its effect to all its children. In a graphical context this is very useful, for example, for transformations. Scene graph technologies that were considered to be useful for this project are OSG and OGRE. Both are described below with the focus on differences rather than similarities.

2.1 OSG

OSG **?** is a cross-platform, open source and legacy-free scene graph Application Programming Interface (API). It is based upon the concept of a scene graph providing an object-oriented framework completely written in Standard C++ on top of Open Graphics Library (OpenGL). Therefore, a developer does not have to be concerned about optimising low-level graphics calls.

2.1.1 Features

The goal of OSG is to make the benefits of a scene graph freely available to commercial as well as non-commercial users. The key strengths of OSG are performance, portability and productivity.

Performance
: In the core scene graph, OSG supports view-frustum culling, occlusion culling, small feature culling, Level Of Detail (LOD) nodes, OpenGL state sorting, vertex arrays, vertex buffer objects, OpenGL Shader Language and display lists. More features can be added by installing different plugins.

Productivity
: OSG makes it possible to rapidly develop high-performance graphics applications by encapsulating the majority of OpenGL functionality. The developer can concentrate on content and how that content is controlled rather than low-level coding.

Portability
: An application developed on top of OSG can easily be ported to different platforms since the core scene graph is based on Standard C++ and OpenGL. Therefore, it has minimal dependency on any specific platform. Furthermore, OSG is completely independent of windowing systems.

2.1.2 Community

OSG has a large community including users as well as developers. The operational area of OSG is very wide including many scientific areas. The community mostly communicates over mailing lists. There are different mailing lists for developers and users.

2.1.3 Documentation

It is recommended to start with the book *OpenSceneGraph Quick Start Guide***?** written by Paul Martz. Other than that OSG depends on the community for documentation. The official website

of OSG is a wiki. The reason for that is that OSG wants its users and developers to be able to document their features themselves. On the one hand this is very useful since all documentation can be found on the same website. On the other hand this approach leads to a slightly unorganised website.

2.2 OGRE - Object-Oriented Graphics Rendering Engine

OGRE **?** is a 3D engine based on the scene graph concept. As OSG, it is completely written in Standard C++. The underlying libraries are Direct3D or OpenGL. OGRE is available under the LGPL (GNU Lesser General Public License[1]).

2.2.1 Features

OGRE puts emphasis on the design of the engine. The developers of OGRE pursue the strategy of well designed and well documented features instead of a high number of features, or in other words: quality over quantity.

2.2.2 Community

The community of OGRE is comparable with the one of OSG. Besides mailing lists, OGRE comes with forums and an IRC channel for communication.

2.2.3 Documentation

The approach of design-led rather than feature-led is reflected in the documentation of OGRE. Together with multiple books OGRE comes with an API reference, a manual and a wiki.

2.2.4 OSG vs. OGRE

Both of the discussed scene graph APIs were considered for integration with Equalizer. For comparison of the two technologies, it makes sense to use a decision matrix. The scale reaches from 1 to 10 where 10 is the highest value. Each point in the decision matrix can be weighted differently depending on the importance in the current project.

		OSG			**OGRE**		
Criteria	Weight	Value	Points	%Points	Value	Points	%Points
Performance I	25%	160fps	3	75	220fps	7	175
Performance II	25%	260fps	7	175	220fps	3	75
Community	10%	++	4	40	+++	6	60
Similar Projects	20%	eqOSG	7	140	eqOGRE	3	60
Documentation	10%	++	4	40	+++	6	60
Extensions	10%	++	4	40	+++	6	60
Total	**100%**		**510**			**490**	

Table 2.1: Decision Matrix OSG - OGRE

A stress test application was used to test the performance. This application consists of a three dimensional model consisting of about one billion triangles in an empty universe. The workstation

[1] http://www.gnu.org/copyleft/lesser.html

used for the tests was an Intel Pentium Quadcore with an NVIDIA graphics chip. Two different measures were made based on this setup:

Performance I The framerate (frames per second) when the whole model is visible.

Performance II The framerate when only a part of the model on the screen (by zooming into the scene until some parts of the scene are outside of the frustum).

The result of this test setup was that OGRE does not show any differences between the setup with the whole model on the screen and the one where only part of the model is visible. In OSG on the other hand, the framerate increases if part of the model is outside of the frustum.

OGRE wins the competition in the criteria *Community* because OGRE provides more communication channels (forum, mailing list, IRC channel) than OSG and the communication seems to be better organised.

Looking for extensions both of the participants are up-to-date. There are miscellaneous 3rd party extensions that can be installed. Most of them are freely available. The most interesting extensions for this evaluation were physics and haptics plugins. Both are available for OSG and OGRE.

Considering all these criteria, OSG wins the competition as one can see from the result of the decision matrix above. Although, the decision is quite marginal and is mostly based on the fact that there is a project called *eqOSG*, which is currently under development at the University of Siegen (Germany) and that OSG has a large community in science.

3 Equalizer

3.1 Introduction

Equalizer is an API for creating parallel applications based on OpenGL. It allows the creation of scalable graphics applications. Furthermore, these applications can be run unmodified on any multipipe visualisation system such as laptops, workstations, multipipe shared memory system or even graphic clusters.

The framework Equalizer is available as open source under the GNU Lesser General Public License (LGPL) license. Furthermore, it runs on Linux, Windows XP and Mac OS X systems. It supports both 32-bit and 64-bit architecture.

This chapter gives an introduction to Equalizer and how it was used in this project. Used Equalizer graphics are by courtesy of Stefan Eilemann, Equalizer developer.

3.2 Purpose of Equalizer

There are two main motivations for parallelising the rendering.

Multi-View Rendering
: To run an application on systems with multiple displays - so called multipipe display systems. This includes workstations with multiple monitors or projectors as well as immersive environments such as a CAVE or head mounted displays. Immersive environments additionally require active or passive stereo rendering.
Multi-View rendering requires the synchronisation of the output of the different displays in order to provide one coherent image.

Scalable Rendering
: To run an application that hits the limitations of a single graphics system due to the hardware. The rendering can be accelerated by aggregating multiple resources.

Equalizer provides methods to satisfy both of these motivations and they are not even mutually exclusive. This makes Equalizer a very powerful and flexible framework.

There are other existing generic approaches for parallelising rendering. A well-known solutions is Chromium, which was originally developed at the Stanford University. Like Equalizer, it enables applications to use multiple graphics cards for rendering. But other than Equalizer, Chromium is focused on streaming the OpenGL commands through a network of nodes, often initiated from a single server. These OpenGL streams can be very large if they contain texture images in addition to the OpenGL commands. This is a major limitation and often leads to the fact that the network is the performance bottleneck.

To get rid of this problem, Equalizer follows a different approach for rendering. It runs the application, of which each client in the network has a copy, in parallel and uses the network only for synchronisation.

The main features, that characterise Equalizer are:

- Runtime Configurability
- Runtime Scalability
- Distributed Execution
- Support for Stereo and Immersive Environments

The following table shows the main differences between Equalizer and Chromium:

Transparent Layer: Chromium	**Parallel API: Equalizer**
runs unmodified applications	minimally invasive API: application has to be adapted
focus on transparency for applications	focus on parallel rendering performance and scalability
operates on OpenGL command stream	parallelises application
single application rendering thread	multithreaded and potentially distributed application
mostly used for rendering to multiple, planar displays, immersive installations may require application changes	multi-display and immersive installations, many-to-one scalable rendering
limited OpenGL extension support and compatibility	full OpenGL compatibility: does not interfere with OpenGL rendering code

Table 3.1: Chromium vs. Equalizer**?**

Figure **??** visualises the different approaches of Chromium and Equalizer. Chromium streams the OpenGL calls to the clients as shown in figure **??** (c) whereas Equalizer runs parts of the application in parallel on multiple rendering clients **??** (a).

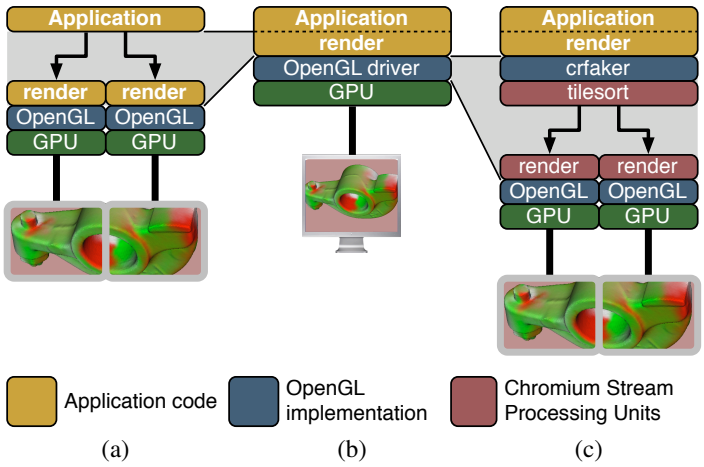

Figure 3.1: A traditional OpenGL application (b) and its equivalents when using Equalizer (a) or Chromium(c)**?**

3.3 Basic Concepts of Equalizer

Equalizer runs parts of the application in parallel on multiple rendering clients. A typical OpenGL application has an event loop which is responsible for performing updates and redrawing the scene. To parallelise the event loop, one has to separate the rendering code from the main event loop. After that, the rendering code can be executed in parallel on different resources. This is the technique applied in Equalizer. Figure **??** visualises the concept.

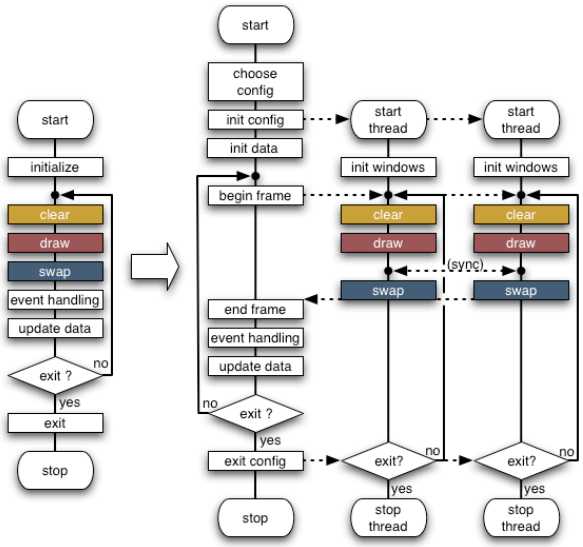

Figure 3.2: Parallelised rendering pipeline as used in Equalizer**?**

Equalizer is based on a client-server model as shown in figure **??**. The different parts and their responsibilities are discussed in detail in the following section:

Server

The Equalizer server is responsible for one single visualisation system. It launches and controls the rendering clients of an application. It is only possible to run one application on a server at a time[1].

Application

The application connects itself to an Equalizer server to receive a configuration. Furthermore, it provides render clients to the Equalizer server, which is responsible to control them. The application itself handles events (and updates data changes caused by events) and controls the rendering.

Render Clients

A render client implements the rendering part of the application. A render client has no main loop and is therefore completely driven by the Equalizer server.

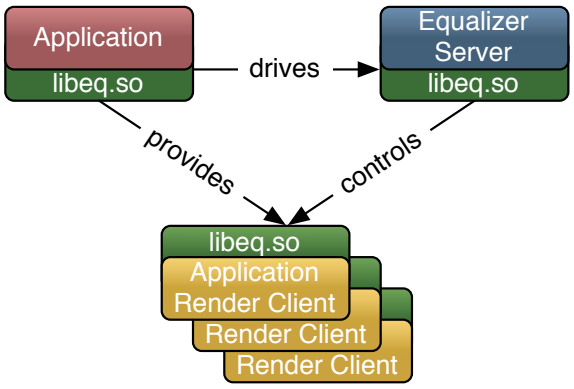

Figure 3.3: Client Server model of Equalizer[**?**, p. 2]

[1] In future releases of Equalizer, it should be possible to run multiple applications on one server.

Figure **???** shows a simplified execution model and visualises how the described components of the Equalizer architecture work together.

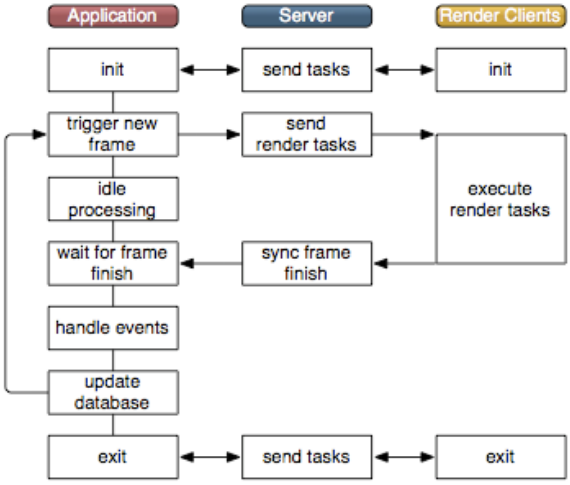

Figure 3.4: Simplified execution model of Equalizer[**?**, p. 9]

3.3.1 Configurability

One of the major benefits of Equalizer is the flexible and scalable configuration of the parallel rendering tasks. An Equalizer configuration always uses a configuration server that allocates and balances the available resources. The server is configured using a hierarchically structured configuration file to describe the available resources and the combination of them for rendering. The structure in the configuration file has to correspond to the physical and logical topology. Equalizer introduces abstract components to describe the rendering resources and their relationship. The following list shows a collection of relevant components for this project. A complete list can be found in the Equalizer Programming Guide [**?**, p. 55ff].

node
: A node represents a single computer in a cluster. Equalizer handles each node as one operating system process of the rendering client. Nodes communicate with each other using *connections*.

pipe
: A *pipe* is an abstraction of a graphics card (GPU). There may be multiple pipes per node.

window
: A *window* represents an OpenGL drawable.

channel
: A *channel* stands for a viewport within a *window*.

connection
: A *connection* is a stream-oriented point-to-point connection between nodes. Nodes either communicate other TCP/IP or SDP.

The configuration for the multiview rendering resources is defined through a compound tree with the following possible components:

channel
Each compound has one channel which is used by the compound to execute the rendering task. A channel may be used by more than one compound.

frustum
Compounds have a frustum description to define the physical layout of the display environment. A frustum can be either a *wall*[2] or a *projection*[3].

attributes
Compounds have attributes to configure the decomposition of the destination channel's rendering.

Consult the Equalizer Programming Guide**?** for a detailed description of all configuration options.

Figure **??** shows an example configuration. The left side of the tree defines the rendering resources and the right side defines the visualisation system which is a four sided CAVE in this case. As one can see, it is possible to combine almost any resources. The matching between the rendering resources and the visualisation output is done via the *channel* identifier. It is necessary to use unique descriptions for the *name* of each channel.

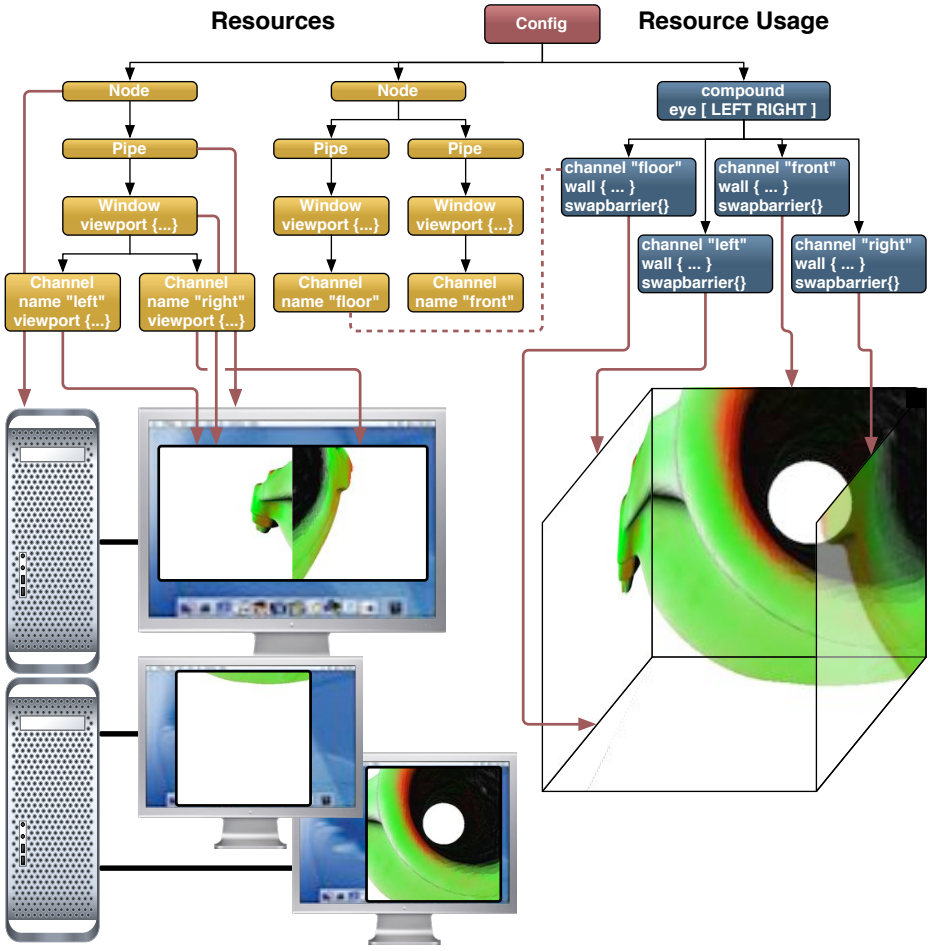

Figure 3.5: Example Configuration for a 4-sided CAVE[**?**, p. 10]

[2]defined by the bottom-left, bottom-right and top-left edge coordinates
[3]defined by the position and the head-pitch-roll orientation

3.3.2 Scalability

Equalizer provides numerous scalability algorithms. Therefore, an application built with Equalizer is inherently scalable. The used algorithm for decomposition and composition can be configured through the Equalizer configuration file. The only thing the programmer has to ensure is that the critical parts of the application (the rendering code) are encapsulated. The rendering code itself can basically be retained. The only thing that has to be changed in the application code is that the application rendering code uses the frustum parameters, viewport and stereo buffer provided by Equalizer. After that, Equalizer handles the distributed execution, synchronisation and final image composition. In other words, the application's rendering code is parallelised, in contrast to other solutions (as Chromium) which operate on the OpenGL command stream produced by a single application thread.

3.3.3 Stereo Compounds

Stereo viewing is realised by rendering two different views, one for each eye. The display setup ensures that each eye sees the correct view, either by time multiplexing the left/right images (active stereo) or by using polarisation filters (passive stereo). Equalizer allows stereo viewing by configuring so called stereo compounds. It uses the notion of an eye pass in the compound specification to assign each eye pass to an individual rendering unit. After the rendering, the resulting images are copied to the appropriate stereo buffer. For passive stereo as used in this project, two output channels are used and projected onto the same surface. The two projected images are polarised in different ways; For example, a left circular polarisation and a right circular polarisation. A viewer wearing polarisation glasses only sees one image per eye which leads to a stereoscopic effect.

3.4 Configuration

As described above, a configuration file is a one-to-one representation of the hardware setup used for rendering. The file format used to describe the setup is an American Standard Code for Information Interchange (ASCII) deserialisation of the server. It uses the same syntactical structure as Virtual Reality Modeling Language (VRML). The basic structure of an Equalizer configuration file can be found in Listing **??**.

```
global { # 0-1 times
  [...]
}
server {
  connection {
    [...]
  }
  config { # 1-n times
    [...]
    node { # 1..n nodes for rendering
      [...]
    }
    compound { # 1..n compounds for rendering usage
      [...]
    }
  }
}
```

Listing 3.1: Basic structure of Equalizer configuration files

`global` The `global` section in a configuration file is optional. It contains default values for various attributes. The notation of these attributes is described in Appendix **??**.

`server` The `server` section in the configuration file is mandatory. It describes the server used in the configuration. Additionally, it may contain connection descriptions for the server listening sockets. A server must at least contain one `config` subsection. Further information is provided in Appendix **??**.

`connection` Equalizer supports two connection types: TCP/IP and SDP. Information about the syntactical description of a connection can be found in Appendix **??**.

`config` The `config` section contains the rendering resources like `node`s and their usage described in `compound`s. A detailed description of resources and their notation in the configuration file can be found in Appendix **??**.

Consult the Equalizer Programming Guide**?** for a detailed description of configuration files.

3.5 Equalizer configurations for different topologies

There are two different topologies described in the following section. It illustrates the Equalizer configuration by example. The output system is for both systems a 4-sided respectively a 3-sided CAVE, the resources to render the output are different.

3.5.1 one-node setup

The first setup covers a single workstation. It contains four graphics cards, each connected with two projectors. The setup is illustrated in figure **??**.

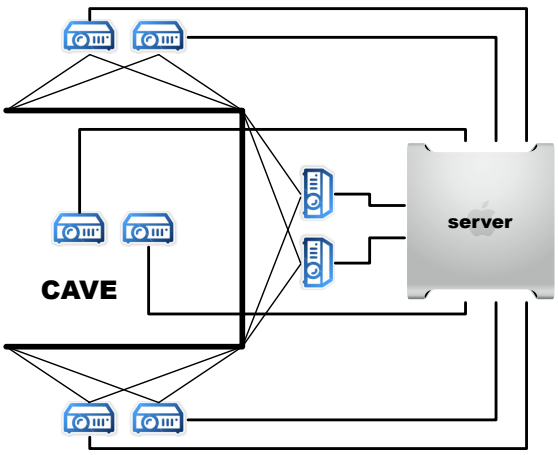

Figure 3.6: Resources for Setup with a single workstation with four graphics cards (2 pipes per card)

Figure **??** shows a graphical representation of the configuration.

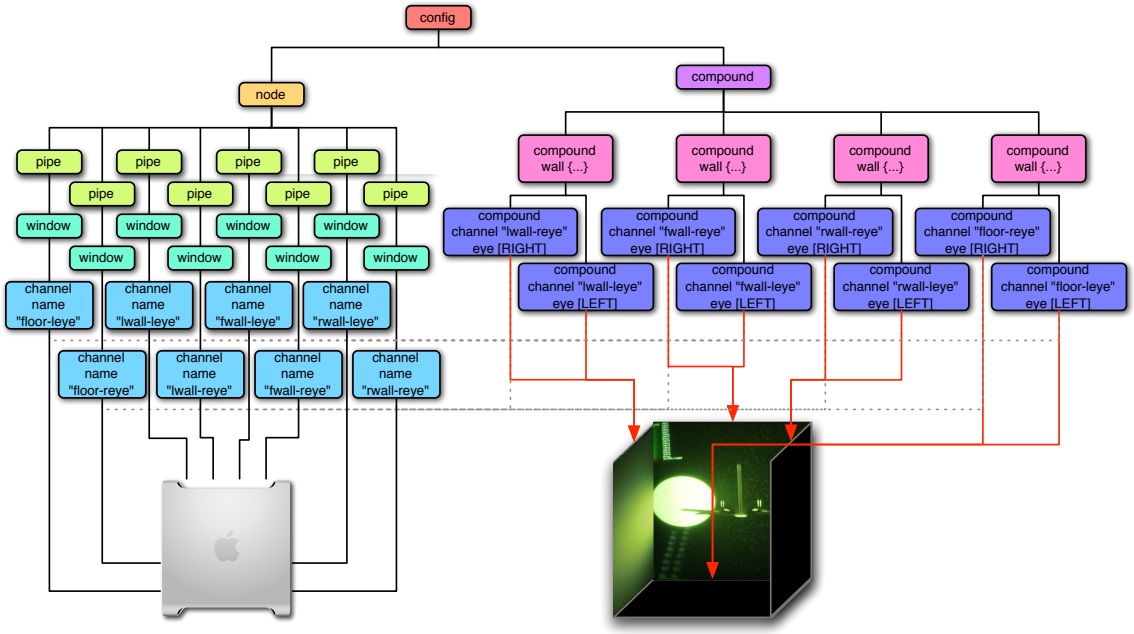

Figure 3.7: Graphical representation of a single node config with a 4-sided CAVE as visualisation system

As shown in figure **??**, there are two pipes per graphics card. This is due to the fact that each graphics card has two outputs, each connected to a projector. Although, Equalizer allows the usage of multiple channels within a single window. This should not be used in combination with the CAVE Rendering Framework.

An example configuration file for this setup can be found in Appendix **??**.

3.5.2 six-node setup

The second setup consists of six separate nodes, each connected to one projector. The setup is illustrated in Figure **??**.

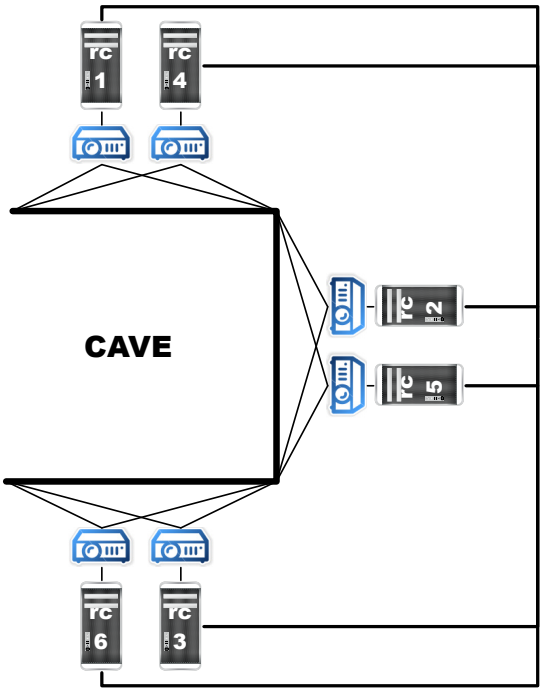

Figure 3.8: Resources for Setup with 6 connected workstations

Figure **??** shows a graphical representation of the configuration.

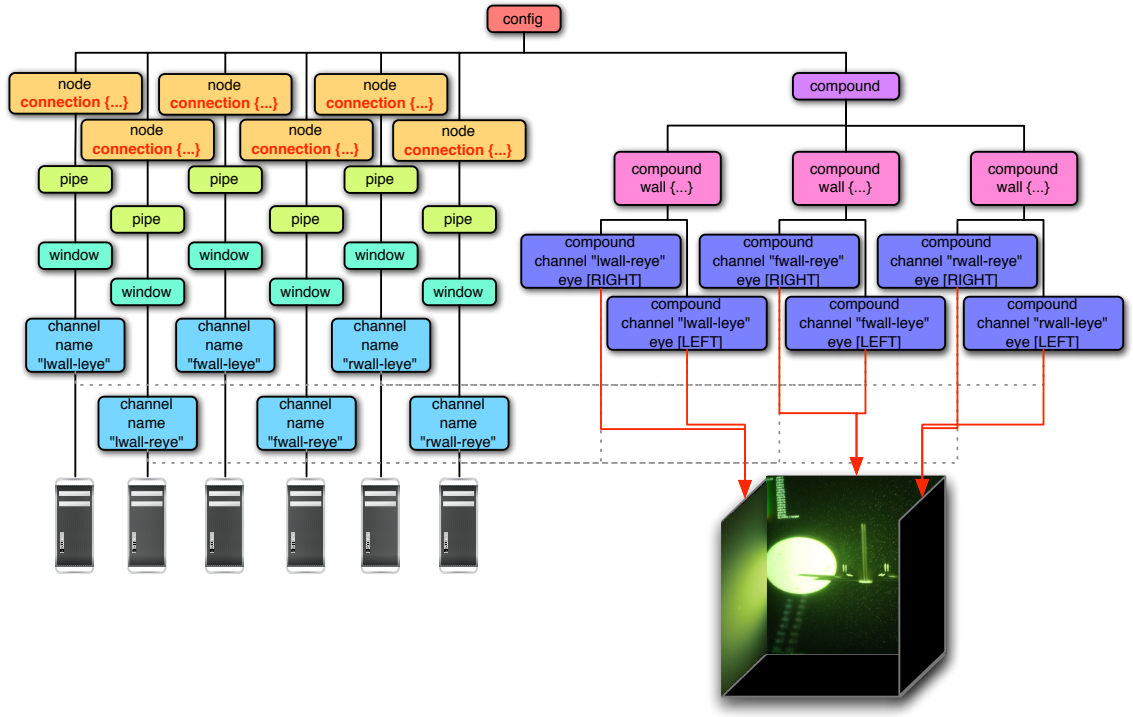

Figure 3.9: Graphical representation of a 6-node config with a 3-sided CAVE as visualisation system

As shown in the figure **??**, each node has its own pipe (Graphics Processing Unit (GPU) representation). To connect the nodes, each node has to contain a `connection` section with at least its hostname or IP address. Furthermore, one of the hosts acts as the server, which hast to be accessible by all other nodes via Secure Shell (SSH); Preferably without password verification. Further information concerning the authentication between hosts can be found in the *User Manual*.

An example configuration file for this setup can be found in Appendix **??**.

4 CAVE Rendering Framework

4.1 Idea

The main goal of this framework is to provide a simple tool library to render OSG applications with Equalizer. OSG developers do not need a lot of knowledge of Equalizer. The CRF offers simple methods to pass the OSG scene graph to Equalizer for basic OSG applications with static scene graphs. Furthermore, it provides additional tools to handle more complex OSG applications with user input and highly dynamic scenes.

Nevertheless, the final application is compatible with most Equalizer configurations and supports most OSG features. For further information about this, have a look at the section **??** or **??**.

4.2 Design

4.2.1 Design Techniques

The CRF abstracts the more complex Equalizer framework. Similar, to many common frameworks, the *Hollywood Principle*[1] plays a major role in the Equalizer framework and therefore also in our CRF. To provide an easy interface for simple OSG applications, we implemented the `crfStarter` class in a facade pattern[2] manor. This class provides easily understandable functions, which pass the OSG scene graph to the framework and run it with Equalizer.

4.3 Namespaces

namespace `eq`
: The low-level standard Equalizer namespace. When using the basic CRF features, one should not have to care about this namespace.

namespace `osg, osgViewer, osgGA, ...`
: Naturally, the CRF uses some of the OSG classes.

namespace `eqOsg`
: The basic implementation for rendering OSG scene graphs with Equalizer. This namespace contains the implementations of all the must-have Equalizer classes to build a complete Equalizer application with a scene graph. Moreover, some basic features, like a camera control and the specialised `EqViewer`, which inherits from the standard `osgViewer`, are implemented here.

namespace `crf`
: This namespace provides classes to abstract the `eqOSG` Equalizer application. Classes like `crfPipe`, `crfChannel` and the facade class `crfStarter` provide easily usable functions to run an OSG scene graph with Equalizer.

[1] *The Hollywood Principle* is a well known paradigm in software engineering. *Don't call us, we call you*, meaning the implemented functions are called by the framework than the developer.

[2] The facade pattern is a software engineering design pattern commonly used in Object Oriented (OO) programming. The name is in analogy to an architectural facade. A facade is an object that provides a simplified interface to a larger body of code, such as a class library.

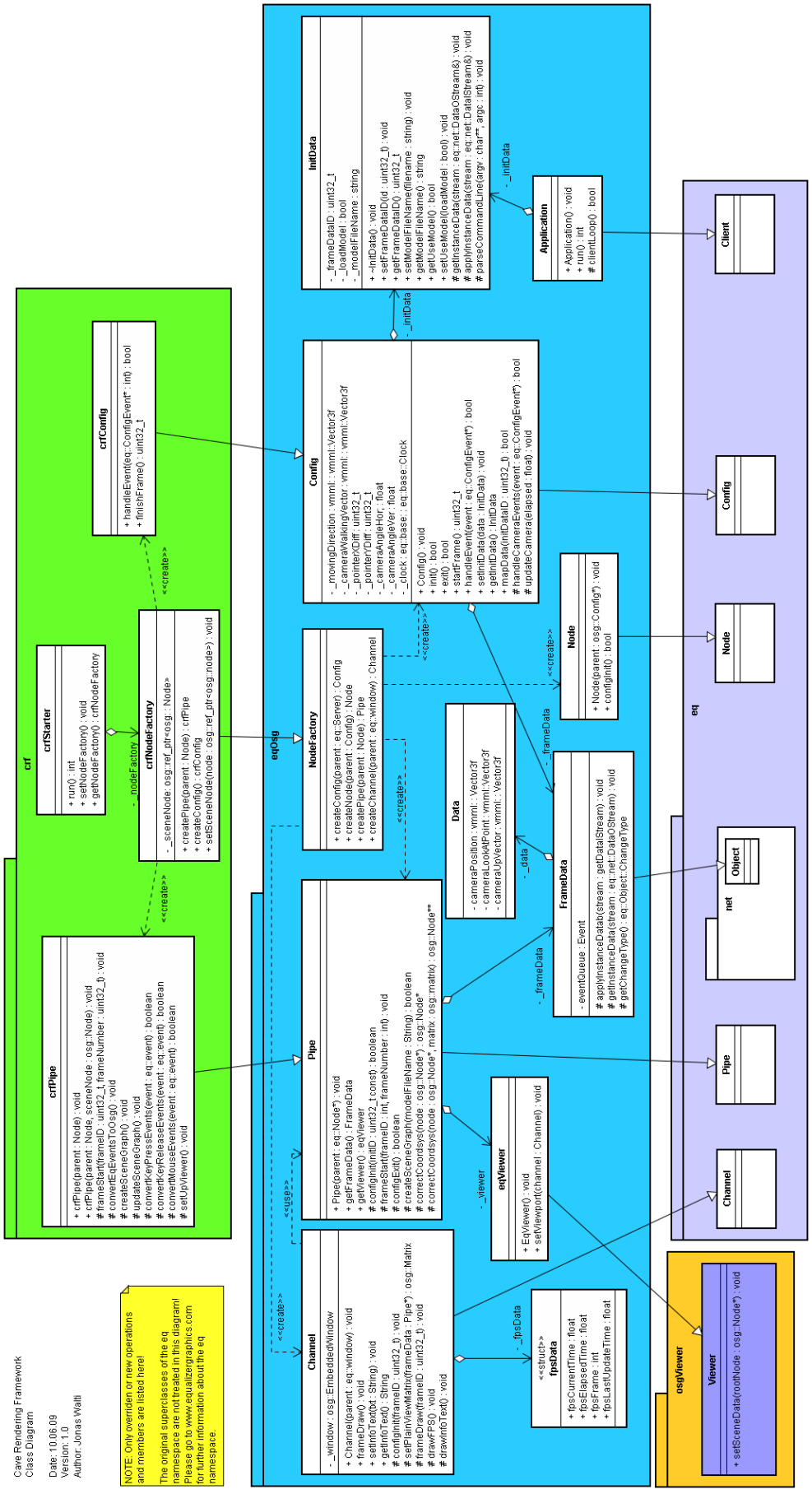

Figure 4.1: The complete class diagram with all three namespaces

4.4 Implementation

4.4.1 About this section

The information provided in this section cares about technical implementation facts. This may help a framework user to fix errors.

IMPORTANT: This part of the documentation requires knowledge of Equalizer. Please consult the previous Equalizer section or the Equalizer Programming Guide **?** for further information.

4.4.2 Overview

Equalizer applications in general are realised by subclassing existing classes of the eq namespace. A good example is the `eqPly` application, shipped with Equalizer. This example makes use of much Equalizer functionality. However, it has nothing to do with OSG.

In the CRF implementation, every pipe holds a complete `eqOsg::EqViewer`, which inherits from `osgViewer::Viewer`. Hence, every pipe holds a complete and fully independent scene graph. This is very important because this can force synchronisation problems and therefore errors in multipipe and/or multinode Equalizer configurations.

The only communication between this different scene graphs is realised with the `FrameData` class. `FrameData` inherits from the `eq::net::Object` class. This object has to be converted to a stream in order to pass it over the network in multinode configurations. In the basic `eqOsg::FrameData` class, several members are present to describe the cameras position, up-vector, viewing direction and current state.

4.4.3 `eqOsg` namepsace classes

Classes in the `eqOSG` namespace provide a basic Equalizer application with an OSG viewer. All needed functionality to render an OSG scene graph with Equalizer are implemented here.

class `EqViewer`

The `EqViewer` class inherits from the original `osgViewer::viewer` and provides a viewer, which is used to render an OSG scene with Equalizer, because the default `osgViewer` does not work. Listing **??** shows the important line of code in the constructor.

```
EqViewer::EqViewer()
{
  //makes the viewer single threaded
  setThreadingModel(osgViewer::Viewer::SingleThreaded);

}
```

Listing 4.1: Constructor of the `EqViewer`

Because a default OSG viewer does not support the multithreaded mode if configured as embedded window (what is needed here, because Equalizer creates the windows), the threading model is set to single. The newly added function `setViewport(eq::channel channel)` sets the frustum and the viewport of the camera, depending on the Equalizer configuration and the passed Equalizer channel.

class `Config`

As in a common Equalizer applications, `osgEq::Config` represents the current configuration of the application. It also updates the frame data class when necessary. Additionally,

the `eqOsg::Config` class calculates the position and viewing direction of the camera. This happens by receiving mouse and keyboard events in `Config::handleEvents()`. Afterwards, calculating the new cameras properties in `Config::updateCamera()`. Finally, the new camera properties are updated in the frame data.
The rest is similar to the eqPly example.

class `FrameData`
The `FrameData` class is a container for values which can be sent to every node and pipe. This is the only way to pass (updated) data to the different render clients and pipes at runtime. In the `eqOsg::FrameData` class, all information concerning the camera is stored in this object.

class `Application`
The `eqOsg::Application` class is similar to the eqPly example. This class starts and stops the application, runs the main loop by calling the `startFrame()`, and `finishFrame()` functions of the `Config` object.

class `NodeFactory`
The `eqOsg::NodeFactory` is needed to create the desired objects during the initialisation of Equalizer.

class `Node`
As in a standard Equalizer application, the node class represents a render client. This class does not contain important changes, because the CRF does not need node-specific customisations.

class `Pipe`
The class `Pipe` abstracts a physical GPU. The derived class `eqOsg::Pipe` holds the two additional members; The `osgEq::EqViewer _viewer` and the `eqOsg::FrameData _frameData`. The viewer holds the complete scene graph and frame data, which is a copy of the distributed data. `eqOsg::Pipe` is a basic and lightweight implementation with an empty scene graph. The `crf::crfPipe` inherits from this class and provides additional functions to create and update the scene graph. More information about this can be found in the following sections.

class `InitData`
The `eqOsg::InitData` class is needed during the initialisation of the application. It also provides a function to parse the command line and set this initial data. Furthermore, it holds the frame data ID, so all clients can synchronise on the distributed frame data object.

class `Channel`
In Equalizer a `Channel` represents an OpenGL viewport. Hence, `eqOsg::Channel` renders the scene graph. This is done with the `frameDraw()` function. Consult **??** for more information. Moreover, when the `configInit` function of the channel is called, the `EqViewer` will be configured as an embedded window. This ensures that a virtual OSG window is defined, which is widely used in given OSG classes (for example `osgViewer::StatsHandler` or `osgGA::FlightManipulator`).

4.4.4 `crf` namepsace classes

The `crf` namespace contains more specialised classes to render OSG applications with Equalizer.

The lightweight eqOsg namespace provides only basic features to render an OSG scene graph. The derived classes of the crf namespace implement additional functionality for showing statistics, scene graph coordinate system correction and the crfStarter class as a facade for simple execution of the application.

class crfNodeFactory
: This class inherits from the eqOsg::NodeFactory and is used by the crfStarter class to create the desired Equalizer objects.

class crfStarter
: The crfStarter is the facade class of our framework. With this class it is possible to run an Equalizer OSG application with two lines of code shown in listing **??**.

```
#include <crf/crfStarter.h>

//1. Create the main method, which calls the crfStart.run()
//method to start equalizer and stuff
int main(int argc, char** argv)
{
    //create the \gls{crf} starter
  crf::crfStarter starter;

  //run the application
  starter.run(argc,argv);
}
```

Listing 4.2: crfStarter main method

In this special case, a HelloWorld application will be started because no node factory and no OSG node have been passed to the starter object. For simple scene graphs, it is possible to pass an OSG root node as third argument of the starter.run(...) method. For dynamic and more complex OSG applications, custom classes have to be implemented (by subclassing the desired eqOsg and crf classes) and a custom node factory has to be passed to the starter with starter.setNodeFactory(yourCustomFactory).

class crfPipe
: This class extends the eqOsg::Pipe with features like pushing Equalizer events to the OSG viewer, scene graph updates, scene graph rotation and more.

4.4.5 OSG Rendering With Equalizer

Overview

This section explains how the scene graph is rendered with Equalizer. In the *eqPly* or the *eqHello* example of Equalizer, there is native OpenGL code in the Channel::frameDraw() function. But in the CRF, the OpenGL commands come from the OSG viewer's camera and have to be rendered considering every channel's viewing matrix.

Frame Rendering

The OSG related rendering is done by the eqOsg::Channel::frameDraw() function. The code of eqOsg::Channel::frameDraw() is shown in listing **??**.

```
void Channel::frameDraw( const uint32_t frameID )
{
  // setup OpenGL State
```

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

  // get the pipe and the viewer
  Pipe *pipe = static_cast<Pipe*>( getPipe() );
  osg::ref_ptr<EqViewer> viewer = pipe->getViewer();

  // set the correct viewport (which can change each frame)
  viewer->setViewport( this );

  // set a view matrix and make sure it is multiplied with the head
  // matrix of Equalizer
  osg::Matrix headView = setPlainViewMatrix( pipe );
  headView.postMult( vmmlToOsg( getHeadTransform() ) );
  //set the channels correct view matrix to the osgViewer's camera
  viewer->getCamera()->setViewMatrix( headView );

  // render the scene graph
  viewer->frame();

  ... more instructions for drawing statistic
}
```

Listing 4.3: `Channel::frameDraw()`

The important difference of the implementation of `Channel::frameDraw()` in the *eqPly* example and in the CRF is that the drawing code is retrieved from the `osgViewer` of the pipe and is not directly realised with common OpenGL commands. The frame of the viewer depends on the current main camera in the scene graph. Therefore, the correct properties (position, up-vector and viewing direction) for the camera need to be set to get the correct frame for each Equalizer channel. This very important step is done by the `eqOsg::Channel::frameDraw()` function, posted above. With `setPlaneViewMatrix(pipe)` the cameras position, up-vector and view direction is updated because the camera can be moved. This step is executed for every channel. After setting up the general camera properties, the cameras view matrix has to be defined. This is different for every channel and depends on the Equalizer configuration and the head matrix (which represents the viewers position and orientation). The `getHeadTransform()` function returns the correct view frustum and orientation for each channel defined in the Equalizer configuration. After multiplying the view matrix of the camera with the head transformation matrix of the channel, the result is set as the new view matrix for the camera of the channel. Afterwards, the drawing code can be executed by calling `viewer->frame()`.

Summary

1. call the base `frameDraw()` of the Equalizer base class to set-up the correct OpenGL state
2. pass the viewport of the channel to the OSG viewer
3. get the current position, view direction and up-vector of the main camera of the scene graph
4. multiply this with the head transformation matrix of the channel
5. set the result as new view matrix for the camera of the channel
6. render the frame

4.4.6 Event Handling

Equalizer Event Handling

In the eqOsg and crf Equalizer implementation, the general event handling is done in the Config class. The Config::handleEvents(ConfigEvent event) function handles the Equalizer events. To get more information about the general event handling in Equalizer, consult the Equalizer Programming Guide **?**.

CRF Event Handling

In OSG applications, it is very common to add multiple event handlers to the osgViewer. The osgViewer receives all the events directly, because in basic OSG applications this viewer creates its own window(s). In this project, Equalizer generates the windows and receives the events. To keep the possibility of using the common OSG event handlers, a mechanism is created that passes all the key and mouse related events from the crf::crfConfig::handleEvents() function via the FrameData's eventQueue to every pipe. In the crfPipe::frameStart() function, Equalizer events are converted to OSG events and passed to the event queue of eqOsg::EqViewer. Everytime crf::crfConfig::frameFinish() is called, the eventQueue of FrameData is cleared because the events have already been pushed to the eventQueue of the OSG viewer by the pipe during the last crfPipe::frameDraw() call.

The three functions convertKeyPressEvents(...), convertKeyReleaseEvents(...) and convertMouseEvents(...) convert the Equalizer events to OSG events and push them to the eventQueue of the viewer. Some basic conversions are done in the crf::Pipe class. To change these conversions and/or support additional keys, these functions have to be overridden. Unfortunately, this part of the crf depends on the operating system because each windowing system reports different ACSII codes. Moreover, Equalizer does not report all codes properly for some special keys. This is already tracked as a bug and should be fixed in future Equalizer releases.

Summary

1. crf::crfConfig::handleEvent(event) pushes the desired events (default: key and mouse events) to the _eventQueue vector of the distributed _framedata object
2. crf::crfConfig::handleEvent(event) calls the eqOsg::Config::handleEvent(event) function
3. depending on the handled events, the _framedata object gets updated
4. eqOsg::Config::frameDraw() commits the _framedata object
5. in crf::crfPipe::frameStart convertKeyPressEvents, convertKeyReleaseEvents and convertKeyPressEvents are called
6. in these functions, the passed events are converted to OSG events and thereafter pushed to the event queue of the viewer (see figure **??**)

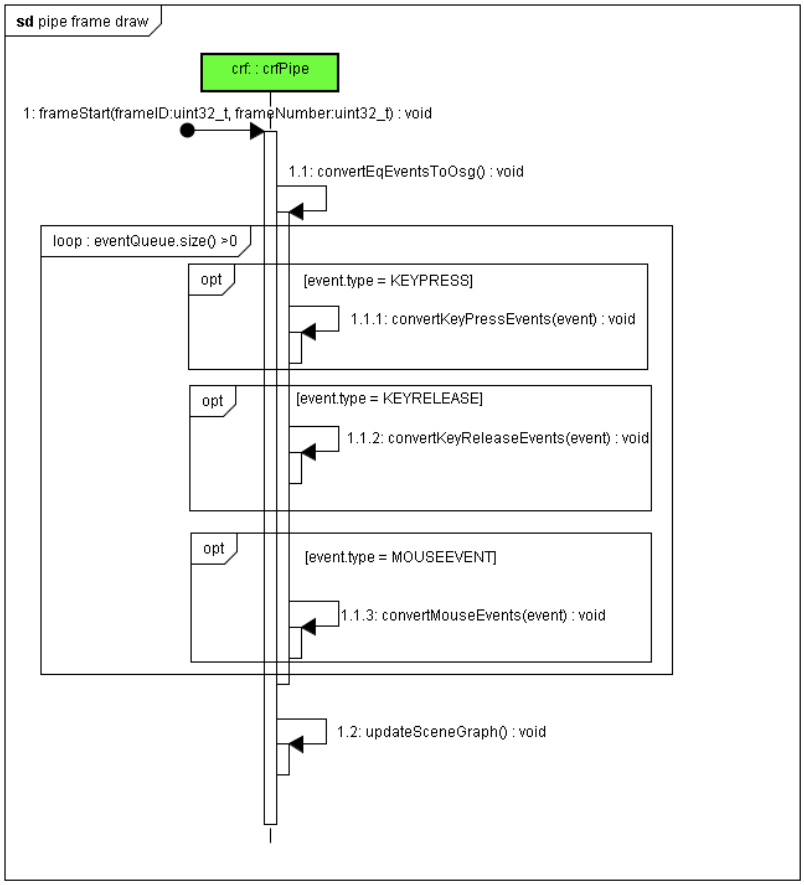

Figure 4.2: the pipe's `frameDraw()` function as a sequence diagram

4.5 Limitations

4.5.1 Scene Graph Creation

Due to the fact that in Equalizer configurations with n pipes on one node n OSG scene graphs are created. The used objects in this creation-process must not be global and/or static. This is because every scene graph has to be completely independent and must not share the same memory. If static objects are used, every scene graph refers to the same memory on the node with several pipes. This causes errors or even segmentation faults.

4.5.2 Dynamic Scene Graphs

Changes in the scene graph have to be synchronised over all pipes and nodes. Due to the fact that distributed objects (the `FrameData` class at runtime and the `InitData` class on start-up in the CRF) provide the only way to communicate between pipes and nodes. Common viewer bound OSG event handlers (like the `osgGA::TrackBallManipulator` or the `osgViewer::Statshandler`) can be used because the CRF sends all mouse and keyboard events to every viewer of every pipe with its `FrameData` class. To add other input devices like a 3D joystick or similar, the new events have to be pushed from the `Config` class over a distributed object to every pipe. Now, the scene graph's changes have to be made in the pipe. Due to the fact that the distributed objects are completely synchronised, every pipe receives the same events at the same time (frame). To

summarise, all changes in a scene graph have to be caused by distributed data. In our case this responsibility is taken by the `FrameData` class.

4.5.3 Model Loading with Multiple Pipes on One Machine

Random segmentation faults occured when loading OSG models with `osgDB::loadNodeFile()` on a multipipe node with just one running instance of the CRF application. After some debugging efforts these random errors seem to be caused by a not thread-safe singleton pattern, used in OSG. This error does not appear in multinode environments where on every node one instance of the CRF application runs per node and therefore the singleton pattern is processed properly.

4.6 Previous Work

This framework is based on the basic OSG-Equalizer application eqOSG, realised by a VR research group of the University of Siegen.

5 Outlook

5.1 Equalizer

Two days ago the chief developer of Equalizer announced the Version 1.0 of Equalizer in the next few weeks/month. Our framework was developed and tested on top of Equalizer version 0.6-rc1. New features can be found at the official website of Equalizer: http://www.equalizergraphics.com.

One of the changes of Equalizer that may affect our framework is the change of the configuration file formats. Equalizer writes on its website:

> Note that this format is the representation for the server's low-level scalable rendering engine. Eventually this file format will be replaced by a higher-level format or API, and may even be partially hidden from the user. Automatic configuration and load balancing are not yet implemented, hence the need to have these low-level configuration files.

As Equalizer 1.0 is considered to contain stable parts only, it would be a future objetive to upgrade it in the framework.

5.2 CAVE Rendering Framework

During our thesis project we could realise our main goal: An easy-to-use framework for the desired integration of OSG into Equalizer. To provide more possibilities, we expanded the framework with more functionality. But there are still a lot of possibilities to advance this project. We would like to point out some of these possible future enhancements.

5.2.1 Distributed Objects

At the moment, the CRF provides no simple mechanism to add more custom distributed data objects to the application. We could not find a simple approach to achieve this feature in a satisfying manor. A distributed object has to be mapped at both sides (master/slave) of the (network) application, committed, synced and counted which leads into a lot of changes just for this new object. To do this, one has to edit more or less all the CRF/eqOsg classes.

5.2.2 Advanced Control Application

Currently we cannot provide a specialised and advanced server application. It is a common approach in Equalizer for more complex applications to create an executable which does absolutely no rendering tasks but handles user input, physics (collision detection), haptics, data distribution or similar. Consult the **?** for further information about differences between client and server applications in Equalizer.

5.2.3 Tracking

It should not be a big deal to introduce a tracker into the CRF. Because of the fact that our key and mouse bound camera handling simulates a kind of tracker, it should be easy to add a real tracker. A straight forward approach would be to add a reference to the tracker's interface in the `crfConfig` class and thereafter pass the tracker's current head matrix to the Equalizer's head matrix in the `crfConfig::startFrame()` function. Of course this would bring along a lot of

calibration work and some special requirements to the OSG scene graph, but technically it should be absolutely possible with the CRF. Unfortunately, there was not enough time to test this use case.

5.2.4 Distributed OSG Scene Graph

Another big effort would be needed to achieve a fully distributed OSG scene graph. A trial can be found in the community **?**, but this project has not been finished yet and is highly likely currently stopped.

6 Testing

6.1 Overview

The CRF has been tested in several environments with multiple applications. Most of them test mainly OSG functionality. Additionally a stress test has been realised with a special application. All test results are listed in the appendix (see **??**).

6.2 Stress Tests

In order to test the performance of the Cave Rendering Framework we wrote a test application. Purpose of this application is to test the framework under extreme circumstances. We used huge models[1] to exhaust the rendering process. The application draws a scene graph with one model on each wall, or floor, respectively. This way, the application should challenge each pipe equally. Additionally, one can let the models rotate and add some light sources to pressure the CRF even more. It turned out that the CRF does not really care about rotations but illumination almost cut the frame rate in half.

The biggest model we could load was the dragon with about 1.1 million triangles. Loading the next bigger model, another dragon with 7.2 million triangles, failed due to memory limitations.

To measure the output we implemented a function to display generic OSG statistics as well as Equalizer statistics. Figure **??** shows a sample output of the stress application with several OSG statistics.

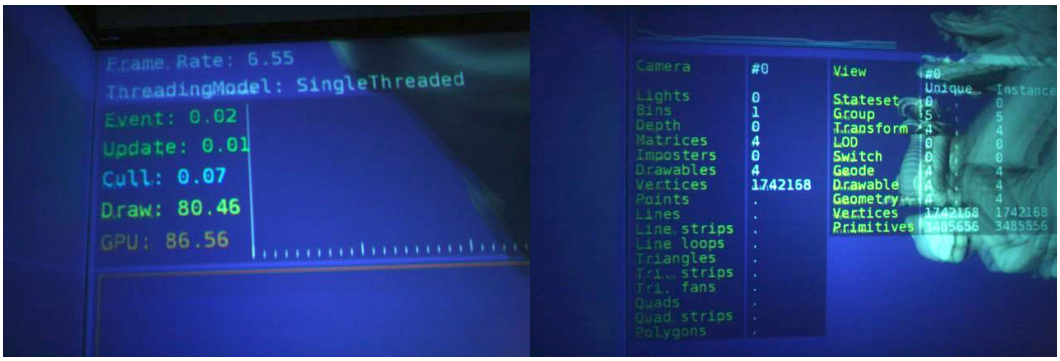

Figure 6.1: OSG statistics

[1] We used models from the Stanford 3D Scanning Repository, provided on http://www-graphics.stanford.edu/data/3Dscanrep. The models range from about 700'000 (Stanford Bunny) up to 28'000'000 (Lucy) triangles.

6.2.1 Test Systems

Equalizer configuration	1 node, 8 pipes	5 nodes, 1pipe each
CPU	Intel Quadcore Q9550 @2.8GHz	2x Intel P4 @3.2GHz
Memory	8GB	2GB
GPU	4x NVIDIA Quadro FX 1700	NVIDIA Quadro 3400
Network	-	1GBit dedicated network
OS	(X)Ubuntu 8.10	
OSG	OpenSceneGraph-2.8.1	
Equalizer	Equalizer-0.6-rc1	
Remarks	1 client with 4 GPUs with 2 channels each = 8 Equalizer pipes	5 rendering clients with 1 GPU and 1 channel each = 5 Equalizer nodes with 1 pipe and 1 channel each

Table 6.1: Hardware Setup of Test Environment

6.3 Test Conclusion

6.3.1 OSG Tests

Most of the tested OSG scenes rendered properly and produced correct images. With the proper Equalizer configuration the stereoscopic immersion was satisfying and worked out of the box. Moreover, all the tested animated scene graphs worked correctly and even dynamic scenes have been rendered fully synchronously. On rendering clients with multiple GPUs and one instance of the running CRF application, it is very important to implemented the OSG scene properly (see **??**) to avoid memory abuse.

6.3.2 Stress Test

The most interesting output for our measurements was the framerate per second (fps). As mentioned before, we tested two different Equalizer setups. One with a single node, using four graphics cards. The other, a distributed setup with six rendering clients and one server. On the latter, we achieved a more or less constant framerate of 100 fps with a model of 1'100'000 triangles. This was about what we expected.

Figure **??** shows the output of Equalizer measured with the multi-node setup. Equalizer orients itself on the slowest rendering client. The green at the bottom of the statistics represents client number six of the setup. It is the slowest client, meaning, it needs the most time to render a single frame. The five other clients listed above need about 1/5 of the time to render a frame.

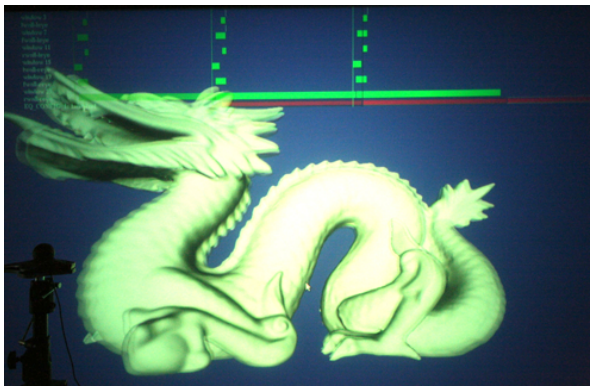

Figure 6.2: Equalizer statistics

The second approach with one computer was less satisfying. With this setup we reached a maximum of 20 fps. Conspicuously, the frame rate decreases linearly with each model, even though, they appear on separate walls and should therefore be rendered in different pipes. This behaviour leads to the conclusion that the entire scene graph is rendered on each graphics card. Concerning this issue, we wrote on the Equalizer mailing list. Stefan Eilemann, the founder and developer of Equalizer pointed out that this might be a problem of the NVIDIA graphics driver which could be solved with a later driver release. Other possible reasons he pointed out were:

- CPU/Memory contention
- Serialisation of the draw calls
- Bad multi-threading driver support
- Lock contention in OSG render traversal.

The last reason seemed unlikely since the OSG community never reported similar problems and the eVolve volume rendering demo, shipped with Equalizer, produces bad results too.

7 Demo Applications

The CRF package comes with some demo applications to demonstrate the functionalities of the framework. The following list gives an overview of these applications and their purpose:

`crfHelloWorld`
: The purpose of this very simple application is to test if the CRF is installed and running properly. Since it is a very slim and simple application, it can be used to test Equalizer configuration files. The application simply shows the word `OSG` written in light blue flickering letters. One can load any desired `*.osg` model files by adding the following parameter: `-model=<filename>`. Screenshots can be found in the Appendix **??**.

`crfShaderDemo`
: The sole purpose of this demo application is to verify whether GL Shading Language (GLSL) shaders can be loaded and executed. It shows the same shader demo provided by the OSG group.

`crfSpaceFlight`
: This last demo application shows some planets rotating around the sun.

Appendix **??** holds screenshots of the expected output for each application.

A Example Configurations

A.1 1node-8pipe configuration

```
# Configuration file for 1 appNode with 8 pipes (4 graphics cards with 2 dvi
    outputs each).
# The rendering output is a 2mx2m CAVE with 3 sides and a floor screen.

# the global section is configures attributes that are valid for all
    configurations listed below.
global {
  EQ_CONFIG_FATTR_EYE_BASE  0.025 # set the eye distance between the left and
      the right eye.
}

server {
  config {
    # the appNode represents the node, on which the application runs.
    appNode {
      pipe {
        device 0 # port number of graphics card
        window {
          # The channel id (name) must be unique. It is the reference for the
              compounds.
          channel { name "fwall-leye" }
        }
      }
      pipe { device 1 window { channel { name "rwall-leye" } } }
      pipe { device 2 window { channel { name "floor-leye" } } }
      pipe { device 3 window { channel { name "lwall-leye" } } }
      pipe { device 4 window { channel { name "floor-reye" } } }
      pipe { device 5 window { channel { name "fwall-reye" } } }
      pipe { device 6 window { channel { name "lwall-reye" } } }
      pipe { device 7 window { channel { name "rwall-reye" } } }
    }
    compound {
      compound {
        # a wall is uniquely defined by 3 points
        wall {
          bottom_left   [ -1.25 -1  1 ]
          bottom_right  [ -1.25 -1 -1 ]
          top_left      [ -1.25  1  1 ]
        }
        # a compound inheriting from the upper compound with the wall.
        compound {
          channel "lwall-leye" # channel to be displayed
          eye [ LEFT ] # eye pass
        }
        compound {
          channel "lwall-reye"
          eye [ RIGHT ]
        }
      }
      compound {
        wall {
          bottom_left   [ -1.25 -1 -1 ]
          bottom_right  [  1.25 -1 -1 ]
          top_left      [ -1.25  1 -1 ]
        }
```

```
        compound {
          channel "fwall-leye"
          eye [ LEFT ]
        }
        compound {
          channel "fwall-reye"
          eye [ RIGHT ]
        }
      }
      compound {
        wall {
          bottom_left    [  1.25 -1 -1 ]
          bottom_right   [  1.25 -1  1 ]
          top_left       [  1.25  1 -1 ]
        }
        compound {
          channel "rwall-leye"
          eye [ LEFT ]
        }
        compound {
          channel "rwall-reye"
          eye [ RIGHT ]
        }
      }
      compound {
        wall {
          bottom_left    [ -1.25 -1  1 ]
          bottom_right   [  1.25 -1  1 ]
          top_left       [ -1.25 -1 -1 ]
        }
        compound {
          channel "floor-leye"
          eye [ LEFT ]
        }
        compound {
          channel "floor-reye"
          eye [ RIGHT ]
        }
      }
    }
  }
}
```

Listing A.1: Configuration file for 1 node, 8 pipes to a 4-sided CAVE visualisation system

A.2 6node-6pipe configuration

```
# Configuration File for 1 appNode and 2 supplement nodes
#Â The rendering output is a 2x2m CAVE with 3 sides and a floor screen

global {
  EQ_CONFIG_FATTR_EYE_BASE  0.055
  EQ_WINDOW_IATTR_HINT_FULLSCREEN ON
}

server {
  config {
```

```
appNode {
  pipe { window { channel { name "lwall-leye" } } }
}
node {
  connection { hostname "n451-rc2" }
  pipe { window { channel { name "fwall-leye" } } }
}
node {
  connection { hostname "n451-rc3" }
  pipe { window { channel { name "rwall-leye" }}}
}
node {
  connection { hostname "n451-rc4" }
  pipe { window { channel { name "lwall-reye" }}}
}
node {
  connection { hostname "n451-rc5" }
  pipe {window {channel { name "fwall-reye" }}}
}
node {
  connection { hostname "192.168.0.6" }
  pipe { window { channel { name "rwall-reye" }}}
}

compound {
  compound {
    wall {
      bottom_left    [ -1.25 -1  1 ]
      bottom_right   [ -1.25 -1 -1 ]
      top_left       [ -1.25  1  1 ]
    }
    compound {
      channel "lwall-leye"
      eye [ LEFT ]
    }
    compound {
      channel "lwall-reye"
      eye [ RIGHT ]
    }
  }
  compound {
    wall {
      bottom_left    [ -1.25 -1 -1 ]
      bottom_right   [  1.25 -1 -1 ]
      top_left       [ -1.25  1 -1 ]
    }
    compound {
      channel "fwall-leye"
      eye [ LEFT ]
    }
    compound {
      channel "fwall-reye"
      eye [ RIGHT ]
    }
  }
  compound {
    wall {
      bottom_left    [  1.25 -1 -1 ]
      bottom_right   [  1.25 -1  1 ]
      top_left       [  1.25  1 -1 ]
```

```
        }
        compound {
          channel "rwall-leye"
          eye [ LEFT ]
        }
        compound {
          channel "rwall-reye"
          eye [ RIGHT ]
        }
      }
    }
  }
}
```

Listing A.2: Configuration file for 6 nodes with 1 pipe each to a 3-sided CAVE visualisation system

B Test Results

B.1 Test Results for Model Loading

Tester	Jonas Walti	**CRF Version**	0.9	**Date**	10.05.09
Model Loading		Load a basic model `cow.osg` by passing the filename as command line argument to the `crfStarter` class.			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	✔	✔	✔	✖
2	Framerate	70	20	100	-
3	SGs in sync over screens	-	✔	✔	-
4	Stereo output	✔	✔	✔	-
5	Camera movement	✔	✔	✔	-
6	Crashless	✔	✔	✔	-
7	Clean exit	✔	✔	✔	-
Remarks	-				

Table B.1: Test Series Model Loading, CRF v0.9

Tester	Jonas Walti	**CRF Version**	1.0	**Date**	09.06.09
Model Loading		Load a basic model `cow.osg` by passing the filename as command line argument to the `crfStarter` class.			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	-	-	✔	✔
2	Framerate	-	-	106	550
3	SGs in sync over screens	-	-	✔	✔
4	Stereo output	-	-	✔	✔
5	Camera movement	-	-	✔	✔
6	Crashless	-	-	✔	✔
7	Clean exit	-	-	✔	✔
Remarks	-				

Table B.2: Test Series Model Loading, CRF v1.0

B.2 Test Results for Scene Graph Tests

Tester	Jonas Walti	**CRF Version**	0.9	**Date**	29.05.09
Adv. Scene Graph		(1) creating a more complex scene graph by overriding CRF functions, (2) creating animations by overriding `updateSceneGraph()`			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	✔	✔	✔	✔
2	Framerate	60	20	60	200
3	SGs in sync over screens	-	✔	✔	✔
4	Stereo output	-	-	✔	✔
5	Camera movement	✔	✔	✔	✔
6	Crashless	✔	✔	✔	✔
7	Clean exit	✔	✔	✔	✔
8	Animation in Sync	-	✔	✔	✔
Remarks	Our own example with a simple universe and its moving planets. Framerate was expected to be higher! Bad EQ config?				

Table B.3: Test Series Model Loading, CRF v0.0

B.3 Test Results for Shader Tests

Tester	Jonas Walti	**CRF Version**	0.5	**Date**	10.05.09
Simple Shader Test		starting `crfStarter` with `glsl_xxx` osg models as argument.			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	✔	✔	-	-
2	Framerate	60	20	-	-
3	SGs in sync over screens	-	✔	-	-
4	Stereo output	-	-	-	-
5	Camera movement	✔	✔	-	-
6	Crashless	✔	✔	-	-
7	Clean exit	✖	✖	-	-
8	Animation in Sync	-	-	-	-
Remarks	Segmentation fault when closing the application. Debugging needed! Problem when deleting a shader program twice. This should not happen.				

Table B.4: Test Series Simple Shader Test, CRF v0.5

Tester	Jonas Walti	**CRF Version**	0.9	**Date**	03.06.09
Simple Shader Test		starting `crfStarter` with `glsl_xxx` osg models as argument.			

#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	✔	✔	✔	-
2	Framerate	60	20	60	-
3	SGs in sync over screens	-	✔	✔	-
4	Stereo output	-	-	✔	-
5	Camera movement	✔	✔	✔	-
6	Crashless	✔	✔	✔	-
7	Clean exit	✔	✔	✔	-
8	Animation in Sync	-	-	✔	-
Remarks	Bug fixed! See **??**				

Table B.5: Test Series Simple Shader Test, CRF v0.9

Tester	Jonas Walti	**CRF Version**	1.0	**Date**	09.06.09
Simple Shader Test		starting `crfStarter` with `glsl_xxx` osg models as argument.			

#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	-	-	✔	✔
2	Framerate	-	-	140	550
3	SGs in sync over screens	-	-	✔	✔
4	Stereo output	-	-	✔	✔
5	Camera movement	-	-	✔	✔
6	Crashless	-	-	✔	✔
7	Clean exit	-	-	✔	✔
8	Animation in Sync	-	-	✔	✔
Remarks					

Table B.6: Test Series Simple Shader Test, CRF v1.0

Tester	Jonas Walti	**CRF Version**	0.9	**Date**	03.06.09
Adv. Shader Test		straight forward ported version of the OSG Shader Example			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	✔	✖	✖	✔
2	Framerate	60	20	70	300
3	SGs in sync over screens	-	✖	✖	-
4	Stereo output	-	-	✔	✔
5	Camera movement	✔	✖	✖	✔
6	Crashless	✔	✔	✔	✔
7	Clean exit	✔	✔	✔	✔
8	Animation in Sync	-	-	-	✔
Remarks	Display errors because this ported OSG example uses a lot of global variables. This (bad) technique does not work when using multiple pipes on one node, because every scene graph uses these global objects. Reimplementation needed for further tests.				

Table B.7: Test Series Advanced Shader Test, CRF v0.9

Tester	Jonas Walti	**CRF Version**	1.0	**Date**	09.06.09
Adv. Shader Test		straight forward ported version of the OSG Shader Example			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	-	-	✔	✔
2	Framerate	-	-	60	250
3	SGs in sync over screens	-	-	✔	✔
4	Stereo output	-	-	✔	✔
5	Camera movement	-	-	✔	✔
6	Crashless	-	-	✔	✔
7	Clean exit	-	-	✔	✔
8	Animation in Sync	-	-	✔	✔
Remarks	Our own example with a simple universe and its moving planets. Framerate was expected to be igher! Bad EQ Config?				

Table B.8: Test Series Advanced Shader Test, CRF v1.0

Tester	Jonas Walti	**CRF Version**	1.0	**Date**	09.06.09
Adv. Shader Test		straight forward ported version of the OSG Shader Example			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	-	-	-	✔
2	Framerate	-	-	-	300
3	SGs in sync over screens	-	-	-	✔
4	Stereo output	-	-	-	✔
5	Camera movement	-	-	-	✔
6	Crashless	-	-	-	✔
7	Clean exit	-	-	-	✔
8	Animation in Sync	-	-	-	✔
Remarks	Our own example with a simple universe and its moving planets. Framerate was expected to be igher! Bad EQ Config?				

Table B.9: Test Series Advanced Shader Test, CRF v1.0

B.4 Test Results for Event Tests

Tester	Jonas Walti	**CRF Version**	0.9	**Date**	29.05.09
OSG Event Test		disabled eq Camera, added `osgTrackballManipulator` to the viewers camera, added the `osgStatsHandler` to the viewer (which can be toggled by key press events)			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	✖	✖	-	-
2	Framerate	60	20	-	-
3	SGs in sync over screens	-	✔	-	-
4	Stereo output	-	-	-	-
5	Camera movement	✔	✔	-	-
6	Crashless	✔	✔	-	-
7	Clean exit	✔	✔	-	-
8	Animation in Sync	-	-	-	-
Remarks	event forwarding works, but EQ does not report the proper ASCII codes for some key events. Tracked in the Technical Documentation. Windows reports uppercase keys! This has to be respected when implementing `convertEventsToOsg()`				

Table B.10: Test Series OSG Event Tests, CRF v0.9

B.5 Test Results for Blender Tests

Tester	Jonas Walti	**CRF Version**	0.9	**Date**	24.05.09
Blender Test		Exporting lender scenes with the OSG Exporter blender plugin. Loading the `.osg` model with the `crfStarter` as argument			

#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	✔	✔	✔	✔
2	Framerate	variable	variable	60	270
3	SGs in sync over screens	-	✔	✔	✔
4	Stereo output	-	-	✔	✔
5	Camera movement	✔	✔	✔	✔
6	Crashless	✔	✔	✔	✔
7	Clean exit	✔	✔	✔	✔
8	Animation in Sync	-	-	-	-

Remarks	when the blender-to-osg limitations have been respected (well documented on the internet) everything works fine! framerate not reported, because of the fact, that we tested several models.

Table B.11: Test Series Blender Tests, CRF v0.9

Tester	Jonas Walti	**CRF Version**	1.0	**Date**	09.06.09
Blender Test		Exporting lender scenes with the OSG Exporter blender plugin. Loading the `.osg` model with the `crfStarter` as argument			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	-	-	✔	✔
2	Framerate	variable	variable	60	550
3	SGs in sync over screens	-	-	✔	✔
4	Stereo output	-	-	✔	✔
5	Camera movement	-	-	✔	✔
6	Crashless	-	-	✔	✔
7	Clean exit	-	-	✔	✔
8	Animation in Sync	-	-	-	✔
Remarks	when the blender-to-osg limitations have been respected (well documented on the internet) everything works fine! framerate not reported, because of the fact, that we tested several models.				

Table B.12: Test Series Blender Tests, CRF v1.0

B.6 Stress Tests

Tester	Jonas Walti	**CRF Version**	0.9	**Date**	03.06.09
Stress Test		Stress demo with one rotating Stanford dragon per wall.			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	-	-	✔	✔
2	Framerate	-	-	18	97
3	SGs in sync over screens	-	-	✔	✔
4	Stereo output	-	-	✔	✔
5	Camera movement	-	-	✔	✔
6	Crashless	-	-	✔	✔
7	Clean exit	-	-	✔	✔
Remarks					

Table B.13: Test Series Stress Tests, CRF v0.9

Tester	Jonas Walti	**CRF Version**	1.0	**Date**	09.06.09
Stress Test		Stress demo with one rotating Stanford dragon per wall.			
#	**Equalizer Config:**	1 node, 1 pipe	1 node, 3 simulated pipes	1 node, 8 pipes (CAVE)	5 nodes (CAVE Network)
1	Runs as expected	-	-	✔	✔
2	Framerate	-	-	23	108
3	SGs in sync over screens	-	-	✔	✔
4	Stereo output	-	-	✔	✔
5	Camera movement	-	-	✔	✔
6	Crashless	-	-	✔	✔
7	Clean exit	-	-	✔	✔
Remarks					

Table B.14: Test Series Stress Tests, CRF v1.0

C Demo Applications

C.1 Sample Outputs for `crfHelloWorld`

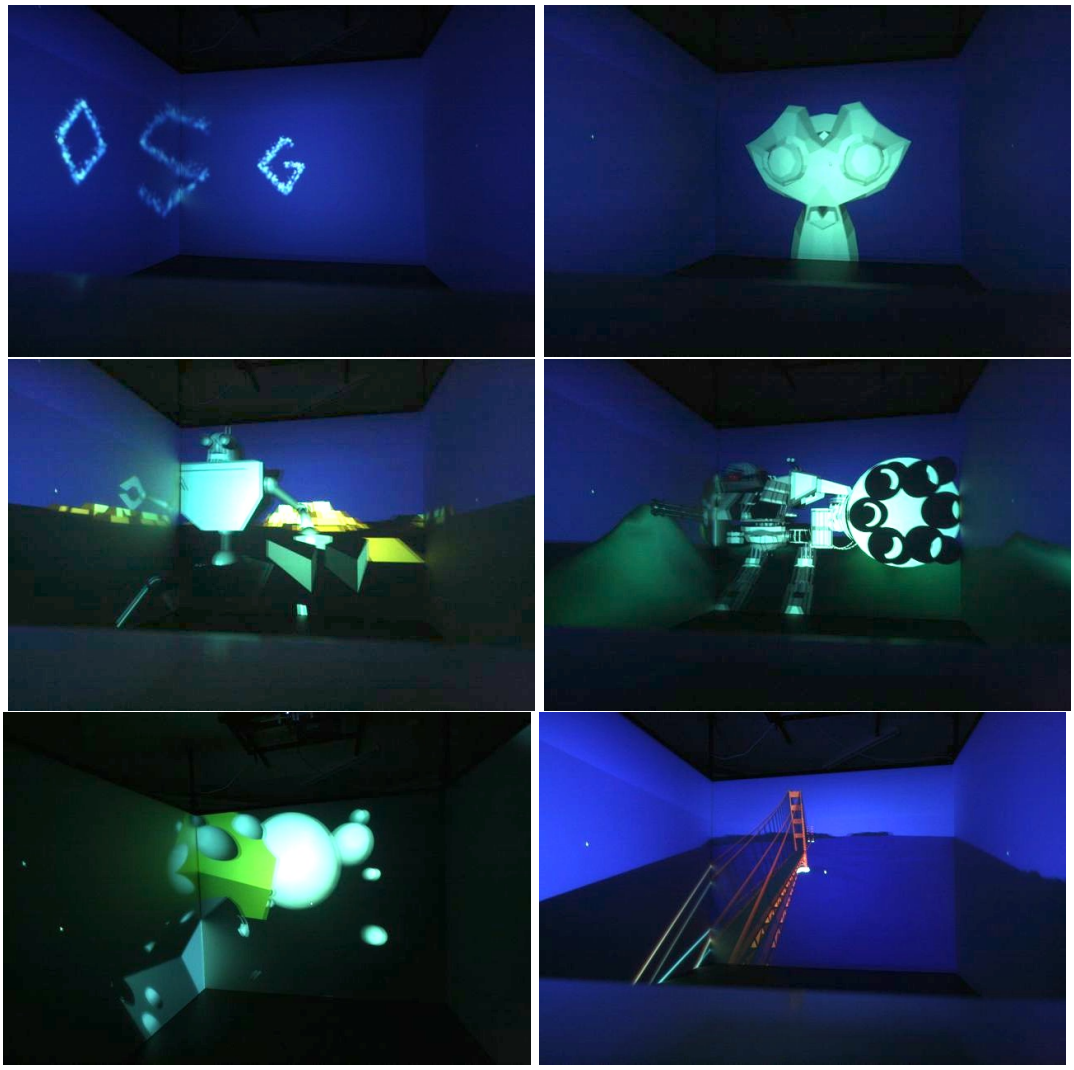

Figure C.1: `crfHelloWorld` with different models

C.2 Sample Outputs for `crfShaderDemo`

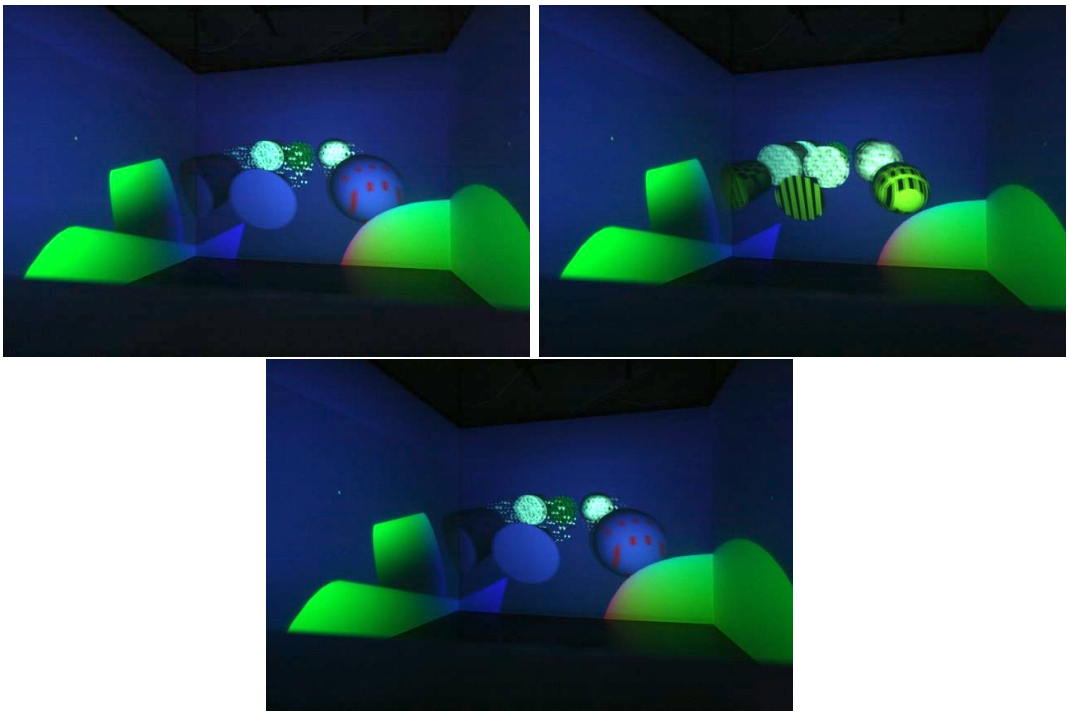

Figure C.2: `crfShaderDemo`

C.3 Sample Outputs for `crfDemo`

Figure C.3: `crfDemo`

C.4 Sample Outputs for `crfStressTest`

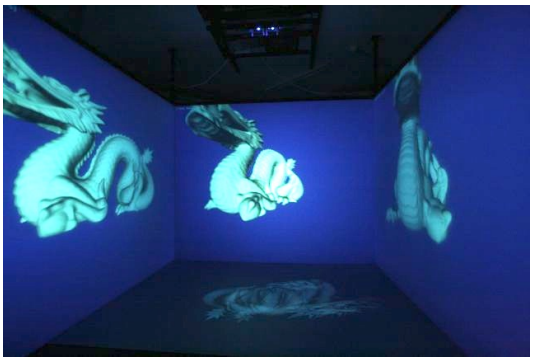

Figure C.4: `crfStressTest`

D Equalizer Notation

D.1 Global section

The global section contains default values for various attributes which are described with the following notation:

`EQ_<ENTITY>_<DATATYPE>ATTR_<ATTR_NAME>`

`ENTITY`
: refers to the corresponding section in the configuration file

`DATATYPE`
: is a single capital letter for the type of the attribute's value. Possible types are `S` for strings, `C` for chars, `I` for integers and `F` for floating point values.

`ATTR_NAME`
: is the name of the attribute

Global attributes can be overwritten with an environment variable of the same name in the appropriate section. Listing **??** shows an example of a global section in a configuration file. The distance between the left and the right eye is set to $0.025m$ per default. Stereo is set to anaglyph, which means that one image is rendered in green and the other in red which leads to a stereoscopic effect with common anaglyph glasses. Furthermore, the Equalizer renders images in fullscreen mode.

```
global {
  # sets the distance between left and right eye to 0.025m
  EQ_CONFIG_FATTR_EYE_BASE  0.025
  # set the stereo mode to anaglyph
  EQ_COMPOUND_IATTR_STEREO_MODE ANAGLYPH
  # enables fullscreen mode
  EQ_WINDOW_IATTR_HINT_FULLSCREEN ON
}
```

Listing D.1: Example for a global section in a configuration file

A more detailed description of the global section and its possible attributes can be found in the Equalizer Programming Guide [**?**, p. 60ff.].

D.2 Server Section

Each configuration requires that a node acts as the server. This node provides the Equalizer configuration file to all render clients. A server in a configuration file is the parent of all render clients. Additionally, it may contain a connection description for the server listening sockets. Currently, only the first configuration is considered no matter how many sections are defined in the file. This may change in a future release of Equalizer.

D.3 Connection Section

By now, Equalizer supports two connection types: TCP/IP and SDP. The connection section describes the network parameters of an Equalizer process. A connection description has to follow the structure shown in listing **??**.

```
connection { # 0-n times, listening connections of the server
  type     TCPIP | SDP   # connection type
  TCPIP_port  unsigned  # connection port for the socket.
  hostname  string     # optional hostname. Used for debugging
}
```

Listing D.2: Definition of a connection in Equalizer

D.4 Config Section

Possible chlidren of the `config` section are listed below and a sample is shown in listing **??**:

- *latency*: Defines the maximum number of frames the slowest operation may fall behind the application thread.
- *attributes*: Defines attributes for a specific server configuration. These attributes override the attributes in the global section.
- *node*: describes rendering resources.
- *compound*: Uses rendering resources (channels) defined as children of the *nodes*.

```
server {
  config {
    latency int # number of allowed latency frames
    attributes {
      [...] # attributes to define or override
    }
    node { # 1..n nodes for rendering
      [...]
    }
    compound { # 1..n compounds for rendering usage
      [...]
    }
  }
}
```

Listing D.3: config section in Equalizer configuration

D.5 Node

A node represents a single machine in a cluster and is one process in the operating system. Listing **??** shows a definition of a node in the configuration file.

```
node {
  name  string
  connection {
    type     TCPIP | SDP
    TCPIP_port  unsigned
    hostname  string
  }
  attributes {
    thread_model  ASYNC | DRAW_SYNC | LOCAL_SYNC
    hint_statistics FASTEST | NICEST | OFF
  }
```

```
  pipe {
    [ ... ]
  }
}
```

Listing D.4: node section in configuration file

- *name*: The name of the machine in the cluster. It has no influence on the execution, but may be useful for debugging reasons.
- *connection*: A node may have $0-n$ connection description(s). The connection descriptions are defined in the same as in the server connection section (see **??**).
- *pipe*: Each node has at least one pipe.

Consult the Equalizer Programming Guide [**?**, p. 66] for more information about the node section.

D.6 Pipe

A pipe represents a graphics card (GPU) and is one execution thread. Listing **??** shows the definition fields of a pipe in a configuration.

```
node {
  ...
  pipe {
    name  string
    port  unsinged
    device  unsigned
    viewport [viewport]
    attributes {
      ...
    }
    window { ... }
  }
}
```

Listing D.5: Pipe section in Equalizer configuration

- *name*: The name of the pipe. It has no influence on the execution but may be useful for debugging reasons.
- *port*: The port is only used for GLX window systems. It identifies the port number of the X server, i.e. the number after the colon in the *DISPLAY* description (:**0**.1).
- *device*: The device identifies the graphics adapter.
- *viewport*: The viewport can be used to override the resolution of the pipe. It is defined in pixels. The default view is automatically detected. Therefore, the viewport usually does not need to be set.

The pipe is the device that depends the most on the underlying operating system. Windowing specific information can be found in the official Equalizer Programming Guide [**?**, p. 67].

D.7 Window

A window represents an OpenGL drawable and holds an OpenGL context and its definition is listed in listing **??**.

```
window {
  name     string
  viewport   [ viewport ]
  attributes {
    ...
  }
  channel {
    [ ... ]
  }
}
```

Listing D.6: Window section in Equalizer configuration

- *name*: The name of the window. It has no influence on the execution but can be useful for debugging reasons.
- *viewport*: The viewport is relative to the pipe. It can be specified in relative or absolute coordinates.
- *attributes*: There are numerous attributes that can be set for a window. A complete list can be found in the Equalizer Programming Guide [**?**, p.67-68].

D.8 Channel

A channel is a two-dimensional area within a window. Its definition is shown in listing **??**.

```
channel {
  name     string
  viewport   [ viewport ]
  attributes {
    ...
  }
}
```

Listing D.7: Channel section in Equalizer configuration

- *name*: The name of the channel is used to identify the channel in the respective compounds. It has to be unique within a *config* section.
- *viewport*: The viewport is relative to the window. It can be specified in relative or absolute coordinates. The default viewport is `[ 0 0 1 1]`, which accords to the full window.

D.9 Compound

Compounds describe the rendering setup. They use channels for the rendering. These channels are defined in the node section described above. The structure of a compound definition is shown in listing **??**.

```
compound {
  channel    string
  eye        [ CYCLOP LEFT RIGHT ]
  wall {
    bottom_left    [ float float float ]
    bottom_right   [ float float float ]
    top_left       [ float float float ]
  }
}
```

Listing D.8: Compound section in Equalizer configuration

- *channel*: The channel keyword references the first channel with the same name in the *config* section described above.
- *eye*: defines whether to use monoscopic or stereoscopic view
- *wall*: A wall description is used to define the view frustum of the compound.

Consider the Equalizer Programming Guide [**?**, p. 68ff] for a detailed description of the compound section.