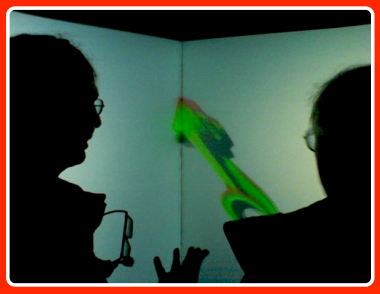
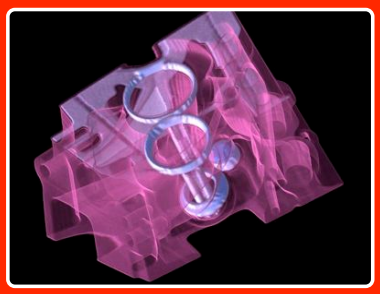
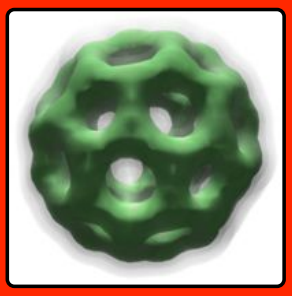
Equalizer Programming Guide

Eyescale Software GmbH







Version 1.1.2 for Equalizer 0.4+svn

November 19, 2007

Equalizer 0.4 Programming Guide

Contributors

Written by Stefan Eilemann.
Engineering contributions by Maxim Makhinya, Jonas Bösch and Christian Marten.

Copyright

©2007 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. 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 guide, please contact eile@equalizergraphics.com.

Previous Page

The images on the front page show: a terrain rendering application on a six-node display wall [top left], the eqPly polygonal renderer in a three-sided CAVE [middle], two volume rendering results from eVolve[1] [bottom left and middle] and a six-node sort-last database decomposition with parallel direct-send recomposition[2] [bottom right].

[1] Data sets courtesy of General Electric, USA and AVS, USA
[2] Data set courtesy of Stanford University Computer Graphics Laboratory

Contents

1 Introduction **1**

2 Getting Started **1**
2.1 Installing Equalizer and running eqPly . . . 1
2.2 Equalizer Processes . . . 1
2.2.1 Server . . . 1
2.2.2 Application . . . 2
2.2.3 Render Clients . . . 2

3 Hello, World! **2**

4 The Programming Interface **3**
4.1 Task Methods . . . 3
4.2 Execution Model and Thread Safety . . . 3
4.3 Config . . . 4
4.3.1 Node . . . 5
4.3.2 Pipe . . . 5
4.3.3 Window . . . 6
4.3.4 Channel . . . 6
4.4 Compounds . . . 6
4.4.1 Compound Channels . . . 6
4.4.2 Frustum . . . 6
4.4.3 Compound Classification . . . 7
4.4.4 Decomposition - Attributes . . . 7
4.4.5 Recomposition - Frames . . . 7

5 The eqPly polygonal renderer **7**
5.1 The main Function . . . 8
5.2 Application . . . 9
5.2.1 Main Loop . . . 9
5.2.2 Render Clients . . . 11
5.3 Distributed Objects . . . 12
5.3.1 Change Handling . . . 12
5.3.2 InitData - a Static Distributed Object . . . 12
5.3.3 FrameData - a Versioned Distributed Object . . . 13
5.4 Config . . . 14
5.4.1 Initialization and Exit . . . 14
5.4.2 Frame Control . . . 15
5.4.3 Event Handling . . . 16
5.5 Node . . . 17
5.5.1 Frame Control . . . 17
5.6 Pipe . . . 19
5.6.1 Initialization and Exit . . . 19
5.6.2 Window System . . . 20
5.6.3 Carbon/AGL Thread Safety . . . 20
5.6.4 Frame Control . . . 20
5.7 Window . . . 21
5.7.1 Initialization and Exit . . . 21
5.7.2 Object Manager . . . 22
5.8 Channel . . . 23
5.8.1 Initialization and Exit . . . 23
5.8.2 Rendering . . . 24

6 Advanced Features **28**
6.1 Event Handling 28
6.1.1 Threading 28
6.1.2 Initialization and Exit 28
6.1.3 Message Pump 29
6.1.4 Event Data Flow 29
6.1.5 Custom Events in eqPixelBench 30
6.2 Image Compositing for Scalable Rendering 31
6.2.1 Parallel Direct Send Compositing 31
6.2.2 Frame, Frame Data and Images 32
6.2.3 Custom Assembly in eVolve 33
6.3 Head Tracking 34

Rev	Date	Changes
1.0	Oct 28, 2007	Initial Version for Equalizer 0.4
1.0.1	Nov 06, 2007	Fixed Typo
1.1	Nov 12, 2007	Adapted distributed objects
1.1.1	Nov 16, 2007	Minor improvements based on lulu.com print
1.1.2	Nov 19, 2007	Minor improvements based on feedback

1 Introduction

Equalizer provides a framework for the development of parallel OpenGL applications. Equalizer-based applications can run from a single shared-memory system with one or multiple graphics cards up to large-scale graphics clusters. This Programming Guide introduces the programming interface, often using the Equalizer eqPly example as a guideline.

Equalizer is the next step in the evolution of generic parallel programming interfaces for OpenGL-based visualization applications. Existing solutions, such as OpenGL Multipipe SDK, Cavelib and VRJuggler, implement a subset of concepts similar to Equalizer. In other areas, e.g., tracking device support, they provide more functionality.

In order to adapt 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. 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 and fully customizable event handling.

If you have any question regarding Equalizer programming, this programming guide, or other specific problems you encountered, please direct them to the eq-dev mailing list[3].

2 Getting Started

2.1 Installing Equalizer and running eqPly

Equalizer can be installed by downloading the distribution[4] and compiling the source code. After installing Equalizer, please take a look at the Quickstart Guide[5] to get familiar with the capabilities of the eqPly example.

Compiling Equalizer is as simple as running make on Linux or building the Equalizer Visual Studio 2005 solution on Windows. On Mac OS X 10.4 (Tiger), some prerequisites have to be installed before running make, as explained in README.Darwin. Mac OS X 10.5 Leopard does have all the prerequisites installed by default.

2.2 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.

2.2.1 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 future.

[3] see http://www.equalizergraphics.com/lists.html
[4] http://www.equalizergraphics.com/downloads.html
[5] http://www.equalizergraphics.com/documents/EqualizerGuide.html

2.2.2 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 database and controls the rendering.

2.2.3 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, based on the rendering tasks received from the server. The tasks are executed by calling the appropriate task methods (see Section 4.1) 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.

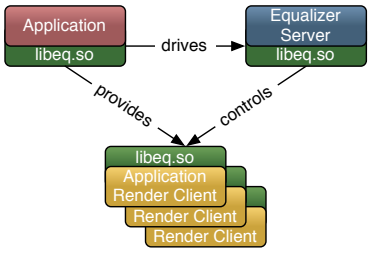


Figure 1: Equalizer Processes

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

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.

3 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.

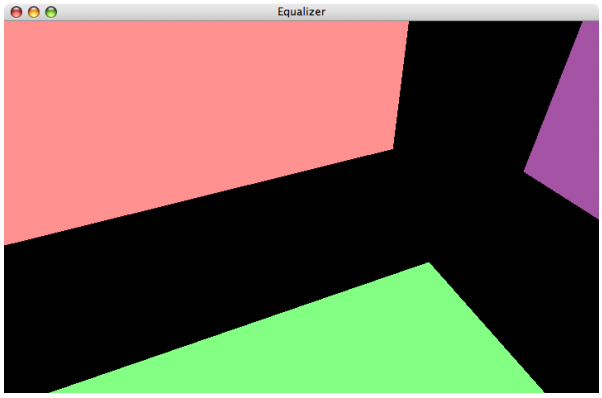


Figure 2: Hello, World!

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 convience function to setup the frustum and other OpenGL state. After setting up some additional lighting parameter, 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
    [...]
}
```

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 5.

4 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 application. It does not provide a scene graph, or interfere in any other way with the application's rendering code. The restructuring work required for 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 other things– Node::frameStart to be called in all active render clients.

4.1 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. The eqPly section will discuss the most important task methods. A full list can be found on the website[6].

4.2 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 context for each thread. If structured correctly, the application rarely has to take care of thread synchonization 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 3.

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.

[6]see http://www.equalizergraphics.com/documents/design/taskMethods.html

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.

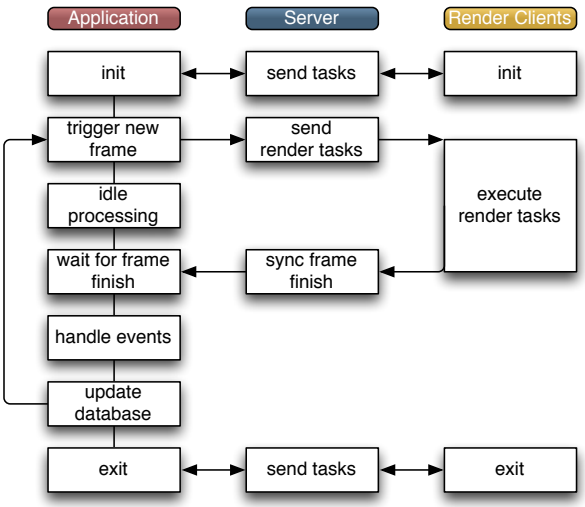


Figure 3: Simplified execution model

Rendering threads on a single node are by default synchronized. 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 application 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 5.5.1. Some scene graphs do implement multi-buffered data.

4.3 Config

The eq::Config represents the current configuration of the application. The configuration is the session in which all render clients are registered. A configuration consists of the description of the rendering resources and the usage description for these resources.

The rendering resources are represented in a hierarchical tree structure which corresponds to the physical and logical resources found in a 3D rendering environment.

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 4.4.

Figure 4 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 deployment, one node and pipe should be used for each computer and graphics card, respectively.

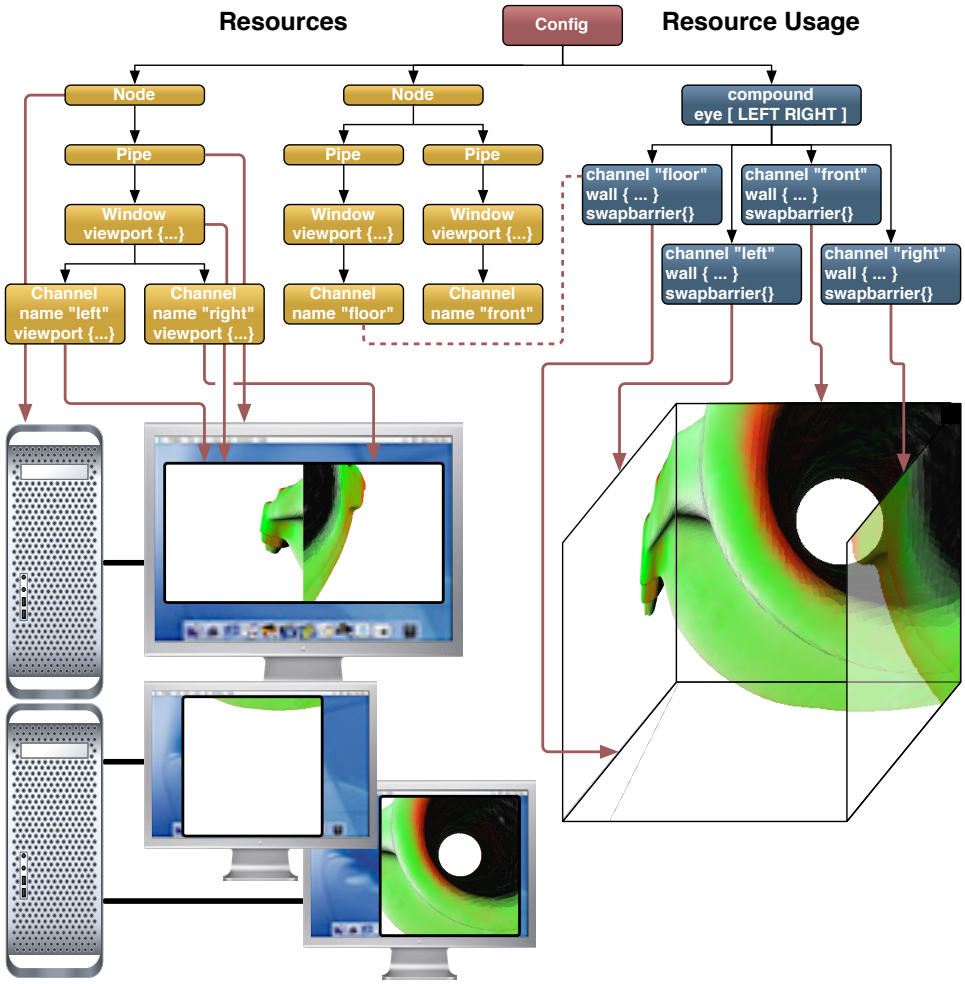


Figure 4: An example configuration

4.3.1 Node

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.

4.3.2 Pipe

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. All pipe, window and channel task methods are executed from the pipe's thread, or in the case of non-threaded pipes from the main application thread[7].

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

[7]see http://www.equalizergraphics.com/documents/design/nonthreaded.html

4.3.3 Window

The eq::Window class holds a drawable and an OpenGL context. The drawable can be an on-screen window or an off-screen[8] PBuffer or framebuffer object (FBO). The window holds window-system-specific handles to the drawable and context, e.g., an X11 window XID and GLXContext for the glX window system.

4.3.4 Channel

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 might be overwritten when it is rendering for another channel during scalable rendering.

4.4 Compounds

Usage of the rendering resources is configured using a compound tree. Although the API does not currently expose compounds, the basic design behind compounds is explained here for a better understanding of the Equalizer architecture. Further information on the configuration of compounds can be found on the Equalizer website[9].

4.4.1 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 are not instantiated during initialization. The rendering tasks for the channels are computed by the server and send to the appropriate render clients.

4.4.2 Frustum

Compounds have a frustum description to define the physical layout of the display environment.

The frustum description is inherited by the children, therefore the frustum is typically defined on the top-most compound.

The frustrum can be specified as a wall or projection description.

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

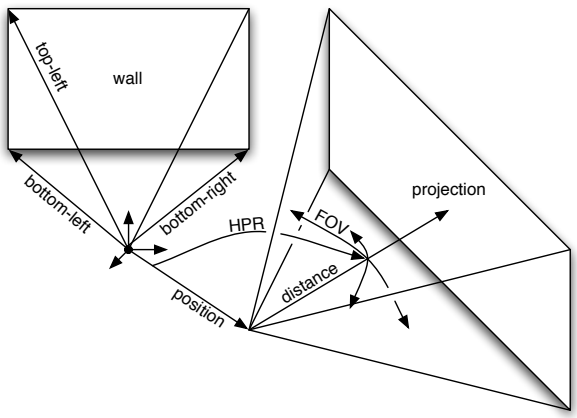


Figure 5: Wall and projection parameters

The 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 5 illustrates the wall and projection frustum parameters.

[8] Off-screen drawables are not implemented yet, but can be created by the application and used with Equalizer

[9] see http://www.equalizergraphics.com/documents/design/compounds.html

4.4.3 Compound Classification

The channels of the leaf compounds in the compound tree are designated as source channels. The top-most channel in the tree is the destination channel. Only source channels execute rendering task. 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 can not be overridden by child compounds.

4.4.4 Decomposition - Attributes

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

4.4.5 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 6.2.

5 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 raised.

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 rendering features or visual quality.

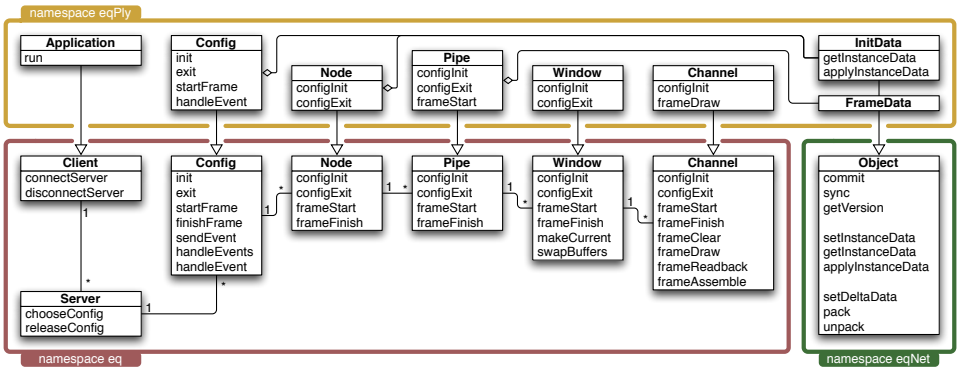


Figure 6: UML diagram of significant Equalizer and eqPly classes and task methods.

All classes in the example are in the eqPly namespace to avoid type name ambiguities, in particular for the Window class which is frequently used as a type in the

global namespace by windowing systems. Figure 6 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 4.3, and two other 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 eqNet::Object. They illustrate the typical usage of distributed objects for static as well as dynamic, frame-specific data.

5.1 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 5.3.2:

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

The second 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 options[10], 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:

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

The third step is to create an instance of the application and to initialize it locally. The application is an eq::Client, which in turn is an eqNet::Node. The underlying network layer in Equalizer is a peer-to-peer network of eqNet::Nodes. The application programmer does not usually have to be aware of the classes in the eqNet namespace, but both the eq::Client and the server are eqNet::Nodes.

The local initialization of a node creates a local listening socket, which allows the eq::Client to communicate over the network with other nodes, such as the server and the rendering clients.

[10] Equalizer-specific options always start with --eq-

```
    // 3. 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:

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

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

```
    // 5. cleanup and exit
    client->exitLocal();
    client = 0;

    eq::exit();
    return ret;
}
```

5.2 Application

In the case of eqPly, the application is also the render client. The eqPly executable has three run-time behaviours:

1. **Application**: The executable started by the user, the controlling entity in 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.

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 potentially –eq-listen <address> and -r[11].

5.2.1 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 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 configuration[12].

```
int Application::run()
{
    // 1. connect to server
    RefPtr<eq::Server> server = new eq::Server;
    if( !connectServer( server ))
    {
        EQERROR << "Can't open server" << endl;
        return EXIT_FAILURE;
    }
```

[11] see http://www.equalizergraphics.com/documents/design/residentNodes.html
[12] see http://www.equalizergraphics.com/documents/design/standalone.html

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
eqBase::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_CUSTOM ) << "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 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 the 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 synchronizes 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 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 exising 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 5.5.1.

Figure 7 shows the execution of (hypothetical) rendering tasks without latency (Figure 7(a)) and with a latency of one frame (Figure 7(b)). With eqPly, a speedup of 15% has been observed on a five-node rendering cluster when using a latency of one frame instead of no latency[13]. A latency of one or two frames is normally not perceived by the user.

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:

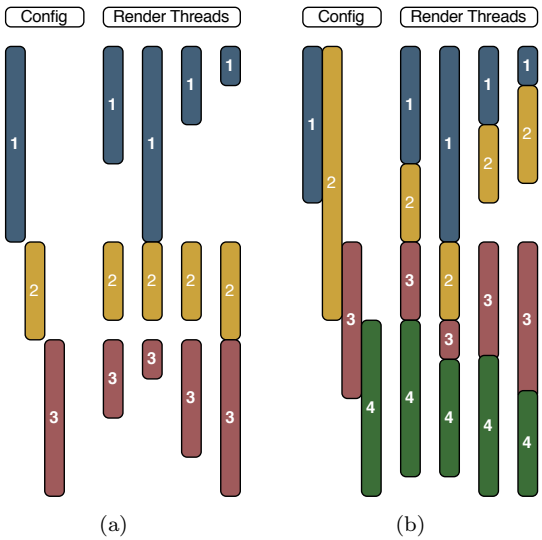


Figure 7: Synchronous and asynchronous execution

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

    clock.reset();
    while( config->isRunning( ) && maxFrames-- )
    {
        config->startFrame();
        // config->renderData(...);
        config->finishFrame();
    }
    const uint32_t frame = config->finishAllFrames();
    const float    time  = clock.getTimef();
    EQLOG( eq::LOG_CUSTOM ) << "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_CUSTOM ) << "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;
}
```

5.2.2 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

[13] http://www.equalizergraphics.com/scalability.html

is never executed. In order 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;
        EQINFO << "One configuration run successfully executed" << endl;
    }
    return true;
}
```

5.3 Distributed Objects

Equalizer provides distributed objects which help implementing data distribution in a cluster environment. The master version of a distributed object is registered with a eqNet::Session, which assigns a session-unique identifier to the object. Other nodes can map their instance of the object to this identifier, thus synchronizing the object's data with the remotely registered master version.

Distributed objects are created by subclassing from eqNet::Object. Distributed objects can be static (immutable) or dynamic. Dynamic objects are versioned.

The eqPly example has a static distributed object to provide initial data to all rendering nodes, as well as a versioned object to provide frame-specific data such as the camera position to the rendering methods.

5.3.1 Change Handling

The way changes are to be handled is determined 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 calling 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.

DELTA_UNBUFFERED The object is versioned, and delta data is used to update slave instances. No data is stored, and no previous versions can be mapped. The instance data is serialized whenever a new slave instance is mapped.

5.3.2 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 LocalInitData 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 either has to provide a pointer and size to its data using setInstanceData, or it has to implement getInstanceData and applyInstanceData. The first approach can be used if all distributed member variables are stored in one contiguous block of memory. The second approach is used otherwise.

The InitData class contains a string of variable length. Therefore it uses the second approach of manually serializing and de-serializing its data. Serialization in getInstanceData and de-serialization in applyInstanceData is performed by streaming all member variable to or from the provided data streams. Efficient buffering and data transport between nodes is implemented in the data streams:

```
void InitData::getInstanceData( eqNet::DataOStream& os )
{
    os << _frameDataID << _windowSystem << _useVBOs << _useGLSL << _filename;
}

void InitData::applyInstanceData( eqNet::DataIStream& is )
{
    is >> _frameDataID >> _windowSystem >> _useVBOs >> _useGLSL >> _filename;

    EQASSERT( _frameDataID != EQ_ID_INVALID );
    EQINFO << "New InitData instance" << endl;
}
```

The data input and output streams perform no type checking on the data. It is the application's responsibility to exactly match the order and types of variables during serialization and de-serialization.

5.3.3 FrameData - a Versioned Distributed Object

Versioned objects have to override getChangeType to indicate how they want to have changes to be handled. The current implementation has the following characteristics:

- Only the master instance of the object is writable, that is, eqNet::Object::commit can be called only on the master instance to generate a new version.
- Slave instance versions can only be advanced, that is, eqNet::Object::sync(version) with a version smaller than the current version will fail.
- Newly mapped slave instance are mapped to the oldest available version.

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 been committed or is still in transmission.

In addition to 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 the delta data happens to be layed out contiguously in memory, setDeltaData might be used. The default implementation of pack and unpack (de)serialize the delta data or the instance data if no delta data has been specified.

The eqPly frame data is layed out in one anonymous structure in memory. It also does not track changes since it is relatively small in size and changes frequently. Therefore, the instance and delta data are the same and set in the constructor. Furthermore, the change type is INSTANCE. The default Object implementation will take care of the distribution of the data:

```
FrameData()
    {
        reset();
        setInstanceData( &data, sizeof( Data ));
        EQINFO << "New FrameData " << std::endl;
    }
```

5.4 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.

5.4.1 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.

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 eqNet::Session used for rendering. Equalizer takes care of the session setup and exit in Client::chooseConfig and Client::releaseConfig, respectively.

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.

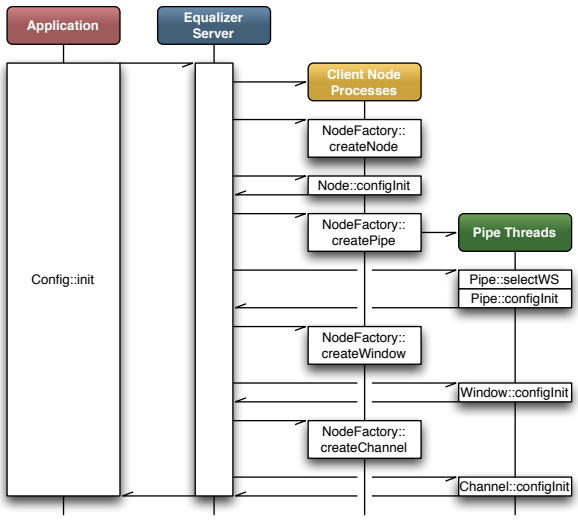


Figure 8: Config Initialization Sequence

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

```
bool Config::init()
{
    // init distributed objects
    _frameData.data.color = _initData.useColor();
    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 implementations[14], which can easily be integrated with Equalizer. The example code in eqPly provides a reference implementation for the integration of such a tracking library. Section 6.3 provides more background on head tracking.

```
    // init tracker
    if( !_initData.getTrackerPort().empty( ))
    {
        if( !_tracker.init( _initData.getTrackerPort() ))
            EQWARN << "Failed to initialise tracker" << endl;
        else
        {
            // Set up position of tracking system in world space
            // Note: this depends on the physical installation.
            vmml::Matrix4f m( vmml::Matrix4f::IDENTITY );
            m.scale( 1.f, 1.f, -1.f );
            //m.x = .5;
            _tracker.setWorldToEmitter( m );

            m = vmml::Matrix4f::IDENTITY;
            m.rotateZ( -M_PI_2 );
            _tracker.setSensorToObject( m );
            EQLOG( eq::LOG_CUSTOM ) << "Tracker initialised" << 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;
}
```

5.4.2 Frame Control

The rendering frames are issued by the application. The eqPly::Config only overrides startFrame in order 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 frustra[15].

The camera position is updated and the frame data is commited, 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:

[14] VRCO Trackd, VRPN, etc.

[15] see http://www.equalizergraphics.com/documents/design/immersive.html

```
uint32_t Config::startFrame()
{
    // update head position
    if( _tracker.isRunning() )
    {
        _tracker.update();
        const vmml::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 );
}
```

5.4.3 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.

For event-driven execution, the application can override Config::handleEvents to blockingly receive events using Config::nextEvent until a new frame has to be rendered.

The eqPly example continuously renders new frames. It 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 with respect to 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 );
            }
            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;
            }
            else if( event->pointerMotion.buttons == eq::PTR_BUTTON3 )
            {
                _frameData.data.translation.x +=
                    .0005f * event->pointerMotion.dx;
                _frameData.data.translation.y -=
                    .0005f * event->pointerMotion.dy;
```

```
            }
            return true;

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

5.5 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 should 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.

During node initialization the init data is mapped to a local instance using the identifier passed from Config::init. The model is loaded based on the filename in the initialization data. No pipe, window or channel tasks methods are executed before Node::configInit has returned.

```
bool Node::configInit( const uint32_t initID )
{
    eq::Config* config = getConfig();
    const bool mapped = config->mapObject( &_initData, initID );
    EQASSERT( mapped );

    const string& filename = _initData.getFilename();
    EQINFO << "Loading model " << filename << endl;

    _model = new Model();
    if ( !_model->readFromFile( filename.c_str() ) )
    {
        EQWARN << "Can't load model: " << filename << endl;
        delete _model;
        _model = 0;
    }

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

The node config exit function deletes the loaded model and unmaps the initialization data:

```
bool Node::configExit()
{
    delete _model;
    _model = NULL;

    eq::Config* config = getConfig();
    config->unmapObject( &_initData );

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

5.5.1 Frame Control

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 its children by calling startFrame, e.g., Node::frameStart has to call Node::startFrame in order 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.

Likewise, Config::finishFrame causes the invokation of the frameFinish task methods. These task methods unlock their parents by calling releaseFrame.

The explicit synchronization of child or parent resources allows the application to optimize the processing, by performing certain, independent operations when the child or parent resources are already unlocked.

The default thread synchronization synchronizes all Channel::frameDraw operations on a **single** node with the node's main thread. This facilitates porting, since the scene database does not have to be multi-buffered as all threads of the node process render the same frame.

The per-node frame synchronization does not break the asynchronous execution across rendering nodes. Advanced applications can remove per-node frame synchronization.

The per-node frame synchronization is achieved through the startFrame→waitFrameStarted and releaseFrameLocal→waitFrameLocal synchronization points. Note that this synchronization is only per-node, different nodes in the cluster still run asynchronously.

The first synchronization point ensures that the application (node) thread is done modifiying the data, typically during event processing. The pipe threads call Node::waitFrameStarted which blocks until the node calls startFrame in Node::frameStart.

The second pair ensures that after Config::finishFrame all pipe threads are done rendering the current frame, and therefore do not access shared data. At this point the application thread can safely modify scene data. The node's frameDrawFinish waits for all local pipes to release the synchronization by calling Pipe::releaseFrameLocal, which happens by default in Pipe::frameDrawFinish. The frameDrawFinish methods are called after all Channel::frameDraw of the corresponding thread have been executed.

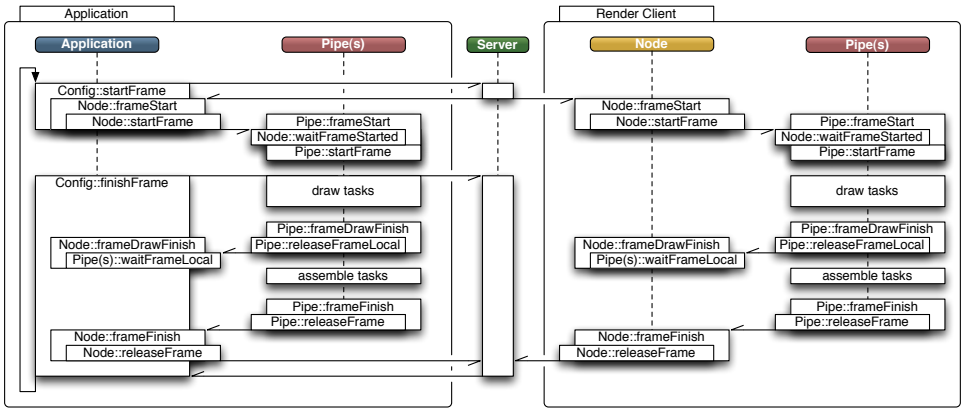


Figure 9: Synchronization of frame tasks

Figure 9 outlines the synchronization for the application, node and pipe classes. 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 implementation of the node task methods does take care of that.

Since eqPly multi-buffers all dynamic data, it can completely remove frame synchronization by:

- releasing the local synchronization early in Node::frameStart
- not waiting for the node to start the frame by not calling Node::waitFrameStarted in Pipe::frameStart
- not waiting for the pipe synchronization in Node::frameDrawFinish

Releasing the local synchronization early is done by calling releaseFrameLocal, and Node::frameDrawFinish is overwritten with an empty implementation:

```
void Node::frameDrawFinish( const uint32_t frameID,
                            const uint32_t frameNumber )
{ /* nop, see frameStart */ }

void Node::frameStart( const uint32_t frameID, const uint32_t frameNumber )
{
    startFrame( frameNumber ); // unlock pipe threads

    // Don't wait for pipes to release frame locally, sync not needed since all
    // dynamic data is multi-buffered
    releaseFrameLocal( frameNumber );
}
```

5.6 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.

5.6.1 Initialization and Exit

Pipe threads are not explicitely 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 eqPly example 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 Node*      node        = static_cast<Node*>( getNode( ));
    const InitData&  initData    = node->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();
}
```

5.6.2 Window System

Equalizer supports multiple window system interfaces, at the moment glX/X11, WGL and AGL/Carbon. Some operating systems, and therefore some Equalizer versions, support multiple window systems concurrently[16].

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 Node*            node     = static_cast<Node*>( getNode( ));
    const InitData&        initData = node->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;
}
```

5.6.3 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.

5.6.4 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

[16] see http://www.equalizergraphics.com/compatibility.html

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 )
{
    // don't wait for node to start frame, local sync not needed
    // node->waitFrameStarted( frameNumber );
    _frameData.sync( frameID );
    startFrame( frameNumber );
}
```

5.7 Window

The Equalizer window holds 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.

5.7.1 Initialization and Exit

The initialization sequence uses multiple, overrideable task methods. The main task method configInit executes a 'child' task method to create the drawable and context. The child task method depends on the window system of the pipe. The default implementations of configInitGLX, configInitWGL or configInitAGL create an on-screen window using OS-specific methods. If the OpenGL drawable and context were created 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.

Figure 10 shows a flow chart of the window initialization. The colored functions are task methods and can be replaced by application-specific implementations.

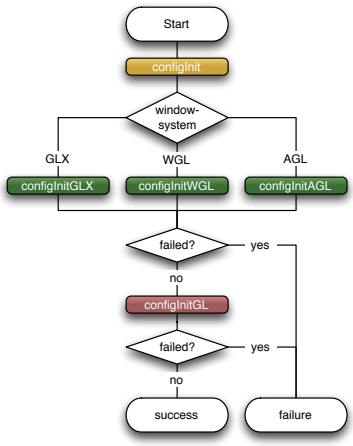


Figure 10: Window Initialization

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

The eqPly window initialization function first calls eq::Window::configInit to use the generic window setup. If that was successful, it initializes a state object and an overlay logo:

```
bool Window::configInit( const uint32_t initID )
{
    if( !eq::Window::configInit( initID ))
        return false;

    eq::Pipe*   pipe        = getPipe();
    Window*     firstWindow = static_cast< Window* >( pipe->getWindow( 0 ));
```

```
    EQASSERT( !_state );

    if( firstWindow == this )
    {
        _state = new VertexBufferState( getGLFunctions( ));
        _loadLogo();

        const Node*      node      = static_cast< const Node* >( getNode( ));
        const InitData& initData = node->getInitData();

        if( initData.useVBOs() )
        {
            const eq::GLFunctions* glFunctions = getGLFunctions();
            // Check if all VBO funcs available, else leave DISPLAY_LIST_MODE on
            if( glFunctions->hasGenBuffers() && glFunctions->hasBindBuffer() &&
                glFunctions->hasBufferData() && glFunctions->hasDeleteBuffers())
            {
                _state->setRenderMode( mesh::BUFFER_OBJECT_MODE );
                EQINFO << "VBO rendering enabled" << endl;
            }
            else
                EQWARN << "VBO function pointers missing, using display lists"
                       << endl;
        }
    }
    else
    {
        _state       = firstWindow->_state;
        _logoTexture = firstWindow->_logoTexture;
        _logoSize    = firstWindow->_logoSize;
    }

    if( !_state ) // happens if first window failed to initialize
        return false;

    return true;
}
```

The state object is used to handle the creation of OpenGL objects in a multipipe, multi-threaded execution environment. It uses an object manager, which is described in detail in Section 5.7.2. It is used in conjunction with a reference-counting smart-pointer here, since it is potentially 'owned' by multiple windows at the same time.

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 ommitted, since the code consists of standard OpenGL calls and is not Equalizer-specific.

The window exit function de-allocates all OpenGL objects when the state object is about to be disposed. The object manager does not delete the object in its destructor, since it does not know if an OpenGL context is still current. Additionally, eq::Window::configExit is called to destroy the drawable and context:

```
bool Window::configExit()
{
    if( _state.isValid() && _state->getRefCount() == 1 )
        _state->deleteAll();

    _state = 0;
    return eq::Window::configExit();
}
```

5.7.2 Object Manager

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

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

The object manager (OM) is a utility class and can be used to manage OpenGL objects across shared contexts. Typically one OM is used for each set of shared contexts and spawns all contexts of a single GPU[17].

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. In eqPly, a key of type const void * is used. The rendering code uses the address of the data item to be rendered as the key to obtain the associated OpenGL object.

All objects managed by the OM are reference counted. If an application releases the objects properly, they are automatically de-allocated. It is also possible to manually manage de-allocation of objects, which might be more convenient in some cases.

Currently, support for display lists, VBO's, textures and shaders is implemented. 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. Increases the reference count of existing objects.

newObject(key) allocates a new object for the given key. Returns FAILED if the object already exists or if the allocation failed. Sets the reference count of a newly created object to one.

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

releaseObject(key | name) decreases the reference count and deletes the object if the reference count reaches zero.

deleteObject(key | name) manually deletes the object. To be used if reference counting is not used.

5.8 Channel

The channel is the heart of the application in that it contains the actual rendering code. The channel is used to perform the various rendering operations for the compounds.

5.8.1 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;
}
```

[17] http://www.equalizergraphics.com/documents/design/objectManager.html

5.8.2 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 appropriatly, 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 in order 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 frustra 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, with the exception of the range which will be used later during rendering:

```
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 the frame data to ensure that all channels render a given frame using the same position.

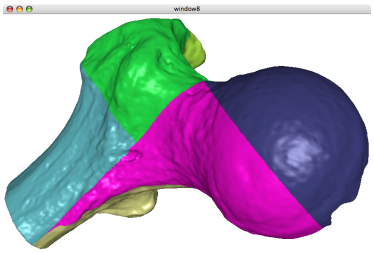


Figure 11: Destination view of an DB compound

Furthermore, a white color is set in case the model does not contain color information, or the color information is not used. In sort-last rendering, eqPly uses a different color for each channel to illustrate the database decomposition, as shown in Figure 11. The Equalizer channel provides a method to obtain a random, but unique color for all channels in the configuration. This color is determined by the server to ensure uniqueness across all channels of the configuration:

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

    glTranslatef( frameData.data.translation.x,
                  frameData.data.translation.y,
                  frameData.data.translation.z );
    glMultMatrixf( frameData.data.rotation.ml );

    Node*             node  = (Node*)getNode();
    const Model*      model = node->getModel();
    const eq::Range& range = getRange();

    if( !range.isFull( )) // Color DB-patches
    {
        const vmml::Vector3ub color = getUniqueColor();
        glColor3ub( color.r, color.g, color.b );
    }
    else if( !frameData.data.color || (model && !model->hasColors( )) )
    {
        glColor3f( 1.0f, 1.0f, 1.0f );
```

```
    }
```

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

```
    if( model )
    {
        _drawModel( model );
    }
    else
    {
        glColor3f( 1.f, 1.f, 0.f );
        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();
    }
}
```

In order 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::FrustumCullerf        culler;

    state.setColors( frameData.data.color &&
                     range.isFull() &&
                     model->hasColors() );
    _initFrustum( culler, model->getBoundingSphere( ));

    model->beginRendering( state );
```

The model data is spatially organized in an 3-dimensional kD-tree[18] for efficient view frustum culling. When the model is loaded by Node::configInit, it is preprocessed into the kD-tree and 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 use.

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 visiblity 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 12 shows a flow chart of the rendering algorithm, which performs efficient view frustum and range culling.

[18]See also http://en.wikipedia.org/wiki/Kd-tree

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

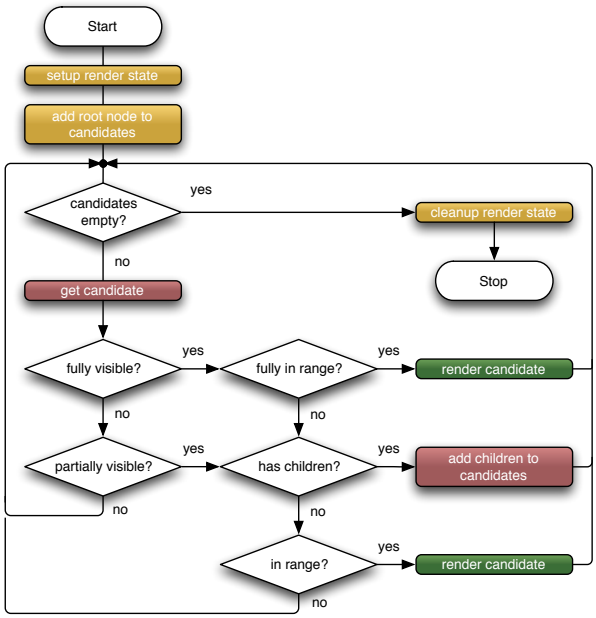


Figure 12: Main Rendering Loop

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 )
            {
                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 Advanced Features

This section discusses some additional important features not covered by the previous eqPly section. Where possible code examples from the Equalizer distribution are used.

6.1 Event Handling

Event handling requires a lot of flexibility. On one hand, the implementation differs slightly for each operating and window system due to conceptual differences in the OS-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 handling in glutMainLoop. More information on event handling can be found on the Equalizer website[19].

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

6.1.1 Threading

Where possible events are received and processed by a separate per-node event thread to allow asynchronous[20] event handling. Currently an event thread is only used by the X11/glX window system. WGL receives and processes the events from the pipe threads that created the windows. AGL receives the events from application or node main thread. Whenever the term **event thread** is used, it refers to the thread receiving the event, such as a per-node thread for glX, the pipe thread for WGL and the main thread for AGL.

6.1.2 Initialization and Exit

[19] see http://www.equalizergraphics.com/documents/design/eventHandling.html
[20] with respect to the rendering

During window and pipe initialization the event handling is set up. For both entities, initEvent-Handler is called to register the pipe or window with an event handler. This method may be overwritten to use a custom event handler, or to not install an event handler at all and so disable event handling. Likewise, exitEventHandler is called to de-initialize event handling.

An event handler consists of two parts: the generic base class providing the interface and generic functions, and the window-system-specific part providing the actual implementation.

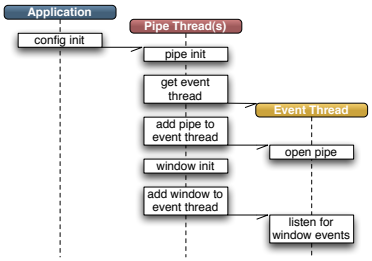


Figure 13: Event Handling Initialization

Event handling is initialized whenever a new window-system-specific pipe or window handle is set. First, exitEventHandler is called to de-initialize event handling for the old handle (if set), and then initEventHandler is called for the new handle.

AGL and glX use an event handler singleton, whereas WGL uses one event handler per window. Pipe event handling is only used for glX, where one Display connection is created and used to subscribe to window events.

6.1.3 Message Pump

For the WGL and AGL window systems, it is required to manually receive and dispatch ('pump') events. On WGL, this has to happen on each thread with windows, whereas on AGL it has to happen only on the main thread. By default, Equalizer pumps these events automatically for the application.

The methods Client::useMessagePump and Pipe::useMessagePump can be overridden to return false to disable this behaviour for their respective threads. On non-threaded pipes, Pipe::useMessagePump is not called.

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.

6.1.4 Event Data Flow

Events are received by an event handler. The event handler finds the eq::Window associated to the event. It then creates a generic WindowEvent, which holds important event data in a window system-independent format. The original event is attached to the generic window event.

The event handler then passes the window event to Window::processEvent, which is responsible for either handling the event locally, or for translating it into a generic ConfigEvent. The config events are send to the application thread using Config::sendEvent.

If the event was processed by processEvent, the function has to return true. If false is returned, the event will be passed to a previously installed, window-system-specific event handling function. The default implementation of Window::processEvent passes most events on to the application.

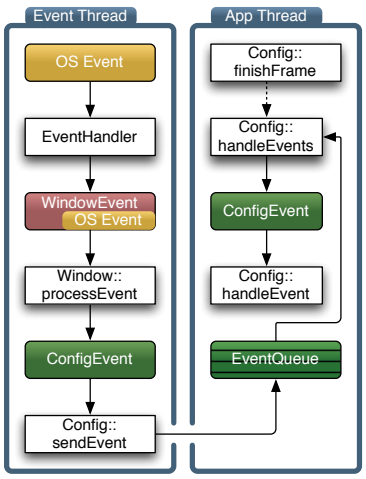


Figure 14: Event Processing

Events sent 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 be implemented.

Later Equalizer versions will introduce Pipe::processEvent and PipeEvent to communicate pipe-specific events, e.g., monitor resolution changes.

6.1.5 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];
    vmml::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 EQ_USER_EVENT_SIZE[21] 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 )
    {
```

[21] currently 32 bytes

```
        case ConfigEvent::READBACK:
        case ConfigEvent::ASSEMBLE:
            cout << static_cast< const ConfigEvent* >( event ) << endl;
            return true;

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

6.2 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, if needed.

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 provide a detailed description of the image compositing API in Equalizer.

6.2.1 Parallel Direct Send Compositing

In order to provide a motivation for the design of the image compositing API, the direct send parallel compositing algorithm is introduced in this section.

The main idea for 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.

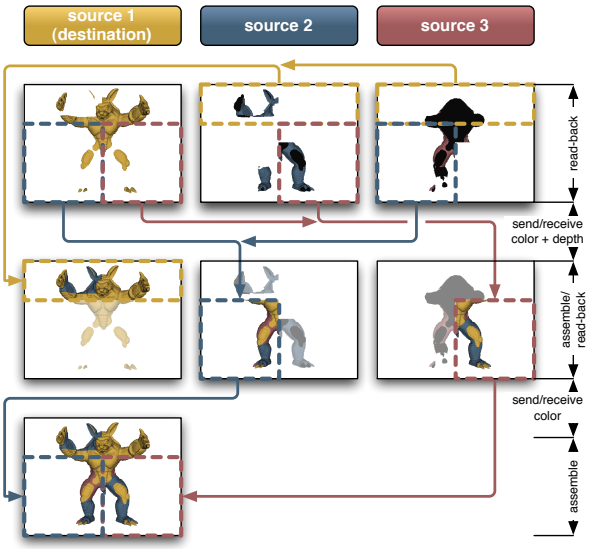


Figure 15: Parallel Direct Send Compositing

In direct send compositing, each rendering channel is also responsible for the sort-last composition[22] 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

[22]depth-based for polygonal data or sorted-blend for volume data

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 an 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 15 illustrates this algorithm for three channels. The Equalizer website contains a presentation explaining this algorithm[23].

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

- Selection of frame buffer attachments: color and/or depth
- 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.

6.2.2 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 signalled by the output frame after readback.

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

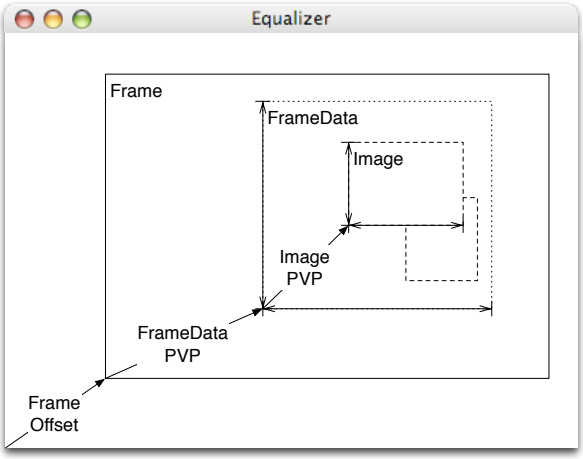


Figure 16: Hierarchy of assembly classes

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.

[23] http://www.equalizergraphics.com/documents/EGPGV07.pdf

The offset of input and output frames characterizes the position of the frame data with respect to the framebuffer, that is, the **window's** lower-left corner. For output frames this is the position of the channel with respect 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 16 illustrates the relationship between frames, frame data and images.

6.2.3 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 together to produce the final image, as shown in Figure 17[24].

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 this mode.

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.

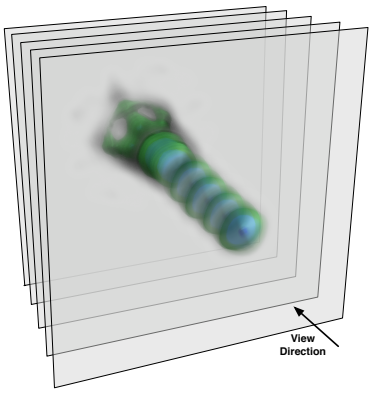


Figure 17: Blending Slices in 3D-Texture-based Volume Rendering

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 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 together 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

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

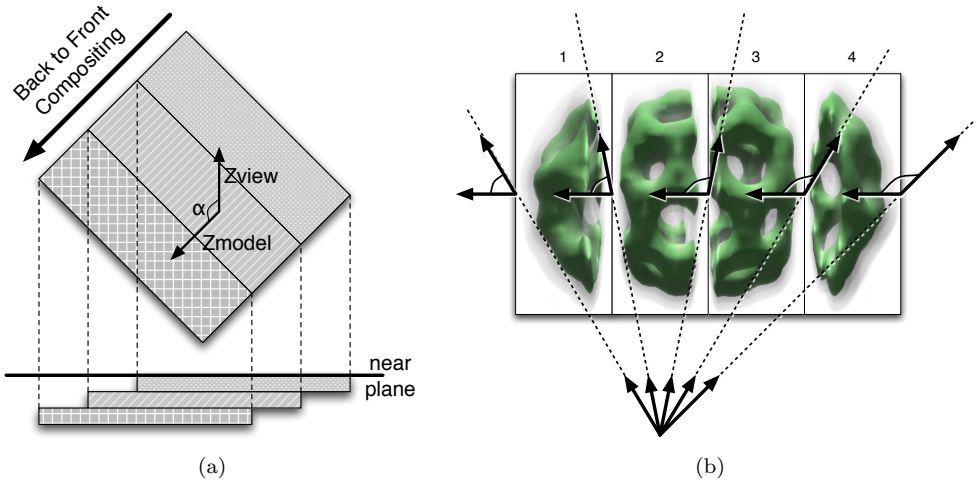


Figure 18: Back-to-Front Compositing for Orthogonal and Perspective View Frustra

buffer images, together with the sorting order, is used to arrange the frames during compositing. Figure 18(a) shows this composition for one view.

Finding the correct assembly order for perspective frustra is more complex. The perspective distortion invalidates a simple orientation criteria like the one used for orthographic frustra. For the view and frustum setup shown in Figure 18(b)[25] the correct compositing order is 4-3-1-2 or 1-4-3-2.

In order to compute the assembly order, eVolve uses the angle between the origin→slice vector and the near plane, as shown in Figure 18(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 19 shows the result of the composition from Figure 18(b).

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

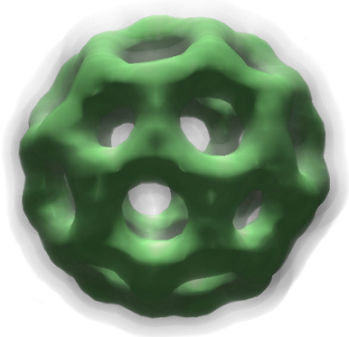


Figure 19: Result of Figure 18(b)

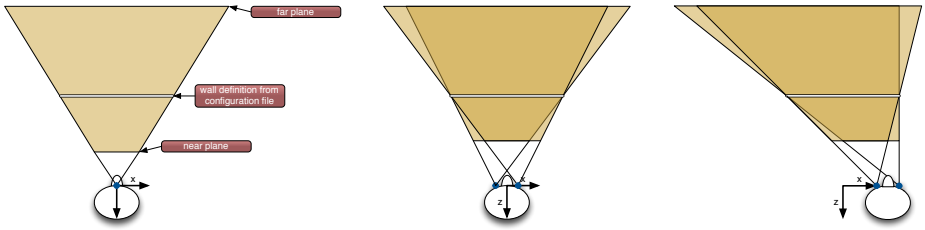
6.3 Head Tracking

The eqPly example contains rudimentary support for head tracking, in order 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.

Figure 20(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 frustra 'moved' by the distance between the eyes, as shown in Figure 20(b).

[25]Volume Data Set courtesy of: AVS, USA



(a) (b) (c)

Figure 20: Monoscopic, Stereoscopic and Tracked frustra

In immersive visualization, the observer is tracked in and the view frustra are adapted to the viewer's position and orientation, as shown in Figure 20(c). The transformation origin → viewer is set by the application using Config::setHeadMatrix, which is used by the server to compute the frustra. The resulting off-axis frustra are positioned using the channel's head transformation.