

Equalizer Programming Guide

Stefan Eilemann*

INCOMPLETE

Version 0.3, August 26, 2007



Version	Date	Changes
0.3	Aug 26, 2007	application and render client code described
0.2	Aug 20, 2007	main function described
0.1	Aug 19, 2007	outlined the basic concepts

<http://www.equalizergraphics.com/documents/Developer/ProgrammingGuide.pdf>

*eile@eyescale.ch

Contents

1	Introduction	1
2	Getting Started	1
2.1	Compiling and running eqPly	1
2.2	Equalizer Processes	1
2.2.1	The Server	1
2.2.2	The Application	1
2.2.3	The Render Client	1
3	The Programming Interface	1
3.1	Task Methods	2
3.2	The Resource Tree	2
3.2.1	Configuration	3
3.2.2	Node	3
3.2.3	Pipe	3
3.2.4	Window	3
3.2.5	Channel	3
3.3	Resource Usage	3
4	The eqPly polygonal renderer	4
4.1	The main Function	4
4.2	Application	5
4.2.1	Main Loop	5
4.2.2	Render Clients	7

1 Introduction

Equalizer provides a framework for the development of parallel OpenGL applications. Equalizer-based applications can run a single shared-memory system with multiple graphics cards (GPU's) or on a distributed graphics cluster. This Programming Guide introduces the programming interface using the eqPly example shipped with Equalizer.

Any questions related to Equalizer programming and this Programming Guide should be directed to the eq-dev mailing list¹.

2 Getting Started

2.1 Compiling and running eqPly

A prerequisite for this Programming Guide is a working eqPly example. The Quick-start Guide² explains how to run it. eqPly can also be executed without a server, which simplifies the development cycle. In this case it will be configured to use one window.

2.2 Equalizer Processes

2.2.1 The Server

An Equalizer server is responsible for managing one visualization system³. Currently it is only useful for running one application at a time, but it will be extended to support multiple applications concurrently and efficiently on one system. The server controls and potentially launches the application's rendering clients.

2.2.2 The Application

The application connects to a server, which chooses a configuration for the application. It provides a render client, to be launched by the server. The application reacts on events and controls the rendering.

2.2.3 The Render Client

The render client implements the rendering part of an application. It is passive, and receives all its rendering tasks from the server. The tasks are executed by calling the appropriate task methods (see 3.1).

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

The rendering client can be the same executable as the application, as is the case with eqPly. Real-world applications often implement a separate, light-weight rendering client.

3 The Programming Interface

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

¹see <http://www.equalizergraphics.com/lists.html>

²<http://www.equalizergraphics.com/documents/EqualizerGuide.html>

³a shared memory system or graphics cluster

application. It does not impose a scene graph or does interfere in any way with the application's rendering code.

3.1 Task Methods

The application subclasses Equalizer objects and overrides virtual functions to implement certain functionality, e.g., the application's OpenGL rendering in `eq::Channel::frameDraw`. These task methods are in concept similar to C function callbacks. The eqPly section will discuss the most important task methods. A full list of all task methods can be found on the website⁴.

3.2 The Resource Tree

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

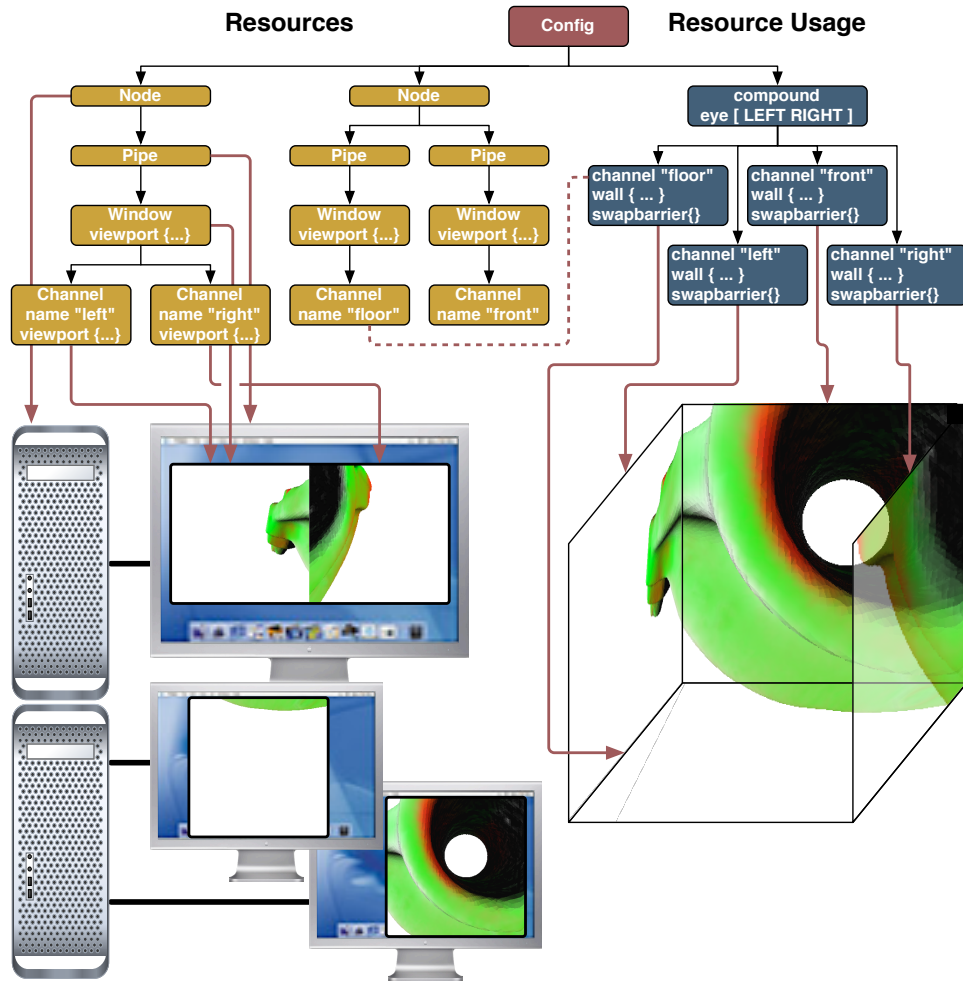


Figure 1: An example configuration

Figure 1 shows one example configuration for a four-side CAVETM, running on two machines (node) using three graphics cards (pipe) with one window each to render to the four output channels connected to the projectors for each of the walls.

⁴<http://www.equalizergraphics.com/documents/design/taskMethods.html>

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.

3.2.1 Configuration

The root of the resource tree is the `eq::Config`, which represents the current configuration of the application. It currently only holds the local node, not all nodes of the configuration.

3.2.2 Node

An `eq::Node` is the representation of a single computer in the system. It is one operating system process of the render client. All node task methods are executed from the main application thread.

3.2.3 Pipe

The `eq::Pipe` is the abstraction of a graphics card (GPU). In the current implementation it is also one operating system thread, unless the pipe's thread hint is set to false. All pipe and child window and channel task methods are executed from the pipe thread for threaded pipes or from the main application thread for non-threaded pipes⁵.

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

3.2.4 Window

An `eq::Window` is an drawable and OpenGL context. The drawable can be an on-screen window or an off-screen PBuffer or FBO⁶.

3.2.5 Channel

The `eq::Channel` is the abstraction of an OpenGL viewport within its parent window. It is the entity executing the actual rendering.

3.3 Resource Usage

How the rendering resources are to be used is configured using a compound tree. Each compound has a channel, which it uses to execute the rendering tasks. The rendering tasks are computed by the server and send to the render clients. At no point the application or render clients have or need knowledge of compounds. The configuration of compounds is not in the scope of this document⁷.

⁵see also <http://www.equalizergraphics.com/documents/design/nonthreaded.html>

⁶off-screen drawables are not yet implemented, but can be created by the application and used with Equalizer

⁷see <http://www.equalizergraphics.com/documents/design/compounds.html>

4 The eqPly polygonal renderer

The eqPly example is shipped with the Equalizer distribution and serves as a simple reference implementation of an Equalizer-based application. Its focus is not on rendering features or visual quality. It serves as a test bed for most of the Equalizer features.

In this section the source code of eqPly is discussed in detail, and relevant design decision and remarks are raised.

All classes in the example are in the eqPly namespace to avoid type name ambiguities, in particular for the Window class.

4.1 The main Function

The main function starts off with parsing the command line into the LocalInitData data structure, which in part will be distributed to all render client nodes. For actual command line parsing is done by the LocalInitData class and will be discussed there:

```
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 options⁸, e.g., the default server location. Furthermore, a node factory 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 for the 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 a 'plain' Equalizer object, thus making it possible to selectively subclass individual entity types. For each rendering resource used in the configuration, one C++ object will be created:

```
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 is an eqNet::Node. The underlying network distribution in Equalizer is a peer-to-peer network structure of eqNet::Nodes. The

⁸Equalizer-specific options always start with -eq-

application programmer rarely is aware of the classes in the `eqNet` namespace, but both the `eq::Client` and the server are `eqNet::Nodes`. The local initialization of nodes creates a local listening socket, so that the node, and therefore the `eq::Client` can communicate over the network with other nodes, such as the server and the rendering clients.

```
// 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 to run the `eqPly` application:

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

After it has finished, the application and `Eqply` is deinitialized and the `main` function returns:

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

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

4.2 Application

In the `eqPly` case, the application is also the render client. It has three run-time behaviours:

1. **Application:** The executable started by the user, which is 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.
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` and `-r`⁹.

4.2.1 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` try first to connect to a server on the local machine using the default port 4242. If that fails, it will create a server running within the applications process with a default 1-channel configuration¹⁰:

⁹see <http://www.equalizergraphics.com/documents/design/residentNodes.html>

¹⁰see <http://www.equalizergraphics.com/documents/design/standalone.html>

```

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;
    }
}

```

The second step is to ask the server for a configuration. The `ConfigParams` are a placeholder for later implementations to provide additional hints and information to the server for choosing a configuration. The configuration is created using `NodeFactory::createConfig`. Therefore it is of type `eqPly::Config`, but the return value is `eq::Config`, making `q` 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` does return after all nodes, pipes, windows and channels are initialized. It returns `true` only if all init task methods were successful. The `EQLOG` macro allows topic-specific logging. The numeric topic values are specified in the respective `log.h` header files:

```

    // 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 actual main loop is executed. The main loop runs until the user exits the configuration or a maximum number of frames, specified as a command-line argument, has been rendered. 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::endFrame`.

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, were they are executed in the order they arrive.

The frame finished is latency frames behind the frame started. The latency is specified in the configuration file, and allows several outstanding frames whose tasks are already queued in the node and pipe threads for execution. This allows overlapped execution and minimizes idle times.

When the main loop is finished, `Config::finishAllFrames` catches up with the latency. It returns after all outstanding frames have been rendered:

```
// 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 the 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;
}
```

4.2.2 Render Clients

In the second and third case, when the executable is used as a render client, `Client::initLocal` never returns. 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;
    }
```