

Equalizer Programming Guide

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

Stefan Eilemann*

INCOMPLETE

Version 0.5, September 3, 2007



Version	Date	Changes
0.5	Sep 3, 2007	added config and partly pipe
0.4	Aug 31, 2007	added distributed objects
0.3	Aug 26, 2007	added application and render client
0.2	Aug 20, 2007	added main function
0.1	Aug 19, 2007	outlined the basic concepts

*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	2
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	3
4.1	The main Function	4
4.2	Application	5
4.2.1	Main Loop	5
4.2.2	Render Clients	7
4.3	Distributed Objects	7
4.3.1	InitData - a Static Distributed Object	8
4.3.2	FrameData - a Versioned Distributed Object	8
4.4	Config	9
4.4.1	Config Initialization and Exit	9
4.4.2	Frame Control	10
4.4.3	Event Handling	11
4.5	Node	11
4.5.1	Frame Control	12
4.6	Pipe	13
4.6.1	Initialization and Exit	13
4.6.2	Window System	13
4.6.3	Frame Control	14

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

All functionality of Equalizer is accessed through the Equalizer client library, which implements all functionality discussed in this document.

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.

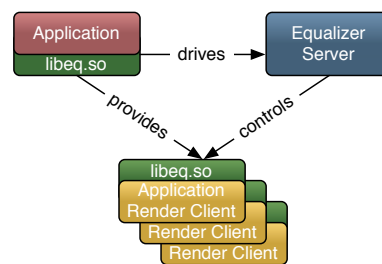


Figure 1: Equalizer Processes

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.

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

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

³a shared memory system or graphics cluster

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

Methods called by the application always have the form `verb[Noun]`, whereas methods called by Equalizer ('Task Methods') have the form `nounVerb`.

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 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 2 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. 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.

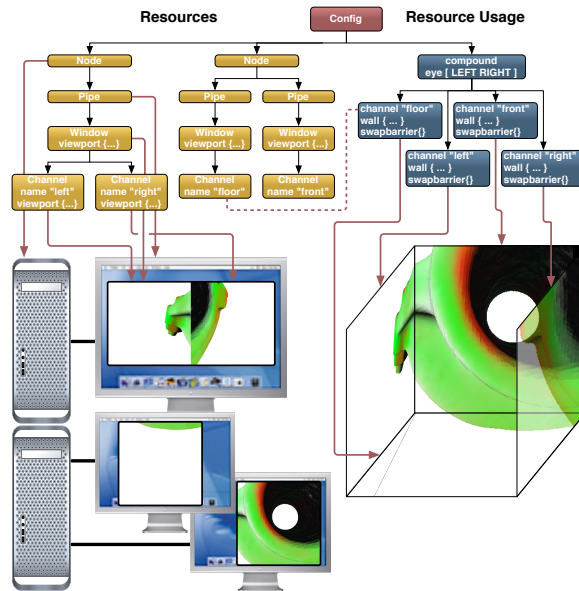


Figure 2: An example configuration

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

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. Each configuration might also use an application node, in which case the application process is also used for rendering.

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.

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.

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>

4.1 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 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:

⁸Equalizer-specific options always start with `-eq-`

```
// 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:

```
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 )
{
}
```

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

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

```

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:" << endl;
    << 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::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 the order they arrive.

`Config::finishFrame` synchronizes on the completion of the frame `current - latency`. The latency is specified in the configuration file, and allows several outstanding frames for which the tasks are already queued in the node and pipe threads for execution. This enables overlapped execution and minimizes idle times. The first latency `Config::finishFrame` return immediately, since they have no frame to synchronize upon. Figure 3 shows the execution of (hypothetical) rendering tasks without latency (3(a)) and with a latency of one (3(b)).

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

```

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

```

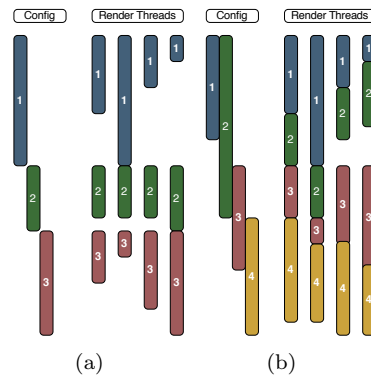


Figure 3: Synchronous and asynchronous execution


```

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. Therefore the application's main loop 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;
}

```

4.3 Distributed Objects

Equalizer provides distributed objects which help implementing data distribution in a graphics cluster. 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 object.

Distributed object 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.

4.3.1 InitData - a Static Distributed Object

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

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 can be outlayed 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 deserializing its data. The serialization is done in `getInstanceData`. The serialized data is cached using a (non-distributed) member variable, which is cleared by each setter of distributed data. Serializing the data is a simple matter of allocating a large enough memory buffer, and copying the data into the buffer:

```
const void* InitData::getInstanceData( uint64_t* size )
{
    *size = sizeof( uint32_t ) + sizeof( eq::WindowSystem ) +
           _filename.length() + 1;
    if( !_instanceData )
        return _instanceData;

    _instanceData = new char[ *size ];

    reinterpret_cast< uint32_t* >( _instanceData )[0] = _frameDataID;
    reinterpret_cast< uint32_t* >( _instanceData )[1] = _windowSystem;

    const char* string = _filename.c_str();
    memcpy( _instanceData + sizeof( uint64_t ), string, _filename.length()+1 );

    return _instanceData;
}
```

After the data is no longer needed, `releaseInstanceData` is called by Equalizer to allow the object to free the data. Since the data is cached for further usage, the release method is not overwritten.

The memory buffer returned by `getInstanceData` is transferred to the remote node during object mapping and passed to `applyInstanceData` for deserialization. The deserialization routine reads the data back into its own member variables:

```
void InitData::applyInstanceData( const void* data, const uint64_t size )
{
    EQASSERT( size > ( sizeof( _frameDataID ) + sizeof( _windowSystem ) ));

    _frameDataID = reinterpret_cast< const uint32_t* >( data )[0];
    _windowSystem = reinterpret_cast< const eq::WindowSystem* >( data )[1];
    _filename     = static_cast<const char*>( data ) + sizeof( uint64_t );

    EQASSERT( _frameDataID != EQ_ID_INVALID );

    EQINFO << "New_InitData_instance" << endl;
}
```

4.3.2 FrameData - a Versioned Distributed Object

Versioned objects have to override `isStatic` to return false, in order for Equalizer to know that they should be versioned. Right now, only the master instance of the object is writable, that is, `eqNet::Object::commit` can be called to generate a new 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 needed to map an object, versioned objects may implement `pack` and `unpack` to serialize or deserialize 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 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, for the instance and delta date are the same and set in the constructor:

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

4.4 Config

The configuration is driving the applications 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.

4.4.1 Config Initialization and Exit

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

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 is necessary to allow a speedy startup of the configuration on large-scale graphics clusters. On the other hand, it means that initialization functions are called even if the parent's initialization has failed.

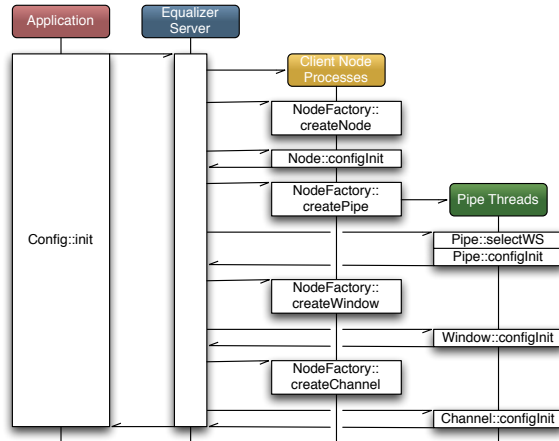


Figure 4: Config Initialization Sequence

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 first, since its identifier is transmitted using the initialization data, which is registered afterwards. The identifier of the initialization data is transmitted to the render client nodes using the `initID` parameter of `eq::Config::init`. Equalizer will pass this identifier to all `configInit` calls of the respective objects:

```

bool Config::init()
{
    // init distributed objects
    _frameData.data.color = _initData.useColor();
    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 device, as this is an orthogonal problem to parallel rendering, and has been solved already by a number of implementations¹¹. The example code in eqPly is one reference implementation for the integration of such a tracking library:

```

    // 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 of the configuration stops the render clients by calling `eq::Config::exit`, and then deregisters 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;
}

```

4.4.2 Frame Control

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

¹¹VRCO trackd, OpenVRPN, etc.

If a tracker is used, the current head position and orientation is retrieved and given to Equalizer, which uses the head matrix together with the wall or projection description to compute the view frustra¹².

The camera position is updated and the frame data is committed, which generates a new version. This version is passed to the rendering callbacks and will be used to synchronize the frame data to the state belonging to the current frame:

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

4.4.3 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:

```
bool Config::handleEvent( const eq::ConfigEvent* event )
{
    switch( event->type )
    {
        case eq::ConfigEvent::WINDOW_CLOSE:
            _running = false;
            return true;

        [...]

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

4.5 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 have only 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.

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

During node initialization the initialization data is mapped to a local instance using the passed identifier 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 = PlyFileIO::read( filename.c_str() );
    if( !_model )
        EQWARN << "Can't load_model:" << filename << endl;

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

The node config exit 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();
}
```

4.5.1 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 entity unlock all its children by calling `startFrame`, e.g., the `Node::frameStart` has to call `Node::startFrame` in order to unlock the pipe threads. Note that certain `startFrame` calls are currently empty, as the synchronization is implicit due to the threading model.

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

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

Figure 5 outlines the synchronization for the application, node and pipe classes. The window and channel synchronization is similar and omitted for simplicity. The `eqPly` example does not override `Node::frameStart` or `frameFinish`, but it is absolutely vital for the execution that `Node::startFrame` or `Node::finishFrame` are called, respectively. The default implementation of the node task methods does take care of that.

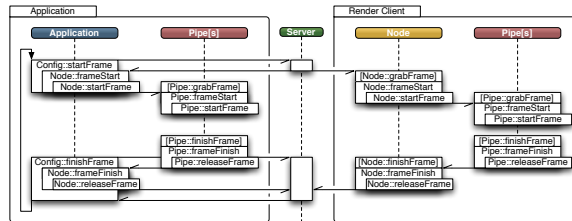


Figure 5: Synchronization of frame tasks

4.6 Pipe

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

4.6.1 Initialization and Exit

Pipe threads are not explicitly synchronized with each other, that is, pipes might be rendering different frames at one 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 GPU-specific initialization, e.g., the display connection is opened when `glX/X11` is used:

```
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 is again symmetric 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();
}
```

4.6.2 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 systems¹³.

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 using `supportsWindowSystem`.

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

```
eq::WindowSystem Pipe::selectWindowSystem() const
{
    const Node* node = static_cast<Node*>( getNode( ) );
    const InitData& initData = node->getInitData();
    const eq::WindowSystem ws = initData.getWindowSystem();
}
```

¹³see <http://www.equalizergraphics.com/compatibility.html>

```

    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;
}

```

4.6.3 Frame Control

TBD

```

void Pipe::frameStart( const uint32_t frameID, const uint32_t frameNumber )
{
    _frameData.sync( frameID );
    startFrame( frameNumber );
}

```