

# Parallel Graphics Programming with Equalizer



# Parallel Programming

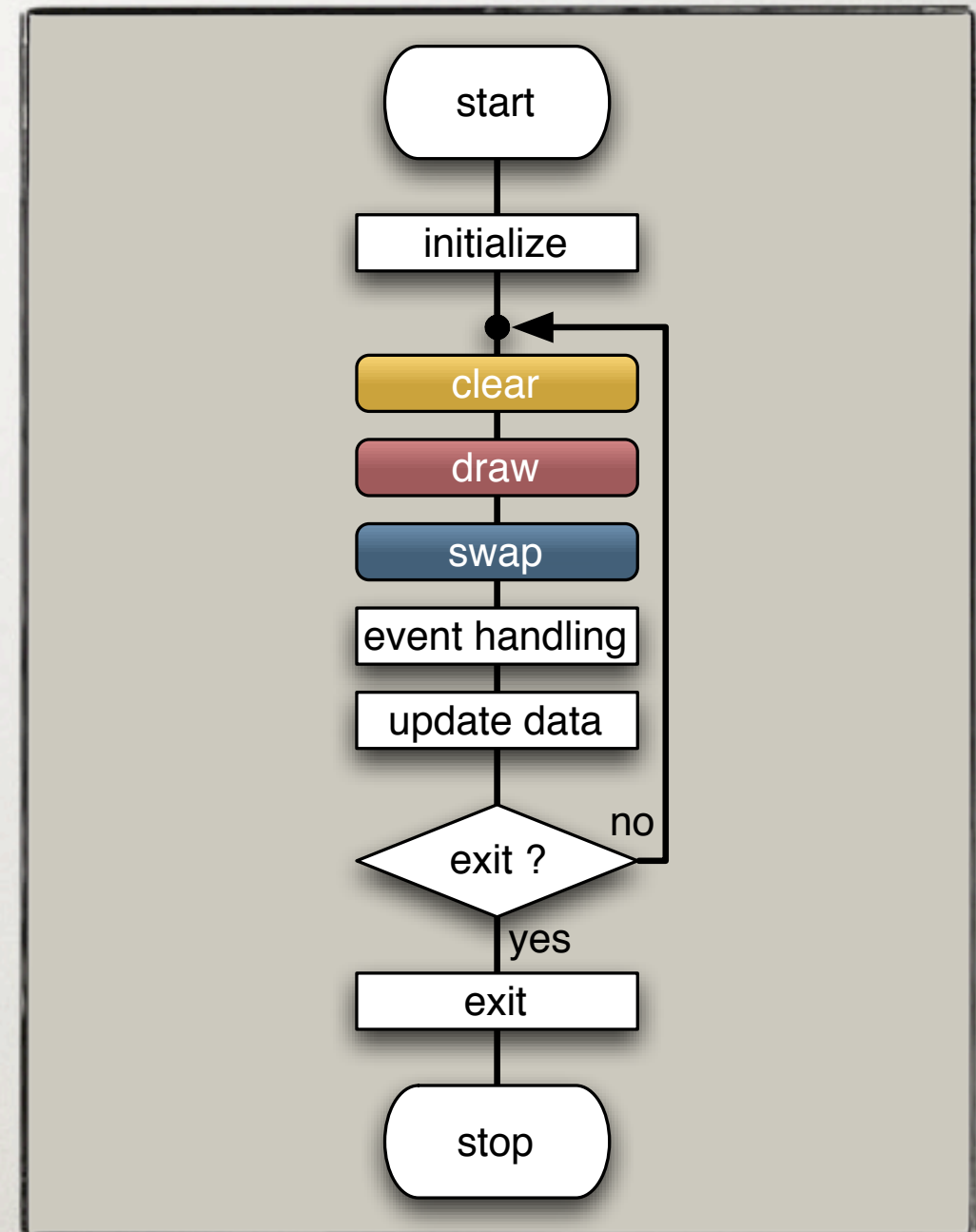
---

- Single pipe application
- Multipipe porting
- Equalizer programming
- Data distribution
- Porting details



# Single Pipe Application

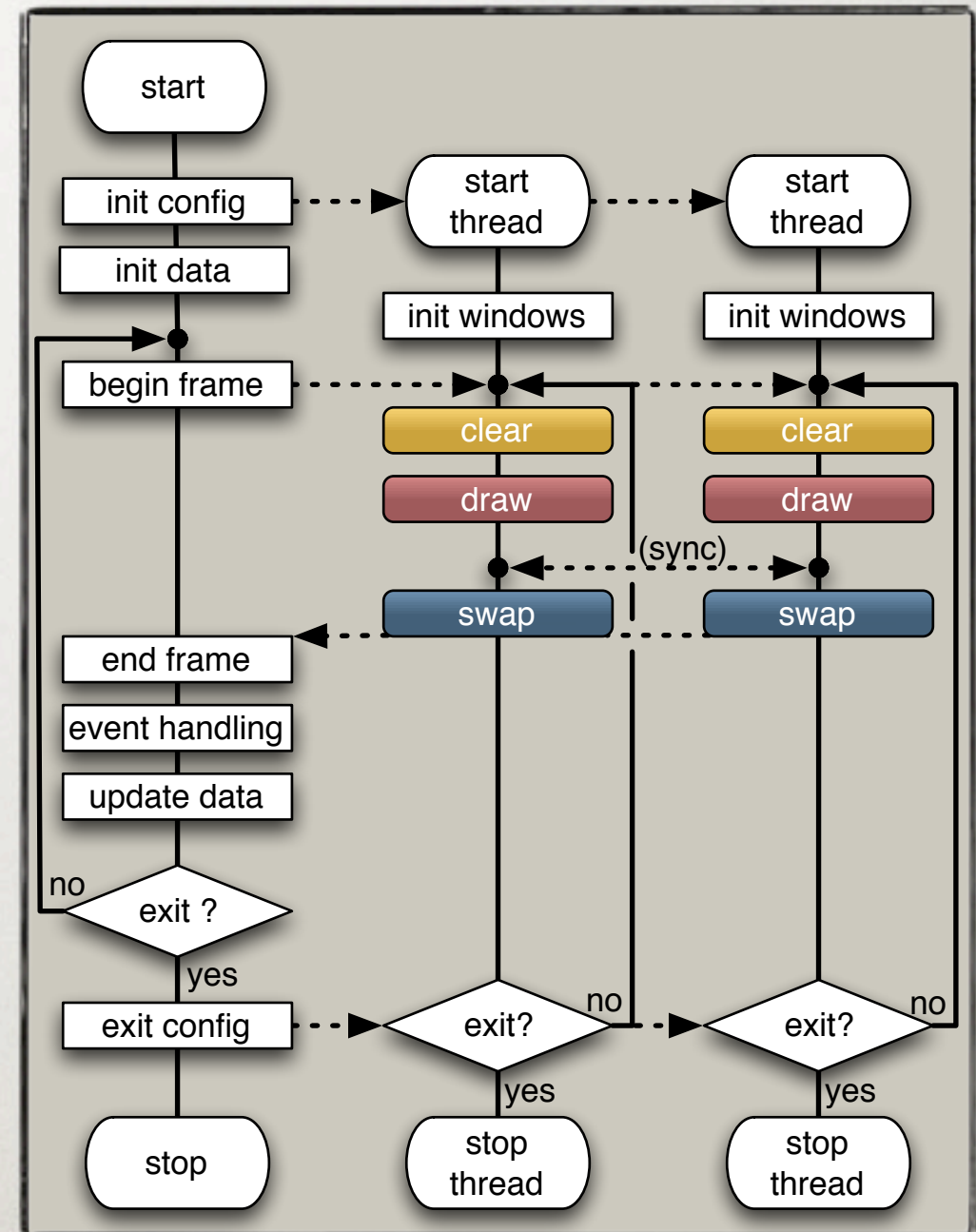
- One thread
- Typical rendering loop
- Stages may not be well separated

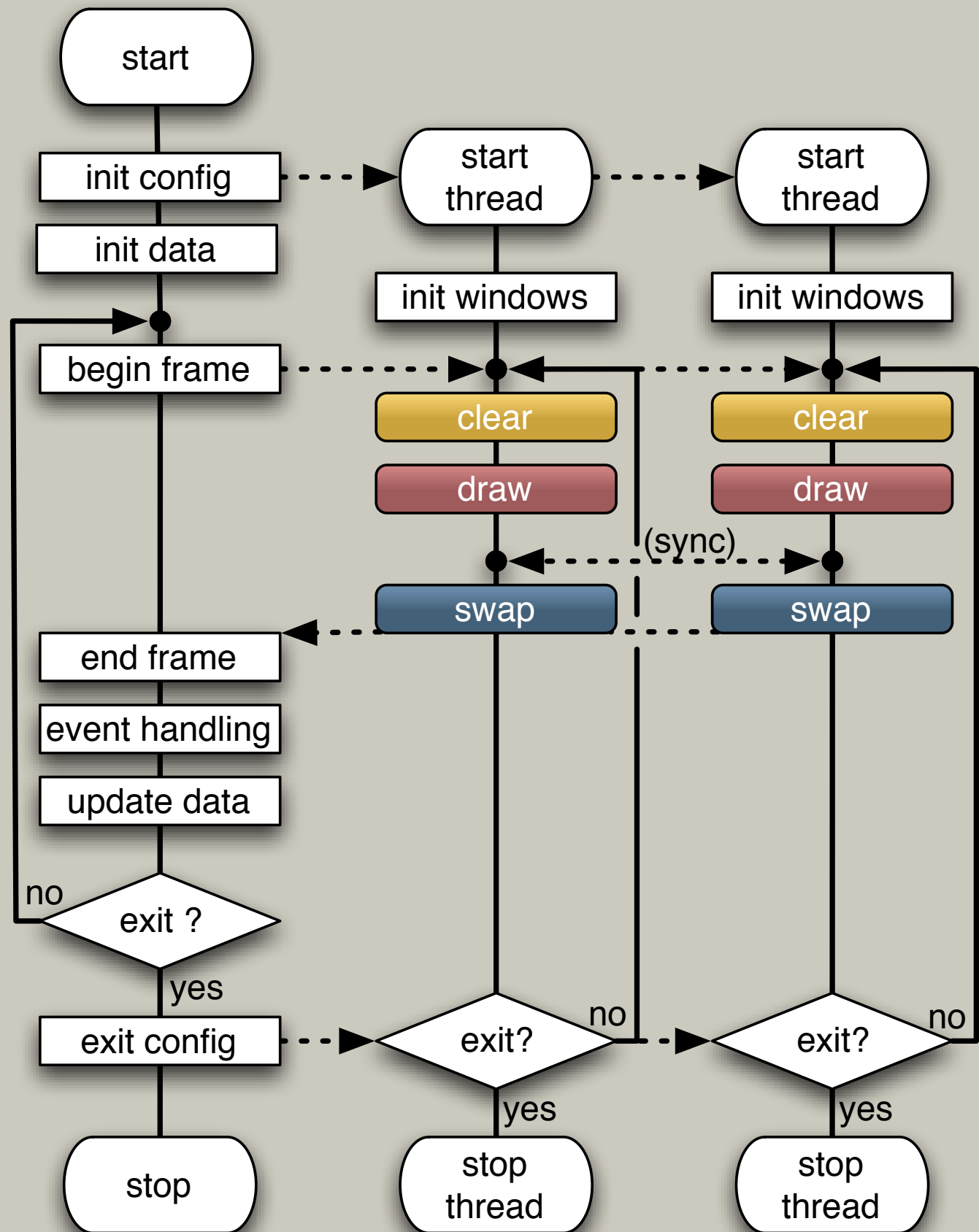
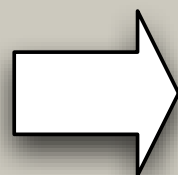
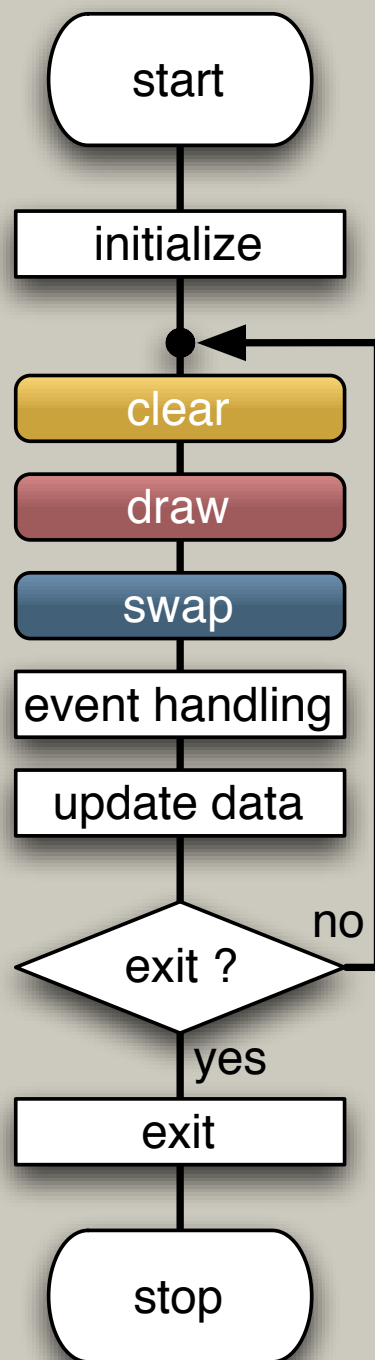




# Multipipe Rendering

- Separate rendering and application
- Instantiate rendering multiple times
- Synchronize parallel execution
- Optional: data distribution for clusters







# Equalizer Programming

---

Applications are written against a *client library* which abstracts the interface to the execution environment

- *Minimally invasive* programming approach
- Abstracts multi-processing, synchronization and data transport
- Supports distributed rendering and performs frame compositing



# Equalizer Programming

---

C++ classes which correspond to graphic entities:

- **Node** is a single computer in the cluster
- **Pipe** is a graphics card and rendering *thread*
- **Window** is an OpenGL drawable
- **Channel** is a viewport within a window

Classes are instantiated by Equalizer multiple times, based on the configuration.



# Equalizer Programming

---

Application subclasses and overrides task methods (“callbacks”), e.g.:

- **Channel::frameDraw** to render using OpenGL
- **Window::configInit** to init drawable and OpenGL
- **Pipe::frameStart** to update frame-specific data
- **Node::configInit** to initialize per-node data

Default methods implement typical use case!



# Example: eqPly

---

- Init
- Main loop
- Config
- Node
- Pipe
- Channel



# eqPly Init

---

```
RefPtr<eq::Server> server = new eq::Server;  
connectServer( server );  
Config* config = server->chooseConfig( configParams );  
config->registerObject( &_initData );  
config->registerObject( &_frameData );  
config->init( _initData.getID( ) );
```

- Server chooses config
- Server launches rendering nodes on Config::init
- Init and frame data are distributed objects
- Init data id is passed to all configInit methods



# eqPly Main Loop

---

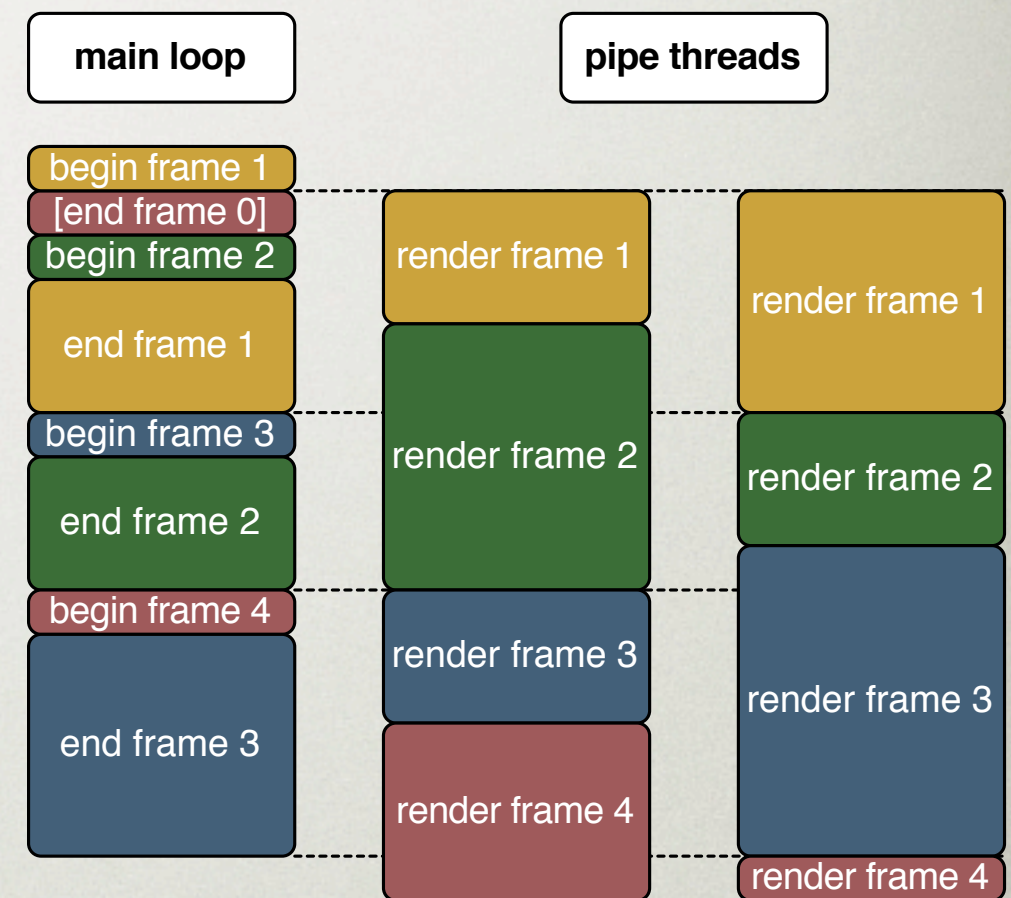
```
while( config->isRunning( ))  
{  
    // update _frameData based on events  
    const uint32_t version = _frameData.commit();  
    config->startFrame( version );  
    config->finishFrame();  
}
```

- `_frameData` is a versioned, distributed object
- Per-frame version is passed to all task methods
- Start frame N, finish frame N-latency
- Processes events within `finishFrame()`



# eqPly Main Loop

- Asynchronous execution
- Per-config latency
- Zero latency enforces synchronous execution
- Minimizes idle times
- Frame-specific data is maintained per pipe





# eqPly Node

---

```
bool Node::configInit( const uint32_t initID )
{
    config->mapObject( &_initData, initID );
    _model = PlyFileIO::read( _initData.getFilename( ) );
    return eq::Node::configInit( initID );
}
```

- `_initData` is a static distributed object
- `initID` was passed to `eq::Config::init()`
- `_model` is static at runtime: one instance per node



# eqPly Pipe

---

```
bool Pipe::configInit( const uint32_t initID )
{
    const InitData& initData = node->getInitData();
    uint32_t frameDataID = initData.getFrameDataID();
    config->mapObject( &_frameData, frameDataID );
    return eq::Pipe::configInit( initID );
}
```

- Maps one FrameData instance per pipe (thread)
- FrameData contains frame-specific information, like camera position
- eq::Pipe::configInit does GPU initialization



# eqPly Pipe

---

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

- Synchronizes \_frameData to frame-specific version (from Config::startFrame)
- startFrame unlocks child resources



# eqPly Channel

---

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

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

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity();
    applyHeadTransform();
    // render using OpenGL
}

...
```



# eqPly Channel

---

...

- apply methods are convenience for 'get; glFunc'
- Buffer: left / right / back buffer for stereo
- Viewport: restrict 2D viewport in window
- Frustum: glFrustum() parameters
- Head Transform: off-axis frustum transformation (head tracking)
- Optional: use getRange() for sort-last DB range



# Distributed Objects

---

- Static and versioned distributed objects
- Versioned objects work like a simplified version control system (cvs)
  - Multi-Buffering of data
  - One master instance commits new versions
  - Slave instances sync to version
  - Versions are typically frame-specific
- See eqPly **InitData** and **FrameData**



# Distributed Objects

---

- How to create distributed objects:
  - Subclass from `eq::net::Object`
  - Use `setInstanceData`, `setDeltaData` or implement your own pack/unpack routines
- How to initialize distributed objects:
  - Master: `Config::registerObject()` assigns id
  - Slaves: `Config::mapObject()` uses id to map to master instance



# Distributed Objects

---

- How to use versioned objects:
  - Master: create and get new version using `Object::commit()`
  - Slave: get version using `Object::sync( version ) [blocking]`
- See also:  
<http://www.equalizergraphics.com/documents/design/objects.html>



# Last Words

---

- LGPL license: commercial use welcome
- Open standard for scalable graphics
- Minimally invasive: easy porting
- Clusters and shared memory systems
- Linux, Windows, Mac OS X
- More on: [www.equalizergraphics.com](http://www.equalizergraphics.com)