

Parallel Graphics Programming with Equalizer

Outline

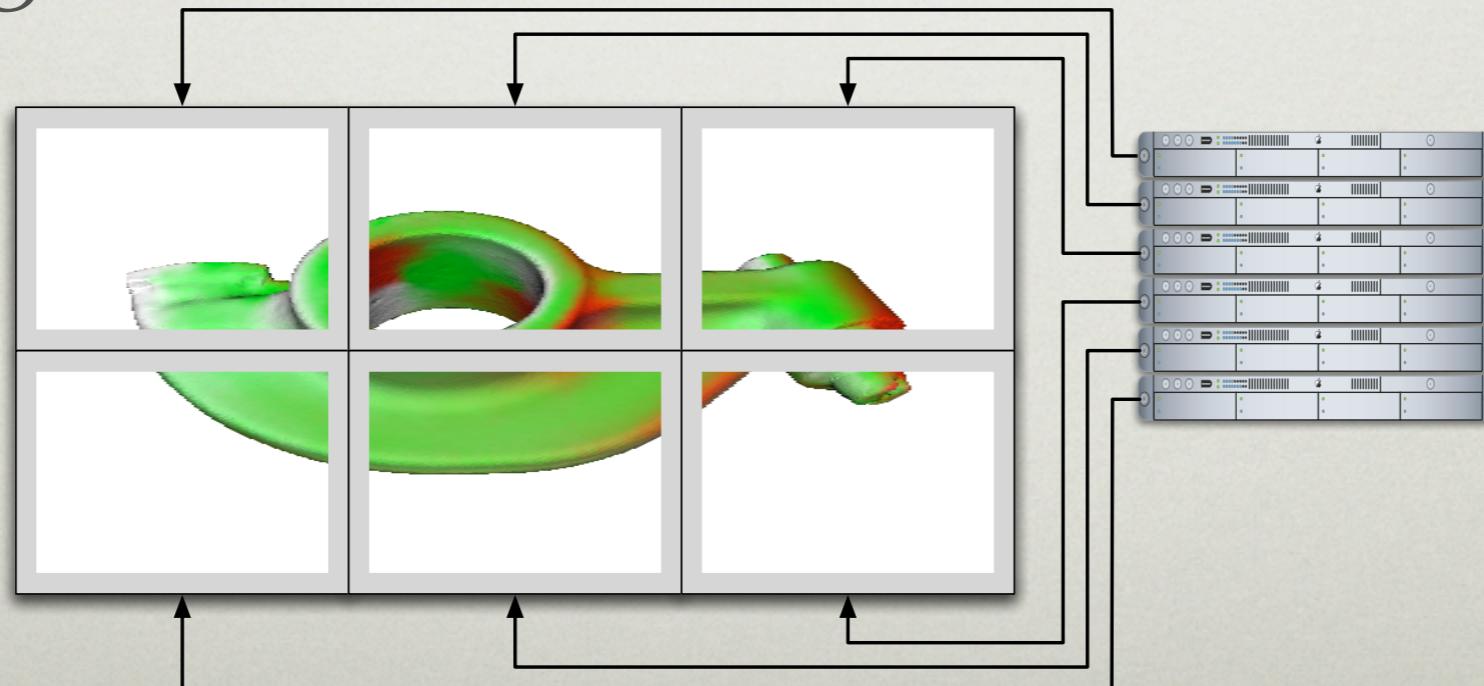
- Application Environments
- Scalable Rendering
- Multipipe Programming with Equalizer

Environment

- Display Walls
- Virtual Reality
- Remote Rendering
- Scalable Rendering

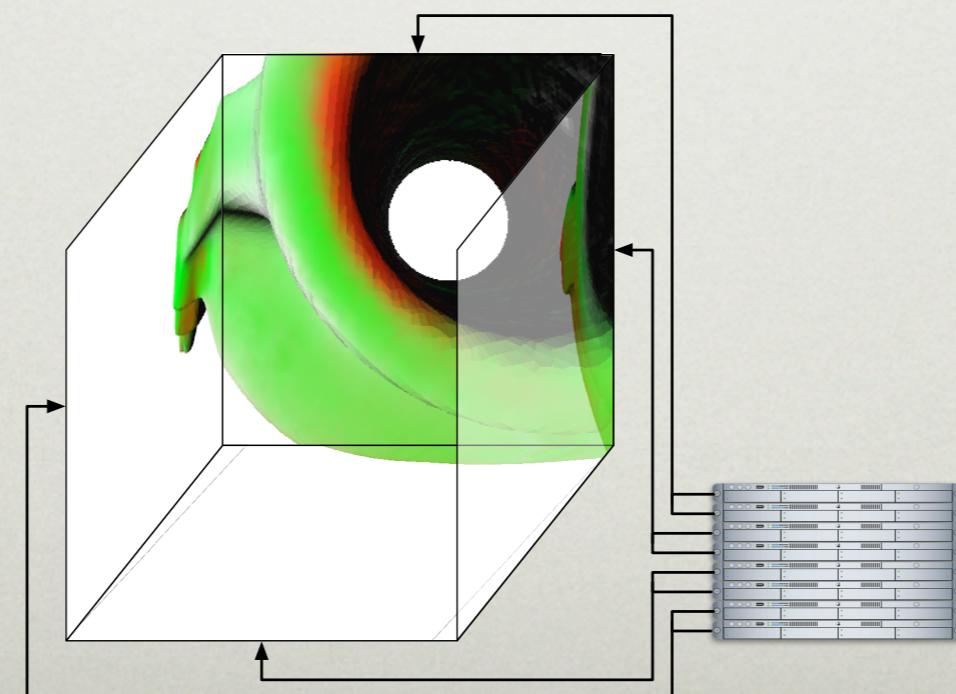
Display Walls

- Group collaboration
- Better data understanding
- One to four displays per computer
- High resolution: 10-100 MPixels



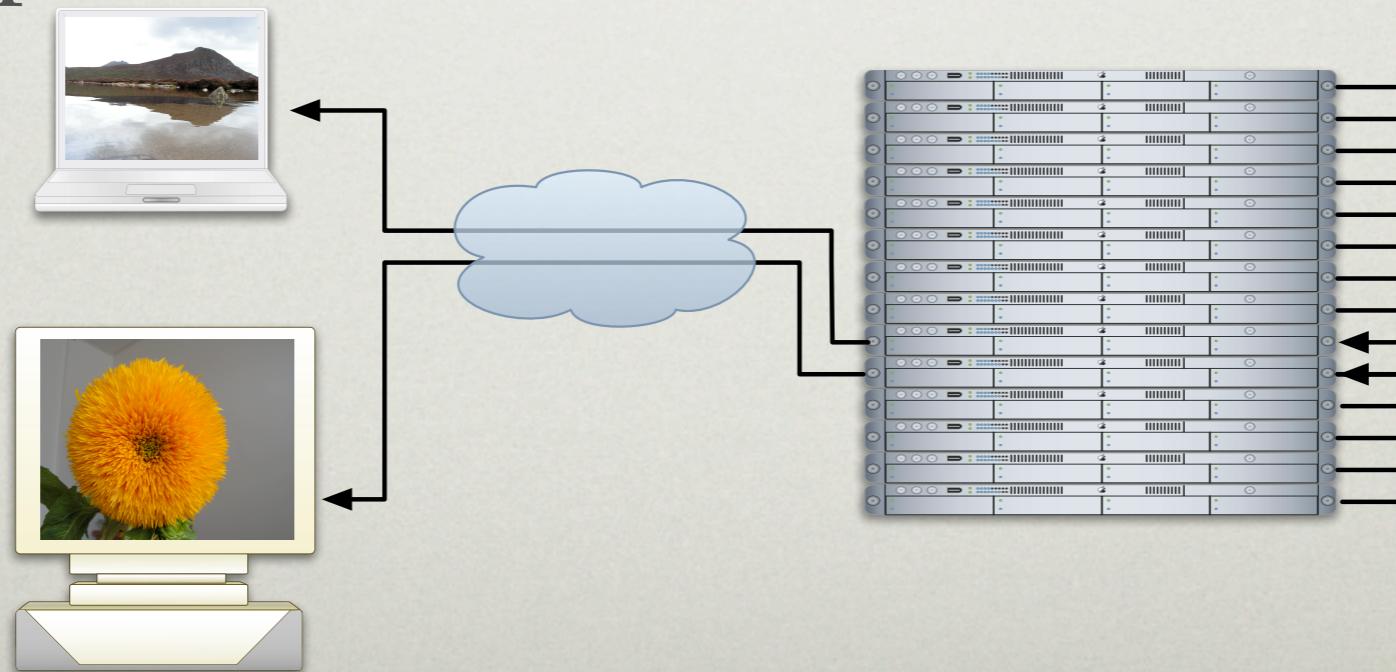
Virtual Reality

- Stereo rendering, head tracking
- High frame rates
- Up to two computers per wall with passive stereo



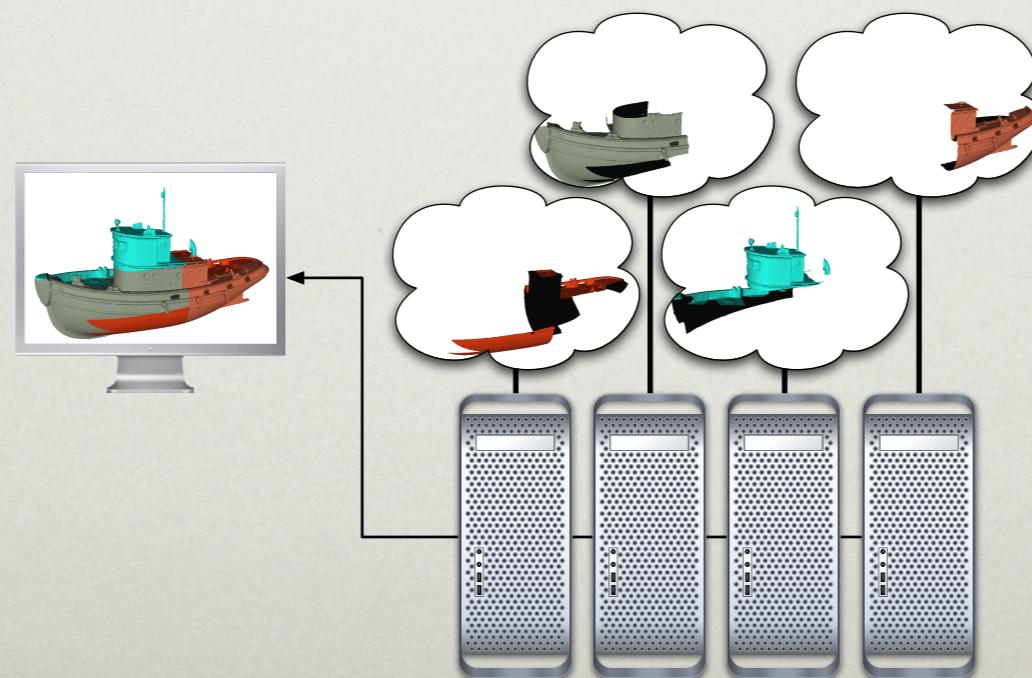
Remote Rendering

- Centralize data, software and hardware
- Combined with scalable rendering
- Avoids copying of HPC result data
- Simplifies administration



Scalable Rendering

- Render massive data sets faster
- Use multiple graphics cards and processors per display
- Different algorithms for parallelization

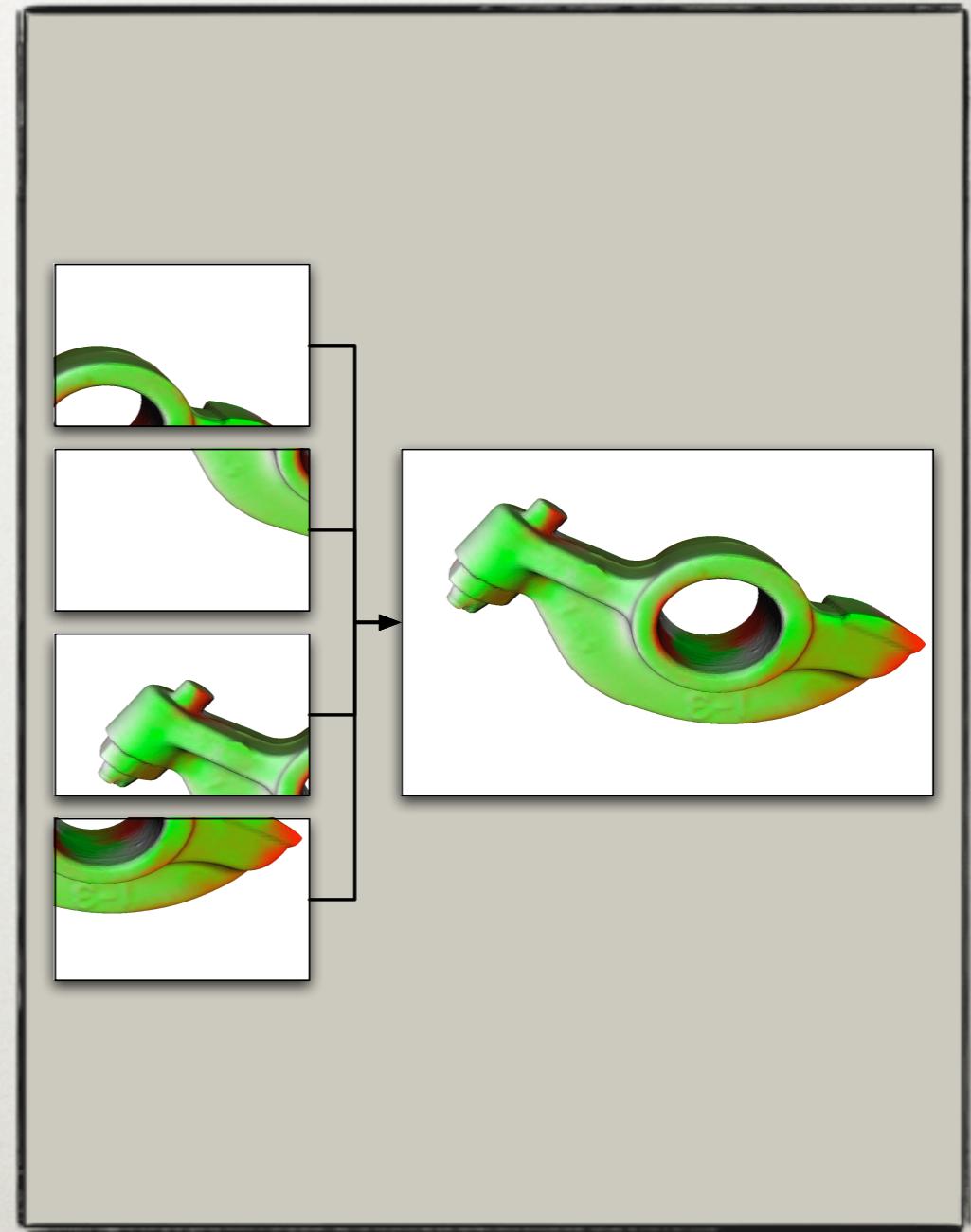


Scalable Rendering

- Single frame decomposition
 - sort-first: screen-space partition
 - sort-middle: only practical on GPU
 - sort-last: database partition
- Entire frame decomposition
 - DPlex: time-multiplex
 - Eye: stereo passes

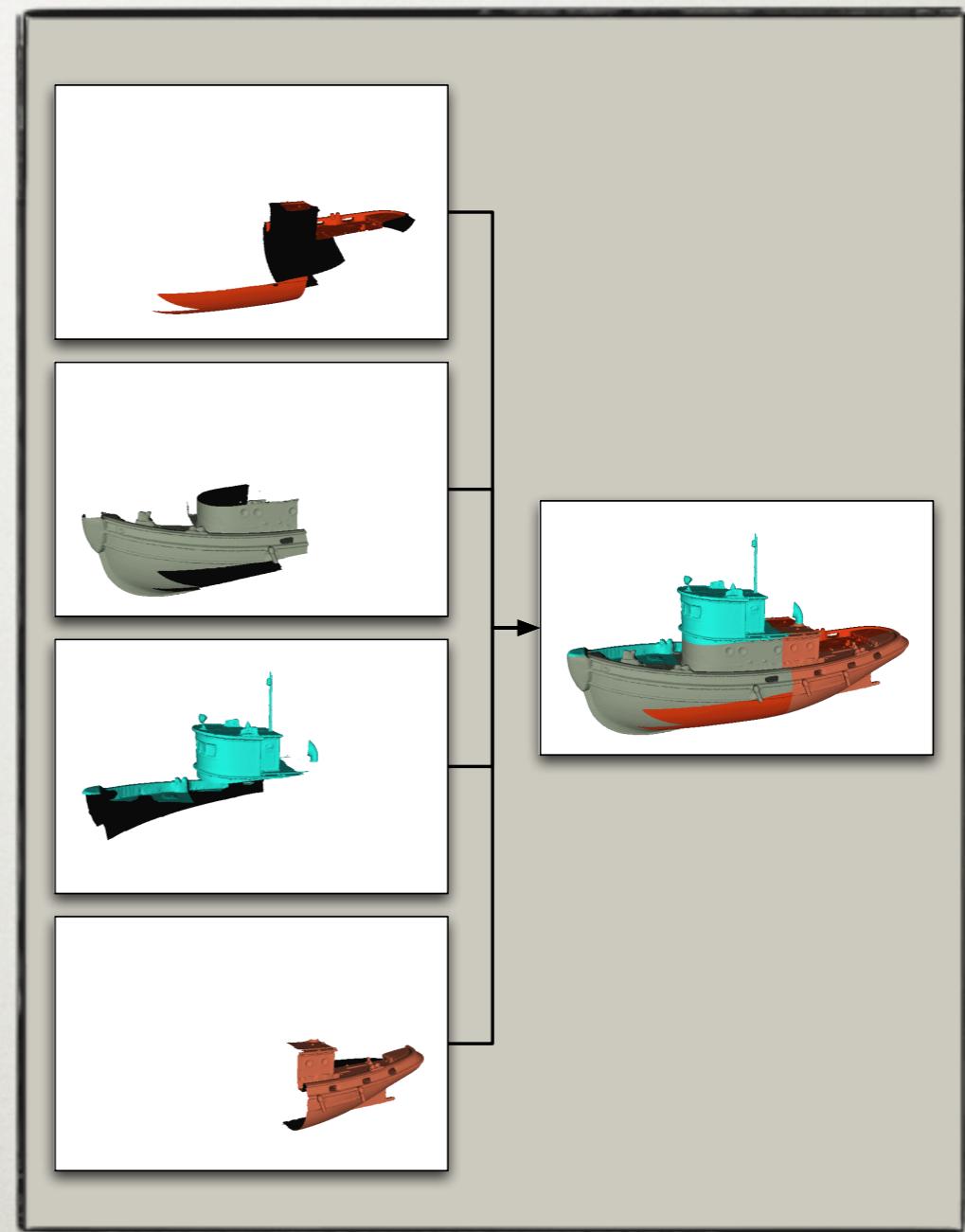
2D / Sort-First

- Scales fillrate / fragment processing
- Scales vertex processing with view frustum culling
- Parallel overhead due to primitive overlap limits scalability



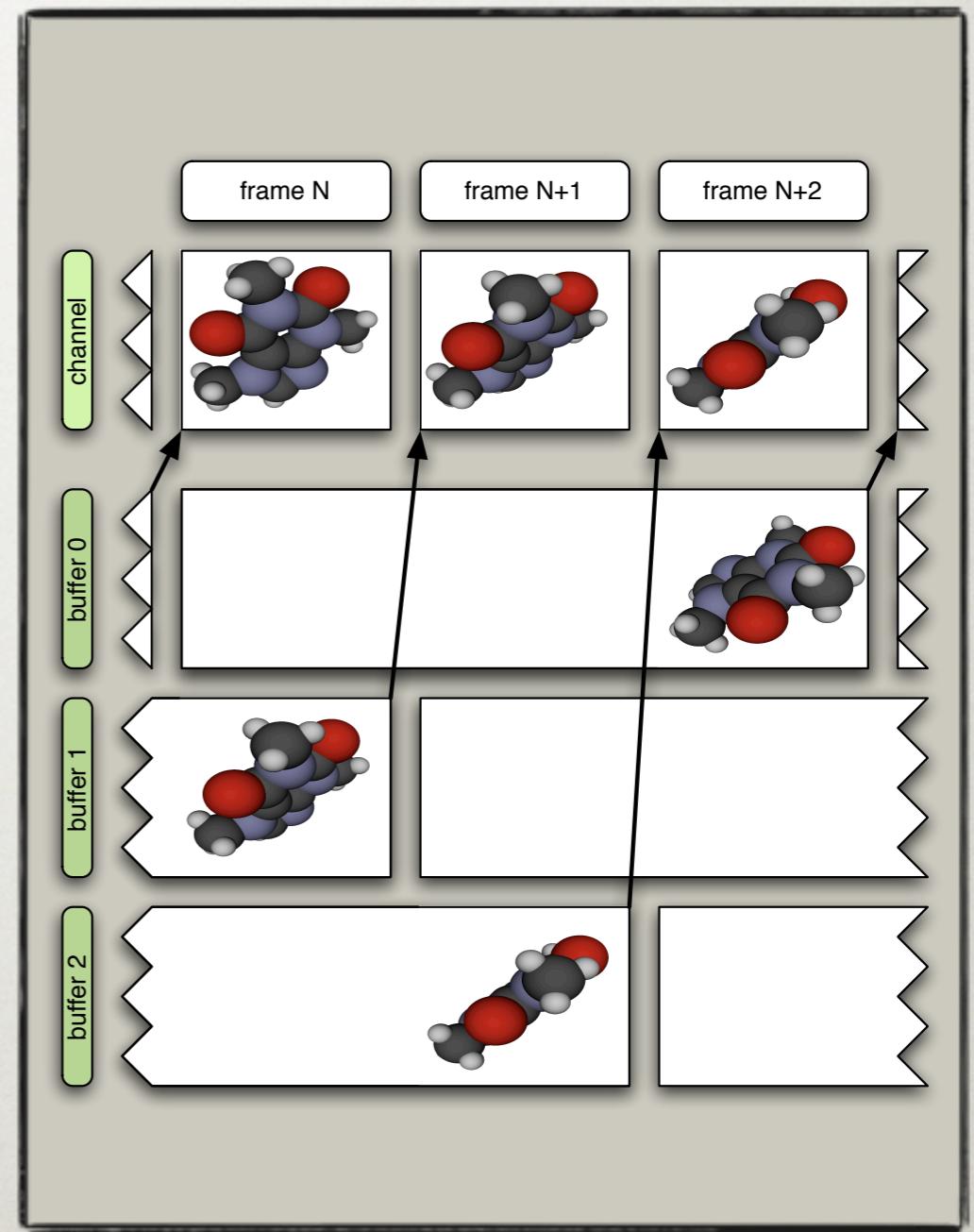
DB/Sort-Last

- Scales all aspects of rendering pipeline
- Application needs to be adapted to render subrange of data
- Recomposition relatively expensive



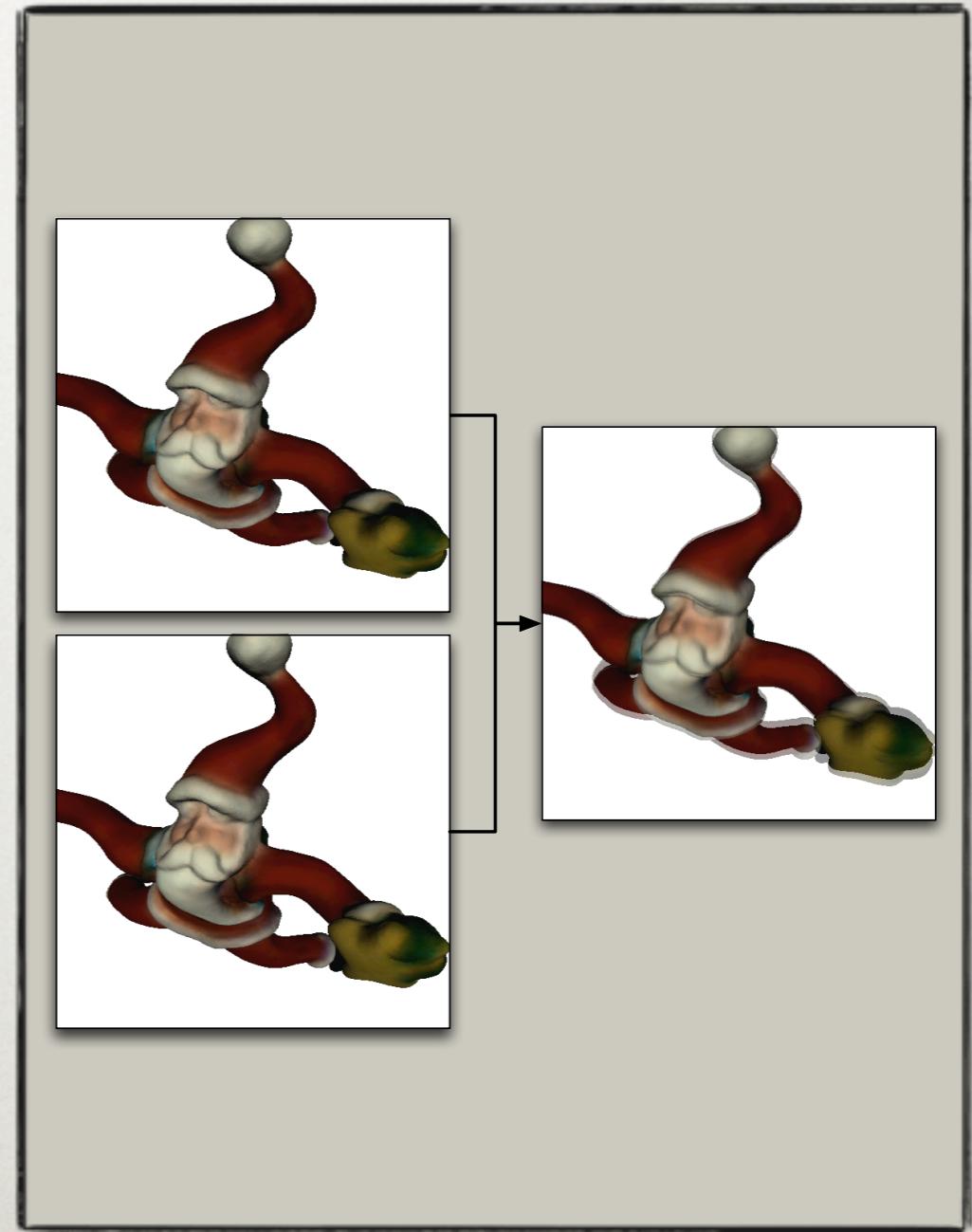
DPlex / Time-Multiplex

- Good scalability and loadbalancing
- Increased latency may be an issue



Eye/Stereo

- Stereo rendering
- Excellent loadbalancing
- Limited by number of eye views



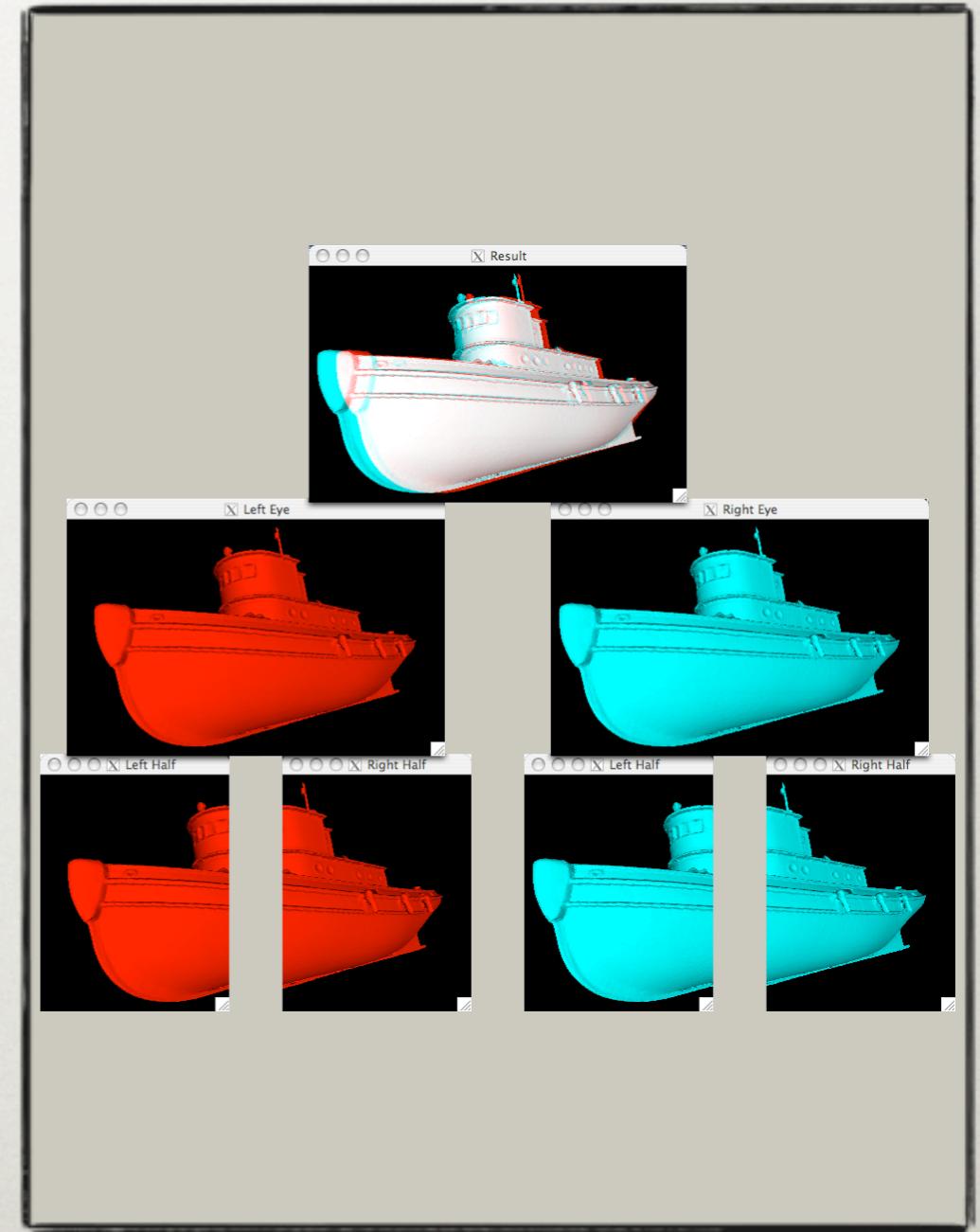
Conclusion

- No ‘magic bullet’
- 2D is ideal for up to eight pipes
- Use Eye if running in stereo
- DB scales best
- Combine modes

	2D	DB	DPlex	Eye
Fillrate	++	++	++	++
Vertex Processing	0	++	++	++
Memory Usage	0	++	0	0
Load Balancing	0	+	++	++
Latency	++	++	-	++
Re-assembly	+	-	+	+

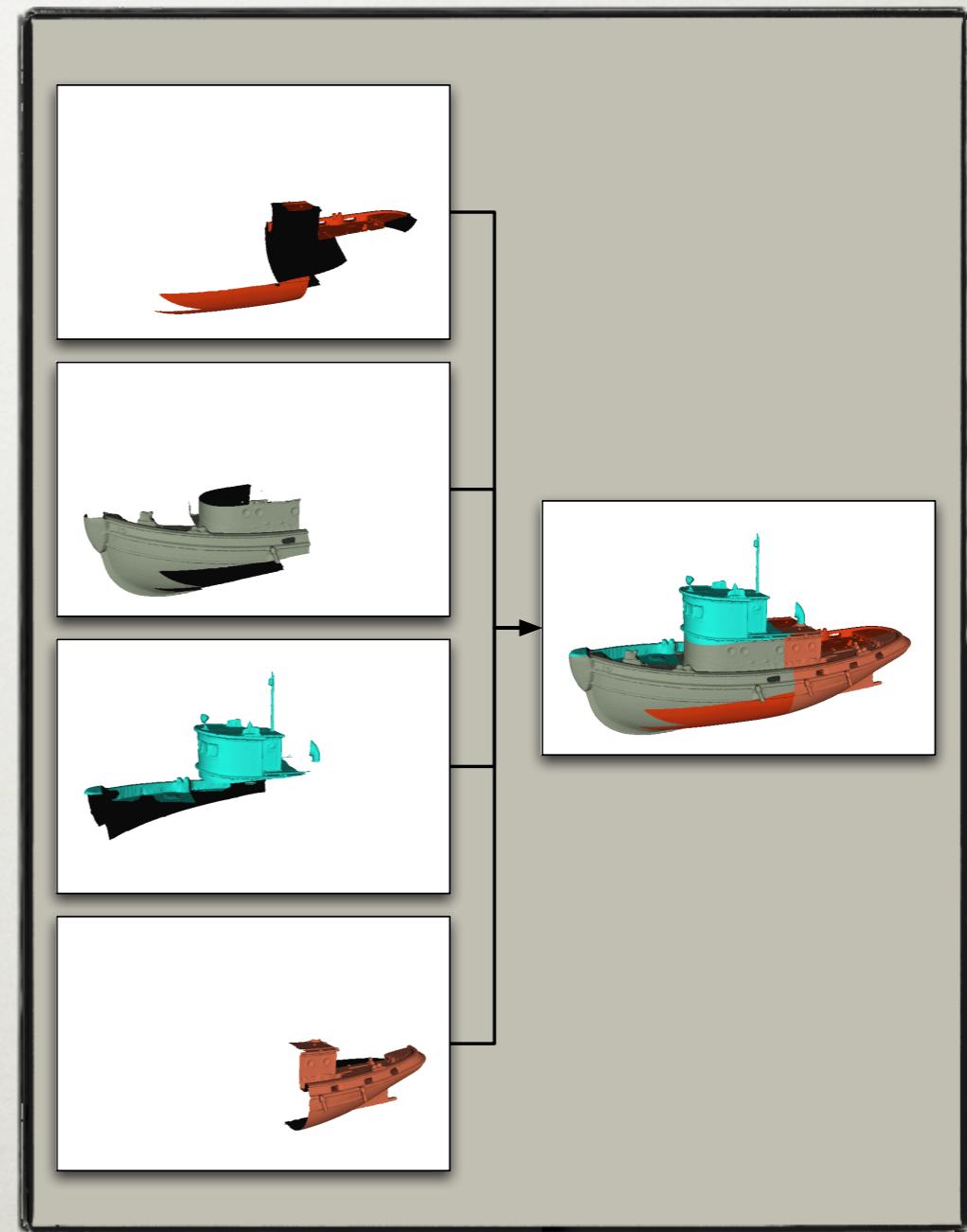
Multilevel

- Combine different algorithms to balance bottlenecks
- Flexible configuration of recomposition algorithm
- Example: Eye + 2D



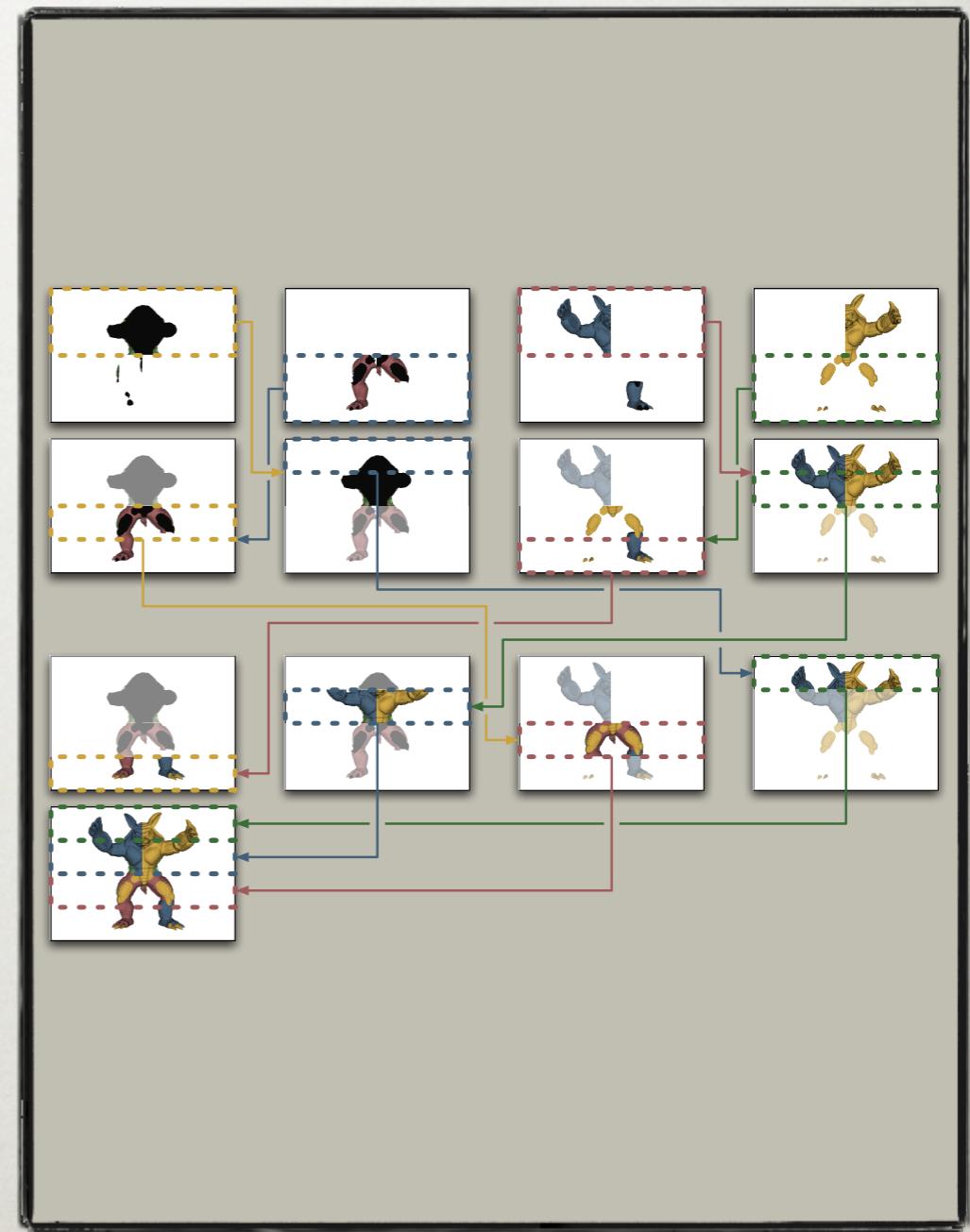
Sort-Last Compositing

- Naive approach
composites everything
on final view
 - $O(n)$ IO requirements
 - $\approx 6\text{MB}/\text{node}/\text{frame}$
- Parallelize compositing



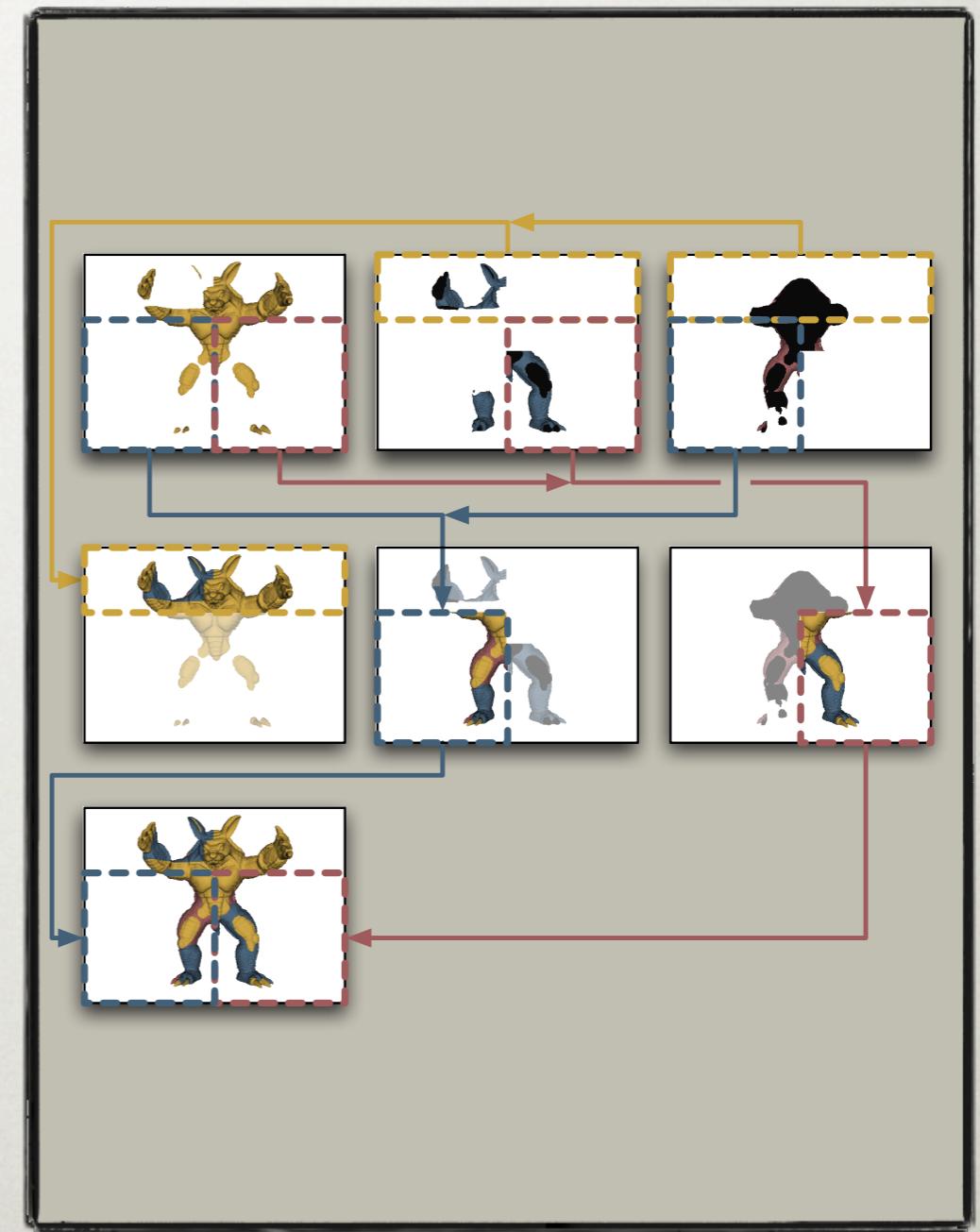
Binary Swap

- Power-of-two nodes
 - 1. Swap half with partner
 - 2. Composite
 - 3. Repeat 1. & 2. until own tile is complete
 - 4. Gather tiles
- $O(1)$, but $O(\log n)$ steps



Direct Send

- Any number of nodes
 1. Composite full tile
 2. Gather tiles
- $O(1)$

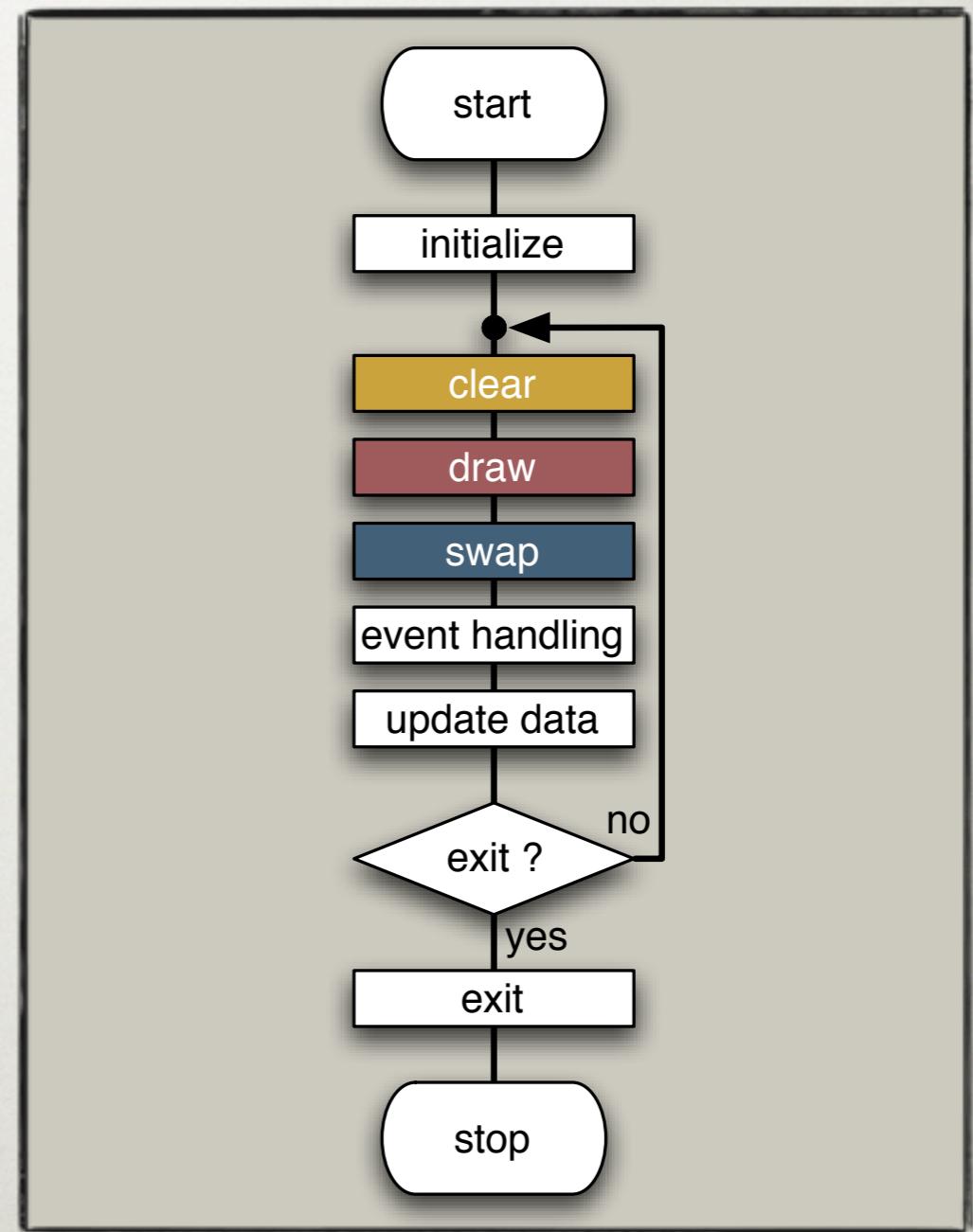


Multipipe Programming

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

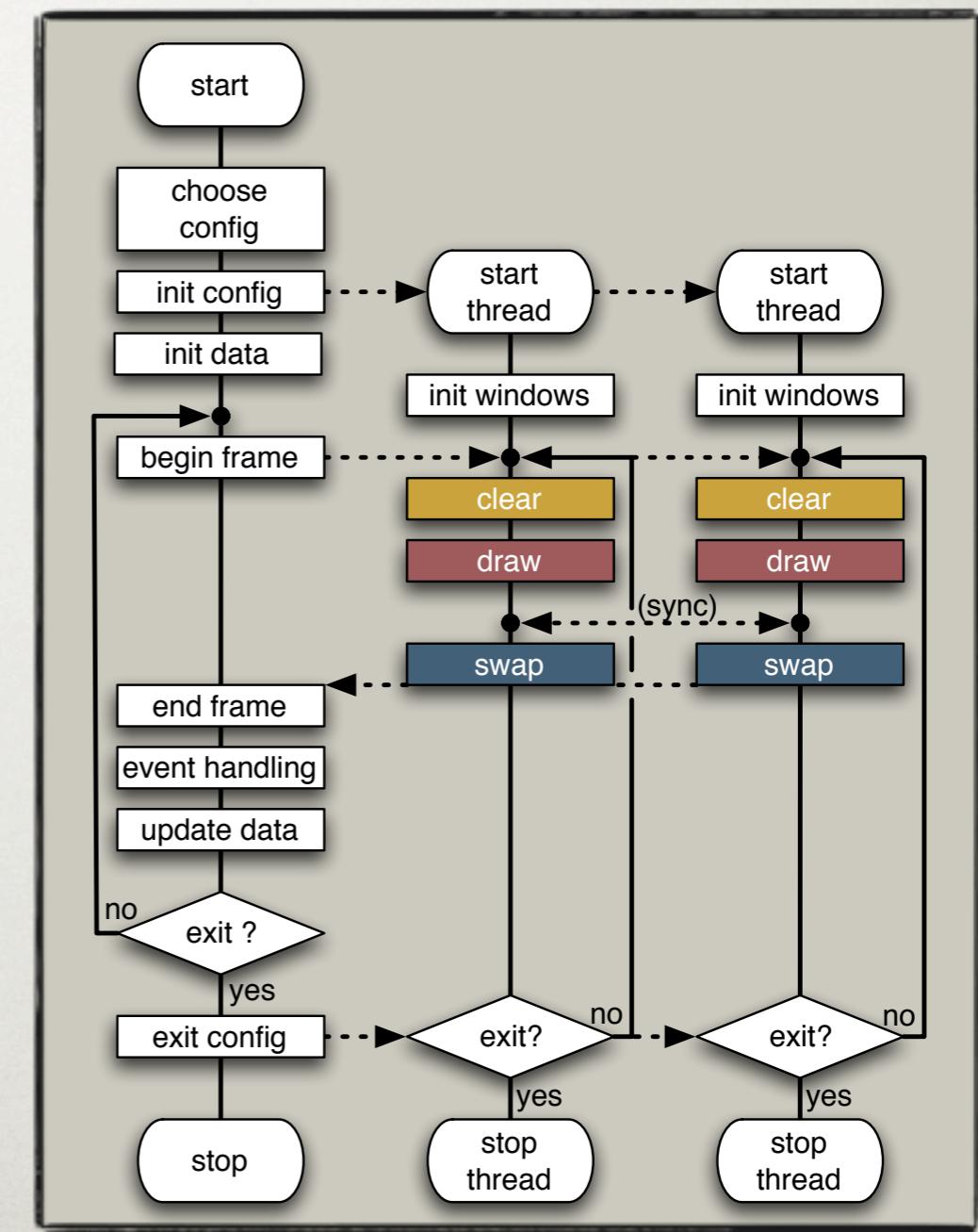
Single Pipe Rendering

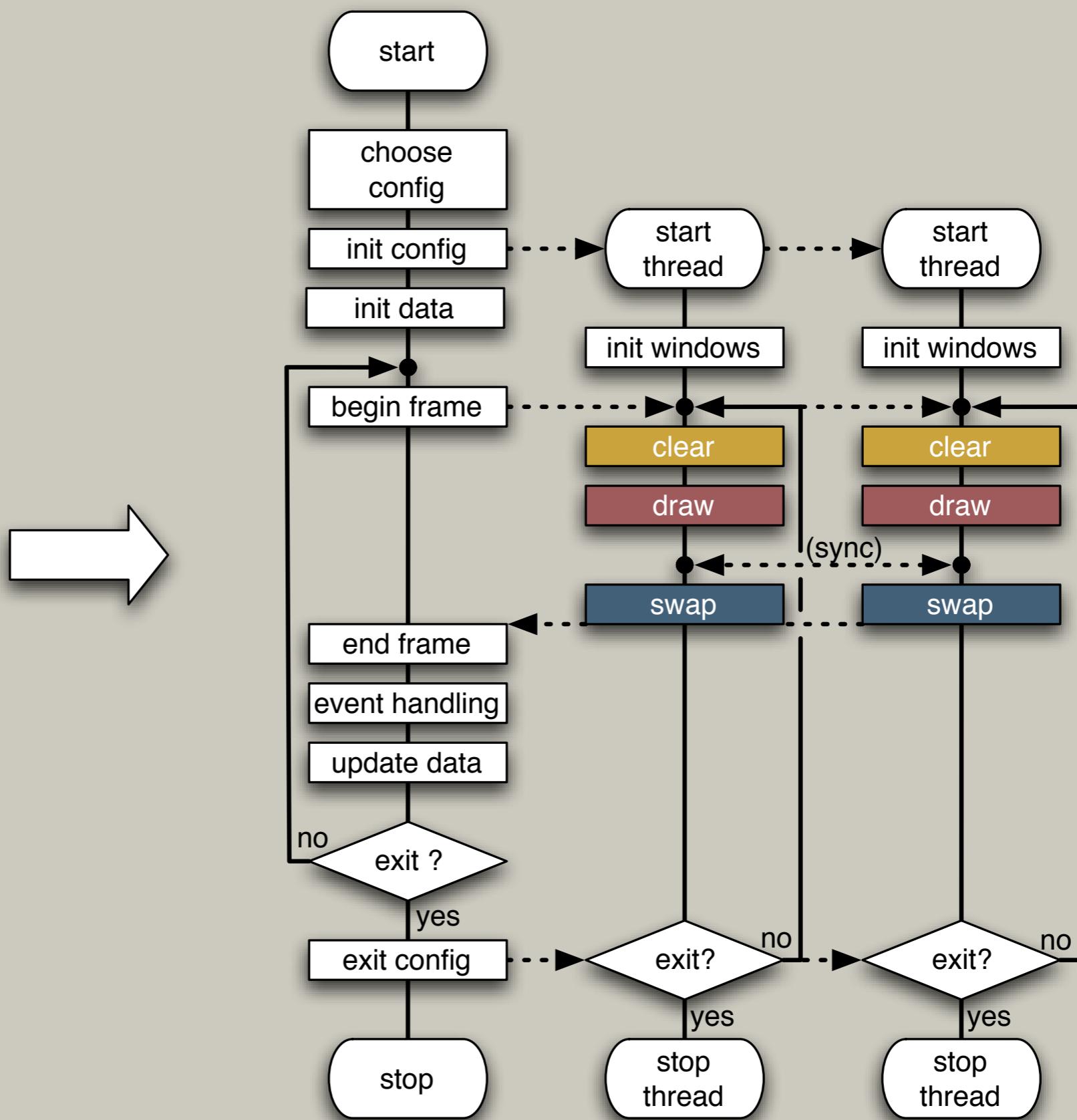
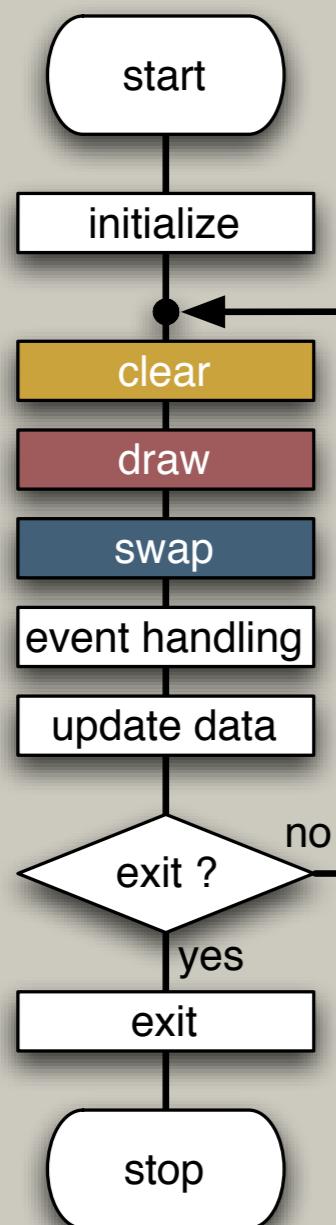
- Typical rendering loop
- Stages may not be well separated



Multipipe Rendering

- Separate rendering and application
- Instantiate rendering multiple times
- Synchronize parallel execution
- 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

Equalizer Programming

Application subclasses and overrides “callback” methods, e.g.:

- **Channel::frameDraw** to render using the provided frustum, viewport and range
- **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!

Data Distribution

- Static and versioned distributed objects
- Versioned objects work like a simple version control system, e.g., cvs
 - Multi-Buffering of data
 - One master instance commits new versions
 - Slave instances sync to version
 - Versions are typically frame-specific
- Use eqPly's **InitData** and **FrameData** as an example

Data Distribution

- How to create distributed objects:
 - Subclass from `eqNet::Object`
 - Use `setInstanceData`, `setDeltaData` to declare data or implement your own pack/unpack routines
- How to use distributed objects:
 - Master instance: `Config::registerObject()` assigns unique identifier
 - Slave instances: `Config::mapObject()` uses unique identifier to map to master instance

Porting Details

- Configuration
- Multithreading
- Synchronization
- Rendering decomposition
- Data distribution for clusters

Configuration

- Display environment
- Rendering resources
- Rendering mode (stereo, mono)
- Equalizer provides:
 - Simple ASCII configuration files
 - Central resource management

Multithreading/Synchronization

- Threading model
- Display synchronization
- Data synchronization
- Equalizer provides:
 - Abstraction of threading library
 - Swapbarrier, lock, sema, barrier, ...

Rendering Decomposition

- Task computation
- Result recomposition
- Equalizer provides:
 - Automatic task generation based on config
 - Flexible decomposition and recomposition
 - Task-specific callbacks

Data Distribution

- Who needs which data?
- Equalizer provides
 - Versioned, distributed objects
 - ‘Userdata’ mechanism, e.g.:
 - Config::beginFrame(**id**) is input parameter to Channel::draw(**id**)
 - ID identifies object or object version

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