Virtual Reality Applications with Equalizer

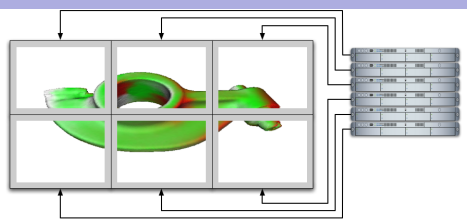
Computer Graphics and
Multimedia Systems Group

University of Siegen

Overview

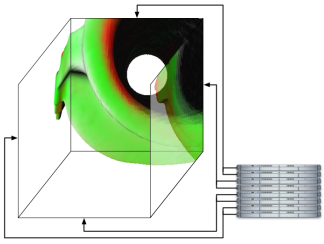
- **Virtual Reality Applications**
- **Equalizer**
  - Overview
  - Usage for VR
  - Concept
  - Class hierarchy
  - Data synchronization
  - Process synchronization
  - Configuration file
- **Examples**
  - eqBox: Minimal Example
  - eqFlyingThings: Example for the VR Lab
  - Using Equalizer in the VR Lab

Virtual Reality Applications



Powerwall:

Six computers, each with one graphics card



Four-sided Cave, active stereo:

Four computers, each with one stereo graphics card

VR Lab in Siegen, passive stereo:

Six computers, each with two graphics cards

Virtual Reality Applications

- VR Applications are *distributed*:
  - Multiple computers
  - Multiple graphics cards per computer
  - Additional computers e.g. for sound, tracking, . . .
- Distributed applications must be *synchronized*:
  - Data synchronization
    - Scene state: all graphics cards render the same scene at a given point in time
    - Scene view: different graphics cards render different views of the scene
  - Process synchronization
    - Frame sync: all graphics cards must finish rendering the current frame before it can be displayed (software synchronization)
    - Buffer swap: all graphics cards swap front and back buffer at the same time (hardware synchronization if supported; otherwise emulation by software)

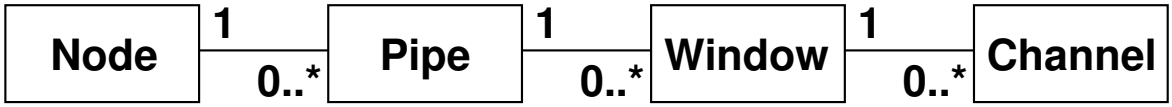
Equalizer: Overview

- Equalizer
  - C++ API for writing distributed OpenGL applications
  - Free and Open Source Software (LGPL)
  - Portable
  - Active development; currently at version 1.2.1
- Concepts
  - Splitting of rendering tasks into different hierarchy levels
  - Software: splitting into
    - State (scene state, parameters)
    - Logic (OpenGL code)
  - Processes: splitting into
    - Client: OpenGL application
    - Server: Equalizer process
  - VR Lab (cluster, geometry, . . .) is only known by the server
  - The application runs without changes in different setups

Equalizer: Overview

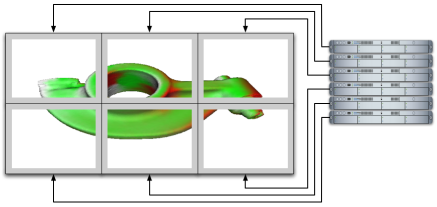
- Splitting of rendering tasks into hierarchy levels:

  Node: Process
  Pipe: Graphics card
  Window: Window (with associated OpenGL context)
  Channel: Output area in a window (viewport)

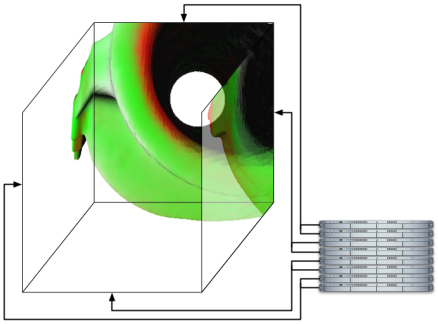


- C++ classes for data synchronization
  - Base class for objects that can be synchronized over a network
  - Event handling (network transparent)
- Typical VR configuration:
  - Each computer runs one or more nodes
  - Each node has one or more pipes
  - Each pipe has exactly one window with exactly one channel (fullscreen mode)

Equalizer: Usage for VR



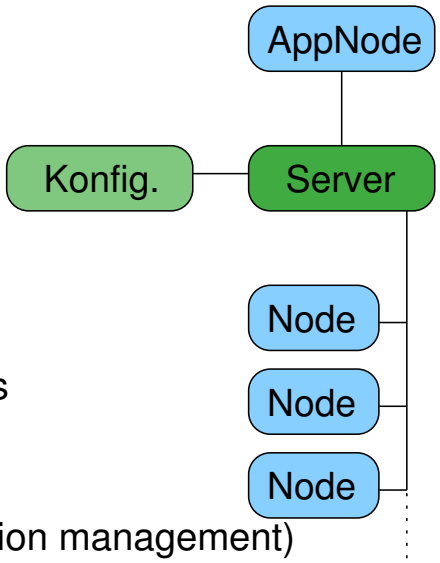
- Powerwall: six projection areas,
  six computers, one graphics card each
  $\Rightarrow$ 1 node per computer, 1 pipe per Node
- Four-sided Cave: active stereo,
  four computers
  $\Rightarrow$ 1 node per computer, 1 pipe per node



- VR Lab Siegen:
  $\Rightarrow$ 1 node per computer, 2 pipes per node
  2 nodes per computer, 1 pipe per node
- Equalizer supports much more:
  - Visualization workstations
  - Cluster for distributed computing on GPUs
  - Load Balancing
  - ...

Equalizer: Concept

- *Configuration file* describes
  - Computers in the cluster
  - Nodes, pipes, windows, channels
  - Geometry of displays
- *Equalizer Server*
  - Reads configuration file
  - Waits to be contacted by the
    AppNode
  - Starts nodes on the configured computers
  - Tells the nodes what to do
- *Equalizer Application*
  - Implements the class `Config` (for application management)
    as well as `Node`, `Pipe`, `Window`, and `Channel`
  - Contacts server and waits to be instructed:
    - *AppNode*: Application thread steers interaction and animation
    - Each node with a pipe: Render thread renders the scene



Equalizer: Class hierarchy

`Config`: Management of the application

- Application contacts server and gets configuration with `eq::getConfig()`
- In the AppNode:
  - Animation
  - Eventhandling (Interaction)
  - Triggering rendering of a new frame
  - Data synchronization
- Everywhere else:
  - Equalizer takes control; `eq::getConfig()` does not return

Equalizer: Class hierarchy

`Node`: Process

- Manages data that is only requires once per process, e.g. the model that needs to be rendered.

`Pipe`: Graphics card

- Manages data that can change between frames, e.g. the position of objects
- Equalizer starts one render thread for each pipe
- Each window of a pipe renders the same frame

Equalizer: Class hierarchy

`Window`: Output window with associated OpenGL context

- Initializes the OpenGL context
- Manages data that is specific to OpenGL contexts, e.g. display lists, textures

`Channel`: Output area in a window (viewport)

- Renders the scene: `channel::frameDraw()`

Equalizer: Data synchronization

- Distributed objects are inherit `co::Object`
  - `getChangeType()` returns synchronization type: `STATIC` for static data, `INSTANCE` orr `DELTA` for variable data
  - `getInstanceData()` and `applyInstanceData()` serialize / deserialize the object
- Each distributed object is assigned to exactly one master instance
  - The master instance is registered with `registerObject()`
  - Changes to the master instance are activated using `commit()`
- Synchronized instances are
  - bound to one master instance with `mapObject()`
  - synchronized with `sync()`

Equalizer: Data synchronization

Typical data management for VR applications:

- *InitData:* Data that is only required once per node and is not changed
  - Managed per node; needs to be synchronized only once
  - Synchronization type `STATIC`
  - Example: a model that needs to be rendered
- *FrameData:* Data that changes from frame to frame
  - Managed per pipe; needs to be synchronized with each new frame
  - Synchronisation type `INSTANCE` or `DELTA`
  - Example: object positions, position of the tracked user
- The master instances are manages in the `config` class (i.e. on the AppNode); only here can changes be made

Equalizer: Process synchronization

Typical process management:

- Frame sync:
  - Main loop in the AppNode:

```
while (config->isRunning()) {
    config->startFrame();
    config->finishFrame();
}
```

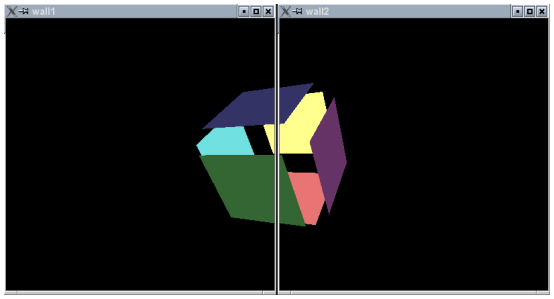
  - Server synchronizes pipes
- Buffer swap:
  - Shared buffer swaps are defined in the configuration file
  - Server synchronizes the swap
  - Equalizer supports hardware and software swapsync

Equalizer: Configuration file

```
global { EQ_WINDOW_IATTR_HINT_FULLSCREEN OFF }
server { config {
    appNode {
      pipe { window {
          viewport [  0   0 400 400 ]
          channel { name "channel1" } } } } }
    node {
      pipe { window {
          viewport [ 410 0 400 400 ]
          channel { name "channel2" } } } } }
    compound {
      compound { channel "channel1"
        wall { bottom_left  [ -1 -0.5 -1 ]
               bottom_right [  0 -0.5 -1 ]
               top_left     [ -1 +0.5 -1 ] } }
      compound { channel "channel2"
        wall { bottom_left  [  0 -0.5 -1 ]
               bottom_right [ +1 -0.5 -1 ]
               top_left     [  0 +0.5 -1 ] } }
} } }
```


Example: eqBox

- eqBox:
  - Displays a rotating box
  - InitData:
    Light sources, rotation axis
  - FrameData:
    Position of the Box, rotation angle
  - Simple interaction via
    cursor keys



- Tasks:
  - Adapt the program
    - Render different object
    - Define new interaction possibilities
    - ...
  - Write configuration files for other scenarios:
    - More windows, multiple computers
    - Configuration for a four-sided Cave
    - Stereo graphics for anaglyph glasses
    - ...

eqBox: InitData

```
class InitData : public co::Object
{
    struct {
        uint32_t frame_data_id;
        float light_pos[4], light_ambient[4];
        float rotation_axis[3];
    } data;

    ...

    void getInstanceData(co::DataOStream &os) {
        os.write(&data, sizeof(data));
    }
    void applyInstanceData(co::DataIStream &is) {
        is.read(&data, sizeof(data));
    }
    ChangeType getChangeType() const {
        return STATIC;
    }
};
```


eqBox: FrameData

```
class FrameData : public co::Object
{
    struct {
        float rotation_angle;
        float head_pos[3];
        float box_pos[3];
    } data;

    ...

    void getInstanceData(co::DataOStream &os) {
        os.write(&data, sizeof(data));
    }
    void applyInstanceData(co::DataIStream &is) {
        is.read(&data, sizeof(data));
    }
    ChangeType getChangeType() const {
        return INSTANCE;
    }
};
```

eqBox: Config

```
class Config : public eq::Config
{
    InitData init_data;          // Master instance
    FrameData frame_data;        // Master instance

    bool init(const InitData &init_data_template) {
        initData.data = init_data_template.data;
        registerObject(&frame_data);
        registerObject(&init_data);
        return eq::Config::init(initData.getID());
    }
    eq::uint128_t startFrame() {
        // Animation: update frame_data ...
        eq::uint128_t version = frame_data.commit();
        return eq::Config::startFrame(version);
    }
    bool handleEvent(const eq::ConfigEvent *event) {
        // Interaction: update frame_data ...
    }
};
```


eqBox: Node

```
class Node : public eq::Node
{
    InitData init_data;          // Synchronized instance

    bool configInit(const eq::uint128_t &init_id) {
        if (!eq::Node::configInit(init_id))
            return false;
        // Map synchronized InitData instance
        // to master instance
        eq::Config *config = getConfig();
        return config->mapObject(&init_data, init_id);
    }

    void frameStart(...);
    void frameFinish(...);
};
```

eqBox: Pipe

```
class Pipe : public eq::Pipe
{
    FrameData frame_data;        // Synchronized instance

    virtual bool configInit(const eq::uint128_t &init_id) {
        if (!eq::Pipe::configInit(init_id))
            return false;
        // Map synchronized FrameData instance
        // to master instance
        ...
        eq::Config *config = getConfig();
        return config->mapObject(&frame_data, frame_data_id);
    }

    void frameStart(const eq::uint128_t &frame_id, ...) {
        frame_data.sync(frame_id);
    }
    void frameFinish(...);
};
```


eqBox: Window

```
class Window : public eq::Window
{
    GLuint box_display_list; // only valid in one GL context

    bool configInitGL(const eq::uint128_t &init_id) {
        if (!eq::Window::configInitGL(init_id))
            return false;
        box_display_dist = glGenLists(1);
        glNewList(box_display_list, GL_COMPILE);
        ...
        glEndList();
        return true;
    }

    void frameStart(...);
    void frameFinish(...);
};
```

eqBox: Channel

```
class Channel : public eq::Channel
{
    void frameDraw(const eq::uint128_t &frame_id) {
        eq::Channel::frameDraw(frame_id);
        const Node *node = getNode();
        const InitData &initData = node->getInitData();
        const Pipe *pipe = getPipe();
        const FrameData &frameData = pipe->getFrameData();
        const Window *window = getWindow();
        glLightfv(GL_LIGHT0, GL_POSITION,
            init_data.data.light_pos);
        glLightfv(GL_LIGHT0, GL_AMBIENT,
            init_data.data.light_ambient);
        glTranslatef(frame_data.data.box_pos...);
        glRotatef(frame_data.data.rotation_angle, ...);
        glCallList(window->box_display_list);
    }
    void frameStart(...);
    void frameFinish(...);
};
```

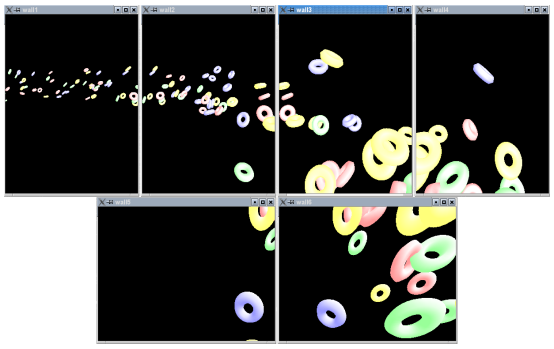

eqBox: main

```
int main(int argc, char *argv[])
{
    NodeFactory node_factory;
    eq::init(argc, argv, &node_factory));

    InitData init_data_template;
    // fill init_data_template...

    Config *config = eq::getConfig(argc, argv);
    // eq::getConfig() only returns on the application node
    config->init(init_data_template));
    while (config->isRunning()) {
        config->startFrame();
        config->finishFrame();
    }
    config->exit();
    eq::releaseConfig(config);
    eq::exit();
    return 0;
}
```

Example: eqFlyingThings



- eqFlyingThings:
  - Stripped-down version of a tutorial task for the VR lecture
  - Animates a stream of different moving objects
- Adapted to the VR Lab:
  - Headtracking via DTrack
    `DTrack.hpp, DTrack.cpp`
  - Dynamic Warping for the four cylindrical projection areas
    `Warp.hpp, Warp.cpp, WarpShader.hpp`
  - Equalizer configuration files
    `vrlab.eqc, vrlab-12node.eqc, vrlab-simulation.eqc`
- Tasks:
  - Run it and test it!
  - Understand the differences in the configuration files
  - Understand how the modules for tracking and dynamic warping are used

Example: Equalizer in the VR Lab

- Setup:
  - Pool: `. /data/staff/install/profiles/equalizer-1.0`
  - VR Cluster: `. /vr/projects/profiles/equalizer-1.0`
- Compile eqBox: `cd eqBox; cmake .; make; ./eqBox`
- Start Equalizer server: `eqServer config.eqc`
  In the VR cluster, the server must run on `vrmaster` (preconfigured with `$EQ_SERVER`).
- Examples for configuration files:
  `/data/staff/install/ubuntu-10.4/`
  `equalizer-1.0/share/Equalizer/configs`
- Documentation:
  - `http://www.equalizergraphics.com/`
  - Mailinglist `eq-dev`