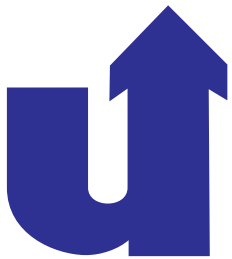
Virtual Reality Application Programming with QVR

Computer Graphics and
Multimedia Systems Group

University of Siegen

November 22, 2026





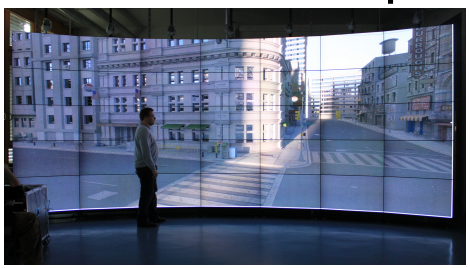
Overview

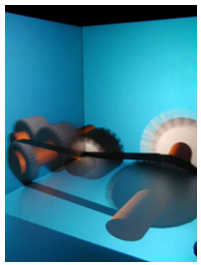
- Challenges for VR frameworks
- Solutions for Multi-GPU / Multi-Host VR
- QVR Overview and Concepts
- QVR Application Interface
- QVR Configuration and Management
- QVR Example Application
- QVR Outlook

Challenges for VR frameworks

Challenges

- VR applications run on a wide variety of graphics and display hardware setups:






- In general, a VR application must handle
  - Multiple hosts (for render clusters)
  - Multiple GPUs on a host
  - Multiple displays devices attached to a GPU

  whereas typical non-VR graphics applications only handle
  - A single display device attached to a single GPU on a single host

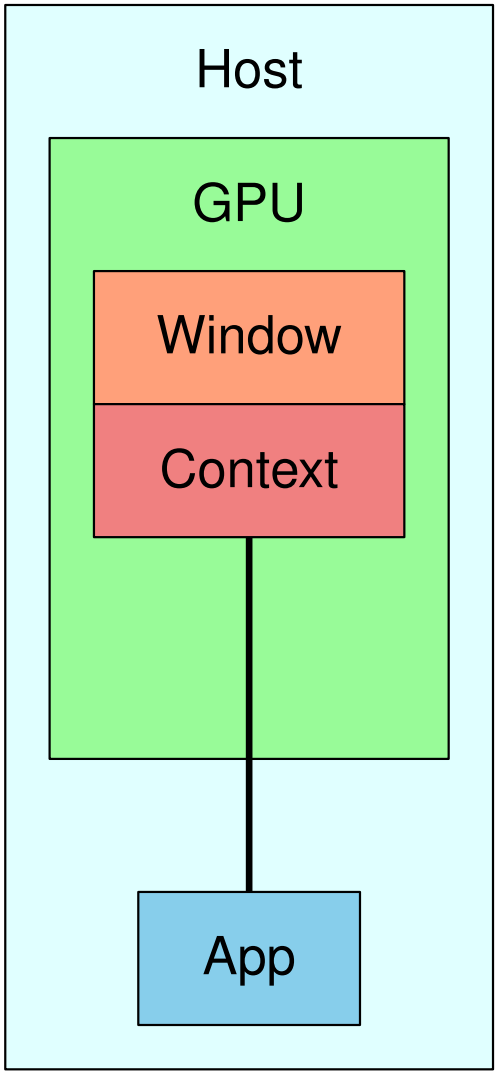
Challenges for VR frameworks

VR frameworks

	Multi-GPU & Cluster	Vive, Oculus	Google VR	Allows own renderer	License	Remarks
Avocado	✓	✗	✗	✗	—	Dead
VRJuggler	✓	✗	✗	?	LGPL	Smelling funny
OpenSG	✓	✗	✗	✗	LGPL	Fraunhofer
ViSTA	✓	✗	✗	(✗)	LGPL	RWTH AC & DLR
Equalizer	✓	(✗)	✗	✓	LGPL	More than VR!
Unreal Eng.	(✗)	✓	✓	✗	Propr.	Epic Games
Unity Eng.	✗	✓	✓	✗	Propr.	Unity Tech.
QVR	✓	✓	(✓)	✓	MIT	

Challenges for VR frameworks

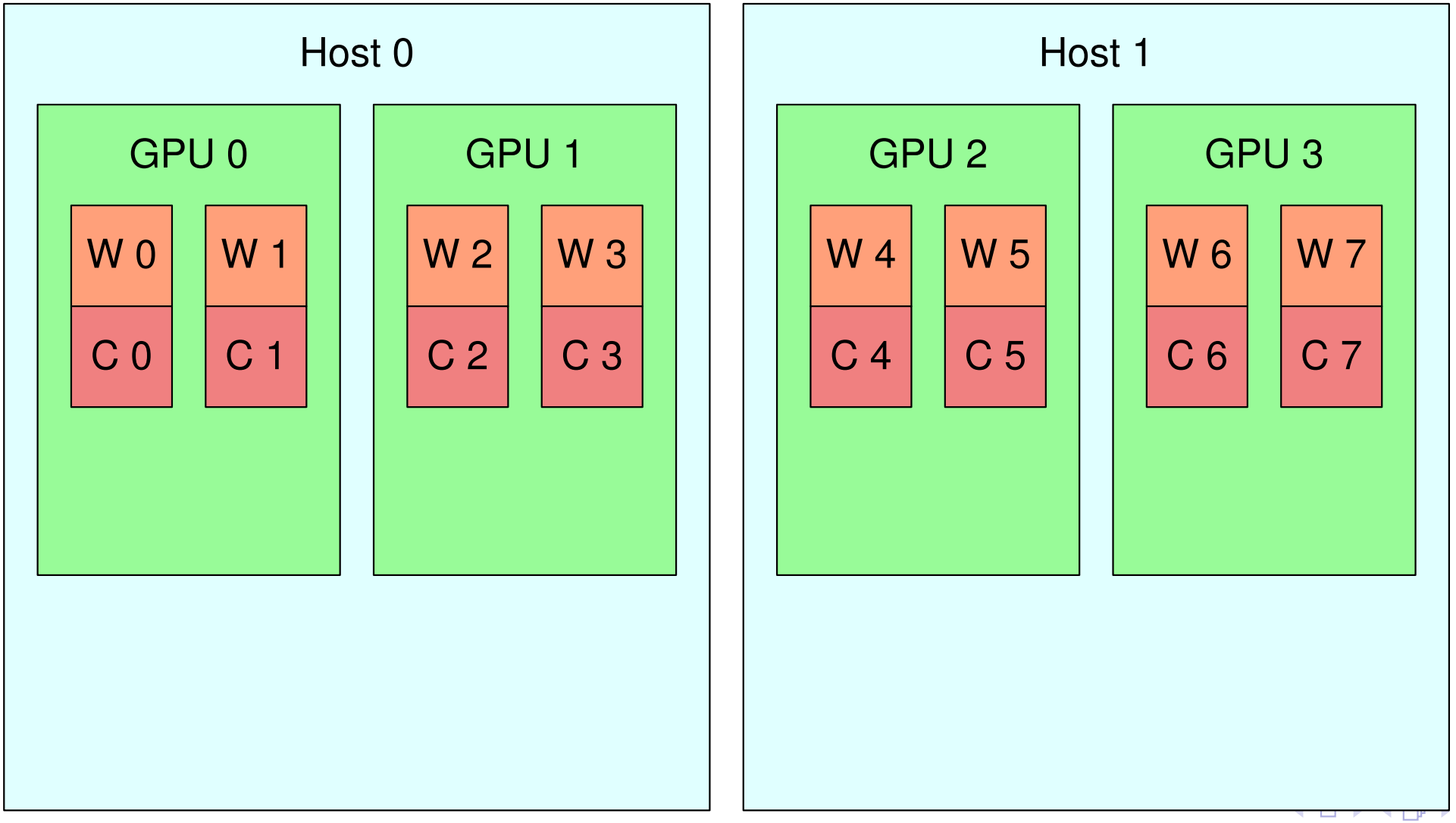
Typical non-VR graphics application



- The application uses a toolkit to create a window
- The toolkit creates an OpenGL context automatically and “makes it current”
- The application never needs to care about the context
  - There is only one context
  - The context is always current

Challenges for VR frameworks

VR application using multiple hosts, GPUs, and displays



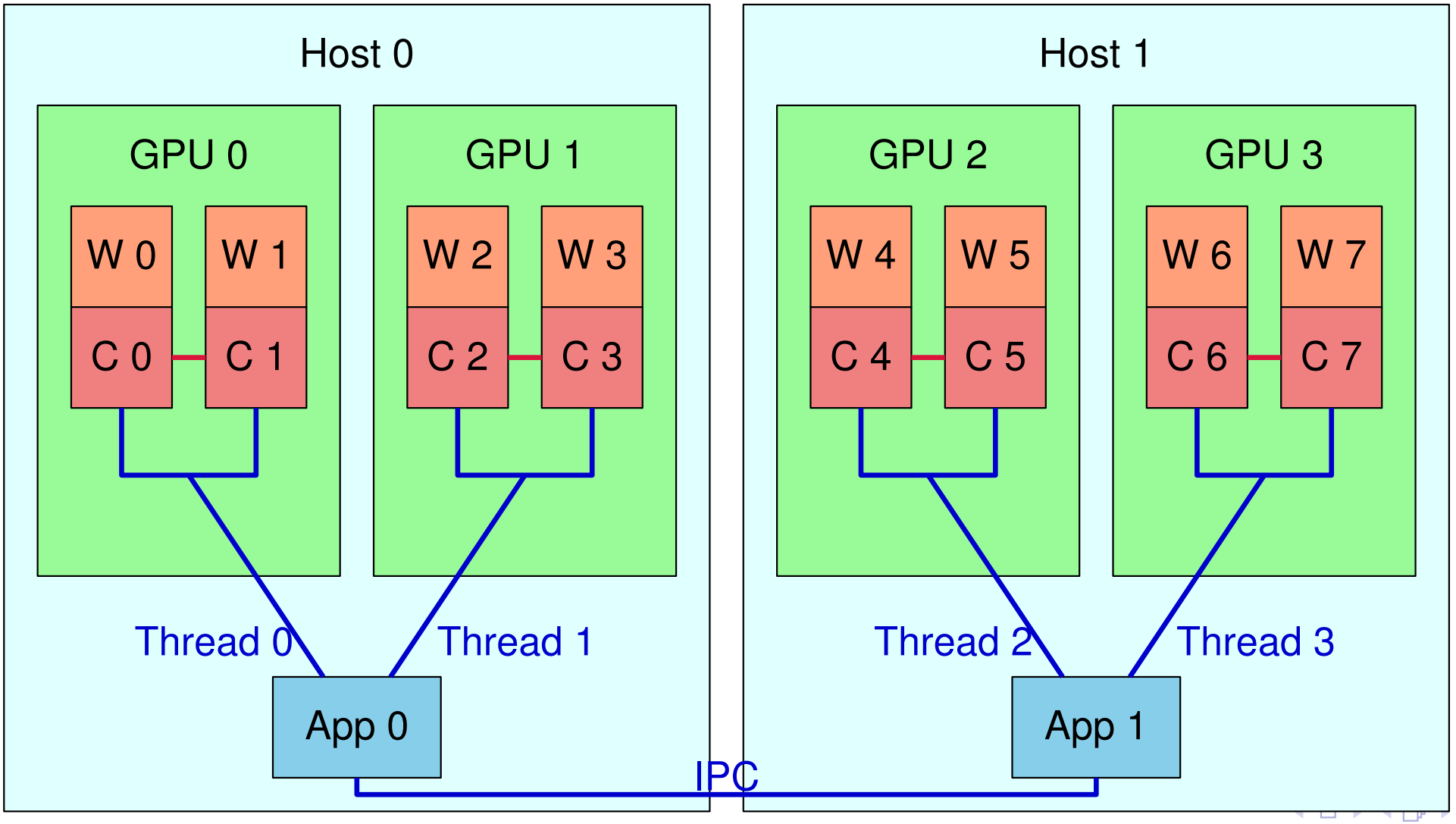
Challenges for VR frameworks

Challenges: OpenGL contexts and threading

- OpenGL contexts on the same GPU can *share* objects such as textures.
  → Only one context should manage OpenGL objects.
- A context can only be current in one thread at a time, and a switch of that thread is expensive.
  → All rendering to a context should happen from only one thread.
- Access to a single GPU is serialized by the driver.
  → Rendering into different contexts on the same GPU should be serialized to avoid context switches.
- The function that triggers swapping of back and front buffers blocks until the swap happened, and the swap is typically synchronized to the display frame rate.
  → The thread in which the context is current is often blocked.

Solutions for Multi-GPU / Multi-Host VR

Multi-Context Multi-Thread Approach



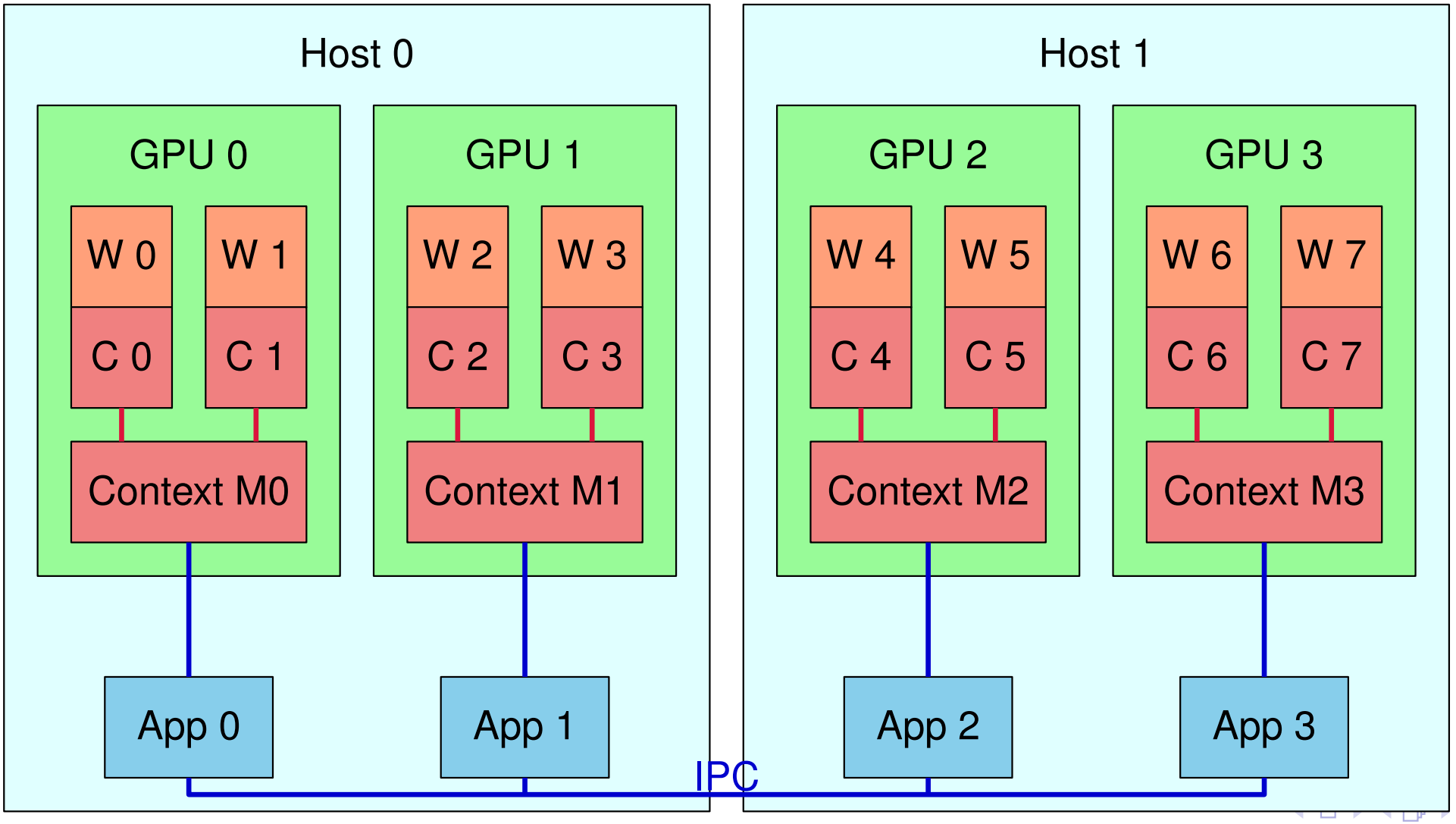
Solutions for Multi-GPU / Multi-Host VR

Multi-Context Multi-Thread Approach

- One process per host
- One context per window
- One thread per GPU
  - Contexts driven by thread share objects
  - Window views driven by thread are rendered sequentially
- An application process must be aware of
  - Multiple rendering threads
  - Multiple contexts that may or may not be sharing objects
- Interprocess communication:
  - Only between hosts

Solutions for Multi-GPU / Multi-Host VR

Single-Context Single-Thread Approach



Solutions for Multi-GPU / Multi-Host VR

Single-Context Single-Thread Approach

- One process per GPU
- One context per process (plus one hidden context per window)
- One thread per process (main thread)
  - Context sharing irrelevant to application
  - Window views are rendered sequentially
- An application process must be aware of
  - Only one thread (rendering threads are hidden)
  - Only one context (window contexts are hidden)
- Interprocess communication:
  - Between hosts
  - Between processes on same host if multiple GPUs are used

QVR Overview and Concepts

The QVR framework

- Implements the single-context single-thread approach for multi-GPU / multi-host support
- Based on Qt (requires nothing else)
- Manages four major types of objects:
  - *Devices* used for interaction, e.g. game controllers
  - *Observers* that view the virtual scene
  - *Windows* that provide views of the virtual scene
  - *Processes* that run on hosts and manage windows
- A VR application implements a simple interface:
  - *render()* to render a view of the scene into a texture
  - *update()* for interactions, animations, and other scene updates
  - Optional: one-time or per-frame actions per process or window
  - Optional: device/keyboard/mouse event handling
  - Optional: serialization, for multi-process support
- Applications run unmodified on different setups

QVR Overview and Concepts

Functionality

- Handles tracking
  Glasses / HMDs, Controllers, ...
- Handles input
  Controller buttons / joysticks, mouse / keyboard
- Handles output
  Puts left/right view onto screen(s) as appropriate

Application View

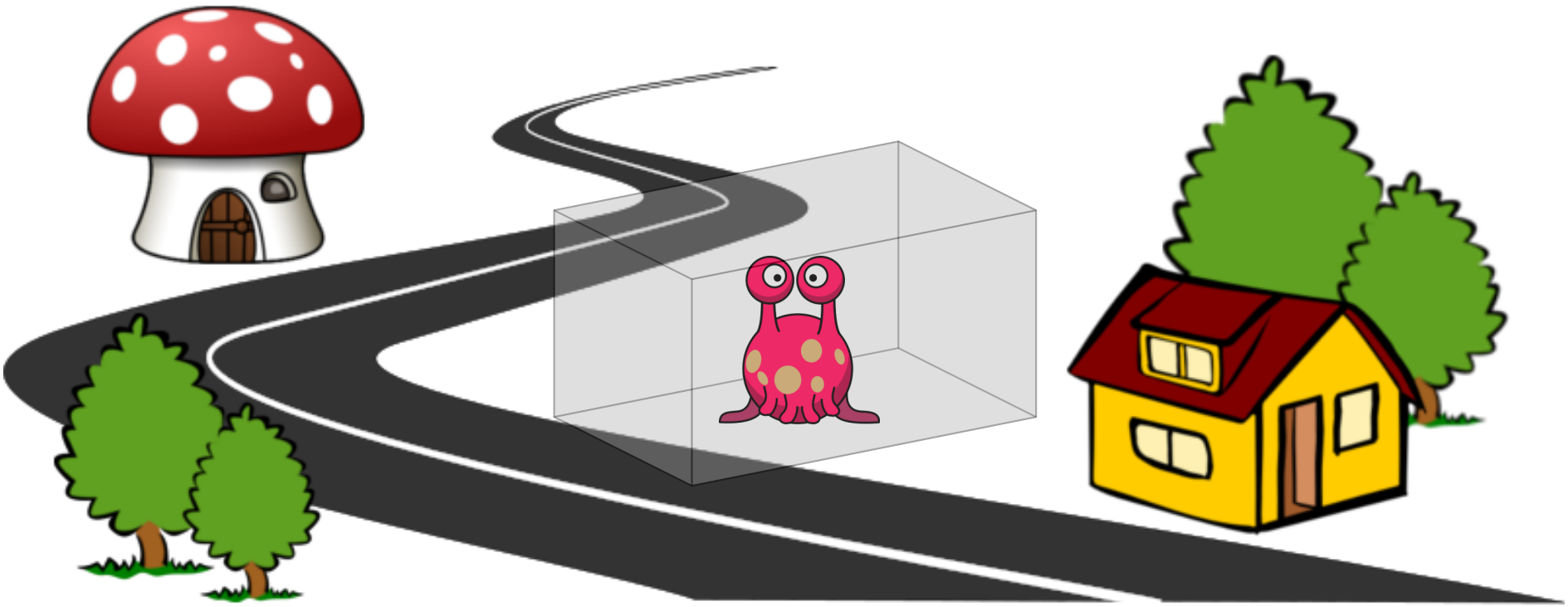
- Implements `QVRApp`
  `render()`, `update()`, ...
- Can access
  `QVRDevice`, `QVRObserver`, `QVRWindow`, `QVRProcess`
- Does not need to care about
  VR hardware, Multi-GPU, threads, OpenGL contexts, ...

VR System View

- Configuration describes available VR hardware
- Autodetected or manually configured
- Configuration classes
  `QVRDeviceConfig`, `QVRObserverConfig`, `QVRWindowConfig`, `QVRProcessConfig`

QVR Overview and Concepts

Illustration



- You are an alien
- Your UFO is a transparent box
- You fly your UFO through a strange world
- You can move freely inside your UFO

QVR Overview and Concepts

Illustration

- The alien views the world through the sides of his UFO.
- The alien flies its UFO through the world.
- The alien moves inside its UFO.

QVR

- An *observer* views the virtual world in *windows*; each *window* provides a view for one *observer*.
- An observer *navigates* through the virtual world.
- An observer's movements are *tracked* inside a tracking space.

QVR Overview and Concepts

Devices (in illustration: for example the UFO remote control)

- Optional: can be tracked inside a tracking space
- Optional: provides buttons and other interaction controls
- Examples:
  - Tracked glasses
  - Traditional game controller
  - HTC Vive controllers
  - ART Flystick
- Configured through QVRDeviceConfig
  - Tracking
    - Type and parameters (e.g. based on VRPN, Oculus Rift)
    - Initial position and orientation
  - Digital buttons
  - Analog elements (triggers, joysticks, trackpads)
- Implemented as QVRDevice
  - Tracking: position and orientation
  - State of buttons and analogs
  - Accessible to the update() function for interaction

QVR Overview and Concepts

Observer (in illustration: the alien)

- Views the virtual world through one or more windows
- Can navigate through the virtual world
- Can be bound to tracked devices, e.g. glasses
- Configured through QVRObserverConfig
  - Navigation
    - Type and parameters (e.g. based on QVR device interaction)
    - Initial position and orientation
  - Tracking
    - Type and parameters (e.g. based on specific devices)
    - Initial position and orientation
    - Eye distance
- Implemented as QVRObserver
  - Navigation: position and orientation
  - Tracking: position and orientation *for each eye*

QVR Overview and Concepts

Window (in illustration: a side of the box-shaped UFO)

- Provides a view of the virtual world for exactly one observer
- Configured through QVRWindowConfig
  - Observer to provide a view for
  - Output mode (left/right/stereo view) and parameters
  - Geometry in pixels
  - 3D geometry in the virtual world (if the window corresponds to a physical screen wall)
- Implemented as QVRWindow
  - Accessible as QWindow for the application, if required
  - Hides its context and rendering thread

QVR Overview and Concepts

Process

- Provides one OpenGL context to the application
- Drives zero or more windows
- Runs one instance of the VR application
- First process is main process; child processes are started automatically when needed
- Configured through QVRProcessConfig
  - GPU to use (system specific)
  - Launcher command (e.g. for network processes)
  - List of window configurations
- Implemented as QVRProcess
  - Accessible as QProcess for the application, if required
  - Hides communication between main and child processes

QVR Application Interface

Application

- Interface specified in the QVRApp class
- All functions except `render()` are optional to implement; the empty default implementation is sufficient
- `void render(QVRWindow* w, const QVRRenderContext& context, const unsigned int* textures)`
  - Called once per window per frame
  - Renders one (mono) or two (stereo 3D) views for window `w` into `textures`
  - The `context` contains all necessary information for the view(s)

QVR Application Interface

Application: `render()`

```
void render(QVRWindow* w, const QVRRenderContext& context,
            const unsigned int* textures) {
    for (int view = 0; view < context.viewCount(); view++) {
        // Get view dimensions
        int width = context.textureSize(view).width();
        int height = context.textureSize(view).height();
        // Set up framebuffer object to render into texture
        setupFBO(textures[view]);
        // Set up view
        glViewport(0, 0, width, height);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        QMatrix4x4 P = context.frustum(view).toMatrix4x4();
        QMatrix4x4 V = context.viewMatrix(view);
        // Render
        ...;
    }
}
```

QVR Application Interface

Application (continued)

- `void update(const QList<QVRObserver*>& observers)`
  - Called once before each frame *on the main process*
  - Updates scene state, e.g. for animations
  - May update observers, e.g. for navigation
  - May use QVR devices for interaction
- `bool wantExit()`
  - Called once before each frame *on the main process*
  - Signals if the application wants to exit
- Optional: `void getNearFar(float& near, float& far)`
  - Called once before each frame *on the main process*
  - Sets the preferred near and far clipping plane

QVR Application Interface

Application (continued)

- Optional: process and window initialization
  - `bool initProcess(QVRProcess* p)`
  - `void exitProcess(QVRProcess* p)`
  - `bool initWindow(QVRWindow* w)`
  - `void exitWindow(QVRWindow* w)`
- Optional: per-frame process and window actions
  - `void preRenderProcess(QVRProcess* p)`
  - `void postRenderProcess(QVRProcess* p)`
  - `void preRenderWindow(QVRWindow* w)`
  - `void postRenderWindow(QVRWindow* w)`

QVR Application Interface

Application (continued)

- Optional: serialization for multi-process / multi-GPU support
  - Data that changes between frames
    - `void serializeDynamicData(QDataStream& ds) const`
    - `void deserializeDynamicData(QDataStream& ds)`
  - Data that is initialized once and remains constant
    - `void serializeStaticData(QDataStream& ds) const`
    - `void deserializeStaticData(QDataStream& ds)`
- Optional: Qt-style event handling for QVR devices, mouse, and keyboard
  - `deviceButtonPressEvent()`, `deviceButtonReleaseEvent()`, `deviceAnalogChangeEvent()`, `keyPressEvent()`, `keyReleaseEvent()`, `mouseMoveEvent()`, `mousePressEvent()`, `mouseReleaseEvent()`, `mouseDoubleClickEvent()`, `wheelEvent()`
  - All keyboard / mouse functions get the Qt event *and the QVRRenderContext from which it came*

QVR Application Interface

Render context

- Implemented as QVRRenderContext
- Relevant for rendering and event interpretation
- Provides:
  - Process index, window index
  - Qt window and screen geometry
  - Navigation pose
  - Window screen wall coordinates (virtual world)
  - Window output mode and required views
  - Per view:
    - Eye corresponding to this view pass (left/right/center)
    - Tracking pose
    - View frustum / projection matrix
    - View matrix

QVR Configuration and Management

Configuration

- Accessible by application:
  - A list of QVRDeviceConfig instances
  - A list of QVRObserverConfig instances
  - A list of QVRProcessConfig instances
    - A list of QVRWindowConfig instances
- Configuration file:
  Corresponds 1:1 to QVR*Config classes
  - List of device definitions
  - List of observer definitions
  - List of process definitions
    - List of window definitions
- Completely defines VR setup
- Application runs unmodified on different setups using different configuration files

QVR Configuration and Management

Example configuration: one window on a desktop computer

```
observer my-observer
    navigation wasdqe
    tracking custom

process main
    window my-window
        observer my-observer
        output red_cyan
        position 800 100
        size 400 400
        screen_is_fixed_to_observer true
        screen_is_given_by_center true
        screen_center 0 0 -1
```

QVR Configuration and Management

Example configuration: Oculus Rift

```
device oculus-head
    tracking oculus head
device oculus-eye-left
    tracking oculus eye-left
device oculus-eye-right
    tracking oculus eye-right

observer oculus-observer
    navigation wasdqe
    tracking device oculus-eye-left oculus-eye-right

process oculus-process
    window oculus-window
        observer oculus-observer
        output oculus
```

QVR Configuration and Management

Example configuration: four-sided CAVE, one GPU per side

```
device glasses
    tracking vrpn DTrack@localhost 0

device flystick
    tracking vrpn DTrack@localhost 1
    buttons  vrpn DTrack@localhost 4 1 3 2 0
    analogs  vrpn DTrack@localhost 1 0

observer cave-observer
    navigation device flystick
    tracking device glasses
```

QVR Configuration and Management

Example configuration: four-sided CAVE, one GPU per side (continued)

```
process main-gpu0
    window back-side
        observer cave-observer
        output stereo
        fullscreen true
        screen_is_fixed_to_observer false
        screen_is_given_by_center false
        screen_wall  -1 0 -2  +1 0 -2  -1 2 -2
process child-gpu1
    window left-side
        observer cave-observer
        output stereo
        fullscreen true
        screen_is_fixed_to_observer false
        screen_is_given_by_center false
        screen_wall  -1 0 0  -1 0 -2  -1 2 0
```

QVR Configuration and Management

Example configuration: four-sided CAVE, one GPU per side (continued)

```
process child-gpu2
    window right-side
        observer cave-observer
        output stereo
        fullscreen true
        screen_is_fixed_to_observer false
        screen_is_given_by_center false
        screen_wall  1 0 -2  1 0 0  1 2 -2
process child-gpu3
    window bottom-side
        observer cave-observer
        output stereo
        fullscreen true
        screen_is_fixed_to_observer false
        screen_is_given_by_center false
        screen_wall  -1 0 0  +1 0 0  -1 0 -2
```

QVR Configuration and Management

Manager

- Singleton, implemented as QVRManager
- Initialized in main(), similar to QApplication
- Reads (or creates) configuration
- Creates devices, observers, processes, windows

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QVRManager manager(argc, argv);

    MyQVRApp qvrapp;
    if (!manager.init(&qvrapp)) {
        qCritical("Cannot initialize QVR manager");
        return 1;
    }
    return app.exec();
}
```

QVR Configuration and Management

Command line options (only the most important)

- `--qvr-config=<config.qvr>`
  Specify a QVR configuration file.
- `--qvr-log-level=<level>`
  Set a log level (`fatal`, `warning`, `info`, `debug`, `firehose`).

QVR Example Application

Putting it all together: a minimal example program

- The virtual scene is a rotating cube with 2m edge length, centered at (0,0,-15)
- The scene is rendered using modern OpenGL
- We let QVR handle navigation and tracking
- We want to exit when the user hits ESC
- We want multi-process support

QVR Example Application

Putting it all together: a minimal example program

- Which functions do we need to implement?
  - To initialize OpenGL objects and state: `initProcess()`
  - Always required: `render()`
  - For animated rotation: `update()`
  - To signal that we want to exit: `wantExit()`
  - To receive the ESC key: `keyPressEvent()`
  - For multi-process support: `serializeDynamicData()` and `deserializeDynamicData()`

QVR Example Application

Putting it all together: a minimal example program

- To initialize OpenGL objects and state: `initProcess()`

```
bool QVRMinimalExample::initProcess(QVRProcess* /* p */) {
    initializeOpenGLFunctions();
    glGenFramebuffers(1, &_fbo);
    glGenTextures(1, &_fboDepthTex);
    // setup _fbo and _fboDepthTex
    glGenVertexArrays(1, &_vao);
    glBindVertexArray(_vao);
    // upload vertex data to buffers and setup VAO
    _vaoIndices = 36;
    _prg.addShaderFromSourceFile(QOpenGLShader::Vertex,
        ":vertex-shader.glsl");
    _prg.addShaderFromSourceFile(QOpenGLShader::Fragment,
        ":fragment-shader.glsl");
    _prg.link();
    return true;
}
```

QVR Example Application

Putting it all together: a minimal example program

- Always required: `render()`

```
void QVRExampleOpenGLMinimal::render(QVRWindow* /* w */,
        const QVRRenderContext& context,
        const unsigned int* textures) {
  for (int view = 0; view < context.viewCount(); view++) {
    // Get view dimensions
    int width = context.textureSize(view).width();
    int height = context.textureSize(view).height();
    // Set up framebuffer object to render into
    glBindTexture(GL_TEXTURE_2D, _fboDepthTex);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, width,
        height, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    glBindFramebuffer(GL_FRAMEBUFFER, _fbo);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_\
        ATTACHMENT0, GL_TEXTURE_2D, textures[view], 0);
    glViewport(0, 0, width, height);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

QVR Example Application

Putting it all together: a minimal example program

- Always required: `render()` (continued)

```
    QMatrix4x4 P = context.frustum(view).toMatrix4x4();
    QMatrix4x4 V = context.viewMatrix(view);
    // Set up shader program
    glUseProgram(_prg.programId());
    _prg.setUniformValue("projection_matrix", P);
    // Render
    QMatrix4x4 M;
    M.translate(0.0f, 0.0f, -15.0f);
    M.rotate(_rotationAngle, 1.0f, 0.5f, 0.0f);
    QMatrix4x4 MV = V * M;
    _prg.setUniformValue("modelview_matrix", MV);
    _prg.setUniformValue("normal_matrix", MV.normalMatrix());
    glBindVertexArray(_vao);
    glDrawElements(GL_TRIANGLES, _vaoInd, GL_UNSIGNED_INT, 0);
  }
}
```

QVR Example Application

Putting it all together: a minimal example program

- For animated rotation: `update()`

```
void QVRMinimalExample::update(const QList<const QVRDevice*>& c
        const QList<QVRObserver*>& customObservers) {
    float seconds = _timer.elapsed() / 1000.0f;
    _rotationAngle = seconds * 20.0f;
}
```

- To signal that we want to exit: `wantExit()`
- To receive the ESC key: `keyPressEvent()`

```
bool QVRMinimalExample::wantExit() { return _wantExit; }
void QVRMinimalExample::keyPressEvent(
        const QVRRenderContext& /* context */,
        QKeyEvent* event) {
    if (event->key() == Qt::Key_Escape)
        _wantExit = true;
}
```

QVR Example Application

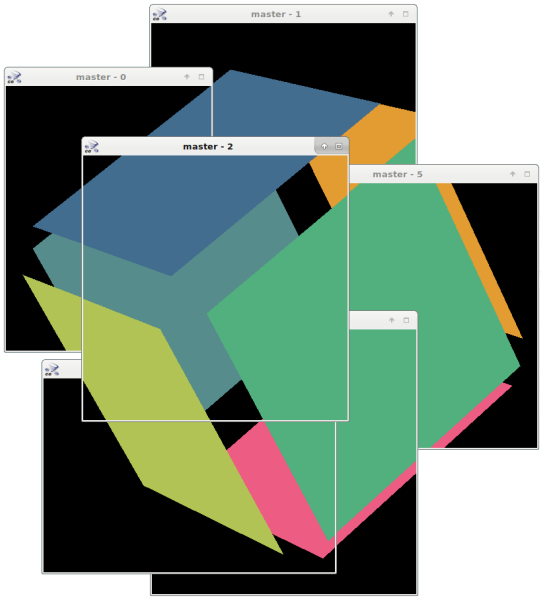
Putting it all together: a minimal example program

- For multi-process support: `serializeDynamicData()` and `deserializeDynamicData()`

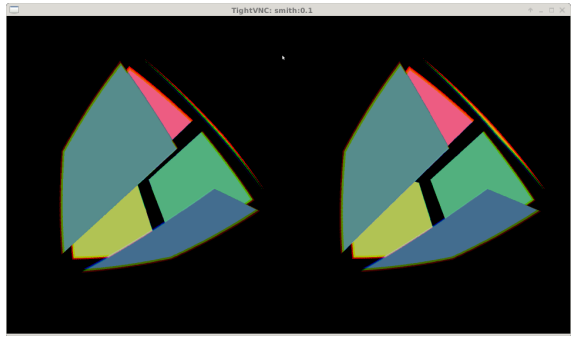
```
void QVRMinimalExample::serializeDynamicData(
        QDataStream& ds) const {
    ds << _rotationAngle;
}
void QVRMinimalExample::deserializeDynamicData(
        QDataStream& ds) {
    ds >> _rotationAngle;
}
```

QVR Example Application

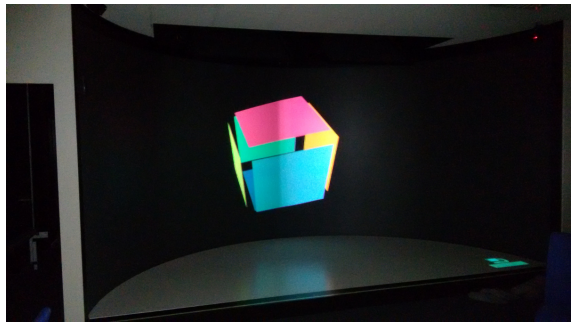
Putting it all together: a minimal example program



Desktop Test



Oculus Rift



VR Lab

QVR Outlook

What else is there?

- Support for VR hardware:
  - HTC Vive via OpenVR
  - Oculus Rift via Oculus SDK
  - Google Cardboard and Daydream via Google VR NDK
  - Tracking / interaction devices via VRPN
- Output plugins for arbitrary postprocessing of views
  - Edge blending, warping, color correction for multi-projector setups
  - Special stereo output modes not covered by QVR
- Example programs
  - qvr-example-opengl-minimal: rotating cube
  - qvr-example-opengl: simple demo scene with ground floor
  - qvr-example-openscenegraph: full-featured OSG viewer
  - qvr-example-vtk: VTK visualization pipeline
  - qvr-sceneviewer: viewer for many 3D model and scene files
  - qvr-videoplayer: a video screen for 2D and 3D videos
  - qvr-vncviewer: VNC viewer (display remote desktops)

QVR Outlook

How do I get it?

- Go to https://marlam.de/qvr
- Download the latest version
- Unpack
- Make sure you have Qt
- Build *and install* `libqvr` first
- Build the examples and link against the *installed* `libqvr`