Virtual Reality Application Programming with QVR

Computer Graphics and
Multimedia Systems Group

University of Siegen

May 2, 2016

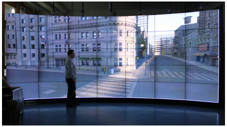
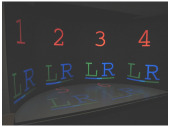
Overview

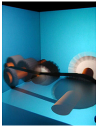
- Challenges for VR frameworks
- Solutions
- QVR Overview and Concepts
- QVR Application Interface
- QVR Configuration and Management
- QVR Example Application
- QVR Outlook and Limitations

Challenges for VR frameworks

Challenges

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

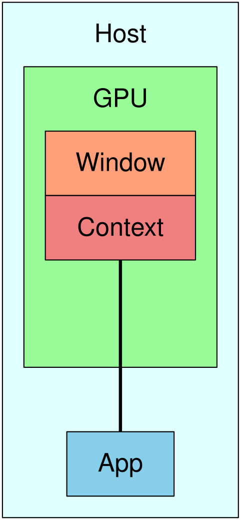


- In general, a VR application must handle
  - Multiple hosts (for render clusters)
  - Multiple GPUs on a host
  - Multiple displays devices attached to a GPU

  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

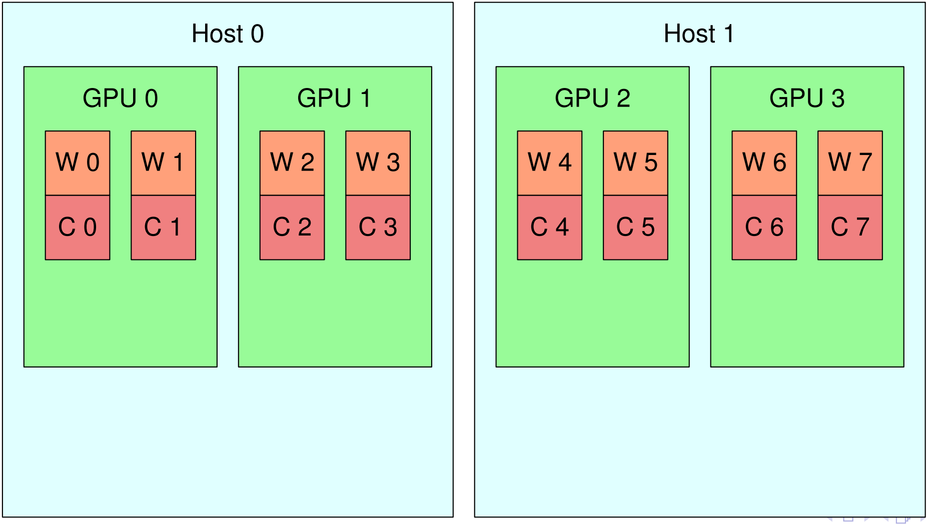
Typical non-VR graphics application



- 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



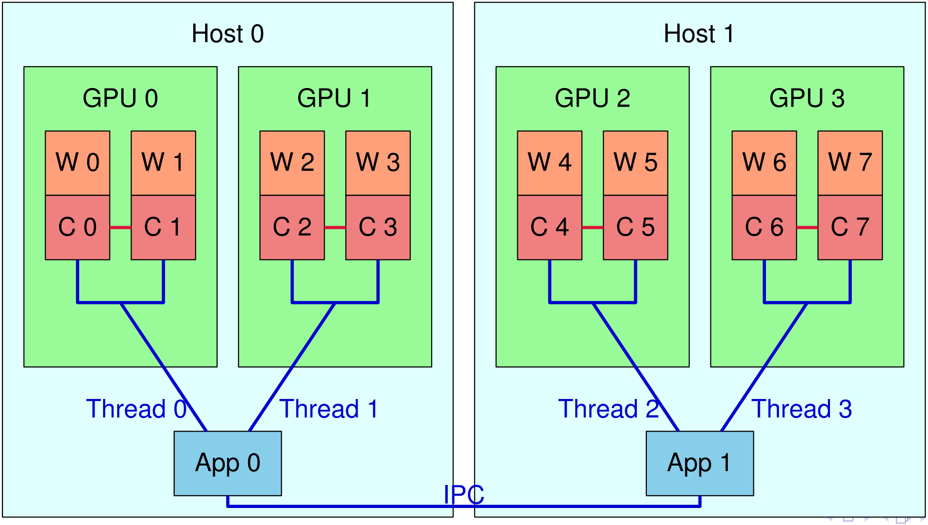
Challenges for VR frameworks

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

Multi-Context Multi-Thread Approach



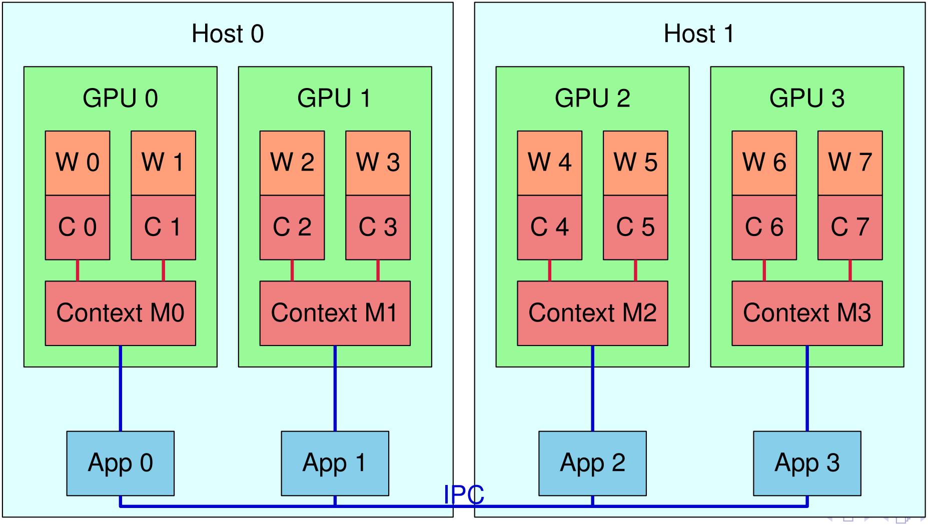
Solutions

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

Single-Context Single-Thread Approach



Solutions

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

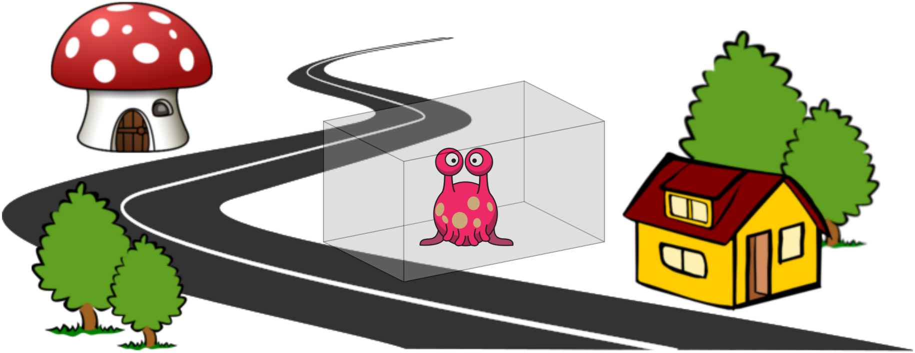
QVR Overview and Concepts

The QVR framework

- Implements the single-context single-thread approach
- Based on Qt (requires nothing else)
- Manages three major types of objects:
  - *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 single interface:
  - *render()* to render a view of the scene for a window
  - *update()* for animations and other scene updates
  - Optional: functions for one-time and per-frame actions per process and per window
  - Optional: serialization functions for multi-process support
  - Optional: keyboard/mouse event handling
- Applications run unmodified on different setups

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 limited space.

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 tracked inside a limited space
- Configured through QVRObserverConfig
  - Navigation
    - Type and parameters (e.g. VRPN, WASDQE)
    - Initial position and orientation
  - Tracking
    - Type and parameters (e.g. VRPN, Oculus Rift)
    - 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
  - For Qt: screen number and window geometry
  - Virtual world coordinates of the window's screen wall
    - Either for screen center (extent computed from display properties)
    - Or for bottom left, bottom right, top left corners
  - Flag: is screen wall fixed to observer?
- 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 master process; slave processes are started automatically when needed
- Configured through QVRProcessConfig
  - Display to talk to (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 master and slave 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, int viewPass, unsigned int t)`
  - Called once or twice per window per frame
  - Renders a view for window `w` into texture `t`
  - Up to two render passes are required for a view: `viewPass` is 0 or 1.
  - The `context` contains all necessary information for the view passes

QVR Application Interface

Application: `render()`

```
void render(QVRWindow* w, const QVRRenderContext& context,
            int viewPass, unsigned int t)
{
    // Set up framebuffer object to render into texture
    setupFBO(t);

    // Set up view
    glViewport(0, 0, textureWidth, textureHeight);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    QMatrix4x4 P = context.frustum(viewPass).toMatrix4x4();
    QMatrix4x4 V = context.viewMatrix(viewPass);

    // Render
    ...;
}
```

QVR Application Interface

Application (continued)

- `void update()`
  - Called once before each frame *on the master process*
  - Updates scene state, e.g. for animations
- `bool wantExit()`
  - Called once before each frame *on the master process*
  - Signals if the application wants to exit
- Optional: `void getNearFar(float& near, float& far)`
  - Called once before each frame *on the master 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 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)`

QVR Application Interface

Application (continued)

- Optional: implementation of custom navigation and tracking
  `void updateObservers(const QList<QVRObserver*>& customObservers)`
  - List contains observers that have navigation type `custom` and/or tracking type `custom`
  - Application can set navigation pose and per-eye tracking poses as it wishes
- Optional: Qt-style event handling for mouse and keyboard
  - `keyPressEvent()`, `keyReleaseEvent()`, `mouseMoveEvent()`, `mousePressEvent()`, `mouseReleaseEvent()`, `mouseDoubleClickEvent()`, `wheelEvent()`
  - All 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 view passes
    - Per view pass:
        - 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 QVRObserverConfig instances
  - A list of QVRProcessConfig instances
    - A list of QVRWindowConfig instances
- Configuration file:
  Corresponds 1:1 to QVR*Config classes
  - 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 master
        window my-window
                observer my-observer
                output stereo 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

```
observer oculus
        navigation wasdqe
        tracking oculus

process master
        window oculus
                display_screen 1
                observer oculus
                output stereo oculus
```

QVR Configuration and Management

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

```
observer cave-observer
        navigation vrpn DTrack@localhost 0 1 0 4 1 3 2
        tracking vrpn DTrack@localhost 0
process master-gpu0
        window back-side
                observer cave-observer
                output stereo gl
                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 slave-gpu1
        window left-side
                observer cave-observer
                output stereo gl
                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 slave-gpu2
        window right-side
               observer cave-observer
               output stereo gl
               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 slave-gpu3
        window bottom-side
               observer cave-observer
               output stereo gl
               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 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

- `--qvr-config=<config.qvr>`
  Specify a QVR configuration file.
- `--qvr-fps=<n>`
  Report frames per second every n milliseconds.
- `--qvr-sync-to-vblank=<0|1>`
  Disable (0) or enable (1) sync-to-vblank. Enabled by default.
- `--qvr-log-level=<level>`
  Set a log level (`fatal`, `warning`, `info`, `debug`, `firehose`).

QVR Configuration and Management

QVRManager main render loop overview
(without handling of slave processes and events)

```
while (!app->wantExit()) {
  app->getNearFar(near, far);
  app->preRenderProcess(thisProcess);
  for (int w = 0; w < windows.size(); w++) {
    app->preRenderWindow(windows[w]);
    renderContext = windows[w]->computeRenderContext(
                                near, far);
    for (int i = 0; i < renderContext.viewPasses(); i++) {
      app->render(windows[w], renderContext, i,
                  windows[w]->texture(i));
    }
    app->postRenderWindow(windows[w]);
  }
  app->postRenderProcess(thisProcess);
  /* all rendering into window textures is now queued */
```

QVR Configuration and Management

QVRManager main render loop overview
(without handling of slave processes and events)

```
  /* wait until window textures are finished */
  glFinish();
  /* render window textures to screen in window threads */
  for (int w = 0; w < windows.size(); w++)
      windows[w]->renderToScreen();
  /* asynchronously trigger buffer swaps in window threads */
  for (int w = 0; w < windows.size(); w++)
      windows[w]->asyncSwapBuffers();
  /* do CPU work while window threads wait for buffer swaps */
  app->update();
  app->updateObservers(customObservers);
  /* wait until all buffer swaps happened */
  waitForBufferSwaps();
}
```

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 core OpenGL 3.3
- 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()`

```cpp
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 QVRMinimalExample::render(QVRWindow* /* w */,
        const QVRRenderContext& context, int viewPass,
        unsigned int texture) {
    GLint width, height;
    glBindTexture(GL_TEXTURE_2D, texture);
    glGetTexLevelParameteriv(GL_TEXTURE_2D, 0,
        GL_TEXTURE_WIDTH, &width);
    glGetTexLevelParameteriv(GL_TEXTURE_2D, 0,
        GL_TEXTURE_HEIGHT, &height);
    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);
    glFramebufferTexture(GL_FRAMEBUFFER,
        GL_COLOR_ATTACHMENT0, texture, 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(viewPass).toMatrix4x4();
    QMatrix4x4 V = context.viewMatrix(viewPass);

    glUseProgram(_prg.programId());
    _prg.setUniformValue("projection_matrix", P);
    glEnable(GL_DEPTH_TEST);

    QMatrix4x4 M;
    M.translate(0.0f, 0.0f, -15.0f);
    M.rotate(_rotationAngle, 1.0f, 0.5f, 0.0f);
    QMatrix4x4 VM = V * M;
    _prg.setUniformValue("modelview_matrix", VM);
    _prg.setUniformValue("normal_matrix", VM.normalMatrix());
    glBindVertexArray(_vao);
    glDrawElements(GL_TRIANGLES, _vaoIndices,
        GL_UNSIGNED_INT, 0);
}
```

QVR Example Application

Putting it all together: a minimal example program

- For animated rotation: `update()`

```
void QVRMinimalExample::update() {
    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

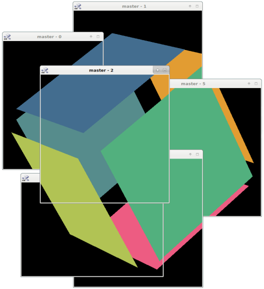
Putting it all together: a minimal example program

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

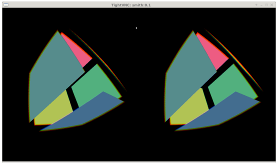
```
void QVRMinimalExample::serializeDynamicData(
        QDataStream& ds) const {
    ds << _rotationAngle;
}
void QVRMinimalExample::deserializeDynamicData(
        QDataStream& ds) {
    ds >> _rotationAngle;
}
```

QVR Example Application

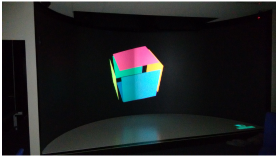
Putting it all together: a minimal example program



Desktop Test



Oculus Rift



VR Lab

QVR Outlook and Limitations

What else is there?

- Output plugins for arbitrary postprocessing of views
  - Specified in configuration file; application does not know
  - Edge blending, warping, color correction for multi-projector setups
  - Special stereo output modes not covered by QVR
- Example programs
  - qvr-minimal-example: rotating cube
  - qvr-helloworld: simple scene with ground floor
  - qvr-sceneviewer: renders many scene/model files
  - qvr-osgviewer: full-featured OpenSceneGraph viewer
  - qvr-vtk-example: VTK visualization pipeline
  - qvr-vncviewer: VNC viewer (display remote desktops)

QVR Outlook and Limitations

Limitations

- Not much tested yet. . .
- Mainly used on Linux so far, limited testing on Windows (but there is no system-dependent code)
- Oculus Rift support still at SDK 0.5 for DK2 (needs updating)
- OpenVR support still missing (for HTC Vive and others)
- OpenGL-based stereo support still missing (will come soon)