VR Juggler
Carlos Andújar, UPC
September 2007

Contents
- What is VR Juggler
- Installing, compiling and running VR Juggler applications (from Getting Started Guide)
- Application objects (from Programmer’s Guide, Chapter 2)
- Sample program: simpleApp
- Helpers: Vec3f & Matrix4ff (from Programmer’s Guide, Chapter 3)
- Getting Input (from Programmer’s Guide, Chapter 3)

What is VR Juggler
- Class library for VR application development.
- Maintained by Iowa State University and Infiscape.
- Released under the GNU LGPL license.
- VR Juggler simplifies common tasks in VR apps:
  - Windowing: window creation and configuration
  - Viewing: computation of stereo cameras
  - I/O devices: supports most typical VR devices
  - Graphical User Interfaces: creation of GUI for app control
  - Spatial sound: simple 3D sound support
  - Networking: support to cluster-based applications
- VR Juggler does not provide a scene graph, but can be used with OpenGL, OpenSG and OpenSceneGraph...

Installing VR Juggler
- Detailed instructions can be found in the Getting Started Guide, http://www.vrjuggler.org
- Recommended distribution: Windows installation wizard for Visual C++ 7.1
- Don’t forget to install all development components.
Where to find sample code

Sample code can be found at:
$VJ_BASE_DIR/share/vrjuggler/samples/OGL/simple

- simpleInput: An application that demonstrates how to get input from the
  wand.
- simpleApp: A very simple OpenGL application that draws a small cube
- contextApp: An application demonstrating how to use OpenGL display lists
  in VR Juggler applications.
- configApp: A relatively simple application that demonstrates how user-level
  code can take advantage of the VR Juggler configuration system, JCCL.

Compiling a sample program

- Detailed instructions on compiling sample code can be found in the
- The simplest way in Windows is to load the preconfigured VC++ projects
  (.vcproj files).

Running a sample program (1)

- The same VR Juggler application can run in many VR
  systems, depending on a configuration file.
- Sample configuration files can be found in
  $VJ_BASE_DIR/share/vrjuggler/data/configFiles
- There are special simulator configurations for running in
  a single PC using mouse and keyboard → most
development and testing can be done on any PC!

Running a sample program (2)

- Simulator configurations
- Virtual VR devices simulated with
  mouse and keyboard input.
- The scene is not shown from the
  user’s viewpoint but from a third-
person view.
- User heads are represented as
  blue spheres.
- Wand is drawn as a green
  device.
- Display surfaces are shown as
  semi-transparent rectangles.

Running a sample program (3)

- Basic simulator modes are described in the
  Getting Started Guide.
- Detailed info on configuration files can be found in the
  VRJConfig Guide.
- The simplest one is standalone.jconf
- We have written config files for CRV’s devices.

Running a sample program (4)

- Starting the application:
  simpleApp standalone.jconf
Running a sample program (5)

Moving the simulated head (*)

(*) In single-window mode, the third-person view also changes!

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Key Press</th>
</tr>
</thead>
<tbody>
<tr>
<td>45° turned right</td>
<td>2 on keypad</td>
</tr>
<tr>
<td>45° turned left</td>
<td>3 on keypad</td>
</tr>
<tr>
<td>90° turned right</td>
<td>6 on keypad</td>
</tr>
<tr>
<td>90° turned left</td>
<td>8 on keypad</td>
</tr>
<tr>
<td>180° turned right</td>
<td>0 on keypad</td>
</tr>
<tr>
<td>180° turned left</td>
<td>9 on keypad</td>
</tr>
<tr>
<td>270° turned right</td>
<td>4 on keypad</td>
</tr>
<tr>
<td>270° turned left</td>
<td>1 on keypad</td>
</tr>
</tbody>
</table>

Running a sample program (6)

Moving the simulated wand:

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Mouse Input Key Press</th>
</tr>
</thead>
<tbody>
<tr>
<td>View head forward</td>
<td>ALT+mouse on mouse backhead</td>
</tr>
<tr>
<td>View head forward</td>
<td>ALT+mouse on mouse front</td>
</tr>
<tr>
<td>View head left</td>
<td>CTRL+mouse on mouse left</td>
</tr>
<tr>
<td>View head right</td>
<td>CTRL+mouse on mouse right</td>
</tr>
<tr>
<td>View head up</td>
<td>CTRL+mouse on mouse up</td>
</tr>
<tr>
<td>View head down</td>
<td>CTRL+mouse on mouse down</td>
</tr>
<tr>
<td>Real world left</td>
<td>SHIFT+mouse on mouse left</td>
</tr>
<tr>
<td>Real world right</td>
<td>SHIFT+mouse on mouse right</td>
</tr>
<tr>
<td>Real world up</td>
<td>SHIFT+mouse on mouse up</td>
</tr>
<tr>
<td>Real world down</td>
<td>SHIFT+mouse on mouse down</td>
</tr>
<tr>
<td>Real world clockwise</td>
<td>Left arrow</td>
</tr>
<tr>
<td>Real world counter-clockwise</td>
<td>Right arrow</td>
</tr>
<tr>
<td>Real hand rotate left</td>
<td>Left mouse button</td>
</tr>
<tr>
<td>Real hand rotate right</td>
<td>Right mouse button</td>
</tr>
<tr>
<td>Real hand rotate up</td>
<td>Scroll up button</td>
</tr>
<tr>
<td>Real hand rotate down</td>
<td>Scroll down button</td>
</tr>
</tbody>
</table>

Running a sample program (7)

[DEMO]

Application objects (1)

- Writing a VR Juggler application basically involves creating a subclass (implementing the interface) of one of the pre-defined application classes.
- Each instance of the derived subclass is an application object (~ process)
  - In a traditional multi-process application → the OS schedules the execution of each process
  - In a VR Juggler application → the VR Juggler kernel is the scheduler, invoking the methods of the application objects.

Application objects (2)

The main() function simply starts the kernel, which will start the app object:

```c
#include <vrj/Kernel/Kernel.h>
#include <simpleApp.h>

int main(int argc, char* argv[])
{
    vrj::Kernel* kernel = vrj::Kernel::instance(); // Get the kernel
    simpleApp* app = new simpleApp(); // Create the app object
    kernel->loadConfigFile(...); // Configure the kernel
    kernel->start(); // Start the kernel thread
    kernel->setApplication(app); // Give application to kernel
    kernel->waitForKernelStop(); // Block until kernel stops
    delete app;
    return 0;
}
```

Application objects (3)

Application objects must provide two kinds of methods:
- Application-dependent: declared in the Application Interface. App
  - API-dependent: declared in the Draw Manager interfaces:
    - GlApp (OpenGL)
    - D3dApp (Direct 3D)
    - PFAp (SDL's Performer)
    - OpenGL SGApp (OpenGL, http://www.openscenegraph.org/)
    - OpenSGApp (OpenSceneGraph, http://www.openscenegraph.org/)
The application interface: App

- **App::init()**
  - Initialize application data (the graphics API has not been started yet)
  - Uses: general initialization that should be done only once per execution.
- **App::apinit()**
  - Graphics API-specific initialization (it will be called once the graphics API has been started)
  - Uses: empty in OpenGL applications

OpenGL app class: GlApp

- **GlApp::contextInit()**
  - Initialize GL (lighting, material...)
- **GlApp::bufferPreDraw()**
  - Called for each context, every frame
  - glClear() the color buffer here.
- **GlApp::draw()**
  - Called for each context, or for each eye, every frame.
  - Modelview and projection matrices already configured.
  - Start clearing the depth buffer
  - Do not change the state of any application variables!

The application interface: App

- **App::preFrame()**
  - Called when the system is about to trigger drawing.
  - Uses: updates in response to device input, master nodes can write to shared data.
- **App::latePreFrame()**
  - Called after shared data is sync among the cluster nodes
  - Uses: nodes that read from the shared data (cluster::UserData<T>::isLocal() returns false) should perform state updates.
- **App::intraFrame()**
  - Executes in parallel with the rendering, while the current frame is being drawn.
  - Uses: any processing that can be done in advance for the next frame (time-consuming computations)
- **App::postFrame()**
  - Called after rendering has completed but before VR Juggler updates devices.
  - Uses: any data updates that are not dependent upon input data and cannot be overlapped with the rendering.

simpleApp: main.cpp

```cpp
#include <simpleApp.h>
#include <vrj/Kernel/Kernel.h>
using namespace vrj;

int main(int argc, char* argv[]) {
    Kernel* kernel = Kernel::instance();           // Get the kernel
    simpleApp* application = new simpleApp();    // Instantiate an instance of the app
    if (argc <= 1) exit(1);
    // Load any config files specified on the command line
    for (int i = 1; i < argc; ++i )
        kernel->loadConfigFile(argv[i]);
    kernel->start(); // Start the kernel
    kernel->setApplication(application); // Give the kernel an application to execute
    kernel->waitForKernelStop(); // Keep thread alive and waiting
delete application;
    return 0;
}
```

simpleApp: simpleApp.h

```cpp
#ifndef _SIMPLE_APP
#define _SIMPLE_APP
#include <vrj/vrjConfig.h>
...
using namespace vrj;
class simpleApp : public GlApp {
public:
    virtual void bufferPreDraw();
    virtual void contextInit();
    virtual void draw();
};
#endif
```

simpleApp: simpleApp.cpp

```cpp
#include <simpleApp.h>
...
vvoid simpleApp::bufferPreDraw(){
    glClear(GL_COLOR_BUFFER_BIT);
}
vvoid simpleApp::draw(){
    glClear(GL_DEPTH_BUFFER_BIT); // don't clear color buffer here!
    drawCube();
}
vvoid simpleApp::contextInit(){
    glLightfv(GL_LIGHT0, GL_AMBIENT, ...);
    glMaterialfv(GL_FRONT, GL_AMBIENT, ...);
    glEnable(GL_LIGHTING);
    ...
}
Helpers: Vec3f, Vec4f

using namespace gmtl;

Vec3f vec(1.0, 1.5, -1.0);
vec[0] = 1.0;
vec.set(1.0, 1.5, 2.0);

// operations (self-explaining)
if (vec1 == vec2) {}
normalize(vec);
float length = length(vec);  
vec2 = 3 * vec;
vec3 = vec + vec2;
gmtl::xform(result, M, vec);  

// access to raw data
glVertex3fv(pos.getData());

Helpers: Matrix44f

Matrix44f uses column-major order (first index varies the fastest, like OpenGL, but unlike C/C++ arrays)

#include <gmtl/Matrix.h>

gmtl::Matrix44f mat; // identity matrix

mat.set(0.0, 1.0, 2.3, 4.1); // (0.0 1.0 2.3 4.1)
8.3, 9.0, 2.2, 1.0;  // (8.3 9.0 2.2 1.0)
1.4, 9.8, 9.7, 2.1;  // (1.4 9.8 9.7 2.1)
3.8, 0.9, 2.1, 0.3;  // (3.8 0.9 2.1 0.3)

gmtl::Matrix44f mat(0.0, 1.0, 2.3, 4.1,  
8.3, 9.0, 2.2, 1.0,  
1.4, 9.8, 9.7, 2.1,  
3.8, 0.9, 2.1, 0.3,  
float val = mat[3][0];  // [row][column]
val = mat[3](0);  // row[column]

Getting input

VR Juggler is designed to avoid applications to depend on specific input devices → uses several intermediate classes:

- **Input devices** (grouped into several categories)
- **Device proxies** (access input devices on the same category through a unique interface)
- **Device interfaces** (enables configuration-based acquisition of input devices, i.e., attachment of a proxy with a particular input device)

### Input devices

- There is an input device class for each input device.
- Grouped into these categories (classes):
  - **Analog** Analog devices
  - **Command** Command-oriented input devices
  - **Digital** Digital (on/off) input devices
  - **Glove** Glove input devices
  - **KeyboardMouse** Keyboard/mouse input handlers
  - **Position** Tracking input devices
  - **String** String (text- or word-driven) input devices

### Device proxies

- A device proxy is an intermediary who forwards information between the application and an input device.
- There is a device proxy class for each type of input device.
- All device proxies have a data request method:
  - **AnalogProxy** getValues() → float in the range [0.0, 1.0]
  - **DigitalProxy** getValues() → gadget::Digital::State, an enumerated type
  - **GloveProxy** getValues() → gadget::GloveData
  - **KeyboardMouseProxy** getValues() → KeyboardMouse::EventQueue
  - **PositionProxy** getValues() → Matrix44f
  - **StringProxy** getValues() → std::string

(*) OFF → Device is in the “off” state.
ON → Device is in the “on” state.
TOGGLE_ON/TOGGLE_OFF → Device was “on” and has changed to “off” since the last frame.
(“) The queue contains all the key and mouse events that occurred since the last frame.
Device interfaces

- **Device interfaces** hide the details of proxy acquisition through the Input Manager.
- Device interfaces act as smart pointers.
- These are the objects we really use in app objects:
  - AnalogInterface
  - CommandInterface
  - DigitalInterface
  - GloveInterface
  - KeyboardMouseInterface
  - PositionInterface
  - StringInterface

Using device interfaces

- All device interface objects must be initialized in `vrj::App::init()`.
- The parameter is the symbolic name/alias of the device (both defined in config files).

```cpp
class myApp : public GlApp {
  // ...
  gadget::PositionInterface head;
};
```

- Access, usually in `preFrame()`:
  ```cpp
  Matrix4f matrix = head->getData(); // feet units!
  ```

Keyboard and mouse events

```cpp
void MyApp::preFrame() {
    gadget::KeyboardMouse::EventQueue evt_queue = mKeyboard->getEventQueue();
    for (auto &event : evt_queue) {
        if (event.type() == gadget::KeyPressEvent || event.type() == gadget::KeyReleaseEvent) {
            gadget::KeyEventPtr key_evt = dynamic_cast<gadget::KeyEvent>(event);
            // Handle the key press event ...
        }
        else if (event.type() == gadget::MouseButtonPressEvent || event.type() == gadget::MouseButtonReleaseEvent) {
            gadget::MouseEventPtr mouse_evt = dynamic_cast<gadget::MouseEvent>(event);
            // Handle the mouse event ...
        }
    }
}
```