

Manual

Making 3D Applications Compatible with Virtusphere

Table of Contents

Who is this manual for?.....	1
Virtusphere - The Virtual Reality System.....	1
What is Virtusphere?	1
Features of Virtusphere as a User Interface System.....	1
Example of Application of Virtusphere with a 3D Program.....	2
Advanced Options and Commands	5
Integration of Virtual Reality Peripheral Network (VRPN) in 3D applications	6
Brief Description of VRPN	6
Why Use VRPN in 3D Application?.....	6
Installation and Setting of VRPN SDK.....	6
How to Use VRPN?.....	8
Links	10

Who is this manual for?

This manual is intended for companies developing user interface subsystems for 3D-applications with support for Virtusphere in their projects.

Virtusphere - The Virtual Reality System

What is Virtusphere?

Virtusphere is a virtual reality system, which fundamentally changes the way of human-computer interaction. The patented method and system of providing infinite spaces allows the most complete immersion into virtual world for training of a wide variety of specialists and for entertainment purposes.

Virtusphere consists of a large hollow sphere located on a special platform allowing the sphere to rotate at 360 degrees. A user wearing a virtual helmet with a 3DOF sensor on his/her head can walk inside the sphere. The sphere allows him/her to move in any direction for any distance avoiding collisions with real world objects. The 3DOF sensor on the virtual helmet determines orientation of the user's head. A sensor under the sphere determines user's movement in space. For applications requiring interaction with virtual objects the user has a manipulator, which can be either mouse-based or gamepad-based. The direction of manipulator's (hand's) action is usually the same as orientation of the user's head. For specific applications with the manipulator a separate 3DOF sensor is used. In this case the direction of manipulator's (hand's) action is independent of the head's orientation and is determined separately. A computer builds a corresponding three-dimensional image based on live data from all sensors and the manipulator and transmits it to the display and the user's virtual helmet. Virtusphere is a natural user interface. It allows users to go deeper into virtual space and interact with it.

Features of Virtusphere as a User Interface System

In order to use 3D applications with Virtusphere the user interface system of 3D engine needs to be modified. It is recommended to use VRPN (Virtual-Reality Peripheral Network) as an interface for interaction between 3D applications and Virtusphere.

Data from 3DOF sensors on the virtual helmet and on the user manipulator can be received via VRPN libraries discussed below. Received coordinates can be directly used for direction of virtual camera.

Data of player's movements in space is provided by the sensor located under the sphere, which is treated by an application as an ordinary optical mouse. It is necessary to use data of changes of the mouse coordinates instead of absolute coordinates in order to provide infinite user's movement. It is recommended to filter data from the sensor to ensure smooth user's movement in the virtual space.

An example of such filter is provided below in section "Example of Application of Virtusphere with a 3D Program". The settings should include user's movement speed in both axes of Virtusphere, inversion of axes directions, and interchange of X and Y axes (turning the coordinate system at 90°) to ensure system's flexibility. If a sensor monitoring height of the 3DOF sensor (Z-coordinate) is not included in Virtusphere's configuration, the height of user's view above the ground camera is given as a constant value (usually 7 feet).

If a standard method of processing user interface via keyboard and mouse is used, the direction of player's movement depends on the direction of his glance. When using Virtusphere the direction of player's movement is not connected to the direction of his/her glance and it is determined only by changes of mouse coordinates. Unambiguous equivalence between the mouse coordinate system and the user's view camera coordinate system should be ensured.

Zero rotation of the user's view camera must correspond to the direction of +Y axis of the mouse. Prior to launching of application orientation of the user's view camera in virtual space and orientation of the user's head in real space should be synchronized. To do this, the user has to enter Virtusphere, turn to the mouse's +Y axis direction (this direction is marked in Virtusphere for convenience), and look directly in front of him/her and in parallel to the ground. Calibration of the system's 3DOF sensor in this application should be performed, the corresponding functionality should be provided in 3D application. Calibration means receiving and saving current orientation of the user's head. During the work cycle orientation of the user's head is corrected based on calibration.

Calibration:

```
Quat zero_angles = UserInput.Head.Quat();
```

Work cycle:

```
Quat head_angles = UserInput.Head.Quat() - zero_angles;
```

At the time of calibration the 3D application user's view camera must have a zero turn. If the 3D application requires user's orientation in direction other than zero, the coordinate system of the user's view camera can be turned correspondingly.

To navigate in the application's menu an additional manipulator in the user's hand can be used, or Virtusphere itself can be used simply as a mouse. For this purpose the settings should also include pointer speed in the menu, inversion of movement direction, and interchange of X and Y axes (turning the coordinate system at 90°).

3DOF sensors should include settings of order of Euler angles for sensor orientation and inversion of each Euler angle. This must provide compatibility with different sensor models.

Example of Application of Virtusphere with a 3D Program

This example shows possible realization of the user interface processing system in 3D applications using Virtusphere with VRPN and integrated quat library. Hypothetic class PlayerControl that processes user input and controls user's virtual character is described below.

Example of the method calculating Euler angles of orientation of the user's view camera - `m_head_angles`, which are members `PlayerControl` class.

```
void PlayerControl::Look()  
{
```

Calculate orientation quaternion of the tracker on the user's head:

```
    q_type q = UserInput.Head.Quat();  
  
    if( q[0] == 0f && q[1] == 0f && q[2] == 0.0f && q[3] == 0f ) {  
        q[3] = 1f;  
    }  
}
```

Calculate Euler angles from quaternion:

```
    q_vec_type angles;  
    q_to_euler( angles, q );
```

Calculate order of Euler angles from configuration:

```
    int pitch_i, yaw_i, roll_i;  
    GetTrackerAnglesOrderIndices( Config.HeadTrackerAnglesOrder,  
        pitch_i, yaw_i, roll_i );
```

Convert Euler angles from radians into degrees, make order of Euler angles to be in accordance with configuration, and invert each angle, if indicated in configuration:

```
    m_head_angles.Pitch = RAD2DEG( angles[ pitch_i ] ) *  
        (Config.HeadTrackerPitchInv ? -1.0f : 1.0f);  
    m_head_angles.Yaw = RAD2DEG( angles[ yaw_i ] ) *  
        (Config.HeadTrackerYawInv ? -1.0f : 1.0f);  
    m_head_angles.Roll = RAD2DEG( angles[ roll_i ] ) *  
        (Config.HeadTrackerRollInv ? -1.0f : 1.0f);
```

Correction of orientation based on calibration value and turn of the camera coordinate system at `m_delta_view_angles` to provide a random orientation of the character in 3D space at the moment of application launch:

```
    m_head_angles = m_head_angles - m_head_zero_angles +  
        m_delta_view_angles;  
}
```

Example of the user's new position calculation method – m_pos included into PlayerControl class:

```
void PlayerControl::Move()  
{
```

To provide virtual character orientation in space at the beginning of the application launch, turn the camera orientation system at m_delta_view_angles:

```
Vec3 x_axis = Vec3( 1, 0, 0 );  
Vec3 y_axis = Vec3( 0, 1, 0 );  
  
m_delta_view_angles.RotatePoint( x_axis );  
m_delta_view_angles.RotatePoint( y_axis );
```

Mouse movement (Virtusphere movement) along each axis is obtained; user movement speed from configuration is applied to mouse movement.

```
float mdx = UserInput.Mouse.FilteredDeltaX() *  
           Config.UserSpeedX;  
float mdy = UserInput.Mouse.FilteredDeltaY() *  
           Config.UserSpeedY;
```

User's new position in virtual space is obtained. Y-axis of Virtusphere corresponds to X-axis of virtual space (a zero turn), and X-axis of Virtusphere corresponds to Y-axis of the virtual space. X and Y axes of Virtusphere are interchanged according to configuration:

```
Vec3 xmove = x_axis * Config.SwapMouseAxis ? mdy : mdx;  
Vec3 ymove = y_axis * Config.SwapMouseAxis ? mdy : mdx;  
m_pos += xmove + ymove;  
}
```

Example of data filtration from Virtusphere movement sensor built in Mouse class:

A part of declaration of Mouse class, m_filer class member is declared:

```
class Mouse  
{  
    ...  
private:  
    float m_filer[2];  
    ...  
}
```

Method, called for object initialization, resets the filter:

```
void Mouse::Init()  
{  
    ...
```

```

memset( m_filter, 0, sizeof( m_filter ) );

...

}

```

Method, called at each work cycle, before processing of the user input:

```

void Mouse::Loop()
{
    ...
    m_filter[0] = (m_filter[0] * 6.0f + DeltaX() * 1.0f) /
        ( 6.0f + 1.0f );
    m_filter[1] = (m_filter[1] * 6.0f + DeltaY() * 1.0f) /
        ( 6.0f + 1.0f );
    ...
}

```

Methods, returning filtered movement of Virtusphere along X and Y axes correspondingly:

```

float Mouse::FilteredDeltaX() const
{
    return m_filter[0];
}

float Mouse::FilteredDeltaY() const
{
    return m_filter[1];
}

```

Advanced Options and Commands

Advanced Options Recommended for Introduction into the Application Settings (in .ini, .cfg and similar file types)

- User's movement speed along X and Y axes of Virtusphere with features of entering negative values for axes inversion separately for menu navigation and movement in the virtual space;
- Interchange of X and Y axes (turning Virtusphere coordinate system at 90°), separately for menu navigation and movement in the virtual space;
- Order of Euler angles for head tracker and for manipulator tracker;
- Inversion of each Euler angle, separately for head tracker and for manipulator tracker;

Additional Commands Recommended for Introduction into the Application

- Restart of VRPN subsystem;
- Calibration (initialization) of trackers;

Integration of Virtual Reality Peripheral Network (VRPN) in 3D applications

Brief Description of VRPN

VRPN is a class library and existing servers developed for implementation of transparent network interface between applications and physical devices used in virtual reality systems. Its main purpose is allocation of a separate PC or host for each virtual reality system station of the control periphery (trackers, button devices, analog devices, tactile devices, audio devices, etc.). VRPN provides connection of the application with all devices using an appropriate mechanism for each type of device. The application does not need to be concerned about network topology and physical location of peripherals, no matter if either a remote host or a local computer is used for running of the application. Also VRPN-server and VRPN-client can be introduced either as separate applications or as a single application without changing of program code.

VRPN provides such level of abstraction which allows using all devices of a single basic class in the same manner, e.g. all trackers are used via single `vrpn_Tracker` basic class. It means that all trackers provide uniform data. Along with that, the application which requires use of specific features of a certain tracker model (e.g. signaling a certain tracker type on data frequency transfer) can obtain a class intended for the specific tracker type. Currently, the following basic classes have been introduced: Analog, Button, Dial, ForceDevice, Sound, Text, and Tracker. Each of these abstractions represents a set of rules for a specific type of device. For each type of device there are one or more servers and a client class for data reception from the device and control of the device.

Why Use VRPN in 3D Application?

First of all, in first person view 3D applications, VRPN can be used for reception of data on user's glance direction from 3DOF sensor, i.e. tracker. VRPN use ensures application compatibility with a great number of existing trackers. 3DOF tracker transfers data on its space orientation as quaternion which can be converted into Euler angles any time. Therefore, tracker on the user's head can be used for acquisition of the user's head position in space and also substitute for the mouse traditional for first person view games. Combination of this technology with the virtual reality helmet allows much deeper immersing of the player into the virtual world than before.

Installation and Setting of VRPN SDK

The most recent version of VRPN SDK can be found at <ftp://ftp.cs.unc.edu/pub/packages/GRIP/vrpn/>. This manual corresponds to version 07.20. Unpack downloaded SDK archive to any suitable folder (for convenience let it be `c:\vrpn`). Support of trackers by Inter Sense requires InterSense SDK, which can be downloaded at <http://isense.com/support.aspx?id=373>. This manual corresponds to version 4.03. To activate support of InterSense trackers in VRPN create 'isense' folder in folder `c:\vrpn\`, copy and paste 3 files from InterSense SDK into this folder:

- `isense.c`;
- `isense.h`;
- `types.h`,

activate InterSense use by changing file `c:\vrpn\vrpn\vrpn_Configure.h`, line #135 must be changed:

```
//#define VRPN_INCLUDE_INTERSENSE
```

to

```
#define VRPN_INCLUDE_INTERSENSE
```

As this version of VRPN SDK was created using InterSense SDK v. 3.45, VRPN SDK original code must be changed to provide compatibility with InterSense SDK v. 4.0.3. The following lines in 'vrpn_Tracker_isense.c' file must be changed:

lines #352-355:

```
ISD_TRACKER_DATA_TYPE data;  
int i;  
  
if(ISD_GetData(m_Handle,&data)) {
```

to:

```
ISD_TRACKING_DATA_TYPE data;  
int i;  
  
if(ISD_GetTrackingData(m_Handle,&data)) {
```

lines #429-438

```
    d_quat[0] = data.Station[station].Orientation[1];  
    d_quat[1] = data.Station[station].Orientation[2];  
    d_quat[2] = data.Station[station].Orientation[3];  
    d_quat[3] = data.Station[station].Orientation[0];  
} else {  
    // Just return Euler for now...  
    // nahon@virttools needs to convert to radians  
    angles[0] = DEG_TO_RAD*data.Station[station].Orientation[0];  
    angles[1] = DEG_TO_RAD*data.Station[station].Orientation[1];  
    angles[2] = DEG_TO_RAD*data.Station[station].Orientation[2];
```

to:

```
    d_quat[0] = data.Station[station].Quaternion[1];  
    d_quat[1] = data.Station[station].Quaternion[2];  
    d_quat[2] = data.Station[station].Quaternion[3];  
    d_quat[3] = data.Station[station].Quaternion[0];  
} else {  
    // Just return Euler for now...  
    // nahon@virttools needs to convert to radians  
    angles[0] = DEG_TO_RAD*data.Station[station].Euler[0];  
    angles[1] = DEG_TO_RAD*data.Station[station].Euler[1];  
    angles[2] = DEG_TO_RAD*data.Station[station].Euler[2];
```

In order to use VRPN in applications, folders with header files (folder C:\vrpn\vrpn) and files of static linking libraries (folder location depends on the platform, compiler and project configuration; exact location can be found in project settings of IDE in use), and VRPN SDK

must be included into corresponding settings of IDE in use. For MS Visual Studio 2005/2008 choose Tools->Options->Projects and Solutions->VC++ Directories->Include files and Tools->Options->Projects and Solutions->VC++ Directories->Library files correspondingly. You can choose VRPN version for static or dynamic linking, these are vrpn and vrpn.dll projects correspondingly.

How to Use VRPN?

As mentioned above, VRPN was introduced together with the client-server technology; for this reason VRPN implementation can be divided into two independent steps.

Client VRPN

To operate client VRPN trackers file `vrpn_Tracker.h` must be included:

```
#include <vrpn_Tracker.h>
```

The next step is building a tracker client object and connecting to a server:

```
vrpn_Tracker_Remote *tkr;  
trk = new vrpn_Tracker_Remote( "Tracker0@localhost" );
```

Initialization procedure and VRPN connection to the server are performed in constructor of `vrpn_Tracker_Remote` class. Constructor parameter is full name of the tracker, which includes the name of device, "@" symbol, and the host name to which the device with VRPN server is connected.

The next step is registration of callback functions to receive data from the tracker:

```
tkr->register_change_handler( NULL, handle_tracker );
```

The first parameter includes the user's data which can be transferred to callback function; leave it blank (make it NULL). The second parameter is callback function address.

Callback function may be given by:

```
void VRPN_CALLBACK handle_tracker( void *userdata,  
                                  const vrpn_TRACKERCB t )  
{  
    printf("Handle_tracker\tSensor %d is now oriented to "  
          " (%g,%g,%g,%g)\n",  
          t.sensor,  
          t.quat[0], t.quat[1], t.quat[2], t.quat[3]);  
}
```

This is just one of the allowed prototypes of the tracker callback function. Possible variants include:

- `vrpn_TRACKERCB` - for obtaining tracker position/orientation
- `vrpn_TRACKERVELCB` - for obtaining tracker speed
- `vrpn_TRACKERACCCB` - for obtaining tracker acceleration

For a complete list of admissible variants of callback functions see VRPN project site or VRPN SDK original codes.

Tracker initialization is complete. Now `mainloop()` method (of `trk` object) must be called as often as possible.

```
while( play ) {
```

```

...
    trk->mainloop();
...
}

```

As the game is completed, free the resources in use:

```

trk->unregister_change_handler( NULL, handle_tracker );
delete trk;

```

Quat library included into VRPN SDK and located in folder C:\vrpn\quat can be used for work with quaternion of tracker orientation received by the callback function.

Server-end

The easiest way to implement application server-end is to use the existent VRPN server included into VRPN SDK (vrpn_server project). It will allow quick reception of the running system. A standard VRPN server is configured with vrpn.cfg file located in the folder of vrpn_server project (C:\vrpn\vrpn\server_src\vrpn.cfg). The example of vrpn.cfg file is shown below:

```

vrpn_Tracker_InterSense Tracker COM1
#vrpn_Tracker_InterSense Tracker COM2

vrpn_Keyboard Keyboard

```

In this file, each device is recorded in one line and includes device class, device ID, and advanced options. The set and admissible values of advanced options vary with different device classes. The set of supported device classes can be found in VRPN server original codes. Device ID is a name referring to the device from the client-end. Allowed set of advanced options can be found in standard VRPN server original codes. Lines beginning with “#” symbol are treated as comments and are ignored, as well as empty lines. In the example above, two devices with identifiers Tracker (of vrpn_Tracker_InterSense class), and Keyboard (of vrpn_Keyboard class) are built. Tracker device must be connected to COM1 port as specified in advanced options.

Of course, use of a separate VRPN server for the end-product may be impossible, so the option of implementation of the server embedded into the game is considered below.

At the beginning, include required header files:

```

#include <vrpn_Connection.h>
#include <vrpn_Tracker.h>

```

Besides the required header files, header files of the classes of specific tracker models must be also included. For example, include Inter Sense trackers:

```

#include <vrpn_Tracker_ISense.h>

```

Now create object of vrpn_Connection class:

```

vrpn_Connection *connection = vrpn_create_server_connection();

```

vrpn_create_server_connection() function has several variants of implementation with different options. For detailed information see VRPN project site or VRPN SDK original codes.

Now, tracker server object can be created:

```

vrpn_Tracker *tracker;

```

```
tracker = new vrpn_Tracker_InterSense( "Tracker0", connection, 2 );
```

Options of server class constructors differ with different tracker models. Tracker Inter Sense described in the example accepts tracker name, connection object, and computer port number to which tracker is connected as parameters.

Now, in the main loop of the game `mainloop()` method must be called by tracker and connection objects:

```
while ( play ) {  
    ...  
    tracker->mainloop();  
    connection->mainloop();  
    ...  
}
```

As the game is completed, free the resources in use:

```
delete tracker;  
delete connection;
```

Of course, this server should provide initialization of tracker objects for different device models and determination of different settings of the trackers (computer port number, data exchange rate, etc.) depending on user-defined setting.

Client and Server in One Process

If both VRPN client and server parts are performed in one process you can avoid using network protocol for data transfer, what will have a positive effect on user input processing rate. For this purpose, constructor of client-end tracker object should be given the same connection object as the server-end tracker object, as a parameter.

```
trk = new vrpn_Tracker_Remote( "Tracker0", connection );
```

Tracker name has changed, now “@” symbol and host name are excluded.

Links

Virtusphere Project: <http://virtusphere.com>

VRPN Project: <http://cs.unc.edu/Research/vrpn/index.html>

InterSense SDK <http://isense.com/support.aspx?id=373>.